EZAsm

COLLABORATORS			
	TITLE : EZAsm		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 17, 2023	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

EZAsm

1.1 EZAsm documentation

```
                                        EZAsm

              Version 1.9   May '94
                By Joe Siebenmann



DISCLAIMER
  STATEMENTS

New features

What is EZAsm?

Addition  Subtraction

Setting up EZAsm

Multiplication  Division

Addressing modes

And  OR  Exclusive-OR

Using functions

Shift  Left/Right

Tag list functions

Assignment

Importing functions

If Else Syntax

Option switches
```

1.2 DISCLAIMER

1.3 New Features, Bug Fixes

```
  Bug fixes:

  1.8:

  -  Size arguments for "++" and "--" stopped working.


    foo = Function()  foo = Function()
    foo != 0 Quit   foo <op> <operand>

  -  In trying to optimize by eliminating the following "tst"
     it could remove the 2nd statement by mistake.

  -  Problems using "MACRO", "ENDM" above your code.



  New Features:

  1.8:

o Function argument syntax is now much more relaxed.
  You can now use: "Permit()", "OpenWindow(&NewWin)", etc..

o New optimizations using PC relative addressing.
  A size decrease of over 11% was achieved on "scrwin".

o Better handling for allowing assembly statements to
  come before EZAsm statements, and better detection of where
  your code starts.

o Improved handling of paths for source and output files.


  1.9:

o All 3.0 functions are supported, including the CD32 libraries
  lowlevel and nonvolatile, also reqtools and powerpacker
  libraries.

o Doc files are now in AmigaGuide format, viewable with
  MultiView or standard text viewers.

o EZAsm is now more than 50% faster!

o To avoid possible problems with some assemblers, the leading "."
  for labels, string constants etc. has been changed to "_".

o Comments are more flexible now, beginning at ANY column
  using "*" or ";".

o Use of "=" is supported as an alternative to "EQU".
```

1.4 What is EZAsm?

EZAsm was written to make programming in 680X0 assembly language MUCH easier! EZAsm combines 680X0 assembly language with parts of C. The result is highly optimized code, with almost HALF the development time!

A68k (by Charlie Gibbs) and Blink (from The Software Distillery) are included, so you'll have everything you need to start producing executables. (except Commodore's include files)

Here are some of its advantages:

o Your code is converted into the fastest possible assembly statement(s), so you automatically write "good" code.

o More structured. Compare and bit test statements can have braces and "else" like C. Being able to use braces lets you use assembly language in a whole new way!

o C-like Amiga function calls! Every 3.0 function in every library is supported.

o Import functions from other libraries using ".fd" files.

o All needed libraries including resources, and those imported from ".fd" files, are opened and closed automatically.

o Your code is much more readable, and easier to debug. You can code nearly twice as fast, with fewer syntax errors.

o No more having to constantly look up which condition code to use for compares, bit tests, or maximum numbers for "moveq" or "addq" etc.

o You can freely mix assembly statements with "EZAsm statements".

You need to know a little about assembly language and C operators before you dive right in. If you're new to 68000 assembly language, I suggest looking at one of the many available books on the subject. The source code for "Mk", "scrwin" and "scrwintags" are included as examples.

1.5 Setting Up EZAsm

Create a directory, and copy these files into it:

EZAsm, A68k, Blink, Mk, ez.lib, nm, dat

You can either CD into your directory or take
advantage of the new path handling:

1. Add "assign EZASM: [path][YourEZAsmDir]".

2. Add EZASM: to your path: "path EZASM: add"
(Use "path" to check that it worked)

(Mk and EZAsm will first check to see if you're
CD'd into the directory, then will try looking for
the ASSIGN)

Paths:

If a path to the source file is given, the output file
will go to same directory. If no path, EZAsm uses the
current directory for source and output files.

(Another good method is to create an ALIAS to execute
a file that loads everything into VD0:, SD0:, or RAD:,
CD into it, and work from there)

Recoverable ram disk fans! Check out "StatRam" (SD0:)
on Fred Fish #915. I got VD0: to work under 3.0,
but this is better!

Mk

Mk is a make-like program, written in EZAsm, that will
Execute() the necessary programs for you, creating an
executable file with a minimum of effort. It also
supports an optional include path for A68k, and support
for a ".mak" file. The source code is included.

(!! WShell users might need to rename Mk !!)

Usage:

Mk [-iPathToIncludes] [path]file

(where "file" is your source file, NO extension)

Execute()s these commands:

```
ezasm file.s
a68k [-iPathToIncludes] file.asm
blink FROM file.o LIB ez.lib TO file
```

.mak

For times when you need to supply more options for A68k or
Blink, or invoke other assemblers or linkers, Mk supports
a .mak file. Just create a file with the same name as your
source file, but with a ".mak" extension. The file should

```
  contain the EZAsm, A68k, Blink or other commands, with any options
  or switches you need, as if you were entering them from the CLI.
  Any comments should begin with a non-alpha character.
  If Mk finds the .mak file, it will Execute() those commands,
  otherwise it will Execute() the standard commands.


Getting Started:

  A good way to begin using EZAsm is to take your existing
  code and gradually make these changes:

  -  Eliminate all your opening and closing of libraries,
     and any "SECTION" statements, EZAsm takes care of all
     that for you.  For data, you might need to move your "END".

  -  Declare your variables.

  -  Start by replacing the statements that make up your Amiga
     function calls with the simple C-style calls.

  As you get used to the easier compares and other
  "EZAsm statements" you can incorporate them into new, or other
  sections of your code.
```

1.6 Addressing Modes

```
                  Operand Legends
                                    Operand Type

     Mode       [A] [B] [C] [D] [E] [F]

Dn               *   *   *   -   *   -
An               *   -   *   -   -   -
(An)             *   *   *   *   *   *
(An)+            *   *   *   *   *   *
-(An)            *   *   *   *   *   *
d16(An)          *   *   *   *   *   *
d8(An,Xn)        *   *   *   *   *   *
16 bit addr      *   *   *   *   *   *
32 bit addr      *   *   *   *   *   *
d16(PC)          *   -   -   -   *   *
d8(PC,Xn)        *   -   -   -   *   *
immediate        *   -   -   -   *   -

bd(An,Xn)        *   *   *   *   *   *     68020/68030
([bd,An],Xn,od)  *   *   *   *   *   *     68020/68030
([bd,An,Xn],od)  *   *   *   *   *   *     68020/68030
bd(PC,Xn)        *   -   -   -   *   *     68020/68030
([bd,PC],Xn,od)  *   -   -   -   *   *     68020/68030
([bd,PC,Xn],od)  *   -   -   -   *   *     68020/68030
```

```
declared variables:

"foo"  becomes  "foo(a5)"  ( d16(An) )
```

1.7 Operand Legends

```
<1-8> 1 - 8
<q> -128 - 127
<n> any byte, word, or long size number

Dn  d0 - d7
An  a0 - a7

{B} byte data size not allowed for An operands

bd  32 bit displacement
od  32 bit outer displacement

Xn      a0 - a7  d0 - d7

     Options:              Examples:

     size:  .w  .l         a2.w  a0.l
     scale: *1 *2 *4 *8    a1*2  a0.w*4  ( 68020/68030 )
```

1.8 Addition Subtraction

```
                     OPERANDS           SIZES
                  Addressing modes

                  Operand Legends
                 ++
--

    [C] <op>            L,W,{B}   *

+=
-=

     Dn <op> [A]        L,W,B

     An <op> [A]        L,W       *

    [D] <op> Dn         L,W,B

    [B] <op> <n>        L,W,B

    [C] <op> <1-8>      L,W,{B}   *
```

```
Examples:

  Total ++
  d1 += 10

Optional Args:

  l, w, b

* When appropriate, these are optimized by making the size word
  (.w) instead of long (.l) for "An" operands.  It's 4 cycles
  shorter, and the upper two bytes are correctly handled!
```

1.9 Multiplication Division

```
                     OPERANDS          SIZES
                  Addressing modes

                  Operand Legends
                 *=

    Dn *= [E]            W

    Dn *= ##             W    *


/=

    Dn /= [E]            W


Examples:

  d0 *= d1
  d2 /= 2

Optional Args:

  w,
  s ( signed  divs/muls )


* ( Improved! ) This optimization results in code that's larger then
  "mulu" or "muls", but will execute much faster.  Not all numbers
  can be optimized.  If the number doesn't work, "mulu"
  or "muls" will be used.

  ( where ## is a word or byte length number )
```

1.10 And OR Exclusive-OR

```
                     OPERANDS          SIZES
                  Addressing modes

                  Operand Legends
                 &=
|=

     Dn <op> [E]         L,W,B

    [B] <op> <n>         L,W,B

    [D] <op> Dn          L,W,B

^=

    [B] <op> Dn          L,W,B

    [B] <op> <n>         L,W,B


Examples:

  Mask &= %11010000
  Flags |= $f0

Optional Args:

  l, w, b
```

1.11 Shift Left/Right

```
                     OPERANDS          SIZES
                  Addressing modes

                  Operand Legends
                 <<
>>

     Dn <op> Dn          L,W,B

     Dn <op> 1-31        L,W,B    *

    [D] <op> 1           W


Examples:

  d2 << d0
  d1 >> 4
```

```
Optional Args:

  l, w, b,
  a = Preserves the sign bit by use of "asr" and "ext.l".
      Use only with right shifts ( ">>" ).

* Normally you're limited to shifting 1-8, or using a
  data register to hold higher shift values.  This
  optimized version is faster, and saves using a data register!
```

1.12 Assignment

```
                     OPERANDS           SIZES
                  Addressing modes

                  Operand Legends
                 =

    [B] = [A]           L,W,{B}

     An = [A]           L,W


Examples:

  temp = Total
  (a1)+ = 0 w

Optional Args:

  l, w, b
```

1.13 If Else Syntax

```
Compare and Bit test statements use this syntax:


    operand <op> operand label



    operand <op> operand {
        .
        .
    }
```

```
    operand <op> operand {
        .
        .

    } else {
        .
        .
    }
```

Size arguments should be placed AFTER the label or brace

1.14 Compare

```
                     OPERANDS           SIZES
                  Addressing modes

                    Brace syntax

                  Operand Legends
                 >=
<=
!=
>
<
=

     Dn <op> [A]         L,W,{B}

     An <op> [A]         L,W

    [B] <op> <n>         L,W,B

  (An)+ <op> (An)+       L,W,B
```

Examples:

```
  Total >= 100 Over

  Buf != 0 {
    FreeMem( Buf 100 )
  }
```

Optional Args:

```
  l, w, b,
  s ( signed )
```

1.15 Bit Test

```
                    OPERANDS        SIZES
                Addressing modes

                  Brace syntax

                Operand Legends
               =
!=


   Dn:0-31 <op> 0-1     L

     Dn:Dn <op> 0-1     L


   [F]:0-7 <op> 0-1     B

    [F]:Dn <op> 0-1     B


Examples:

  d1:0 != 1 EvenRtn

  ($bfe001):6 = 0 LMBDown

  d2:d0 = 1 {
    rts
  }


Optional Args:  ( ignored )

Rules:

o No spaces inside first operand.

o Right operand can only be 0 or 1.
```

1.16 Statement Arguments

```
    b   forces operation to be byte

    w     "       "        "   word

    l     "       "        "   long

    a   preserves sign bit     ( >> )

    s   signed                 ( *=, /=, compares )
```

```
( it's OK to use multiple arguments ( "w s" etc. ))

( these are RESERVED and can't be used as variables or labels
( unless you use upper case ) )
```

1.17 Using Functions

```
               o All 3.0 functions are supported, including the CD32 libraries
  lowlevel and nonvolatile, also powerpacker and reqtools libraries.
  "OBSOLETE" functions are included for backward compatibility.
  See "nm" for supported functions.

o All needed libraries including resources, and those imported
  from ".fd" files, are opened and closed automatically, and their
  bases are added to your variables. ( resources don't need to
  be closed ) ( See also
                XFAIL, LVER
                )



  All functions within these 3.0 libraries and resources
  are supported:

  amigaguide.library    _AmigaGuideBase
  asl.library     _AslBase
  battclock.resource    _BattClockBase
  battmem.resource    _BattMemBase
  bullet.library      _BulletBase
  card.resource     _CardResource
  ciaa.resource     _CiaABase
  ciab.resource     _CiaBBase
  commodities.library   _CxBase
  datatypes.library   _DataTypesBase
  diskfont.library    _DiskfontBase
  disk.resource     _DiskBase
  dos.library     _DOSBase
  exec.library
  expansion.library   _ExpansionBase
  gadtools.library    _GadToolsBase
  graphics.library    _GfxBase
  icon.library      _IconBase
  iffparse.library    _IFFParseBase
  intuition.library   _IntuitionBase
  keymap.library      _KeymapBase
  layers.library      _LayersBase
  locale.library      _LocaleBase
  lowlevel.library    _LowLevelBase
  mathffp.library     _MathBase
  mathieeedoubbas.library   _MathIeeeDoubBasBase
  mathieeedoubtrans.library _MathIeeeDoubTransBase
  mathieeesingbas.library   _MathIeeeSingBasBase
  mathieeesingtrans.library _MathIeeeSingTransBase
  mathtrans.library   _MathTransBase
```

```
  misc.resource     _MiscBase
  nonvolatile.library   _NVBase
  potgo.resource      _PotgoBase
  powerpacker.library   _PPBase
  reqtools.library    _ReqToolsBase
  rexxsyslib.library    _RexxSysBase
  translator.library    _TranslatorBase
  utility.library     _UtilityBase
  workbench.library   _WorkbenchBase


  All functions for these .devices are supported:

  console.device      _ConsoleDevice
  input.device      _InputBase
  ramdrive.device     _RamDriveDevice
  timer.device      _TimerBase


  You're responsible for calling OpenDevice() and
  loading the base to use these .device functions:


BYTE  TimeRequest[40]
LONG  _TimerBase


  OpenDevice( "timer.device" 0 &TimeRequest 0 )
  bne Exit

  a0 = &TimeRequest
  _TimerBase = 20(a0) ;TimeRequest->tr_node.io_Device
    .
    .



o The cia.resource functions AbleICR(), AddICRVector(),
  RemICRVector(), and SetICR() have been changed to
  simplify the OpenResource().  The functions are now
  AbleICRA() and AbleICRB() etc. and open the corresponding
  ciaa.resource or ciab.resource.

o The leading underscores of the bases are necessary so you can
  use includes without your assembler complaining.  Some library
  bases are already defined in some ".i"s resulting in
  "multiply defined symbol" errors.


                AllocMem(100 #CLEAR_PUBLIC)
                           ^
                           | space required

o Function argument syntax is now much more relaxed!
  Arguments must be separated by a space or tab.

o Multi-line function arguments are supported.  As with tag list
```

```
  arguments, the closing parenthesis ( ")" ) shouldn't be on a
  separate line. You should have at least one argument before it.


              a2 = ViewAddress()
              d1 = Read( "DF0:myfile" Buf BufLen )

o Address or data registers can be used for returns if
  they're more convenient.  Return variables can be ANY size.
  Variables used in function arguments must be LONG.


Arguments:

o You can pass registers to function arguments directly.
  If the proper register is already loaded, just pass "*".

o You can also use these, and other addressing modes for function
  arguments and returns: "(An)", "(An)+", "-(An)", "d16(An)" etc.

o EZAsm supports argument strings surrounded by double quotes.
  Strings are automatically NULL terminated.
  The following C character constants are supported:


    \b  backspace
    \f  form feed
    \n  newline
    \r  carriage return
    \t  horizontal tab
    \v  vertical tab
    \\  backslash
    \"  double quote
    \'  single quote
    \nnn  octal character value
    \xnn  hex character value


  Examples:

    "Hello, World!\n"

    "\x1b[32mEZAsm  1.9\x1b[39m\n"


o Sometimes you have to "play around" with using "#", "&",
  variables, constants etc.. Also taking into account if it's
  being pushed onto the stack.  You might be giving it the
  CONTENTS of a variable, when it needs its ADDRESS. ( etc. )

o It keeps track of the current library base.  As long as no user
  labels, or close braces are hit, it will skip reloading
  the base register for functions which use the same library base.
```

1.18 Use of PC Relative code

```
  END

NewWin  ds.w  0 ;align
  dc.w  0,0,640,200
  dc.b  -1,-1
  .
  .

o PC relative addressing is now used, when appropriate,
  to reference data labels.


  The following are used with FUNCTION ARGUMENTS and STATEMENTS
  when referencing declared variables and data labels:

  Address of:

    &NewWin
    #NewWin

    ( These can go to an address or data register:
    "move.l #NewWin,Dn", "lea NewWin(pc),An" )

  Contents of:

    NewWin



  The following directives will not appear in the output assembly
  file, and can occur any number of times.  PC relative
  optimizations should be turned off when the data references
  are beyond 32K of the current position.


  NEAR
    All data references after this will use PC relative
    addressing.  This is turned on by default at the start
    of your code.

  FAR
    All data references after this will not use PC relative
    addressing.
```

1.19 Using Tag list functions

```
  OpenScreenTag( 0
    SA_Width 640
    SA_Height Height
    SA_Depth 2
```

```
    SA_DetailPen -1
    SA_BlockPen -1
    SA_Type $f
    SA_Quiet 1
    SA_DisplayID $8000
    TAG_END )


  Tag list functions are supported.  When using the
  "varargs" version of a function ( above ), the tag list
  arguments are built as data, and the function argument is
  converted to a pointer to the data structure.

  EZAsm relies on the following:

    FunctionA() <- Pointer to tag list version
    Function()  <- "varargs" version


  The following functions HAVE BEEN CHANGED for consistancy:

         ( was )                    ( now )

   _DOSBase:

        System                     SystemA
        [SystemTags]               System

        AllocDosObject             AllocDosObjectA
        [AllocDosObjectTags]       AllocDosObject

        CreateNewProc              CreateNewProcA
        [CreateNewProcTags]        CreateNewProc

   _AslBase:

        AllocAslRequest            AllocAslRequestA
        [AllocAslRequestTags]      AllocAslRequest

        AslRequest                 AslRequestA
        [AslRequestTags]           AslRequest

   _GfxBase:

        ExtendFont                 ExtendFontA
        [ExtendFontTags]           ExtendFont

   _IntuitionBase:

        OpenWindowTagList          OpenWindowTagA
        [OpenWindowTags]           OpenWindowTag

        OpenScreenTagList          OpenScreenTagA
        [OpenScreenTags]           OpenScreenTag


o It knows about every 3.0 tag ( "SA_Height", "WA_Width" etc. )
```

```
  and they'll automatically be converted to their
  equivalents ( $80000024, $80000066 ), no includes are needed.
  All the supported tags are listed in the file "nm"
  below the function names.

o It's important to use these tags!  That's how it differentiates
  between 'regular' and tag list function arguments.

o Tag lists MUST be terminated with "TAG_DONE", "TAG_END"
  ( with NO '0' tag data ) or "TAG_MORE" ( with a tag list address ).

o Be sure to add an include when using tag data arguments like:
  "TRUE", "FALSE", "CUSTOMSCREEN" etc.

o The closing parenthesis ( ")" ) shouldn't be on a separate line.
  You should have at least one argument before it. 8^(

o Variables used for tag data arguments can be ANY size, and are
  correctly loaded automatically.

o When importing tag list functions from libraries and using
  "unknown" tags, I'd recommend setting up a tag list data structure
  like EZAsm does, use includes, and pass its address.
```

1.20 Importing functions from .fd files

```
Usage:

PROTO [path]LibraryName [path]FDName


Example:

PROTO req.library req.fd


  Where PROTO is in upper case, begins at column 1, and
  is placed ABOVE your variable declarations.  The ".fd"
  file is read, and all its functions, both "##public"
  and "##private", become available for use.  The
  library base is added to your variables, and the
  opening and closing of the library is handled
  automatically.

  To avoid any conflicts with other function names, these
  are used first.  Up to 10 ".fd" files can be used.

  "PROTO" is most useful for importing functions from
  PD libraries or new libraries from Commodore.
  You can also fix a "built-in" function's offset
  or argument register(s).

  Here's an example:
```

```
##base _DOSBase
##bias 102
Examine()(D1,D2)  ;( arguments within the first "()" are ignored )
##end

  ( the default bias is 30 )
```

1.21 Option switches

```
   These switches should be placed above your variable declarations,
   in upper case, and should not begin in column one.


   Using command line arguments:

  Because the automatic opening of libraries will
  corrupt A0 and D0, the following "switches" are provided:


ARGS  A0 is loaded into the variable "Args" ( automatically
  created ) and "clr.b -1(a0,d0.w)" is used to NULL terminate
  the argument string.  The length in D0 is lost.
  ( see Mk.s for an example of using "ARGS" )

SAVEARGS
  A0 and D0 are pushed onto the stack with "movem.l d0/a0,-(sp)".
  You're responsible for restoring and using them.



   Other useful switches:

SAVEREGS
  Pushes registers you request onto the stack at the start of
  your code, and restores them before exiting.

  Usage:    SAVEREGS  register list

  Example:  SAVEREGS  d2-d4/a0-a3/a6


  Where register list contains no spaces, and will generate
  these statements: "movem.l d2-d4/a0-a3/a6,-(sp)"
        "movem.l (sp)+,d2-d4/a0-a3/a6"

OPT4  Uses "$4.w" instead of "$4" to load ExecBase, and
  "lea n.w,An", "pea n.w" when appropriate.
  ( If you don't use A68k, I'd recommend using this.
  A68k makes these optimizations, but doesn't like the
  "$4.w" syntax )

EXT Allows you to specify the extension of the output assembly
  file.  The default is ".asm".  Example: EXT .a

NOEXTRAS
```

```
  Removes the loop clearing the stack frame, and
  "moveq #0,d0", before the "rts" at the end of the code.

XFAIL Prevents your program from exiting if OpenLibrary(), for
  the specified library, fails.  Useful when trying to open
  another library if the first attempt fails.

  Example:  XFAIL reqtools.library

LVER  Allows you to specify a version number for an OpenLibrary().

  Example:  LVER  intuition.library 36
```

1.22 Optimizations

```
  In addition to the many "standard" optimizations performed
  on statements, the following are also performed:


  STATEMENT          BECOMES       NOTE


   An = 0           sub.l An,An

   [B] = 0          clr [B]          1


 ( compares )

   [B] < 0
   [B] >= 0                          5

   [B] = 0
   [B] != 0         tst [B]          6
                    bxx label

   ----------------------------------

   [B] += <q>
   [B] -= <q>

   [B] &= <q>
   [B] |= <q>
   [B] ^= <q>

   [B] = <q>        moveq   <q>,d7   2
                    [opr].l d7,[B]


   ( compares )

   Dn <op> <q>      moveq   <q>,d7   2
                    cmp.l   d7,dn
                    bxx     label
```

```
   An <op> <q>      moveq   <q>,d7   2
                    cmpa.l  d7,an
                    bxx     label

   ----------------------------------

   An += <i>        lea  i(An),An
   An -= <i>        lea  -i(An),An   3, 2

   An = <n>         lea  n,An        4, 2
```

<i> maximum: -= 32768   += 32767

Notes:

1 For byte and word sizes the code size is smaller, for long
it's smaller and faster than "move #0,[B]".

2 Only apply to long sized operations.

3 (1-8 handled by "addq", "subq" (see note in Addition Subtraction))

4 (0 handled by "sub.l An,An")

5 Taken as signed.

6 In some cases "tst" is eliminated, or D7 is loaded, see below.

More optimizations:

```
  Buf != 0 {
    FreeMem( Buf 100 )
  }
```

```
          ( standard )                    ( improved )

  tst.l Buf(a5)      move.l  Buf(a5),d7
  beq .l2      beq .l2
  movea.l $4,a6      movea.l $4,a6
  movea.l Buf(a5),a1    movea.l d7,a1
    .        .
    .        .
```

o Instead of doing a "tst", the variable is loaded into D7
(flags set, and same size and number of cycles) where it can
be utilized by a following function call. In this case it
loads D7 into A1, saving 2 bytes, and taking only 4 cycles
instead of 13.

```
  Fhandle = Open( "df0:myfile" 1005 )

  Fhandle = 0 {
    .
    .
  }

    .       .
    .       .
  move.l  d0,Fhandle(a5)    move.l  d0,Fhandle(a5)

  tst.l Fhandle(a5)   bne .l5
  bne .l5
```

o The flags are set by the "move", so the "tst" instruction
  can be eliminated, saving 4 bytes and 9 cycles.
  (a label before the "Fhandle = 0 {" statement disables this)
  Several other optimizations of this type are also supported.

```
  foo > 10 {
    .
    .
    [jmp statement]
  } else {
    .
    .
  }
```

o In cases where a [jmp statement] ("jmp", "bra", "bra.s",
  "rts", "rte", "rtd" or "rtr") is immediately above
  the "else" (above), normal label generation (etc.) for
  jumping past "else" is eliminated.

1.23 Statement Information

o Statements can be indented as you like. Operands, operator,
  and arguments must be separated by at least one space or tab.

o Braces can be nested up to 30 deep.

o Comments can begin in ANY column with "*" or ";", or after
  statements (separated from last argument or operand) using ";".
  Most comments are transferred to the output file.

o Operands supported: @141 (octal), $61 (hex), %1100001 (binary),
  'a' (ASCII), 97 (decimal). ASCII strings in operands can
  contain a maximum of 4 characters.
  (no quotes within quotes permitted)

```
o To make labels and symbols as compatible as possible, they aren't
  checked for illegal characters.  Typically the first character
  must be a letter, underscore "_", or a period ".".
  The rest of the characters can be any of these plus 0-9.

o Labels and symbols are limited to a length of 127.
  ( Check your assembler to find what length they're
  significant to ( usually around 30 ))

o Labels that don't begin at column 1 must be followed immediately
  with ":", otherwise, use of ":" is optional.

o Temporary labels of the form n$, where n consists of decimal
  digit(s), are supported.  These labels are only in effect till
  the next non-temporary label is encountered.  ( be careful
  of "hidden" labels generated by braces )

o "moveq #0,d0", "rts" is automatically inserted for you in the
  closing block of code.  ( See also
                NOEXTRAS
                )

o D7 is used as a scratch register for many optimizations, so be
  careful if you use it.

o Labels and constants that are generated are in the range:

  "_l0" to "_ln"  Program labels
  "_c0" to "_cn"  String constants
  "_t0" to "_tn"  Tag list labels

  Try to avoid accidentally using them.


o "SP", "PC", "CCR", "SR" & "USP" ( upper or lower case )
  are supported, but you must ensure it's used correctly.


IMPORTANT! :

  Don't use register A5!  It holds the base address of the
  stack frame used for variable storage.
```

1.24 Variable Declaration

```
LONG  foo bar
WORD    DMASave
BYTE  Sw RegSave[32] ViewPort[40]


o "var[n]" reserves n consecutive blocks of given size.
  This is a convenient way of allocating a chunk
  of the stack frame for structures, register save areas etc..
  These are aligned on 8 byte boundaries similar to AllocMem().
```

```
  Be sure to use "&" when appropriate, otherwise your data will
  go to $0, resulting in a crash.

o Variables must be separated by at least one space, or tab.

o Word alignment is assured for all variables.

o Declarations must occur JUST BEFORE your program code,
  and begin at column 1, with LONG, WORD, or BYTE in upper case.

o EZAsm uses "link" to allocate a stack frame for storage.
  The maximum size of this space is 32K or greater, depending
  on which processor you're using.  Since this space
  contains "garbage", code is added to clear it out before use.
  The frame size is rounded up to the nearest LONG.


  ( Sometimes variables are "adjusted" for proper loading, and
  "-2(a5)" or other displacement may appear in the output file
  instead of "foo(a5)" or other variable name you might be expecting. )
```

1.25 Argument Information

```
o It's now much more forgiving about your placement of
  assembler directives, and smarter about detecting where
  your code starts.  If you experience any problems
  ( it hit an assembly directive it didn't recognize )
  try placing your data statements below "END",
  and "equ", "xref" and "xdef" type directives above your
  variable declarations, or below "END".
  ( In the final .asm file "END" will be adjusted )

o Instruction size:  IN MOST CASES YOU WON'T NEED
  A SIZE ARGUMENT.  It knows the size of your variables,
  address and data registers, and is smart enough to know
  what size to use.  You'll need to specify a size if
  the data is smaller than the instruction size you want,
  ( d1 = $20 w ) or it can't know the size of the operands
  ( (a2)+ = 3(a0) l ).

  Caution:  Be aware that if you load small variables into
      larger ones, the upper bytes aren't cleared and
      may garbage your result.  Always initialize those
      variables, or restrict the size of further
      operations to "b" or "w".

o Try not to overuse the size arguments.  Once you get confident
  it's sizing your instructions correctly ( by checking the
  .asm file ) you'll find you can eliminate their use almost
  entirely.  By needlessly restricting a long operation to
  a word, or byte, you can miss the "quick" optimization.

o Since "(a1)" refers to the CONTENTS of the byte, word, or long
  that A1 points to, "($dff180)" is used in a similar way.
  ( decimal addr's are also valid: "(14675968)" )
```

```
o Numbers are "converted" for you. ( $f2 -> #$f2  37 -> #37 )
  Operands it doesn't recognize, like "wd_UserPort(a2)",
  are output "as is".
```

1.26 General Information

```
o If at any time you're unsure of what a statement is being
  output as, or want to check something out, just look
  at the output .asm file.  Well placed blank lines in your
  source code can enhance its readability.

o For best viewing of the output file, set your tabs to 8 spaces.

o I think it's a good idea to get away from using include files.
  Most assembly source files I see do this.
  It speeds up the assembler tremendously, and saves
  endless wear and tear on your drives.
```

1.27 Converting C to EZAsm

```
  It's fairly easy to convert any C code into
  EZAsm if you follow these guidelines:


  Allocating structures:

  Whenever you see "struct VSprite SpriteA" etc.,
  you need to either allocate some memory for the
  structure, or set up your own:



BYTE  SpriteA[<structure size>]


---------- or ----------------


CLEAR_PUBLIC  equ $10001

LONG  SpriteA


  SpriteA = AllocMem( <structure size> #CLEAR_PUBLIC )
  beq Exit

---------- or ----------------

SpriteA ds.w  0 ;align
  dc.l  0 ;NextVSprite
```

```
  dc.l  0 ;PrevVSprite
  .
  .
```

(you can look-up the size of the structure in the "StructOffsets" file, or use "#vs_SIZEOF" etc. and let the assembler get its size from the includes)

Some functions will create the structures for you and return a pointer to it (OpenWindow(), OpenScreen() etc.) just declare a LONG, and load the returned pointer:

```
LONG  Window

  Window = OpenWindow( &NewWindow )
  beq Exit
```

Accessing structure elements:

Often you'll need to get at data inside structures. First load the "structure pointer" into an address register, then use the offset of the structure element. Here's a typical example:

```
  Move( Window->RPort, 20, 20 );  /* C */


  a0 = Window
  Move( wd_RPort(a0) 20 20 )

  ---- or ----

  a0 = Window
  Move( 50(a0) 20 20 )
```

(add an include to your source, and the assembler will look-up "wd_RPort" and substitute its offset or, even better, use "50(a0)")

!!! See the included file StructOffsets !!!

Be sure to convert BPTRs (left shift 'um 2 bits) before using them:

```
    a3 = _DOSBase

    Forbid()

    a3 = 34(a3)   ;dl_Root
    a3 = 24(a3)   ;rn_Info
```

```
    add.l a3,a3 ;convert BPTR
    add.l a3,a3

    .
    .
```

1.28 Errors

```
"Illegal argument"

  The argument found was not valid for the operator.  See
  the list of "Optional Args" for the operator.  It must
  be lower case, and be separated from the operands
  by at least a space or a tab.

"Illegal operand"

  One, or both, of the operands are:  not valid for the operator,
  have an invalid number, or byte size was specified for an
  "An" operand ( {B} ).  In most cases it's looking for "Dn" or "An"
  as an operand.  ( look under "OPERANDS" for a valid combination )

"Illegal size"

  The size argument you specified is not valid for the operator.
  Check "SIZES" for valid size arguments.

"Needs size argument"

  It doesn't have enough size information about the operands to
  calculate an instruction size.
  You need to add an l, w, or b size argument.

"Label not found"

  No matching label was found.

"Brace mismatch"

  Checks are made when a closing brace ( "}" ) is hit, and when "END"
  is hit.  If the brace stack is "messed up" at that time, an error
  is given.  If "}" is shown, look from there up.  Both "}" and "END"
  may appear.  If just "END", look for a "{" or "} else {" without a
  matching "}".

"Function not found"

  No function matching your function name was found.  Check case and
  spelling of function name, and be sure there isn't a space before
  the "(".  Check the list of supported function names in the
  file "nm".

"Function argument count incorrect"

  Check the number of arguments you used for the function.  Too many
```

or not enough were used. For tag lists, don't use the "A" version of a function with "varargs". With string arguments, check for a missing '"'. Also look for a missing end ")".

(EZAsm doesn't do much checking of normal assembly statements. A68k and Blink will catch any problems missed by EZAsm and bring them to your attention)

1.29 Acknowledgments

Thanks to the following people who have helped to make EZAsm better:

Martin Combs, for all the testing, bug reports, ez.lib improvements and suggestions.

Wayne Lutz, for the 3.0 info.

Andreas Ackermann, for the suggestions, bug reports, and the PC relative optimization idea.

And others, who sent bug reports and suggestions.

1.30 Contacting the author

Try using EZAsm, I don't think you'll want to go back to "ordinary" assembly language programming. 8^)

Why isn't EZAsm shareware? I don't think you can make much money with shareware. I'd like to get EZAsm into the hands of as many interested people as possible because I believe it'll really make a difference. If you have any suggestions for improvements, found bugs, or just want to say "hello", please write, I'd like to hear from you!

Enjoy!

You can reach me at:

Joe Siebenmann
2204 Pimmit Run Lane Apt 1
Falls Church, VA 22043
(USA)

(703) 893-2579