ADEV11 Development System for AmigaDOS

Version 2.0

A 68HC11 C compiler, assembler, linker, librarian and downloader for the Amiga
Public Domain

Stan Burton
1978 26 St. SE
Medicine Hat, Alta, CANADA
T1A 2G8

Table of Contents

ADEV11 Development System for AmigaDOS
Users Manual

Version 2.0

Acknowledgements

I would like to acknowledge the assistance and cooperation of a number of people that have contributed to the improvement of this product either through the supply of files to incorporate into the distribution or their services as bug finders and reporters.

Contributions

Paul Isaacs - AmigaGuide version of the documents
Ron Eirich - HCload utility

Bug reports

Paul Issacs, Keith Vasilakes ...

NAME

SAsm

SYNOPSIS

SAsm [options] <srcfile> [options]

DESCRIPTION

SAsm is the assembler for the DEV11 system. It is a high level macro cross assembler for the Motorola 6803, 6805, 68HC11 and 68HC16 families and for the Hitachi 6303 family. It is a highly modified version of the publicly distributable DASM V2.12. The 68HC16 code generation has not been well tested since I do not have an HC16 to test it on. If you find any problems with it let me know (Stan).

SAsm produces a relocatable file which can be linked together (Slink) with other modules and/or library elements to produce an executable file. Naturally this includes the ability to have multiple segments within a module and BSS or uninitialized segments.

COMMAND LINE

srcfile: if no extension is specified in the name, .a is added
outfile: if no other output file is specified the file generated is the extension-less srcfile name with a .o extension

The following options are available:

-l[name]	generate list file, if no name is specified extension-less srcfile with .lst extension is used
-oname	generate output to a specific file
-s[name]	generate symbol file, if no name is specified extension-less srcfile with .sym extension is used
-v#	select verboseness 0-4 (default 0, see below)

- 0 (default) Only warnings and errors are generated
- 1 Segment list information is generated after each pass, Include file names are displayed and reasons why another pass is required are given.
- 2 Mismatches between program labels and equates are displayed

on every pass (usually none occur in the first pass unless you have re-declared a symbol name).

3 Unresolved and unreferenced symbols are displayed every pass (unsorted, sorry)

4 An entire symbol list is displayed every pass to STDOUT. (unsorted, sorry)

-p#	select number of passes 2-9 (default 2)
-d	debug mode
-DSYMBOL	predefine a symbol, set to 0
-DSYMBOL=EXPRESSION	predefine a symbol, set to expression

Example: asm master.asm -lram:list -v3 -DVER=4

The verbose options can provide additional information about reasons for failure to assemble a file. Part of this is the reason code:

R1,R2 reason code: R3

where

R1	is the number of times the assembler encountered something requiring another pass to resolve.
R2	is the number of references to unknown symbols which occured in the pass (but only R1 determines the need for another pass).
R3	is a BITMASK of the reasons why another pass is required. See the end of this document for bit designations.

-expressions, as in C. (all expressions are computed with 32 bit integers)
-no real limitation on label size, label values are 32 bits.
-complex pseudo ops, repeat loops, macros, etc....

The following special characters are used in the symbol dump:

????	unknown value
str	symbol is a string
eqm	symbol is an eqm macro
(r)	symbol has been referenced
(s)	symbol created with SET or EQM pseudo-op

LABELS and SYMBOLS

A label consists of one or more characters from the set A-Z, a-z, 0-9. The first character must be alphabetic. There is no limit to the name length. The value the label assumes the range of a 32 bit integer.

The label will be set to the current segment counter either before or after a pseudo-op is executed. Most of the time, the label is set before the pseudo-op is executed. The following pseudo-op's labels are created AFTER execution of the pseudo-op:

SEG, ALIGN

PROCESSOR MODEL

The processor model is chosen with the PROCESSOR pseudo-op and should be the first thing you do in your assembly file. Only one PROCESSOR pseudo-op may be declared in the entire assembly.

SEGMENTS

The SEG pseudo-op creates/sets the current segment. Segments are used to separate the various parts of a program, for example CODE and BSS_DATA; later the linker could place the CODE in the EPROM address range and the data at the ram address range. The use of DS or RMB statements in a segment planned for ROM is not logical.

As a result of the use of relocatable segments the ORG statement found in simpler assemblers is not used; its functionality is passed to the linker.

'Uninitialized' (.U) segments do not produce output. Therefore, output generating statements are not allowed in these segments.

MULTIPLE SOURCE FILES

The pseudo-ops XREF and XDEF make separately assembled source files possible. XREF tells the assembler that a symbol that you will be using is available from another source file. XDEF tells the assembler to make a symbol available to other source files. So, to have a successful reference, one and only one source file must XDEF the symbol and one or more source files must XREF it. Later, when all the component source files have been assembled, the linker takes these deferred references and changes them to specific addresses.

There is currently an unnecessary limitation in the use of XREF'd symbols - only one XREF'd symbol may be used per instruction. The example below should be legal but is not legal due to this limitation.

```
        XREF    stack,stack_size
        LDS     #stack+stack_size-1
```

The limitation was the easiest way to disallow the similar, but not legal, situations shown in the example below. The current object module format (based on the Amiga format) cannot handle these as no native Amiga assembler can - in fact the SAS assembler 'asm' (5.10b) does not handle the above situation either.

```
        XREF        stack,stack_size
        LDS         #stack-stack_size-1
        LDS         #stack+stack_size*4
        LDAA        #-b
```

I hope to eventually eliminate the un-needed limitation and perhaps some day in the far far future I can eliminate the object format based limitation.

There is one more very small limitation. The assembler will report an error for the following code.

```
        XREF        s
        LDAA        #s-129
```

It does not know what s will evaluate to and assumes 0 which gives a negative value greater than can be handled in an 8 bit value. If this proves to be a severe hardship to anyone, let me know.

MACROS

You cannot have a macro definition within a macro definition, but can nest macro calls.

Arguments passed to macros are referenced with: {#}. The first argument passed to a macro would thus be {1}. You should always use LOCAL labels (.name) inside macros which you use more than once. {0} represents an EXACT substitution of the ENTIRE argument line.

GENERAL

? The other major feature in this assembler is the SUBROUTINE pseudo-op, which logically separates local labels (starting with a dot). This allows you to reuse label names (for example, .1 .fail) rather than think up crazy combinations of the current subroutine to keep it all unique.

PSEUDOPS

INCLUDE "name"

Include another assembly file.

[label] SEG[.U] name
[label] RSEG[.U] name

This sets the current segment, creating it if neccessary. If a .U extension is specified on segment creation, the segment is an UNINITIALIZED or BSS segment. The .U is not needed when going back to an already created uninitialized segment, though it makes the code more readable. The .z force may be used to declare that any symbols declared in this segment may be accessed by direct (page zero) addressing where appropriate. If you need a .U and .z segment, first declare it as .U and then immediately redeclare it as .z.

[label] DC[.BWL] exp,exp,exp ...
[label] FDB exp,exp,exp ...
[label] FCB exp,exp,exp ...
[label] FCC exp,exp,exp ...

Declare data in the current segment. The default size extension for DC is a byte. FCB allows only byte size and FDB allows only word size.

[label] DS[.BWL] exp[,filler]
[label] RMB exp[,filler]

Declare space (default filler is 0). Note that the number of bytes generated is exp * entrysize (1,2, or 4). The default size extension for DS is a byte. RMB allows only the size of byte. Note that the default filler is always 0.

[label] DV[.BWL] eqmlabel exp,exp,exp....

This is equivalent to DC, but each exp in the list is passed through the symbolic expression specified by the EQM label. The expression is held in a special symbol dotdot '..' on each call to the EQM label.

See EQM below

[label] HEX hh hh hh..

This sets down raw HEX data. Spaces are optional between bytes. NO EXPRESSIONS are allowed. Note that you do NOT place a $ in front of the digits. This is a short form for creating tables compactly. Data is always layed down on a byte-by-byte basis.

Example: HEX 1A45 45 13254F 3E12

ERR

Abort assembly.

[label] XDEF symbol,symbol,symbol
[label] PUBLIC symbol,symbol,symbol

Defines which symbols/labels are available outside of this module.

[label] XREF symbol,symbol,symbol
[label] EXTERN symbol,symbol,symbol

Declares which symbols/labels from outside modules are used in this module. The .z force may be used to declare that the symbols may be accessed by direct (page zero) addressing where appropriate.

PROCESSOR model

Do not quote. Model is one of: 6803,HD6303,68705,68HC11, 68HC16. Can only be executed once, and should be the first thing encountered by the assembler.

ECHO exp,exp,exp

The expressions (which may also be strings), are echod on the screen and into the list file

[label] ALIGN   N[,fill]

Align the current PC to an N byte boundry. The default fill character is always 0.

[label] SUBROUTINE name

This isn't really a subroutine, but a boundry between sets of temporary labels (which begin with a dot). Temporary label names are unique within segments of code bounded by SUBROUTINE:

```
CHARLIE subroutine
        ldx     #10
.1      dex
        bne     .1
BEN     subroutine
        ldx     #20
.qq     dex
        bne .qq
```

Automatic temporary label boundries occur for each macro level. Usually temporary labels are used in macros and within actual subroutines (so you don't have to think up a thousand different names)

symbolEQU   exp

The expression is evaluated and the result assigned to the symbol.

symbolEQM   exp

The STRING representing the expression is assigned to the symbol. Occurances of the symbol in later expressions causes the string to be evaluated for each occurance. Also used in conjuction with the DV psuedo-op.

symbolSET exp

Same as EQU, but the symbol may be reassigned later.

END [symbol]

Optional. If used with a symbol name the value of that symbol will be entered into the output file as the starting address of the program. May only be used once and only at the end of the file. Be careful - if a symbol is not given and a comment without a preceding ';' is used the first word of the comment will be interpreted as the symbol.

MAC name
MACRO name

Declare a macro. lines between MAC and ENDM are the macro. You cannot recursively declare a macro. You CAN recursively use a macro (reference a macro in a macro). No label is allowed to the left of MAC or ENDM.

ENDM

End of macro def. NO LABEL ALLOWED ON THE LEFT!

MEXIT

Used in conjuction with conditionals. Exits the current macro level.

[label] IFCONST exp
[label] IFD exp

Is TRUE if the expression result is defined, FALSE otherwise and NO error is generated if the expression is undefined.

[label] IFNCONST exp
[label] IFND exp

Is TRUE if the expression result is undefined, FALSE otherwise and NO error is generated if the expression is undefined.

[label] IF exp

Is TRUE if the expression result is defined AND non-zero. Is FALSE if the expression result is defined AND zero. Neither IF or ELSE will be executed if the expression result is undefined. If the expression is undefined, another assembly pass maybe required.

[label] ELSE

ELSE the current IF.

[label] ENDIF
[label] ENDC
[label] EIF

Terminate an IF. ENDIF and EIF are equivalent.

[label] REPEAT exp
[label] REPEND

Repeat code between REPEAT/REPEND 'exp' times. if exp == 0, the code repeats forever. exp is evaluated once.

```
Y       SET     0
        REPEAT  10
X       SET     0
        REPEAT  10
        DC      X,Y
X       SET     X + 1
        REPEND
Y       SET     Y + 1
        REPEND
```

generates an output table: 0,0 1,0 2,0 ... 9,0 0,1 1,1 2,1... 9,1, etc...

Labels within a REPEAT/REPEND should be temporary labels with a SUBROUTINE pseudoop to keep them unique.

The Label to the left of REPEND is assigned AFTER the loop FINISHES.

FORCED ADDRESSING MODES

[label] XXX[.force] operand

XXX is some mnemonic, not necessarily three characters long. The .FORCE optional extension is used to force specific addressing modes. Force extensions are also used with DS,DC, and DV to determine the element size.

example: lda.z charlie

Force	Description	Alternate Force Extension	
i	implied		
ind	indirect word		
0	implied		
b	byte address	z d	(zeropage, direct)
bx	byte address indexed x		

by	byte address indexed y		
w	word address	e a	(extended, absolute)
l	longword (4 bytes) (DS/DC/DV)		
r	relative		
u	uninitialized (SEG)		

EXPRESSIONS

Some operators, such as ||, can return a resolved value even if one of the expressions is not resolved. Operators are as follows:

NOTE WELL! Some operations will result in non-byte values when a byte value was wanted.
Example: ~1 is NOT $FF, but $FFFFFFFF.
Preceding it with a < (take LSB of) will solve the problem.
ALL ARITHMETIC IS CARRIED OUT IN 32 BITS. The final result will be automatically truncated to the maximum handleable by the particular machine language (usually a word) when applied to standard mnemonics.

PRECEDENCE

UNARY

20	~exp	one’s complement.
20	-exp	negation
20	!exp	not expression (returns 0 if exp non-zero, 1 if exp zero)
20	<exp	take LSB byte of a 16 bit expression
20	>exp	take MSB byte of an expression

BINARY

19	*	multiplication
19	/	division
19	%	mod
18	+	addition
18	-	subtraction
17	>>,<<	shift right, shift left
16	>,>=	greater, greater equal
16	<,<=	smaller, smaller equal
15	==	equal to. Try to use this instead of =
15	=	exactly the same as == (exists compatibility)
15	!=	not equal to
14	&	logical and
13	^	logical xor
12	|	logical or
11	&&	left expression is true AND right expression is true

10	||	left expression is true OR right expression is true
9	?	if left expression is true, result is right expression, else result is 0. [10 ? 20] returns 20
8	[]	group expressions
7	,	separate expressions in list (also used in addressing mode resolution, BE CAREFUL!

CONSTANTS

nnn decimal
0nnn octal
%nnn binary
$nnn hex
'c character
'c' character
"cc.." string (NOT zero terminated if in DC/DS/DV)
[exp]d the constant expressions is evaluated and it's decimal result turned into an ascii string.

SYMBOLS

.. holds evaluated value in DV pseudo op
.name represents a temporary symbol name. Temporary symbols may be reused inside MACROS and between SUBROUTINES, but may not be referenced across macros or across SUBROUTINEs.
. current program counter (as of the beginning of the instruction).
name beginning with an alpha character and containing letters, numbers, or '_'. Represents some global symbol name.

WHY codes:

Each bit in the WHY word (verbose option 1) is a reason (why the assembler needs to do another pass), as follows:

Bit	Meaning
0	expression in mnemonic not resolved
1	
2	expression in a DC not resolved
3	expression in a DV not resolved (probably in DV's EQM symbol)
4	expression in a DV not resolved (could be in DV's EQM symbol)
5	expression in a DS not resolved
6	expression in an ALIGN not resolved
7	
8	???ALIGN: Normal origin not known (if in ORG at the time)

9 EQU: expression not resolved
10 EQU: value mismatch from previous pass (phase error)
11 IF: expression not resolved
12 REPEAT: expression not resolved
13 a program label has been defined after it has been referenced (forward reference) and thus we need another pass
14 a program label's value is different from that of the previous pass (phase error)

Certain errors will cause the assembly to abort immediately, others will wait until the current pass is over. The remaining allow another pass to occur in the hopes the error will fix itself.

HIGH LEVEL CONSTRUCTS

The assembler allows for some high level C-like constructs that help to make your assembler source file self documenting and clearer. The expressions allowed in these constructs are quite different, check the documentation below.

[label] _IF exp	If the expression (condition) evaluates to true then the statements following this statement and including up to the associated _ELSE or _ENDIF statement are executed.
_ELSE	marks then end of the true condition portion of the _IF block and the start of the false condition portion
_ENDIF	marks the end of an _IF block
[label] _FOR [exp]\exp\exp	The optional first expression (assignment) is executed and then the second expression (condition) is evaluated. As long as it evaluates to true, the statements following this statement and including up to the associated _NEXT statement will be executed. Each time the _NEXT statement is reached the third expression (assignment) is executed and then the second expression is re-evaluated.
_NEXT	marks the end of a _FOR block

If an _IF, _ELSE, _FOR or _NEXT statement causes an branch address range error you can instruct that extended addressing be used with the .w force.

There are currently two types of expression: condition and assignment. Each consists of three parts and the parts must appear in the defined order.

<condition> := [!]<register><comparison><argument> | [!]<CCRbit>
<assignment> := <register><equate><argument>
<register> := <A|B|D|X|Y>
<comparison> := <|<=|==|!=|>|>=|.HI.|.LO.|.HS.|.LS.
<argument> := <number>|<label>|[<number>](X)|[<number>](Y)|(number)|(label)
<CCRbit> := CCRc|CCRm|CCRv
<equate> := =|+=|-=
<number> := $<hex number>|<decimal number>

Although not currently allowed, expect that multiple comma separated assignment expressions will be allowed and that a reversed order assignment will be allowed.

Some examples of the use of these expressions are given below.

a) if register a is greater than 3 then add 6 to a and shift left otherwise just shift left

```
_IF     A>3
        adda    #6
        lsla
_ELSE
        lsla
_ENDIF
```

b) convert a mixed case null terminated string to lower case

```
_FOR  A=(X)\A!=0\A=(X)
        _IF     A>='A'
                _IF     A<='Z'
                        adda #'a'-'A'
                _ENDIF
        _ENDIF
        inx
_NEXT
```

High level expressions can be nested up to 8 levels deep. High level expressions are only available for the 68HC11 currently, if you want them to be available for another processor, let me know. It is possible to generate some syntactically legal assignment statements that produce illegal opcodes. An example of these would be X+=4. I have not implemented these yet due to the absence of convenient opcodes (ADDX, SUBX) in the HC11 and the complexities in faking them, so don't use them.

NAME

SC11

SYNOPSIS

SC11 <options> <file name>

Options		
	-a	Annotate the assembler listing with source.
	-b	Generate inline assembler for builtins.
	-D symbol	Define a preprocessor symbol.
	-I	Directory for include files.
	-l	Create a source listing file.
	-n	Turn off optimization.
	-o filename	Specify name for output.
	-p [0|filename]	Create or use precompiled header.
	-P	Generate for use in PROM.
	-q	Run quietly.
	-r	Use library for integer math.
	-U symbol	Undefine a preprocessor symbol.
	-w	Convert all auto variables to static

DESCRIPTION

SC11 is a C compiler that produces 68HC11 source output that is ready for assembly. The compiler is reasonably compliant with the ANSI-C standard. It supports all C constructs except bit definitions. Some HC11 extensions are provided.

This product has had a good amount of testing, but due to its complex nature, there could still be many bugs and, as this is a public domain product, I cannot advise its use for applications that may be critical to someones life, income or career. If you feel that you must do so anyway, I recommend that you contact me before you start, to ensure that I am still supporting this product.

BUILT-IN FUNCTIONS

The following functions are available as built-ins. As such, these functions are quite a bit faster than the corresponding library functions.

strlen	strcmp	strcat
strcpy	bzero	bcopy

As well the following HC11 specific functions are provided only as built-in functions

void enable_intr(void)	generates an HC11 'CLI' instruction
void disable_intr(void)	generates an HC11 'SEI' instruction
void wait_for_intr(void)	generates an HC11 'WAI' instruction

MEMORY FUNCTIONS

The functions malloc and free are available. If you use them, you will get a 2K byte space of memory to allocate from. If you do not use them, no memory is mapped for allocation.
If you want to change the size of this area you must create, assemble and link with a file that looks a lot like this.

```
        PROCESSOR 68HC11

        XDEF  __first_mem_location,__last_mem_location

        RSEG.U        BSS
__first_mem_location
        rmb     2048
__last_mem_location
```

INTERRUPTS

If an interrupt routine is to call a C routine it must preserve all registers and the D extension and mirror temporaries and of course restore them on exit. If it will be doing floating point math, it must also preserve the 10 bytes of math registers. This is a fairly high overhead to pay. For those that insist on having interrupts call C routines I plan to add an interrupt keyword for functions that forces them to preserve these registers.

Currently, many of the library routines in C_lib.lib and math.lib are not re-entrant. If another task, process or interrupt calls a non re-entrant function that is used by the main program or another interrupt, if nested interrupts are possible, bad and unpredictable things will happen. Better yet there is currently no list of which functions are and are not re-entrant. BE SAFE - DON'T DO IT.

Instead of calling C functions with interrupts, it is reccommended that the interrupt handler service the interrupt then set a flag that the main program checks and on finding it set executes the calls.

PRINTF, SPRINTF, FPRINTF, SCANF, SSCANF, FSCANF

There are 4 versions of each function in the printf family of functions and 4 versions of each function in the scanf family of functions. The default versions are ANSI compatible. The printf family uses the function __print which is very large and it would be a shame to use an 8 Kbyte function to

```
        printf("Hello World");
```

Similarly the scanf family uses the function __print which is also very large and it would be a shame to use an 8 Kbyte function to

```
scanf("%s",string);
```

For something as simple as the examples above you should consider using puts or fputs or gets or fgets, they are smaller and faster. If you need printf or scanf, the linker can be told to substitute one of the other formatting functions, __sm_print, __med_print or __lg_print for __print and __sm_scan, __med_scan or __lg_scan for __scan. The differences are;

__print	Everything is included
__lg_print	Floats excluded
__med_print	Floats and longs excluded
__sm_print	Floats and longs excluded and width, precision, 'l' and 'h' are ignored
__scan	Everything is included
__lg_scan	Floats excluded
__med_scan	Floats and longs excluded
__sm_scan	Floats, longs and octal excluded and 'l' and 'h' are ignored,

Currently none of the long or float handling is in the library. In their place are stubs that do nothing.

ERROR REPORTING

The error detecting/reporting of this compiler is quite poor. I have never had it crash but it will hang. If you have it hang I reccomend that you run another compiler, such as SAS/C, over the code to find the problem.

STACK USAGE

As on most compilers there is a need for more stack for a larger function. With this compiler it doesn't take much of a function to exceed the default stack size of 4000 bytes. Stack checking saves your system from a crash - so it is only a small hassle to increase the stack and try again. I plan to give the program more stack by default.

ZERO PAGE USAGE

It is very important from the execution time perpective that you have the 6 bytes of pseudo registers in page zero. This does not seem like an unreasonable demand so the system defaults to that way. If you cannot use any zero page ram, you will have to modify the C_lib.inc file to remove the .z from the zpage segment and then reassemble all C compiler generated sources including the library sources and then link with zpage at its correct location.

Similarly, if you choose to use floating point routines, another 10 bytes of page zero will be required.

HC11 REGISTER USAGE

A C routine may use any of the HC11 registers. In addition to this it may use 6 bytes of pseudo registers which are used to speed calculations. Two of these bytes act as an extension to the D regster for use with long variables. The other 4 bytes act as mirror temporaries of the extended D register.

L99999999	L99999990	D register	
L99999995	L99999996	L99999997	L99999998

If a C routine is to call an assembly routine, the assembly routine must preserve only the Y and S registers. The subroutine will be called with all parameters pushed on the stack, as is traditional with the C language. The parameters are pushed onto the stack in the reverse order that they are listed in the function prototype. The Y register points to the data frame for the calling function which should not be changed. The result of the routine, if there is one, will be in the B register if it is a char size result, in the D register if it is a word size result or in the D register and the D extension word.

In order for an assembly routine to call a C routine, the assembly routine must setup like a C routine. Parameters must be pushed to the stack and the stack must have enough space for the auto (local) variables of the C routine(s).

C source symbols have an underscore prepended to them when they are translated to assembler. Therefore, if the C program calls an assembler function "find_mouse", the assembler functions name in the assembler source would be "_find_mouse".

FUNCTION RETURN VALUES

For the results of functions returning either char or long sized to be used properly in following expressions it is ESSENTIAL that the function prototype be known before the function is used. If it is not known the compiler will assume int meaning that a byte of garbage will be appended to the char as the MS byte, or that the upper word of the long will be lost.

FUNCTION PARAMETERS

In using var args type functions such as fprintf, fscanf and their derivatives and unprototyped functions, it is important to understand how arguments are pushed to the stack for these calls. All expressions (a single variable is consider a simple expression) are pushed to the stack in the size that the expression evaluates to:

a char variable is pushed as 8 bits

an int,short or pointer variable is pushed as 16 bits
a long variable is pushed as 32 bits
an expression is pushed as the longest type used in the expression
an expression that mixes signed and unsigned types of the same size is promoted to the next larger size unless the size is long
a numeric constant is pushed as the smallest signed size that it can fit into. Ex. 127 => char, 128 => int

So if an unprototyped or var args function is expecting an int and you pass it the number 4, it will not work the way you expect. You will have to cast it to the required size or if the function is fully prototyped (not var args) the cast will be done automatically for you.

For integer specifiers %d, %x, etc., fprintf and fscanf have a default size of int. If you wish to use a long you must you the l specifier (%ld, %lx) and either pass a long variable, long expression or long cast. The %c specifier has a default size of char. If you pass it an int it will not work as you expect, in this case you must cast to a char.

This concern is not present in 68000 C compilers because they always promote the type to long since that is the natural size of the processor. But to produce efficient code for the HC11 we must do it this way. Intel processors (at least with 16 bit compilers) suffer some of these problems.

INPUT/OUTPUT

You are expected to provide 4 routines if you plan to do character I/O. putchar, getchar, puts, gets; these are documented in any C book.

BIT FIELDS

This compiler does not support bit fields

CUSTOMIZING C.O

You are strongly advised to link with c.o for all your C programs. However, c.o is not provided with this release (or any other). Only the source file (c.a) is provided since it MUST be customized to suit your environment. It is a very small program, currently the only thing that must be setup is the stack pointer. Once this is done use SAsm to assemble c.a and produce c.o.

THE PROM (-P) OPTION

This option causes all non-constant initialized data to be generated in the data segment IDATA. This name has special meaning to the linker, causing it to generate a mirror BSS allocation for it. The linker also generates 3 symbols, __IDATAfrom, __IDATAend and IDATAto. If there is no IDATA segment, these symbols are generated with unknown but identical values. Using these symbols a simple routine in C_lib called __init_data copies the initialization data to the BSS segment. This allows for ram variables in a BSS segment to have initial values coming from the PROM. When linking, the IDATA segment should be located in RAM memory, the initialization values are automatically tacked onto the end of the CODE segment or a CODE segment is created. If you link with the module c.o all of this will be taken care of for you.

BIT OPERATION OPTIMIZATIONS

Certain operations can be optimized into bit operations which are much faster than other operation. To use them you must design your program to take advantage of them. Any variables that you will be doing bit operations on must be char or unsigned char type and must be accessible via a dereference, which includes function parameters, auto variables in functions and pointer referenced variables.

For the first type of bit optimization you must use the &= or |= operations and you must use a constant as the bit pattern. This will make use of the 6811 instructions BSET and BCLR. For example;

```
char r;
void main(unsigned char *s) {
        char t;
        char *u_ptr;

        r |= 4;              <== NOT Optimized, not pointer accessible
        t = t | 4;           <== NOT Optimized, not |= operator
        s &= 5;              <== Optimized
        s &= ~6;             <== Optimized
        *u_ptr |= 0x30;      <== Optimized
}
```

The second type of optimization is employed when a dereferrenced variable is anded with a constant as part of a conditional expression. This uses the 6811 instructions BRSET and BRCLR For example;

```
unsigned char *ptr;
void main( void ) {
        if (*ptr & 0x80) {      /* if MSB set then ...

        }
}
```

The above constraints are a function of the limitations of the bit operations of the HC11.

CODING FOR MAXIMIZING EFFICIENCY

The auto increment/decrement reference is much faster as

```
*++aptr
```

than as

```
*aptr++
```

particularly on int and long sized data.

STANDARD C FUNCTIONS (C_lib.lib)

The following functions are available through the C_lib.lib library. They are currently not prototyped here since such prototypes are available from many other sources and because to do so would take time from more worthwhile enterprises.

strlen	sprintf
fprintf	printf
isalnum	isalpha
iscntrl	isdigit
isgraph	islower
isprint	ispunct
isspace	isupper
isxdigit	atoi
atol	ungetc
fputc	putc
fgetc	getc
getchar	putchar
feof	ferror
puts	fputs
malloc	free
memcpy	memset
sscanf	fscanf
scanf	

The following functions are useful functions that I have seen in other compilers but are not standardized ... and currently not documented, sorry.

stci_d	stcu_d
stci_h	stcl_d
stcul_d	stcl_h
stcc_d	stcuc_d
stcc_h	stci_o
stcl_o	stcc_o

NAME

SDis

SYNOPSIS

SDis <options> <file name>

Options	-a<num>	additional address specification (n in hexadecimal)
	-f<name>	filename of additional address specifications
	-l8	toggle labels for 8 bit immediates (default off)
	-l16	toggle labels for 16 bit immediates (default on)
	-x	generate XREF's rather than EQU's for labels external to S-record file

DESCRIPTION

SDis is the dis-assembler for the DEV11 system. It accepts a Motorola S-record format file and outputs a symbolic dis-assembled file to stdout, by default the console. The output file is assembler (SAsm) ready and would assemble to be exactly the same as the file that was dis-assembled.

SDis is what I call a tracking dis-assembler; which means that it must have the address of an executable instruction to start from and it follows the executable code thru statement by statement building up a map of where executable code is found. Normally it gets the starting address from the S9 record, if your file has one. Otherwise or if other addresses must be known, as might be the case with a monitor ROM with many entry points, the dis-assembler asks for an address(es) to start from or address(es) can be provided on the command line or from a file. A tracking dis-assembler will not attempt to dis-assemble data; it knows the difference (but it can be intentionally fooled).

If a file is used to supply additional code entry addresses, the format of the file is one Hex address per line in the file. Blank lines are ignored

Unfortunately, some compilers (including SC11) produce code from a switch statement that fools SDis. The construct is;

```
jsr     switch_handler
fcb     case1_val
fdb     case1_vector
fcb     case2_val
fdb     case2_vector
```

The handler never returns from the jsr, instead it fiddles the stack and jumps to one of the vectors. If you see non-sense instructions or '???' not in a comment this is what you have run into.

The above example also shows how the disassembler can miss some code areas, thinking they are data instead. Since it cannot know the format of the data following the JSR or any vector lookup table, it cannot know that there is code at the vector addresses. This is where a human brain is required. After the first disassembly analyse the data areas to see if they are vector lookup tables; if so get the addresses and enter them in by hand during the next disassembly.

If SDis finds a place in the file that it believes is data but does not have a label, i.e. that address is not accessed by any of the code that it knows about, it will print out "??? no label." Generally, this means that the data is not accessed by the program. If you inspect the the hex dump of this data you may discover that it is in fact executable code. Again, this would be code that is not used in the program. Unused code or data is a sign that a linker and library routines have been used to build the program as this process often results in the inclusion of extraneous routines.

No disassembler can know the difference between a constant and an address in an immediate statement. For example;

```
        label       rmb     1
                    ldaa    #label
                    ldaa    #$30
```

If 'label' evaluates to an address of $30 then no difference exists between the two ldaa statements. But the difference is critcal if the dis-assembled code is to be relocated. In hopes of helping, there are two options that determine how these cases are disassembled - as constants or labels and for 8 bit immediates if the value is in the range of ASCII a comment containing the character is appended. BUT, you will still have to sort out which are really constants if you wish to relocate!

NAME

SSim

SYNOPSIS

SSim [NTSC|PAL] [LORES] [NOLACE]

Note: the screen defaults to DBL_NTSC HIRES LACE

NOTES

Special thanks go to Motorola Inc. for supplying the reference materials that were essential to this product and consequently for the education as to the depth and breadth of the ’HC11 family.

The large number of peripherals and variations of peripherals in this family has made difficult to model everything in the time that I can devote to a public domain product. Unmodelled members of the family are disabled in the menus. Additionally, the following peripherals and functions are not modelled on any processor.

EEPROM programming SPI

If these are particularly important to you, let me know, I’ll see what I can do.

DESCRIPTION

SSim is a highly capable simulator for the 68HC11 processor. It is different from Sim68 (for IBM PC’s) in many ways. First, it is more GUI based, second, it simulates the memory and the operation of regisiters specific to the selected processor and third, it allows for simulating external peripherals via an ARexx interface. The ARexx simulation could be done by CanDo or AmigaVision to provide a graphical display of, say, an operators panel.

DISPLAYS

The display windows show various representations of information that are important while simulating. The displays change during the simulation as the data that they display changes. Most displays can be set to display in hex, binary or decimal format.

The Register window shows the contents of the internal registers. Initially the registers contain ’XXXX’ indicating that the contents are not known. It is possible for the simulation to work but the program to not work if your program relies on the simulators assumption that the registers default to 0. For example: if the first instruction to execute after reset is ’ORAA #4’ the register display will show the contents as ’04’ when only the state of one bit can be known for sure.

The Stack window shows the bytes in the stack above the current stack pointer. This is particularly useful for resolving problems with values passed to functions via the stack (C style). Higher address memory is shown above the stack pointer.

The Memory window can display and eventually modify the contents of 64 bytes of memory.

The Source window shows the instructions that have been and will be executed. Note that this is somewhat different from what you would expect from a source level debugger, which would show the source lines before and after the current line. This display choice was made due to the nature of assembler programs, i.e. it takes 20 lines of assembler to do anything. It is more import to know where you have been than where you could have been.

The Source window contains a '>' that marks the next instruction to be executed. Above this is a history of statements that have been executed so far. Below it is a disassembly of the code following the next instruction.

The Control window has 5 buttons that control the simulation;

a) Run - This button tells the simulation to continue executing statements until a breakpoint is hit or an error occurs.

b) Stop - This button, which is only useful while Running, causes the simulation to stop.

c) Step - This button tells the simulation to step to the instruction following the next instruction. If the next instruction is a JSR the simulation will continue executing until the following instruction is reached, thereby stepping over the next instruction.

d) Step Into - This button tells the simulation to step to the instruction that will execute after the next instruction. If the next instruction is a JSR the simulation will 'step into' the subroutine.

e) Reset - This button performs the same function as applying a low level to the processors ~Reset pin. The reset vector is fetched from memory and the processors registers are set accordingly.

The Serial window has a display for the text that is output by the simulated serial port output (Tx) and two string entry gadgets for entering characters to go to the serial port input (Rx & RxHex). All printable characters and most control characters can be entered into the Rx gadget. CR and LF can be entered by pressing ctrl-M and ctrl-J respectively. The RxHex input allows for hexadecimal entry of characters to the serial port. Entry to this gadget MUST always be multiples of two characters. The RxHex gadget is read from after the Rx gadget is empty.

The Ports window shows the output values of port A thru H (if applicable) for the simulated processor.

The Break Points window allows for the entry of addresses at which the simulation will halt.

The Analog window allows for the entry of floating point voltage values for the analog inputs. The values entered must be between 0.000 and 5.000 inclusive.

OPERATION

When the program is first started no processor is selected and therefore there is no memory or registers. It would be pointless to try to load a program, view memory or set breakpoints at this time.

The first operation is to select a processor or load a previously saved simulation, which contains the processor choice. Currently the most common processors are available but eventually all will be available. Once a processor has been selected the simulation will have the appropriate memory for that processor. This memory map can be viewed and eventually modified. Processor operations on registers that would remap these memories in the real processor will cause them to be relocated in the simulation.

If a processor was selected, it may be necessary to inform the simulation about external memory and peripherals; this is done with the window raised by the Configure/Memory Map menu item. External memory may be added to the internal maps for the processor. The type for this kind of map is User since external (to SSim) simulation is not required. For external peripherals, those that attach to the bus or ports, the map sets the address for the peripheral and the type is set to ARexx. This will cause the simulation to try to exchange data with an external ARexx program when that address is read or written (see the section on ARexx for details of the exchange).

An S-record file can be loaded into the simulation. If the loading process attempts to write to non-existant memory locations you will get an error message. More than one load can be done to merge various pieces of code.

Now that a processor has been chosen, the map has been set and the code has been loaded, which is expected to contain vectors, the processor can be reset by pressing the reset button on the middle panel. This should load the reset vector and display the disassembled code in the source window. Much later on I hope to have it display the source from the original file which would include comments and for C programmers the original C statements.

The control panel in conjunction with the other windows can now be used to debug the program.

SIMULATION OPTIONS

There are currently 2 options, both respecting the slow speed of the simulation (and I have an A4000).

a) Run Fast - stops the continuous display updates of the Registers, Stack and Memory windows while the simulation is running as opposed to stepping. With Run Fast active during a long run there is no indication that anything is happening.

b) Baud x 16 - this departs from strict simulation by allowing the serial output to run 16 times faster than usual. If you don’t require strict simulation this option will keep you from getting old waiting for character output.

AREXX

Whenever an instruction results in a change to any of the processors external ports or any read or write to any area of the memory map that has been mapped as AREXX, an ARexx interaction is required. If a NOTIFY message has been received at the ARexx port, an ARexx reply is formed and the NOTIFY message is returned to the sender port. Also, at the end of every instruction, if a INSTR message has been received by the program, the result value of the message is set to the number of E clock cycles that passed during that instruction and the message is replied. If an AREXX program wishes to receive all messages it must keep a NOTIFY message continually on the AREXX port. If it can send more than one such message that is best, otherwise it will have to send another NOTIFY message as soon as the current one is returned. To give this a better chance of success, SSim runs at a priority of -1.

The ARexx messages are:

Notify	SSim will never send this
Instr	
Port A/B/C/D/E/F/G/H <data>	
Mem <WO> <W/B> <addr> <data>	or
Mem <RO> <W/B> <addr>	
Serial <in/out> <"string">	
Analog <in/out> <#> <value>	value is 0 to 4095 which spans range of VRL and VRH
QUIT	arexx program never sends this

data values can be in decimal or hex if preceded by a ’$’

If a Mem command is given with a read mode, the simulation will stop until there is a response from the arexx program. The response must be another Mem command with the same address.

I suspect that AmigaVision 1.70z or better is required to properly access this data by using the ?<forgot>? command which only appears in the readme.

CAVAETS

SSim requires AmigaDOS 2.04 or better, in fact, 2.04 may leave open some possibilities for crashes that are not possible on 2.1 and up.

It is not possible to successfully save a simulation until both a processor has been chosen and reset.

NAME

slib

SYNOPSIS

slib [-a<1>] [-d<1>] [-l[<1>]] [-r<1>] [-w<1>] [-x<1>] <library name>

<1> - one file

DESCRIPTION

Slib is the librarian for the ADev11 system. This version, done by Paul Issacs, replaces the old slib, now called xlib. Xlib will be phased out but is included now since the two programs are significantly different. The following options are recognized by slib:

-a Specifies an object file that will be added to the library. The object file must not already be in the library. The -a option may be used more than once.

-d Specifies an object module that will be deleted from the library.

-l Causes a listing of the library to be sent to a file or to stdout if the filename is omitted. The listing includes the name of the module, its size (not the code size), and the symbols that are available from that module. The -l option can be used more than once.

-r Specifies an object module to be replaced in the library. If the module is not in the library, the specified module is added to the library. The -r option may be used more than once.

-w Specifies a file that contains option commands to be processed by slib. The file may be broken into lines, but each line must start with an option. A command file can specify another command file and more than one command file can be used. An example file might look as shown below.

```
-adownload.o
-rcowtown.o
-l
```

-x Specifies an object module in the library that is to be copied to the modules original filename.

One and only one library must be specified for each invocation of slib.

NAME

xlib

SYNOPSIS

xlib [-a<1>] [-r<1>] [-l] [-c<1>] <library name>

<1> - one file

DESCRIPTION

Xlib is the old librarian for the ADev11 system. It is being phased out and new developments should not use it. The following options are recognized by slib:

-a Specifies an object file that will be added to the library. The -a option may be used more than once.

-c Specifies a file that contains option commands to be processed by slib. The file may be broken into lines, but each line must start with an option. A command file can specify another command file and more than one command file can be used. An example file might look as shown below.

```
-adownload.o
-rcowtown.o
-l
```

-l Causes a listing of the library to be sent to the console (unless redirected). The listing includes the name of the module, its size (not the code size), and the symbols that are available from that module. The -l option can be used more than once.

-r Specifies an object module to be removed from the library. The -r option may be used more than once.

One and only one library must be specified for each invocation of slib.

NAME

slink
slink1.3

SYNOPSIS

slink [FROM <1+>] [TO <1>] [WITH <1>] [LIB <1+>] [MEM <range> <1+>]
[MAP <1>]

<1> - one file
<1+> - one or more files (separated by commas)
<range> = <hex address>-<hex address>

DESCRIPTION

Slink is the linker for the DEV11 system. Slink accepts keyword commands to control the process of linking. Keywords are not case sensitive. The following keywords are recognized by slink:

FROM Provides a list of object files that will become the root of the output file. FROM must be used once and may be used more than once with each use adding to the root, but you must specify at least one file for each use.

LIB Provides a list of library files to be scanned to resolve symbols not found in the root files. Only the modules containing the unresolved symbols will be included in the output file. LIB can be used more than once.

MEM Specifies an area of memory and provides a list of the segment names that are to be located in that area. The linker must know where to place every segment that is used. MEM is usually used more than once.

TO Specifies the output file to create. The file will be an S-record format file. TO must be used once and only once.

WITH Specifies a file that contains keyword commands to be processed by slink. The file may be broken into lines, but each line must start with a keyword. A WITH file can WITH another file and more than one WITH file may be used. An example file might look as shown below.

```
FROM download.o
TO download.sr
MEM 100-7fff EXT_RAM
```

MEM 8000-83ff INT_RAM
MEM 0-ff ZPAGE
MEM fe00-ffd5 EEPROM, CODE
MEM ffd6-ffff VECTORS

MAP Causes a link map listing to be generated. This file contains information such as the addresses of all global symbols and the addresses of the hunks from each module.

There is a reserved segment name called CHKSUM that works like any other segment except that it is generated by the linker. To use the CHKSUM feature, you link the CHKSUM segment in with the rest of the segments in the EPROM (doesn't matter where). This produces a word size value in the EPROM such that the sum of all the words of the rom is zero. This segment name can only appear once in a link. The following assumptions have been made about the use of CHKSUM: that some code exists within 255 bytes of the start of the rom, the rom extends to 0xFFFF and unused rom is erased to 0xFF. The linker determines the start of the rom by anding the lowest address that it sees with 0xFF00. See the example chksum in the examples directory. If you have segments that generate S-records that are not in the EPROM, which you shouldn't, these will screw-up the computation.

The segment name IDATA has special meaning to the linker, causing it to generate a mirror BSS allocation for it. The linker also generates 3 symbols, __IDATAfrom, __IDATAend and __IDATAto. If there is no IDATA segment, these symbols are generated with unknown but identical values. Using these symbols a simple routine in C_lib called __init_data copies the initialization data to the BSS segment. This allows for ram variables in a BSS segment to have initial values coming from the PROM. When linking, the IDATA segment should be located in RAM memory, the initialization values are automatically tacked onto the end of the CODE segment or a CODE segment is created.

EXAMPLE

The following line invokes the linker using the command file download.sln and creating the linker map file download.map.

Slink MAP download.map WITH download.sln

ADEV11 Development System for AmigaDOS
Library Reference

Version 1.0

Function List

FUNCTION	**DESCRIPTION**	**LIBRARY**
__asc2byte	**convert ASCII string to byte**	**lib11.lib**
__find_spc	**find first space ' ' character in string**	**lib11.lib**
__put_asc	**print out word size number (signed)**	**lib11.lib**
__put_asc_sm	**print out byte size number (signed)**	**lib11.lib**
__put_asc_str	**make string from word size number (signed)**	**lib11.lib**
__put_asc_str_sm	**make string from byte size number (signed)**	**lib11.lib**
__put_asc_str_u	**make string from word size number (unsigned)**	**lib11.lib**
__put_asc_str_usm	**make string from byte size number (unsigned)**	**lib11.lib**
__put_asc_u	**print out word size number (unsigned)**	**lib11.lib**
__put_asc_usm	**print out byte size number (unsigned)**	**lib11.lib**
__put_hex	**print out word size hex number (signed)**	**lib11.lib**
__put_hex_sm	**print out byte size hex number (signed)**	**lib11.lib**
__put_hex_str	**make string from word size hex number (signed)**	**lib11.lib**
__put_hex_str_sm	**make string from byte size hex number (signed)**	**lib11.lib**
__put_oct	**print out word size oct number (signed)**	**lib11.lib**
__put_oct_sm	**print out byte size oct number (signed)**	**lib11.lib**
__put_oct_str	**make string from word size oct number (signed)**	**lib11.lib**
__put_oct_str_sm	**make string from byte size oct number (signed)**	**lib11.lib**
__search	**search for string in string list**	**lib11.lib**
__skip_spc	**find first non-space ' ' character in string**	**lib11.lib**
__str_lookup	**find string in string list from ordinate**	**lib11.lib**

*The names used above are the ones that would be used to access the functions from an assembler program. From a C program one less leading underscore would be used, although none of these functions would have any value to a C program due to the different ways of handling argument passing.

NAME

__asc2byte

SYNOPSIS

IN:	X	pointer to string
OUT:	A	value result
	X	pointer to first non-numeric character
	B	modified
	Y	not modified

DESCRIPTION

This function converts an ASCII number in a string to a binary value. The conversion stops at the first non-numeric character.

NAME

__find_spc
__skip_spc

SYNOPSIS

(__find_spc)

IN:	X	pointer to string
OUT:	X	pointer to space character or NULL
	A	space character or NULL
	B,Y	not modified

(__skip_spc)

IN:	X	pointer to string
OUT:	X	pointer to non-space character
	A	non-space character
	B,Y	not modified

DESCRIPTION

These functions search through a string to find the presence or absence of a space.

NAME

__put_asc
__put_asc_sm
__put_asc_u
__put_asc_usm

SYNOPSIS

(__put_asc, __put_asc_u)

IN:	D	value to convert
OUT:	D,X	modified
	Y	not modified

(__put_asc_sm, __put_asc_usm)

IN:	B	value to convert
OUT:	D,X	modified
	Y	not modified

DESCRIPTION

These functions convert a binary number to ASCII and print it. They use a common variable called __z_blank that determines how many of the leading digits are to be considered for zero blanking. For example, with __put_asc which is a 5 digit routine;

__z_blank	value = 99	value = 9	value = 0
3	99	09	00
4	99	9	0
5	99	9	

Normally, with an N digit routine, a __z_blank of N-1 would be used. __z_blank is modified by the function.

NAME

__put_asc_str
__put_asc_str_sm
__put_asc_str_u
__put_asc_str_usm

SYNOPSIS

(__put_asc_str, __put_asc_str_u)

IN:	D	value to convert
	X	address of string to store result
OUT:	D,X	modified
	Y	not modified

(__put_asc_str_sm, __put_asc_str_usm)

IN:	B	value to convert
	X	address of string to store result
OUT:	D,X	modified
	Y	not modified

DESCRIPTION

These functions convert a binary number to ASCII and store it in the string pointed to by the X register. They use a common variable called __z_blank that determines how many of the leading digits are to be considered for zero blanking. For example, with __put_asc_str which is a 5 digit routine;

__z_blank	value = 99	value = 9	value = 0
3	99	09	00
4	99	9	0
5	99	9	

Normally, with an N digit routine, a __z_blank of N-1 would be used. __z_blank is modified by the function.

NAME

__put_hex
__put_hex_sm

SYNOPSIS

(__put_hex)

IN:	D	value to convert
OUT:	D	modified
	X,Y	not modified

(__put_hex_sm)

IN:	B	value to convert
OUT:	A,B	modified
	X,Y	not modified

DESCRIPTION

These functions convert a binary number to HEX ASCII and print it. They use a common variable called __z_blank that determines how many of the leading digits are to be considered for zero blanking. For example, with __put_hex which is a 4 digit routine;

__z_blank	value = $99	value = 9	value = 0
2	99	09	00
3	99	9	0
4	99	9	

Normally, with an N digit routine, a __z_blank of N-1 would be used. __z_blank is modified by the function.

NAME

__put_hex_str
__put_hex_str_sm

SYNOPSIS

```
(__put_hex_str)
IN:   D     value to convert
OUT:  D     modified
      X,Y   not modified

(__put_hex_str_sm)
IN:   B     value to convert
OUT:  A,B   modified
      X,Y   not modified
```

DESCRIPTION

These functions convert a binary number to HEX ASCII and store it in the string pointed to by the X register. They use a common variable called __z_blank that determines how many of the leading digits are to be considered for zero blanking. For example, with __put_hex_str which is a 4 digit routine;

__z_blank	value = $99	value = 9	value = 0
2	99	09	00
3	99	9	0
4	99	9	

Normally, with an N digit routine, a __z_blank of N-1 would be used. __z_blank is modified by the function.

NAME

__put_oct
__put_oct_sm

SYNOPSIS

(__put_oct)

IN:	D	value to convert
OUT:	D	modified
	X,Y	not modified

(__put_oct_sm)

IN:	B	value to convert
OUT:	A,B	modified
	X,Y	not modified

DESCRIPTION

These functions convert a binary number to OCTAL ASCII and print it. They use a common variable called __z_blank that determines how many of the leading digits are to be considered for zero blanking. For example, with __put_oct which is a 5 digit routine;

__z_blank	value = o99	value = 9	value = 0
3	99	09	00
4	99	9	0
5	99	9	

Normally, with an N digit routine, a __z_blank of N-1 would be used. __z_blank is modified by the function.

NAME

__put_oct_str
__put_oct_str_sm

SYNOPSIS

(__put_oct_str)

IN:	D	value to convert
OUT:	D	modified
	X,Y	not modified

(__put_oct_str_sm)

IN:	B	value to convert
OUT:	A,B	modified
	X,Y	not modified

DESCRIPTION

These functions convert a binary number to OCTAL ASCII and store it in the string pointed to by the X register. They use a common variable called __z_blank that determines how many of the leading digits are to be considered for zero blanking. For example, with __put_oct_str which is a 5 digit routine;

__z_blank	value = o99	value = 9	value = 0
2	99	09	00
3	99	9	0
4	99	9	

Normally, with an N digit routine, a __z_blank of N-1 would be used. __z_blank is modified by the function.

NAME

__search
__str_lookup

SYNOPSIS

(__search)

IN:	X	pointer to search string
	Y	pointer to string list
OUT:	X	not modified
	Y	modified
	A	ordinate of string in list or -1
	B	not modified

(__str_lookup)

IN:	X	pointer to string list
OUT:	X	pointer to string
	Y	not modified
	D	modified

DESCRIPTION

These functions perform inverse actions on string lists. ‘search’ searches for a string in the list and returns its ordinal number (first string in list => 0). ‘str_lookup’ finds a string given the ordinal number.

A string list is a linear sequence of strings terminated by a null string, for example:

```
DC      "string0",0
DC      "string1",0
DC      "string2",0
DC      0
```

ADEV11 Development System for AmigaDOS
Examples

Version 1.1

NAME

monitor

DESCRIPTION

Monitor is a simple monitor program that fits in the EEPROM of a 68HC11F1 or any HC11 which meets the following memory requirements; 504 bytes of non-volatile, 404 bytes of ram. It provides some basic I/O calls. It is different from the typical debug monitor in that the program under test can supplement the commands of the monitor and that the program under test is expected to be command driven rather than jumping to an address in the program to start it.

COMMAND SET

The basic command set is very limited with all commands associated with downloading an S-record file.

S0	accept an S0 (identification) record
S1	accept an S1 (data) record
S9	accept an S9 (start address) record. When this is received execution immediately transfers to that location via a JSR. This code should initialize any variables, registers and/or interrupts, hook up to the command list and then return.

RESOURCES

The following resources are available to the code under test.

Vectors

All of the vectors are available as 3 byte RAM locations. When the associated interrupt occurs a jump is made to the first of these bytes. Normally a jump instruction would be placed here; a $7E would be the first byte and the address of your function would be the second and third (MSB first). The vector names are listed below.

SCI_JMP	SPI_JMP	PULSE_IN_JMP
PULSE_OVF_JMP	TIMER_OVF_JMP	TIMER_IC4_JMP
TIMER_OC4_JMP	TIMER_OC3_JMP	TIMER_OC2_JMP
TIMER_OC1_JMP	TIMER_IC3_JMP	TIMER_IC2_JMP
TIMER_IC1_JMP	RT_INTR_JMP	IRQ_JMP
XIRQ_JMP	SWI_JMP	OPC_JMP
COP_FAIL_JMP	MON_FAIL_JMP	

Variables

Only one variable is available to the code under test.

EXT_TABLE	A non-zero address in this variable is used as a pointer to a command table which the monitor will search if it receives a command that it cannot resolve in its own table. If the command is found, the associated address is jumped to via a JSR. A command table consists of a sequence of individual commands followed by a null byte. The individual commands consist of a null terminated command name string followed by the 2 byte address of the code to be executed when the command is given.

Functions

The following monitor functions are available to the program under test.

GETC	Gets a character from the serial port
GETS	Gets a null terminated string from the serial port
PUTC	Puts a character to the serial port
PUTS	Puts a null terminated string to the serial port
PCRLF	Puts a newline (carriage return/line feed) to the serial port
RDBYTE	Gets a hex ASCII byte from the serial port
str_cmp	Compares two strings

These functions are available by XREFing them and linking with board.o (supplied). A sample link command file (your.sln) is also provided. Note that the board.a segment names (BOARD_VECTORS and BOARD_RAM) must be first in their memory area and must correlate with the memory areas used to generate the monitor.

BUILDING THE MONITOR

To generate the S-record file to program your EEPROM or EPROM configure the file download.sln to match the memory available on your board. There are three _SIZE equates in download.a that may also require change. Perform a make operation using the file lmkfile. If you have the SAS/C compiler this will work directly, other make utilities may require simple changes. If you do not have a make utiltiy then you will have to do the link by hand.

```
slink with download.sln
```

OPERATION

When the processor is reset the monitor does a memory check to see that the memory that it expects is present and functional. It then checks to see wether the reset that started the monitor was due to a reset input or a power up. If it was a reset input then the monitor checks for an EXT_TABLE and if that is found it executes the first command in that table. It then sends a prompt character ('>') to the serial port and waits for a command to be entered.

Due to a shortage of EEPROM memory on the F1 no editing of the command line is possible and the commands, both monitor and user, are case sensitive.

When a command is entered and found in a table a JSR to the associated address is performed. At this point the X register points to the command line string and any additional arguments may be extracted. All registers, except the stack, may be changed. When the command code execution is complete its last statement must be an RTS which returns control to the monitor which issues another prompt.

ADEV11 Development System for AmigaDOS
Utilities

Version 1.1

NAME

HCLoad

SYNOPSIS

HCLoad [-options] srecord_data
HCLoad [-options] -ssrecord_data
where:
srecord_data   a mandatory data file containing S records

DESCRIPTION

This program loads Motorola S record files into the EPROM, EEPROM or RAM of an MC68HC11 type micro-processor. The micro-processor must have it's serial port connected to the Amiga's serial port and the micro-processor must be reset into the bootstrap mode.

HCload, when first run, downloads a bootstrap program to the MC68HC11 processor. The bootstrap program can be either one of the built in bootloaders or an external one custom made for the processor. This bootstrap program then in turn will load a S record file into the appropriate EPROM, EEPROM or RAM. As the program runs a byte counter is displayed to give an indication of progress. If for some reason the MC68HC11 cannot load the data sent to it, HCload will time out and notify the operator.

The bootloader for the 711k4 processor will automatically determine what type of memory is being addressed by the S record and automatically switch to the proper programming algorithm for that memory. This allows RAM, EEPROM and EPROM to be programmed with the same utility.

HCLoad accepts the following options:

-fx.xxx   Where x.xxx is the MC68HC11 xtal frequency in MHz. Default is 8.00. This option is used to adjust the Amiga baud rates to that of the MC68HC11 when using xtals other than 8.00 MHz.

-p#   The Amiga serial port number. Port 0 is the default.

-hxxxxx   Where hxxxxx is the MC68HC11 processor type. Depending on the processor type different bootloaders are used. These are the current built in bootloader selection options.
-hc711k4
-hc811e2
-hc11f1
The default is -hc811e2

-bfile   Where file is the name of a binary file to be used instead of the built in bootloaders. This allows a user to create a bootloader for newly released processor types or to have a custom bootloader. The file must contain the starting character and the correct number of data bytes as specified by Motorola.

EXAMPLE

HCload -stest.srec -f4.95 -p3 -hc711k4

The example will first upload the built in hc711k4 bootloader. Serial port 3 will be used (for those lucky enough to have a multi-serial card) and the Amiga baud rates will be adjusted by a ratio of 4.95/8.00. When the bootloader is finished, the file "test.srec" which contains Motorola S records will be uploaded. If a transfer address was specified in the file then execution will begin at this address.

NOTE

HCload was written by Ron Eirich, its original author, the version made available in the previous releases was a modification of his original work. He has kindly updated and contributed it to this release. If you have any problems with it, contact me.

NAME

MHex

SYNOPSIS

MHex [<options>] in_file_name out_file_name

where options are	-s<num>	EPROM start address
	-e<num>	EPROM end address

DESCRIPTION

MHex is a utility for converting an S-record file into a binary file. After loading the specified S-record file, the utility will ask for the start and end address for the EPROM (or memory block) if these were not specified on the command line. MHex must know this since the binary file has no address information and thus every byte must be generated sequentially from the start of the device to the last used location.

NAME

SProg

SYNOPSIS

SProg

DESCRIPTION

SProg is a Graphical User Interface (GUI) for controlling an EPROM programmer. It was designed with the A.R.T. EPP-2 (available from Jameco Electronics) in mind but with an idea to control other programmers. This programmer is relativly inexpensive, but very capable and handles a wide range of EEPROM devices. Best of all, it is not IBM PC specific. If you use another programmer and cannot use the builtin configurability of this program, send me a copy of its commands and I will see if I can make some changes.