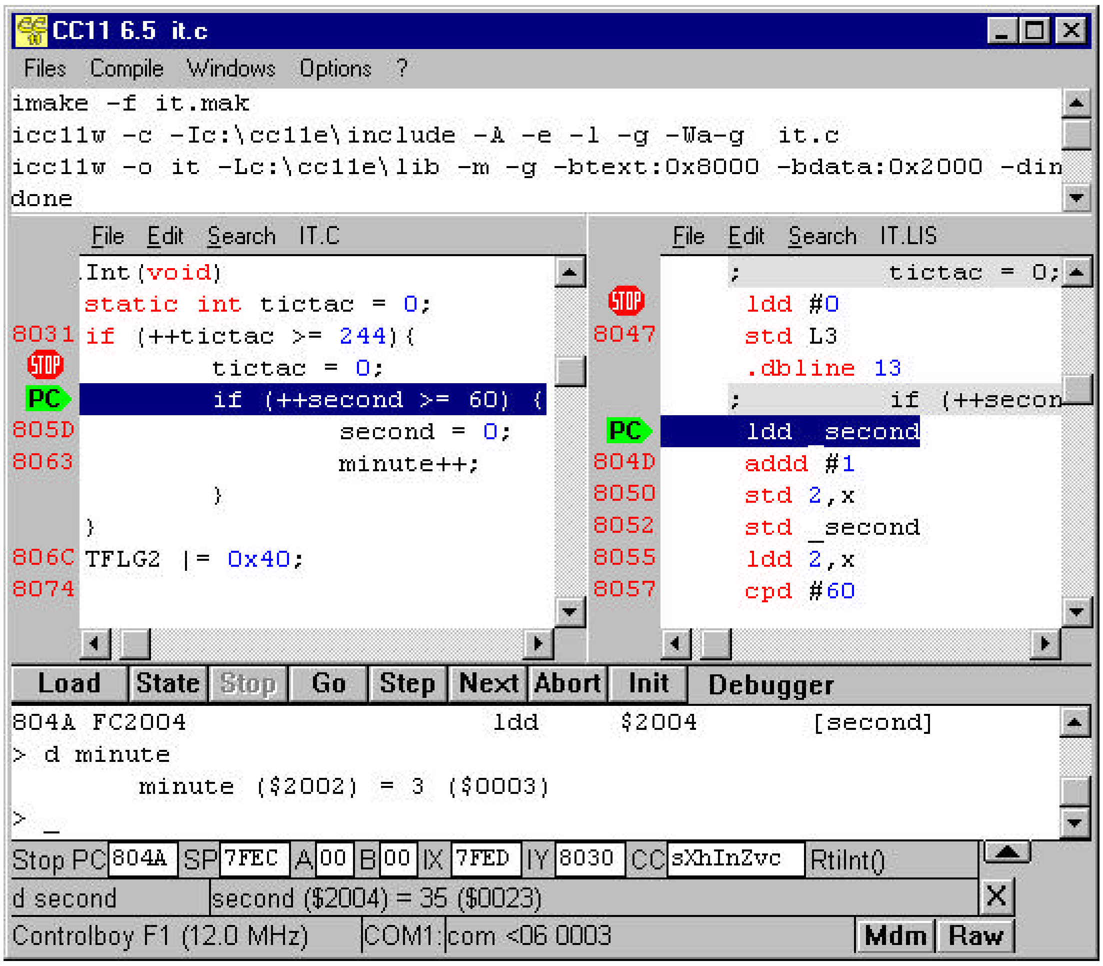
CC11, Basic11, Debugger, Simulator
Development Tools for the 68HC11


CC11 6.5 it.c
Files Compile Windows Options ?
File Edit Search IT.C
File Edit Search IT.LIS
STOP
PC
Load State Stop Go Step Next Abort Init Debugger
Mdm Raw

```
imake -f it.mak
icc11w -c -Ic:\cc11e\include -A -e -l -g -Wa-g  it.c
icc11w -o it -Lc:\cc11e\lib -m -g -btext:0x8000 -bdata:0x2000 -din
done
```

```
     .Int(void)
      static int tictac = 0;
8031  if (++tictac >= 244){
              tictac = 0;
              if (++second >= 60) {
805D                  second = 0;
8063                  minute++;
              }
      }
806C  TFLG2 |= 0x40;
8074
```

```
     ;          tictac = 0;
       ldd #0
8047   std L3
       .dbline 13
     ;          if (++secon
       ldd  _second
804D   addd #1
8050   std 2,x
8052   std _second
8055   ldd 2,x
8057   cpd #60
```

```
804A FC2004                    ldd      $2004         [second]
> d minute
        minute ($2002) = 3 ($0003)
> _
```

```
Stop PC 804A SP 7FEC A 00 B 00 IX 7FED IY 8030 CC sXhInZvc RtiInt()
d second        second ($2004) = 35 ($0023)
Controlboy F1 (12.0 MHz)    COM1:com <06 0003
```

The programming interface allows you to write a program, compile it, download it, and debug it. The program can be written in assembly language, Basic, or C. The window on top is the main window. It allows you to start one or several editor windows to edit the source programs, and to press COMPILE to compile the program. You will get the output of the compilation in this window.
The WINDOWS menu keeps all source files and all files included by the #include directive in these files. If you click on a file in this menu, you will also bring up the file in an editor window.
The OPTION dialog allows you to specify the name of the compiler, compiler options, the files to compile, and others.

Double clicking on a Basic or C source line brings up the assembly source lines generated by the compiler for the high-level source line. Double clicking on an assembly line brings you back to the original high level source line.

The debugger in the bottom window communicates with the target. When the debugger reaches a breakpoint or finishes a single step, you see in the left window the original source line and in the right window the assembly line, generated by the compiler. Clicking on the left margin of a source window (high-level or assembly) sets or removes a breakpoint.

CC11

Controlord
484, avenue des Guiols
F 83210 La Farlede
Tel. 04 94 48 71 74
Fax 04 94 33 41 47
controlord@controlord.com
www.controlord.com

04/2000

1 Introduction

CC11 is a C cross compiler for the Motorola MC68HC11 family of microcontrollers. The compiler runs on a Windows platform to create object files for a 68HC11. The debugger allows you to download this object into the memory of the 68HC11 target and to debug the program.

CC11 accepts the ANSI C language with the following exception: The supplied library is only a subset of what is defined by the standard. Most missing functions do not make sense in an embedded microcontroller environment.

The compiler uses the following types:

unsigned char	8 bits
signed char	8 bits
unsigned short	16 bits
signed short	16 bits
unsigned int	16 bits
signed int	16 bits
unsigned long	32 bits
signed long	32 bits
float	32 bit, exponent 8 bits, mantissa 24 bits
double	32 bit, exponent 8 bits, mantissa 24 bits
pointer	16 bits
void	

The compiler uses three segments.

- The instructions of the program are stored in the TEXT segment. This segment will be loaded into a nonvolatile memory of the target like an EPROM or an EEPROM.
- Initialized data is put into the DATA segment. This segment will be allocated in the RAM of the target. The data will be copied into the RAM before starting the C program.
- Uninitialized data is allocated in the BSS segment. This segment will be allocated in the RAM of the target and initialized to zero before starting the C program.

The result of the compilation is a file in Motorola S-record format and additional information for symbolic debugging.

- ANSI C standard header files, preprocessor
- Standard library in source and object: Printf, isdigit, memcpy, atoi, malloc, …
- Floating point 32 bits: arithmetic, conversation, math functions
- Definition of the 68HC11 target ports
- Interrupt functions
- C program may call an assembly routine, and an assembly routine may call a C routine.
- Program may reside in an EEPROM or in an EPROM
- Examples for Printf on LCD, floating point, interrupt functions
- Linker, library archiver, shared libraries
- Make

Read this:
Brian W. Kerninghan, Dennis M. Ritchie
The C Programming Language
Second Edition. ANSI C
Prentice-Hall, ISBN 0-13-110362-8

2 Directories

After installation you will have the following subdirectories

`Bin`	Executables
`Include`	Header files (#include)
`Lib`	Library files
`Libsrc`	Source files of the library
`Examples`	Programming examples
`Talkers`	Source files of the talkers for the target

3 Using the Compiler

3.1 Example 1: Flashing a LED

Following you find the small program **`../examples/flash.c`**.

```
#include <hc11.h>

void wait(int cnt);
int x;

void
main(void)
{
        DDRG  = 0x01;                       /* PG1 = output */
        for(;;) {
                PORTG ^= 0x01;              /* flash led */
                wait(20000);
                x++;
        }
}

void
wait(int cnt)
{
        for (;cnt>0; cnt--);

}
```

Click on FILE, OPEN to open the file. The program flashes a LED L2 on a Controlboy F1 board. You can easily adapt the program to run on another board flashing a LED on another port. Click on OPTIONS, FLASH.MAK to verify the parameters of the target memory.

-BTEXT	start of the EEPROM of the target	0x8000 for Controlboy F1
-BDATA	start of the RAM of the target.	0x2000 for Controlboy F1
-Binit_sp	stack pointer, end of the RAM	0x7FFF for Controlboy F1

When these parameters are correct, close the dialog, and click on COMPILE in the main window. The Windows IDE starts IMAKE under DOS to interpret the file **`flash.mak`** which will run ICC11W to compile and link the program. All messages from the compiler tools will be shown in the window. When the compiler finishes with no errors, the result is the file **`flash.s19`** . This is the file Motorola S-record that can be loaded into the targets program memory. You can see this file clicking on WINDOWS, FLASH.S19.

To download the program, the debugger must show you the target in the STOP state. When the target runs, click on STOP to stop the current program. When there is no connection with the target, read the chapter Preparing the Software and the Target. Click on LOAD to download the program into the program memory on the target. Click on GO or press the reset button to start the program.

The LED should flash now.

You may click on STOP to stop the program. Two windows then show you on the left the source code of the C program and on the right the assembly listing file, created by the compiler. The left margin of these windows display the addresses in the target memory. A PC indicates the current location of the PC. Type in the debugger window

```
d x
```

to show the value of the variable.

3.2 Example 2: Printf on a LCD

The second example is the file **`../examples/print.c`**. This program uses printf() of the standard library to print something on standard output. As there is no standard output like working with an ASCII terminal, the application must furnish this standard output. All functions using standard output are based on **`putchar(char c)`** . Realizing this function to print a single character on a liquid crystal display, all standard output will be routed to the display. The source file **`print.c`** contains **`main()`** which calls **`printf()`** of the standard library, which calls **`putchar()`** to write to the display.

```
// Program for Controlboy F1: printf on LCD

#include "hc11.h"
#define PORTM     *(unsigned char *)(_IO_BASE + 0x62)
#define PORTN     *(unsigned char *)(_IO_BASE + 0x63)

#include <stdio.h>

void lcdinit(void);

void
main(void)
{     lcdinit();
      printf(" Controlboy F1   ");
}

int
putchar(char c)               /* library function     */
{
...
```

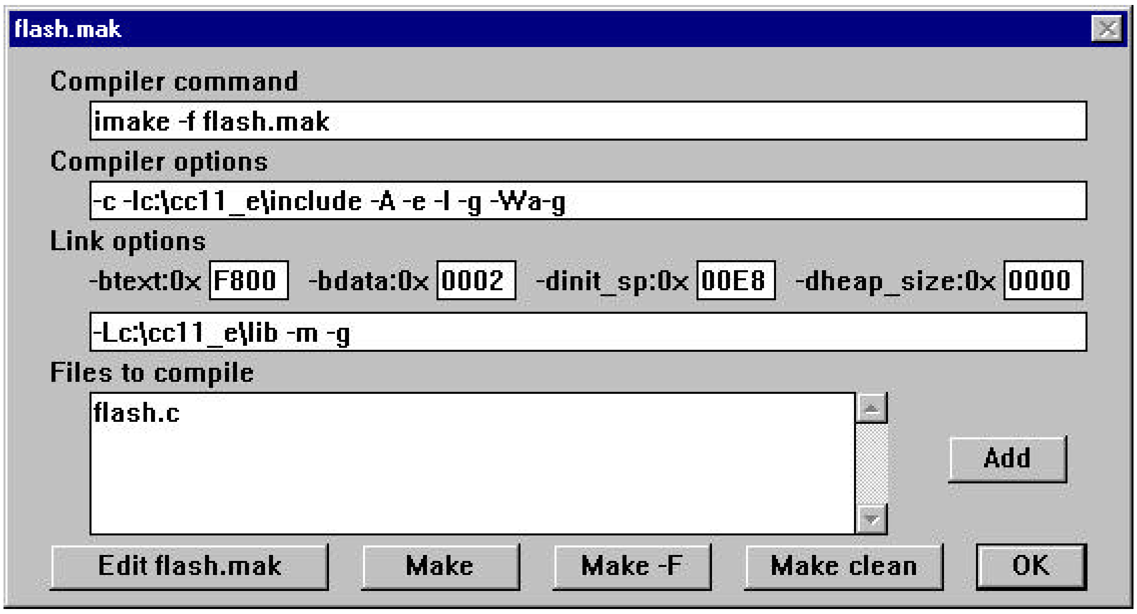
3.3 Example 3: Floating Point

The third example **`../examples/float.c`** calculates the square root of 2 and prints out the result. Note that the linker option **`-lfp11`** must be set in the window FLOAT.MAK

```
// Program for Controlboy F1: printf floating point on LCD

#include "hc11.h"
#define PORTM     *(unsigned char *)(_IO_BASE + 0x62)
#define PORTN     *(unsigned char *)(_IO_BASE + 0x63)

#include <stdio.h>
#include <math.h>
void lcdinit(void);

double e;

void
main(void)
{     lcdinit();
      e = sqrt(2.0);
      printf("sqrt(2)=%f", e);
}
...
```

4 Compiler Options, Make

CC11 uses the tool IMAKE to construct the final executable from the source files. When you have made your source file **`flash.c`** , CC11 automatically creates a **`flash.mak`** file for you. Click on OPTIONS, FLASH.MAK. You may change the most common parameters in the dialog or by clicking on EDIT FLASH.MAK edit directly the file **`flash.mak`**. Click on MAKE to run IMAKE to create the executable. Make will only recompile those files that changed and use existing objects when possible. Click on MAKE -F to recompile all. MAKE CLEAN cleanes the directory from object and intermediate files. Only source files will stay.


flash.mak

Compiler command

imake -f flash.mak

Compiler options

-c -Ic:\cc11_e\include -A -e -l -g -Wa-g

Link options

-btext:0x F800 -bdata:0x 0002 -dinit_sp:0x 00E8 -dheap_size:0x 0000

-Lc:\cc11_e\lib -m -g

Files to compile

flash.c

5 Customizing the Compiler

5.1 Crt11.s, End.s

The source files **`crt11.s`** and **`end.s`** are in the directory **`../libsrc`**. The linker includes the file **`../lib/crt11.o`** as program startup in the executable at the beginning. You may probably adapt this file to your board. **`Crt11.s`** contains declarations concerning the memory and how to use it. The program is executed before any C program. It gives a proper environment for the running of the C programs. It initializes the stack pointer, loads the DATA segment from the copy in the EEPROM, and clears the BSS segment.

It then calls the c program **`main()`**. Note that the compiler always puts an underscore before a C name, it thus calls **`_main()`**. When **`main`** terminates or calls **`exit()`** , the program enters a final endless loop.

The linker includes the file **`end.s`** at the end of the executable. This file contains declarations to calculate the sizes of the segments.

You may change **`crt11.s`** in the directory **`../libsrc`** . There is a file **`crt11.mak`** to compile the file and to install **`crt11.o`** in **`../lib`** .

5.2 Hc11.h

The file **`../include/hc11.h`** contains declarations of the registers of the 68HC11. You may adapt this file to your specific 68HC11 CPU and to your board. Include this file in the C source using:

```
#include <hc11.h>
```

The file contains declarations like the following:

```
#define _IO_BASE  0x1000
#define PORTA     *(unsigned char volatile *)(_IO_BASE + 0x00)
```

You may use these declarations in your program like:

`PORTA = 0x08;`	set the port to 08
`i = PORTA;`	read the port
`PORTA |= 0x08;`	set bit 3 to 1
`PORTA &= ~0x08;`	set bit 3 to 0
`if (PORTA & 0x08)`	examine bit 3

5.3 Putchar

A program may use **`printf()`** of the standard library to print something on standard output. As there is no standard output like working with an ASCII terminal, the application must furnish this standard output. All functions using standard output are based on one single function:

```
int putchar(char c)
```

On realizing this function to print a single character, on realize the complete C standard output.

Following are examples of **`putchar()`** and **`getchar()`** for the RS232 interface of the 68HC11.

```
#include <hc11.h>

#define bit(x)    (1 << (x))
#define RDRF      bit(5)
#define TDRE      bit(7)

int getchar()
      {

      while ((SCSR & RDRF) == 0)
            ;
      return SCDR;
      }

void putchar(unsigned char c)
      {

      if (c == '\n')
            putchar('\r');
      while ((SCSR & TDRE) == 0)
            ;
      SCDR = c;
      }
```

6 Compiler Runtime Architecture

Names are significant up to 32 characters. The compiler puts an underscore(_) before a C name. Function arguments are evaluated from right to left. Bitfields are allocated from left to right.

6.1 Segments

The compiler uses three segments.

- The instructions of the program are stored in the TEXT segment. This segment will be loaded into a nonvolatile memory of the target like an EPROM or an EEPROM.
- Initialized data is put into the DATA segment. This segment will be allocated in the RAM of the target. The data is stored in the IDATA segment, which will be loaded into the nonvolatile memory. The data will be copied into the RAM before starting the C program.
- Uninitialized data is allocated in the BSS segment. This segment will be allocated in the RAM of the target and initialized to zero before starting the C program.

You must also assure enough space for the stack of the program at the end of the RAM.

If your program uses heap functions like **`malloc()`**, you must provide the memory for the heap.

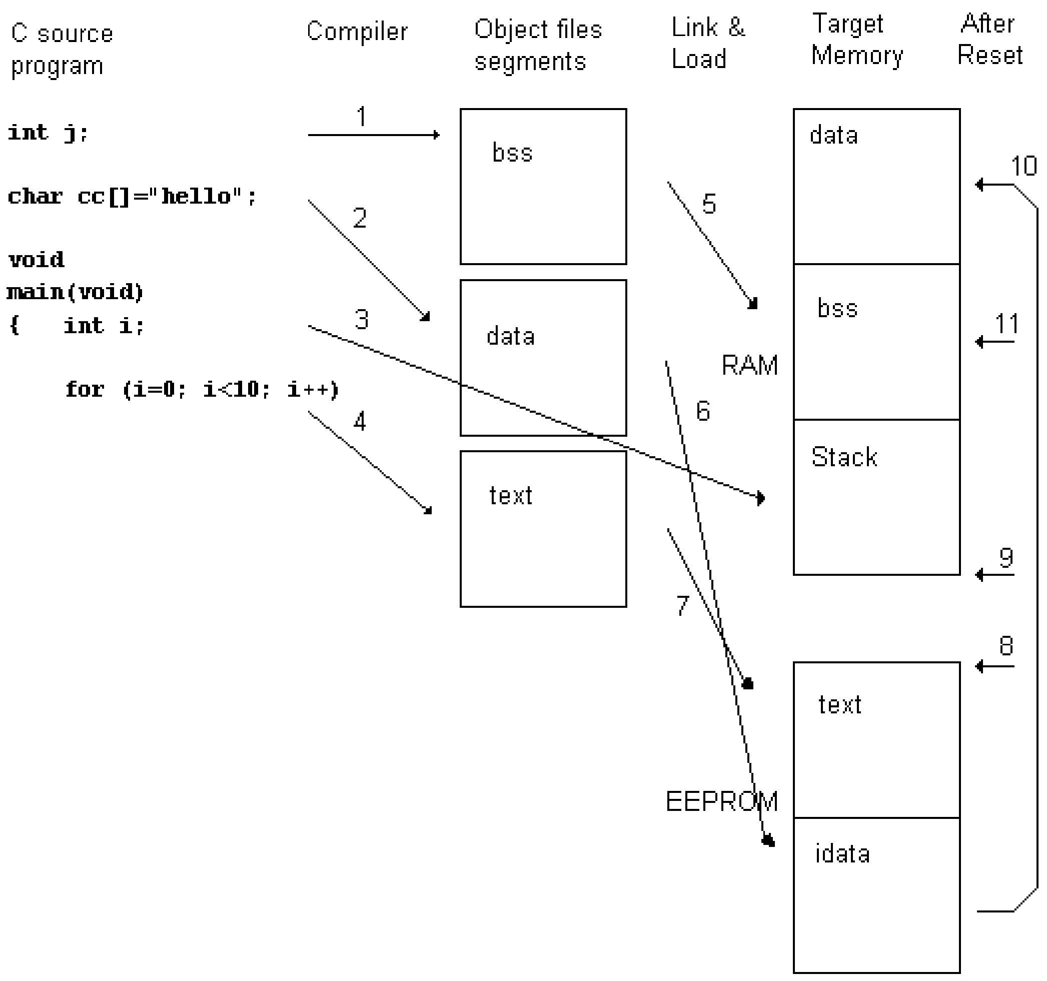
The linker produces a map file with the extension **`.mp`** that shows the final addresses in the target memory space.

Variables declared **`const`** are stored in the TEXT section. See the following examples

`char a[100];`	The variable is in the BSS segment in the RAM. It will be initialized to 0 before starting the C program.
`char b[]="Controlboy";`	The variable is in the DATA segment in the RAM. The data is stored in the nonvolatile memory and copied to the variable before starting the C program.
`const char c[]="Controlboy";`	The variable and the data are in the nonvolatile memory. The program can not write to the variable.

The directive **`#pragma abs_address`** allows to store data at an absolute address.

```
#pragma abs_address:0xFFF0
void (*rtiint_vector)() = RtiInt ;
#pragma end_abs_address
```



C source program
Compiler
Object files segments
Link & Load
Target Memory
After Reset
int j;
char cc[]="hello";
void
main(void)
{   int i;
    for (i=0; i<10; i++)
1
2
3
4
bss
data
text
5
6
7
RAM
EEPROM
data
bss
Stack
text
idata
10
11
9
8

6.2 Types

The compiler uses the following base data types.

unsigned char	8 bits
signed char	8 bits
unsigned short	16 bits
signed short	16 bits
unsigned int	16 bits
signed int	16 bits
unsigned long	32 bits
signed long	32 bits
float	32 bit, exponent 8 bits, mantissa 24 bits
double	32 bit, exponent 8 bits, mantissa 24 bits
pointer	16 bits
void	

The **`char`** type is equivalent to **`unsigned char`**.

Float and double use the IEEE single precision format. Floating point calculation is done in global memory and not reentrant.

The header file **`limits.h`** defines the implementation specific types.

```
#define CHAR_BIT  8
#define CHAR_MAX  SCHAR_MAX
#define CHAR_MIN  SCHAR_MIN
#define INT_MAX        32767
#define INT_MIN        -32768
#define SCHAR_MAX 127
#define SCHAR_MIN -128
#define SHRT_MAX  INT_MAX
#define SHRT_MIN  INT_MIN
#define UCHAR_MAX 255
#define UINT_MAX  65535u
#define USHRT_MAX UINT_MAX
#define LONG_MAX   2147483647L
#define LONG_MIN  (-2147483647L-1)
#define ULONG_MAX 4294967295UL
```

The file **`float.h`** describes the implementation specific float types.

```
#define DBL_MAX   3.402823466e+38f
#define DBL_MIN   1.175494351e-38f
#define FLT_MAX   3.402823466e+38f
#define FLT_MIN   1.175494351e-38f
```

6.3 Interrupt Functions

The following directive declares an interrupt handler.

```
#pragma interrupt_handler <nom> [<nom>]*
```

The example **../examples/it.c** uses an interrupt handler for the real timer. It counts the time in the variables second, and minute. The directive **#pragma abs_address** is used to load the address of the handler as interrupt vector.

```
#include <hc11.h>
#include <stdio.h>
int second;
int minute;

#pragma interrupt_handler RtiInt
void RtiInt(void)
{     static int tictac = 0;
      if (++tictac >= 244){        /* 8 Mhz */
            tictac = 0;
            if (++second >= 60) {
                  second = 0;
                  minute++;
            }
      }
      TFLG2 |= 0x40;
}

void
main(void)
{
      PACTL &= 0xFC;          /* speed */
      TMSK2 |= 0x40;          /* start the timer */
      asm(" cli ");           /* allow ints */
      for (;;);
}

#pragma abs_address:0xFFF0
void (*rtiint_vector)() = RtiInt ;
#pragma end_abs_address
```

Compile and load the program into the target, start the program, and see what happens, using the debugger.

```
> d minute, second
```

7 Assembly Code

The compiler allocates an argument block for a function on function entry. Therefore it does not generate explicit argument pushings and poppings when a function is called. This generates shorter and faster code. Local variables and function parameters are addressed via the X index register. Even though the HC11 has a limitation of only allowing 8 bit offset (256 bytes) from an index register, the compiler generates correct code to access items that are more than an 8 bit displacement away. However, it requires several instructions to do the extra computation, so if you are concerned about code size or speed, you may try not to use more than 256 bytes of local storage. The X register is used as the frame pointer and must be restored to its original value at function exits. Since floating point computation uses a fair amount of code space, it is likely that programs using floating point would not fit in a single chip mode system because such a system only contains a small amount of code storage.

7.1 Calling Conventions

Time or space critical code can be written in assembler language routines or embedded assembly code in your C source. The compiler prepends an underscore to all global functions and data names. Thus, to access such objects in assembly code, you must prepend an underscore to the name of the object. Arguments are promoted to their natural sizes, and pushed from right to left on the stack, except for the first argument which is passed in the D register, unless the first argument is a structure, floating point, or long.

original type	as parameter
Char 8 bits	Int 16 bits
Short 8 bits	Int 16 bits
Int 16 bits	Int 16 bits
Long 32 bits	Long 32 bits
Float 32 bits	Double 32 bits
Double 32 bits	Double 32 bits
Pointer 16 bits	Pointer 16 bits

To call a function that returns a structure or float, the compiler generates code to pass the address of a (temporary) structure using the D register. This structure holds the return value of the function when the function returns. The stack pointer SP points to one byte below the return address . A typical function prologue sequence pushes the first argument from the D register on the stack, then pushes the previous frame pointer X on the stack, and finally sets the X register to the current stack pointer. It then adjusts the stack pointer by the amount of local storage the function needs and transfers the stack pointer to index register X. All local variables and arguments are referenced by using a displacement off the X register.

```
int toto(int a, int b);

void
main(void)
{     int   i;
      i = toto(3, 7);
}
```

```
            .text
_toto::     pshx
            tsx
            addd  4,x   ; 2. parametre
            pulx
  rts
```

7.2 Embedded ASM Statement

In addition to linking with assembly modules, you may embed arbitrary assembly statements in your C programs. The format is:

```
        asm("asm string");
```

The compiler inserts a tab at the beginning of each line and a newline at the end of the specified string. C escape sequences such as \n can also be used to embed a series of assembly instructions with a single asm call. Inside the supplied string, a reference in the form %<variable name>, where <variable name> is an in-scoped data variable, will be replaced by the assembly reference to the variable. Note that the peephole optimizer ignores the instructions inside an asm() statement. This means branches across asm() statements will always be long branches. To generate multiple line embedded assembly statements, you may write multiple asm() calls, or use the '\n' escape character, combined with the string concatenation feature of ANSI C:

```
        int i;
        asm("ldd %i\n" /* note no comma */
        "std 0x1000\n" /* %i will be replaced by */
        "bra .-4"); /* ?,x */
```

You can only use asm() in an expression statement context (i.e., only by itself and not as part of a larger expression), or outside a function definition. You must be careful to preserve the X register in embedded asm since it is the frame pointer.

8 ASSEMBLER

The compiler generates assembler code which is then processed by the assembler. The assembler generates a relocatable object file from the input. You may also write assembler routines and link them into your C program. This chapter describes the format of the assembly language accepted by the
assembler. This format is slightly different from the Motorola Assembler.

Relocatable Sections

Assembly code is grouped into relocatable or absolute sections. The linker combines together sections of the same names from all the object modules. At link time, you specify the start address of each relocatable section and the linker adjusts symbol references to their final addresses. This process is known as relocation.

Notation

A <name> is a sequence of 32 or less characters consisting of alphabets, digits, dots (.), dollars ($), or underscores (_). A name must not start with a digit.

A <number> is a sequence of digits in C format: a 0x prefix signifies a hexa-decimal number, a 0 prefix signifies an octal number and no prefix signifies a decimal number. In addition, 0b signifies a binary number. You may also indiciate hex number by using the $ prefix.

An <escape sequence> is C style \n, \t, etc. plus \0xxx octal constants. \e refers to the escape character.

A <string> is a C string: a sequence of characters enclosed by double quotes (“). A double quote within the string must be prefixed by the escape character backslash (\).

An <exp> is a relocatable expression. It is either:
1. a term, i.e., a dot (which denotes the current program counter value 1), a number, an escape sequence, a name, or
2. an expression enclosed with ‘(‘ and ‘)’, or
3. two expressions joined with a binary operator. These binary operators, with the same meanings as in C, are accepted: >> << + - * / % & | ^
4. a unary prefix operator applied to an expression:
   - upper byte of an expression
   - < lower byte of an expression
   - ‘x character “x”
   - “ab double byte value of characters “a” and “b”
   - and the following C style unary operators: - + ~

Unlike C, all operators have the same precedent so you may have to insert parentheses to group the expressions. Also note that only one relocatable symbol may appear in an expression.

Direct Page Reference

Some instructions take a direct page reference as an operand. That is, the address of the operand is in the first 256 bytes of memory. You cannot specify a direct page variable in C, but you can do so in assembly by prefixing a variable or a direct address with a ‘*’ in the references:

```
bset    *_foo, #0x23
bclr    *0x20, #0x32
```

You must ensure that a direct page variable is defined within the first 256 bytes. Otherwise, you will get an error when you link the program.You may do so by putting such variables in an absolute area:

```
      .area memory(abs)
      .org  0
_foo:: byte 1
```

Format

An assembly file consists of lines of assembly text in the following format. Lines greater than 128 characters are truncated:

[label:] operation [operand]

In addition, comments can be introduced anywhere on the line with the ‘;’ charac-ter. All characters remaining in the line after the comment character are ignored.

A label defines a relocatable symbol name. Its value is the program counter value at the point where the label appears in the final linked executable. Zero, one, or more labels may exist on a source line. A label must end with a single colon ‘:’, or two colons “::”, the latter case signifying that the label is a global symbol (that is, one that can be referenced from another object module). An operation is either an assembler directive or an HC11 opcode.

Assembler Directives

Directives are operations that do not generate code but affect the assembler in certain ways. The assembler accepts the following directives:

.text specifies that the following data and instructions belong to the text section.

.data specifies that the following data and instructions belong to the data section. If you create any data items in the data section in an assembler module, you must define the same values in the idata section immediately following it. At program startup time, the idata section is copied to the data section and the sizes of the two sections must match. In fact, it is better to reserve the space in the data area and define the values in the corresponding idata area. For example:

```
.data
_mystuff::
      .blkb 5
.area idata
      .byte 1, 2, 3, 4, 5 ; _mystuff gets these at startup
```

.area <name> specifies that the following data and instructions belong to a section with the given name. .text is a synonym for .area text and .data is a synonym for .area data. The compiler only uses the text, data and bss sec-tions. <name> may optionally be followed by the attribute “(abs),” signifying that this area is an absolute section and can contain .org directives.

.org <exp> change the program counter to the address specified. This directive is valid only within an absolute area.

.byte <exp>[,<exp.]* (or .db ...) allocates bytes and initializes them with the value(s) specified. For example, this allocates 3 bytes with the values 1, 2, and 3.

```
.byte 1,2,3
```

.word <exp>[,<exp.]* (or .dw ...) allocates words and initializes them with the value(s) specified.

.blkb <number> reserves number bytes of storage without initializing their contents.

.blkw <number> reserves number words of storage without initializing their contents.

.ascii <string> allocates a block of storage large enough to hold the string and initializes it with the string.

.asciz <string> allocates a block of storage large enough to hold the string plus 1, and initializes it with the string followed by a terminating null.

.even forces the current program counter to be even.

.odd forces the current program counter to be odd.

.globl <name>[,<name>]* declares that name(s) are global symbol(s) and can be referenced outside of this object module. This has the same effect as defining the label with 2 colons following it.

.if <exp>, **.else**, and **.endif** implement conditional assembly. If the <exp> is nonzero, then the assembly code up to the matching .else or matching .endif is processed. Otherwise, the assembly code from the matching .else, if it exists, to the matching .endif is processed. Conditionals may be nested up to 10 levels.

.include <string> opens the file named by the double quoted “string” and processes it.

<name> = <exp> assigns the value of the expression to the name.

HC11 Instructions

Instruction operands, if they exist, take the form of an expression, which is usually a constant or memory address expressed using the above format and operators.
Bclr/brclr and bset/brset use different syntax from that of the motorola assembler:

```
bclr [opnd],#mask
brclr [opnd],#mask,label
```

and

```
bset [opnd],#mask
brset [opnd],#mask,label
```

[opnd] must either be

- a direct page reference (e.g., *_foo, or *0x10), or
- an indexed operand (e.g., 12,x or 3,y)

9 Standard C Library and Header Files

The compiler comes with a subset of the Standard C library. Most functions are Standard C conformant, with the most notable exception being that printf() only supports a subset of the format characters since supporting the full set would make the code too large. Functions based on an underlying operating system, like file input output, signals, error functions, date and time functions are not implemented.

You will find in the **`../lib`** subdirectory three standard libraries.

`Libc11.a`	The standard library that will be included automatically by the linker into your program. The printf() of this library does not handle long types nor float or doubles.
`Liblng11.a`	Like the above, but the printf() handles long types. Do include this library, use the linker option **`-llng11`** .
`Libfp11.a`	Like the above, but the printf() handles long types, floats, and doubles. Do include this library, use the linker option **`-lfp11`** .

You will find in the **`../libsrc`** subdirectory:

- The sources of all files of the standard libraries.
- A makefile **`libc.mak`** to compile these files and to install the libraries in **`../lib`**
- The source files **`crt11.s`** and **`end11.s`**
- A makefile **`crt11.mak`** to compile and to install these files.

Printf

int printf(char *fmt, ..) prints out formatted text according to the format specifiers in the fmt string. The format specifiers are a subset of the standard formats:

%d	prints the next argument as a decimal integer
%o	prints the next argument as an unsigned octal integer
%x	prints the next argument as an unsigned hexidecimal integer
%u	prints the next argument as an unsigned decimal integer
%s	prints the next argument as a C nul terminated string
%c	prints the next argument as an ASCII character
%f	prints the next argument as a floating point number (you must include the library libfp11.a to use this)

If a '#' character is specified between '%' and 'o' or 'x', then a leading 0 or 0x is printed respectively. If you specify 'l' (letter el) between % and one of the integer format characters, then the argument is taken to be long, instead of int. (you must include the library liblng11.a to use this)

int sprintf(char *buf, char *fmt) prints a formatted text into buf according to the format specifiers in fmt. The format specifiers are the same as in printf().

In addition, to ease porting programs from other environments, stdout and stderr are defined as 0, and FILE is typedefed as void, and fprintf(FILE *, char *fmt, ...) is the same as printf(char *fmt, ..).

10 Shared Library

What is it good for?

Look at the following example:

```
void lcdinit(void);
void
main(void)
{     lcdinit();
      printf("   10/3= %f", 10.0 / 3.0);
}
```

This program contains about 100 bytes. The linker will add code for the printf, the floating point arithmetic, and the driver for a display. The final object is about 7000 bytes long. A shared library contains the most common used functions. These are stored in a specific place in the target EEPROM. The application program can use the functions from the shared library. Thus the program stays small. You do not win in memory space, but shared libraries reduce the download time and reduce the turn around time during debugging.

Creating a shared library

The file **`../libshare/libshare.txt`** contains a list of functions to be included in the shared lib. This file together with the makefile creates a shared library for the Controlboy F1 target. The shared lib will be installed in EEPROM from 0xD000 to 0xFC00 using RAM from 0x200 to 0x220. It contains arithmetic functions for long types and float types, a printf, and a driver for a LCD display. The makefile creates the file **`LIBSHARE.S19`**. Download this file into the target memory. The makefile also creates **`LIBSHARE.A`** and installs the file in **`../LIB`**. The application has to add **`-lshare`** to the linker options to use the shared library.

There are three ways, to use data and bss sections in a shared lib:
- The shared library does not use these sections.
- The shared library only uses the bss section, and this section is not initialized to zero. This is the case in the example above.
- The shared library uses these sections. The application must use a modified crt11.s to initialize the sections.

11 Make

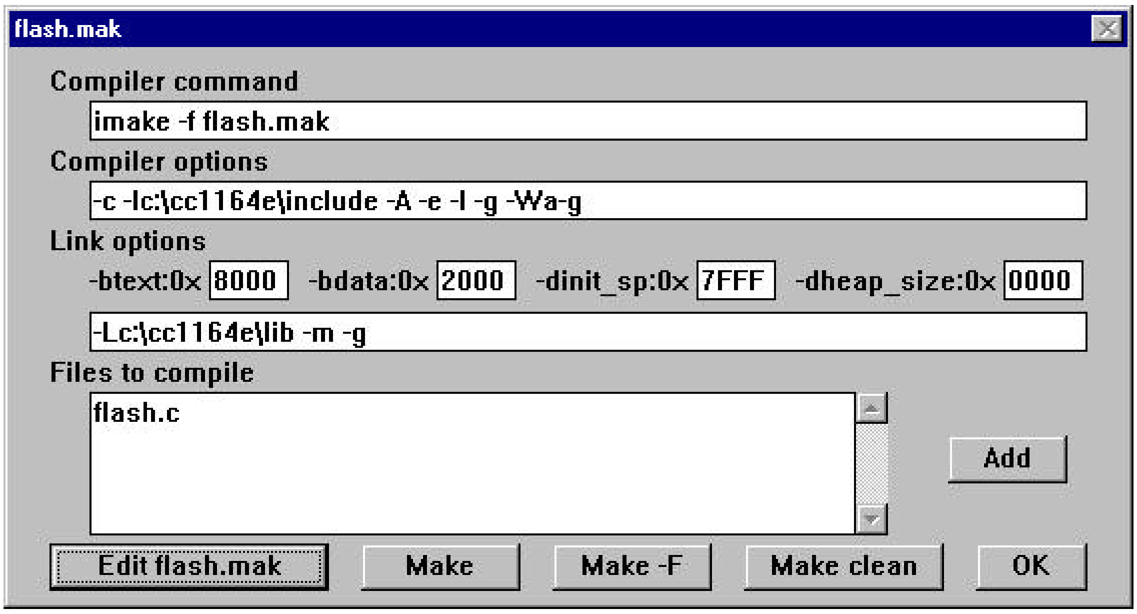
A typical program consists of object files from multiple source files. It is tedious and error-prone to manually compile and link the files together, and especially to remember which files were changed. Also, if a header file changes, then you will have to recompile all source files that include the header file. By using imake and makefile, you let imake handles all this details for you. Imake is a make utility for managing dependencies between a set of files. You create a description file that describes the dependencies between the set of files in a program, and invoke imake to compile files that have changed or compile files that include a header file that has changed since the last time the program was built.

Having a source file **`flash.c`**, CC11 automatically creates a makefile **`flash.mak`** for your. You may change parameters for the compiler and for the linker in a dialog. You may also edit directly the makefile. This chapter is only for those who want to write their own makefiles.

Imake reads an input file containing a listing of dependencies between files and associated rules to maintain the dependencies. The format is generally a target file name, followed by a list of files that it is dependent upon, followed by a set of commands to be used to recreate the target from the dependents. Each dependent is in its own right a target, and so the maintenance of each dependent is performed recursively, before attempting to maintain the current target. If after processing its all of its dependencies, a target file is found either to be missing, or to be older than any of its dependency files, imake uses the supplied commands or an implicit rule to rebuild it. The input file defaults to “makefile,” but you may override it with a command line option -f <filename>. If no target is specified on the command line, imake uses the first target defined in makefile.

Examples of Using Imake

Having a source file **`flash.c`**, CC11 automatically creates a makefile **`flash.mak`** for your. Click on OPTIONS, FLASH.MAK, to see and change all significant parameters.



flash.mak

Compiler command

imake -f flash.mak

Compiler options

-c -Ic:\cc1164e\include -A -e -l -g -Wa-g

Link options

-btext:0x 8000 -bdata:0x 2000 -dinit_sp:0x 7FFF -dheap_size:0x 0000

-Lc:\cc1164e\lib -m -g

Files to compile

flash.c

EDIT FLASH.MAK	See and to change the makefile FLASH.MAK.
MAKE	Start Imake to rebuild the executable.
MAKE -F	Rebuild all, even when no update is needed.
MAKE CLEAN	Removes objects and intermediate files.

You may use the file FLASH.MAK as a good start to create your own makefile.

```
CFLAGS= -c -Ic:\cc1164e\include -A -e -l -g -Wa-g
LFLAGS= -Lc:\cc1164e\lib -m -g -btext:0x8000 -bdata:0x2000 -
dinit_sp:0x7FFF -dheap_size:0x0000
TARGET= flash
SRCFILES= flash.c
OBJFILES= flash.o
.SUFFIXES:  .c .o .s .s19
.RESPONSE:
      icc11w
$(TARGET).s19: $(OBJFILES)
      icc11w -o $(TARGET) $(LFLAGS) $(OBJFILES)
.c.o:
        icc11w $(CFLAGS)  $<
.s.o:
        icc11w $(CFLAGS)  $<
```

Using the description file ‘makefile’.

If a target has no makefile entry, or if its entry has no rule, imake attempts to derive a rule by each of the following methods:

- Implicit rules, read in from a user-supplied makefile.
- Standard implicit rules typically read in from the file default.mk.
- The rule from the .DEFAULT: entry target, if there is such an entry in the makefile.

If there is no makefile entry for a target, and no rule can be derived for building it, and if no file by that name is present, imake issues an error message and stops.

When imake first starts, it reads the environment setting of MAKEFLAGS and scans all present options. Then it reads the command line for a list of options, after which it reads in a default makefile that typically contains predefined macro definitions and target entries for implicit rules. If present, imake uses the file default.mk in the current directory. Otherwise it looks for this file along the search path. Finally, imake reads in all macro definitions from the command line. These override macro definitions in the makefile.

The makefile may contain a mixture of comment lines, macro definitions, include lines, and target lines. Lines may be continued across input lines by escaping the NEWLINE with a backslash ‘\’. A comment line is any line whose first non-space character is a ‘#’. The comment ends at the next unescaped NEWLINE. An include line is used to include the text of another makefile. The first seven letters of the line is the word “include” followed by a space. The string that follows is taken as a filename (without double quotes) to include at this line.

Macros

A macro definition line has the form of ‘WORD=text...’. The word to the left of the equal sign (without surrounding white space) is the macro name. Text to the right is the value of the macro. Leading white space between the = and the first word of the value is ignored. A word break following the = is implied. Trailing white space (up to but not including a comment character) is included in the value. Macros are referenced with a $. The following character, or the parenthesized ( ) or bracketed { } string, is interpreted as a macro reference. imake expands the reference (including the $) by replacing it with the macro’s value. If a macro contains another macro, the interior one is expanded first. Note that this may lead to infinite expansion, if a macro references itself.

Predefined Macros

The MAKE macro is special. It has the value “imake” by default, and temporarily overrides the -n option (which means no execution mode: i.e. commands are printed but not executed) for any line in which it is referred to. This allows nested invocations of imake written as:

```
        $(MAKE) ...
```

to run recursively, with the -n flag in effect for all commands but imake.

There are several dynamically maintained macros that are useful as abbreviations within rules. They are shown here as references; it is best not to define them explicitly.

• $* refers to the basename of the current target, derived as if selected for use with an implicit rule.
• $< refers to the name of a dependency file, derived as if selected for use with an implicit rule.
• $@ refers to the name of the current target.

Because imake assigns $< and $* as it would for implicit rules (according to the suffixes list and the directory contents), they may be unreliable when used within explicit target entries.

A line of the form ‘WORD += text...’ is used to append the given text to the end of a macro. The += must be surrounded by white space.

Target Rules

A target entry in the makefile has the following format:

```
        target: dependency...
                rule
```

The first line contains the name of a target, or a list of target names separated by white space. This may be followed by a dependency, or a dependency list that imake checks in order. Subsequent lines in the target entry begin with a space or TAB, and contain shell commands. These commands comprise a rule for building the target. If a target is named in more than one colon-terminated target entry, the dependencies and rules are added to form the target’s complete dependency list and rule list. To rebuild a target, imake expands macros, strips off initial TABs, and passes each command line to a shell for execution. The first line that does not begin with a space, TAB or # begins another target or macro definition. Macros are expanded during input, for target lines. All other lines have macro expansion delayed until absolutely required.

Special Targets

When used in a makefile, the following target names perform special-func-tions. Many of them act as commands to imake.

.DEFAULT:	The rule for this target is used to process a target when there is no other entry for it, and no rule for building it. imake ignores any dependencies for this target.
.DONE:	imake processes this target and its dependencies after all other targets are built.
.IGNORE:	imake ignores non-zero error codes returned from commands.
.INIT:	This target and its dependencies are built before any other targets are pro-cessed.
.SILENT:	imake does not echo commands before executing them.
.SUFFIXES:	This denotes the suffixes list for selecting implicit rules.
.RESPONSE:	indicates that the following command can take a response file (i.e. @-file) if the command gets too long. For example, `.RESPONSE:` / `ilink` says that if a command line for ilink gets too long, then imake will put the command line into a temporary file and use the @-file option with these commands.

Rules

When processing rules, the first non-space character may imply special handling. Lines beginning with the following special characters are handled as follows:

-	imake ignores any nonzero error code. Normally, imake terminates when a command returns a nonzero status, unless the -i option or the .IGNORE target is used.
@	imake does not print the command line before executing it. Normally, each line is displayed before being executed, unless the -s option or the .SILENT target is used.

Implicit Rules

A target file name is made of a basename and a suffix. The suffix may be null. When a target has no explicit target entry, imake looks for an implicit target made of an element from the suffixes list concatenated with the suffix of the target. If such an implicit target exists, a dependency file name consisting of the basename and the suffix from the suffix list is recursively made. If successful, the implicit rule is invoked to build the target.
An implicit rule is a target of the form:

```
.DsTs:
        rule
```

Ts is the suffix of the target, .Ds is the suffix of the dependency file, and ‘rule’ is the implicit rule for building such a target from such a dependency file.

The Suffix List

The suffix list is given as the list of dependencies for the .SUFFIXES: special-func-tion target. The default list is contained in the SUFFIXES macro. You can define additional .SUFFIXES: targets; a SUFFIXES target with no dependencies clears the list of suffixes. Order is significant within the list; imake selects a rule that cor-responds to the target’s suffix and the first dependency-file suffix found in the list. To place suffixes at the head of the list, clear the list and replace it with the new suf-fixes, followed by the default list:

```
.SUFFIXES:
.SUFFIXES: suffixes $(SUFFIXES)
```

12 Command Line Tools

12.1 ICC11W Compiler Driver

Format

```
icc11w [options] file1 file2...
```

You usually only interact with the compiler driver and do not need to use the other tools individually when you are compiling. The compiler driver takes your input files and processes them according to your specified options, the default options, and the input file types. Some options are passed to the compiler passes directly, such as the -D switch to define macro names. In any case, you can pass any option you want to a particular compiler pass by using the -W option. If any of the passes returns a failure code, the compiler driver aborts with the sub-program failure code. You may request the compilation process to stop after a particular pass. For example, -S means compile to assembly only. Unknown options and file types are passed directly to the linker.

-A	Warn about function declarations without prototypes, and other non-conform-ancewith strict ANSI C requirements.
-c	Produce object files only. Do not link.
-D<name>[<=def>]	Define a preprocessor macro name. If no definition is given, then the value 1 is implied.
-E	Preprocess the input C files only. Do not compile, assemble or link. The generated output has the same name as the input C file but with a .i extension.
-e	Turn on the preprocessor extension which accepts C++ style comments
-l	Emit interspersed C and assembly code for the debugger.
-l<f>	link in the library file lib<f>.a. For example, use -lfp11 to include the floating point capable printf function.
-L<dir>	Specify the directory in which to search for the library files, crt11.o and end11.o.
-o<file>	Name the output executable S record file. The executable has the default extension .s19
-R	Do not link in startup file crt11.o and end11.o.
-s	Be silent. By default, if you specify more than one input file, the driver prints out each file name as it is processed.
-S	Produce assembly files only. Do not assemble or link.
-U<name>	Undefine a preprocessor macro name.
-v	Be verbose. Print out the command lines used to invoke the other passes and print out version information. If you specify -v more than once, then it prints the commands but does not actually execute them.
-w	Suppress warning diagnostics such as unreferenced variables.
-W<pass><arg>	Pass an argument to a particular compiler pass. <pass> must be p, f, a, or l; referring to the preprocessor (icpp), the parser and code generator (iccom11), the assembler (ias6811), and the linker (ilink) respectively. <arg> is passed directly to the compiler so you will need to specify the switch character '-' if it is a switch option. For example: icc11 -Wl-m foo.c passes the -m switch to the linker.

12.2 ICCPW Preprocessor

Format

icppw [options] <input file> [<output file>]

The preprocessor reads the input file, processes the macro preprocessor directives in the file, and writes the output. The input file usually has a .c extension and the output file has an .i exten-sion. The following are the valid options. If no output file is specified, then the preprocessed text is written to standard output.

-11	Assume an HC11 target. Use the ICC11 specific environment variables and files. If specified, this must be the first argument to icpp. The default is to use ICC11 environment variables.
-D<name>[<=def>]	Define a preprocessor macro name. If no definition is given, then the value 1 is implied.
-E	Ignore errors and return success status except when files cannot be written
-e	Turn on the preprocessor extension which accepts C++ style comments.
-I<path>	Include path. The preprocessor uses include paths to search for system include files (ones that are enclosed in < and >). You may supply multiple -I options. In addition, the preprocessor searches the directory specified by the environment
-U<name>	Undefine a preprocessor macro name

12.3 ICCOM11W Compiler Parser and Code Generator

Format

iccom11w [options] <input file> [<output file>]

This is the main piece of the compiler system. It takes a preprocessed C input file and generates an assembly output file. The compiler accepts ANSI C language and not the older K & R style C. Typically, the input file has a .i extension and the output has a .s extension. If you do not specify an output file, then the output is written to standard output. Note that if a C program does not have any preprocessor directive, you may pass it to iccom11 directly.

Les options acceptées sont:

- A	Warn about function declarations without prototypes, and other non-conformance with strict ANSI C requirements.
-data:<name>	Use <name> instead of “data” for the name of the data section. Note: data not put in the “data” data section will not get intialized by the startup code.
-e<number>	Set the error limit. The compiler aborts when the number of errors reaches the error limit. The default is 20.
-l	Emit interspersed C and assembly code for the debugger.
-text:<name>	Use <name> instead of “text” for the name of the text section
-w	Suppress warning diagnostics such as unreferenced variables.

12.4 IAS6811W Assembler

Format

ias6811w [options]<input file>

The assembler processes an assembly file as input and produces a relocatable object file as output.

-l	Generate a listing file (.lis)
-o<file>	Specify the name of the output file. The default is the base name of the input with a .o extension.

12.5 ILINKW The Linker

Format

ilinkw [options] <file1><file2>...

Ilink combines object files together to form an executable image. It automatically includes the start-up file crt11.o, end11.o and the library file libc11.a. The linker searches library files last after the object files are processed, so you may place a library anywhere on the link command line.

-11	Assume an HC11 target. Use the ICC11 specific files. If specified, this must be the first argument to ilink. The default is to use ICC11 environment variables.
-btext:<start address>[.end address] [:<start address>[.<end address>]]*	Set the address ranges of the text section. Default is 0 to 0xFFFF.
-bdata:<start address>[.end address] [:<start address>[.<end address>]]*	Set the address ranges of the data section. Default is immediately after the text section to 0xFFFF.
-b<section name>:<start address> [.end address] [:<start address>[.<end address>]]*	Set the address ranges of the named section. Default is immediately after the last section encountered.
-d<symbol>:<value>	Define a linker symbol which can be used to satisfy a unresolved reference or as a value for defining a section (-b flag) address
-g	Generate a file for the debugger
-l<f>	Link in the library file lib<f>.a.
-L<dir>	Specify the directory in which to search for the library files, including crt11.o and end.o.
-m	Create a map file with a name based on the output file but with a .mp extension and create a .lst file from concatenated .lis files with final addresses.
-R	Do not link in startup file crt11.o and end11.o.
-o<file>	Specify the name of the output file, which will automatically be given a .s19 extension. The default name is the base name of the first input file name.
-s<old>:<new>	Treat the section named <new> as if it has the name <old>.
-u<startup file>	Specify a non-default startup file. The default is crt11.o
-w	Do not emit warning messages.

12.6 ILIBW Library Archiver

Format

```
ilibw [ options ] <library archive> <object file 1> <object file 2> ...
```

The library archiver creates and manipulates libraries of object modules. Note that the library file (e.g., libc.a) must appear before any object files, after the options.

-a	Add the object modules to the archive. Create the archive if it does not exist. If the object module already exists in the archive, it is replaced.
-t	Print the names of the object modules in the archive.
-x	Extract the named object modules from the archive. This overwrites any existing object files of the same name.
-d	Delete the object modules from the archive.

Examples:
To place a new version of putchar.o into the library:

```
ilib -a libc11.a putchar.o
```

To list the contents of a library:

```
ilib -t libc11.a
```

12.7 IMAKE Make Utility

Format

```
imake [-f filename] [ options ] [target ...] [macro=value ...]
```

The make utility is used to maintain, update, and regenerate groups of programs. If no makefile is specified with a -f option, make reads a file named ‘makefile’, if it exists. If no target is specified on the command line, make uses the first target defined in makefile. If there is no makefile entry for a target, and no rule can be derived for building it, and if no file by that name is present, make issues an error
message and stops.

-f<makefile>	Use the description file ‘makefile’. A - as the makefile argument denotes the standard input. The contents of ‘makefile’, when present, override the standard set of implicit rules and predefined macros. When more than one -f makefile argument pair appears, make uses the concatenation of those files, in order of appearance.
-d	Display the reasons why make chooses to rebuild a target; make displays any and all dependencies that are newer.
-F	Force all target updates. Build target and all dependencies even when no update is needed according to file time/date.
-i	Ignore error codes returned by commands. Equivalent to the special-function target .IGNORE:.
-k	Abandon building the current target as soon as an error code is returned during building. Continue with other targets.
-n	No execution mode. Print commands, but do not execute them. Even lines beginning with an @ are printed. However, if a command line contains a reference to the $(MAKE) macro, that line is always executed.
-r	Do not read in the default file (default.mk).
-s	Silent mode. Do not print command lines before executing them. Equivalent to the special-function target .SILENT:.
-S	Undo the effect of the -k option. With this switch, the -k option is undone, which means that any error code returned by a child process during building will halt make and display the error code.
-v	List the current version number of make.
macro=value	Macro definition. This definition remains fixed for the make invocation. It overrides any regular definition for the specified macro within the makefile itself.

BASIC11

BASIC11 Rev 10/99

Controlord
484, avenue des Guiols
F 83210 La Farlède
Tél. (33) 494 48 71 74
Fax (33) 494 33 41 47
controlord@controlord.fr
www.controlord.fr

1 Introduction

The compiler Basic11 allows you to write Basic programs on a P.C. host for MC68HC11 based target systems.

Basic (Beginner's All-purpose Symbolic Instruction Code) was invented 1964 by John Kemeny and Tom Kurz at the Dartmouth College. Basic was designed as a language, easy to learn and to be used in a very short time. The first Basic systems were interpreters that ran on computers long before the time of microprocessors and P.C. Basic had a second life as interpreter in the ROM part of microprocessors.

Basic is not normalized, even if there exists a 'ANSI Standard for minimum Basic'. Each implementation of Basic has its own properties due to the microprocessor used but also due to the limitations of the target.

You often find on small microprocessors a Basic which is limited to a minimum, like the byte as single data type, IF, FOR, GOTO and GOSUB.

Basic11 uses a dialect which is oriented to high level languages like Pascal or C. You will nevertheless still find the famous GOTO and GOSUB, but there are other language elements that replace these statements a little bit to basic.

Basic11 is not an interpreter but a real compiler that translates the source program into an object program. This object is ready to be loaded into a ROM, PROM, EPROM, or EEPROM. The program will run on the 68HC11 without any help of an interpreter or an operating system.

What is the difference between an interpreter and a compiler?

- An interpreter interprets a program. Thus the program is very slow. A compiled program is much faster.
- An interpreter needs some place in the RAM and the ROM section of the target. When it is in a real ROM, you have to buy the interpreter, else you have to buy the memory for the interpreter for each target. It is obviously limited in comfort and richness of the language.
- An interpreter runs directly on the target. You do not need a host computer to compile the program, just some kind of a terminal.
- Working on a P.C. as host allows you to handle and archive the source and the documentation of the program.

The compiler BASIC11 and the assembler AS11 run under DOS. The compiler produces as output an assembly program which will be translated by the assembler into an object file of Motorola S-records. The program WBASIC11 runs under Windows 3.1 or Windows 95 and allows you to edit the source programs, run the compiler et get all messages of the compiler and the assembler like error messages. The debugger allows you to download the program into the EEPROM of the target and to debug the program.

2 Example: A simple Program

Once you have established a connection between the P.C. and your target board, you may run your first program on it.

Click on FILE, OPEN, and select the program FLASH.BAS.

```
' Target Controlboy F1
ProgramPointer    $8000 ' modify accoring to target
DataPointer       $0002 ' modify accoring to target
StackPointer      $7FEF ' modify accoring to target

byte DDRG  at $1003
byte PORTG at $1002

int  i

DDRG.0=1                ' PG0 = output
ASM   cli               ; enable debugger
do                      ' for ever
      if PORTG.0=0 then PORTG.0=1 else PORTG.0=0
      if PORTG.1=1 then i=20000 else i=5000
      for i = i to 0 step -1  ' tempo
      next i
loop
```

This program flashes a LED at the output PG0.

You have to adapt the first three lines, as indicated in the preceding chapter. And obviously you have to change the program if you have a LED at another output pin of the 68HC11.

Click on COMPILE to compile the program. If the compiler does not show any errors during compilation, you may now download the program.

In the debugger window, the debugger has to show you a target in the STOP state. If the target is in the RUN state, just click on STOP to stop the program on the target. If the debugger has no connection with the target board, you must load the talker into the target as indicated in the preceding chapter.

Click on LOAD to download your program into the target memory. Click on GO to start your program in the target.

The LED must now flash regularly.

If you click on STOP, your program stops. You see in one window the Basic program and in another window the assembly program that the compiler has produced. The debugger displays on the left indent the program pointer of the target. Enter at the prompt of the debugger

```
d i,c
```

to display the contents of the variables i and c. Click on STEP to execute a single assembly instruction on the target. If you click on the left indent of a source window, you set a breakpoint in the target memory, or you remove an existing breakpoint.

3 The Basic11 Language

Basic11 is a language without format restrictions. There is one statement per line. A statement may start in the first column but this is in no way necessary.

3.1 Keywords

Keywords are reserved words that may not be used as names for variables.

```
AND ASM AT BYTE DATAPOINTER DO ELSE EXTERN END EXIT FOR FUNCTION GOSUB
GOTO IF INT INTEGER INTERRUPT LOOP MOD NEXT NOT OPTION OR PRINT
PROGRAMPOINTER REM RETURN STACKPOINTER STEP THEN TO UNTIL WHILE XOR
```

Keywords may be written in capital or in small letters. All of the three following lines are correct.

```
PROGRAMPOINTER    $E000
datapointer       $0002
StackPointer      $01FF
```

3.2 Comments

A comment is preceded by the key word REM for remark, an apostrophe or two slashes. When using the apostrophe or the two slashes, a comment may finish a line of a statement.

```
REM The program flash.bas
‘ This program does really nothing
A = 30            ‘ starting value
A = A + 1         // A now is 31
```

3.3 Compiler options

```
<prologue>      =     OPTION <string>
```

The compiler accepts several options on the DOS command line. You may also enter options using the option directive. This directive must stand at the beginning of the program. An option must be surrounded by quotation marks.

b	Undefined variables will be automatically declared as BYTE.
i	Undefined variables will be automatically declared as INTEGER.
L<répertoire>	Specifies the directory for library files. The default is /BASIC11. When the compiler does not find a source file, it will search the file in this directory.

Example:

```
Option       "b"
```

3.4 Pointer declarations

```
<prologue>      =     PROGRAMPOINTER <constant expression>
                |     DATAPOINTER <constant expression>
                |     STACKPOINTER <constant expression>
```

The pointer declarations are essential for the target program. These declarations must be at the beginning of the program like the option directive.

```
ProgramPointer    $E000
```

The compiler stocks the program and the constant data in the memory beginning at this address. This address should thus be the beginning of the ROM, PROM, EPROM, or EEPROM of the target address space.

```
DataPointer       $0002
```

The compiler reserves space for the variables in the memory beginning at this address. This address should thus be the beginning of the RAM. The RAM of the 68HC11 always starts at the address $0000. It is nevertheless advised to avoid the two first bytes, since wrong programs tend to write to this address.

```
StackPointer      $00E8
```

The stack pointer should be placed at the end of the RAM that can be used by the program, since the stack grows down from this address. You must provide enough space for the stack. The stack is used for function calls, local variables within functions, complicated expressions, and finally for interrupt handling.

A program without the declaration of the ProgramPointer is not considered to be a main program. The compiler will translate the program into an assembly program but will not generate an object program.

3.5 The #include directive and the file start.bas

This preprocessor directive must start in the leftmost column of a line.

```
#include <file name>
```

The directive includes another file in the source code. The compiler translates this file first and will then continue after this line. The included file may include another source file.

It is advised to include the file START.BAS in each Basic program. This file includes the pointer declarations. Thus you have to change these lines before using the file due to the target you are using. Following there is a small assembly program which will be executed before running your program. This program initializes all global variables to 0, which avoids lots of common problems. The file also includes the declarations for the ports of the 68HC11.

```
ProgramPointer    $E000 ' change this for your target
DataPointer       $0002 ' change this for your target
StackPointer      $01FF ' change this for your target

             sect   text
             cli                  ; enable debugger
             ldx    #_data_s
             bra    _crt2
_crt1        clr    0,x           ; clear data area
             inx
_crt2        cpx    #_data_e
             bne    _crt1

             sect   data
_data_s      equ *

byte SCONF          at $0420      ' Controlboy 2 /3
byte PORTB          at $0410      ' with X68C75
byte PORTBI         at $0430
byte PORTC          at $0408
byte PORTCI         at $0428

'byte PORTB         at $1004      ' Controlboy 1
'byte PORTC         at $1003      ' mode single-chip
'byte DDRC          at $1007

byte PORTA          at $1000
byte PIOC           at $1002
byte PORTCL         at $1005
byte PORTD          at $1008
byte DDRD           at $1009
byte PORTE          at $100A
byte TMSK2          at $1024
byte TFLG2          at $1025
byte PACTL          at $1026
byte PACNT          at $1027
byte BAUD           at $102B
byte SCCR1          at $102C
byte SCCR2          at $102D
byte SCSR           at $102E
byte SCDR           at $102F
byte ADCTL          at $1030
byte ADR            at $1031
byte OPTIONS        at $1039
```

3.6 Declaration of variables, names, data types

<declaration>	=	<type> <variable> [,<variable>]*
	|	<type> <variable>(<constant expression>)
<type>	=	BYTE | INTEGER | INT

A variable has a name of up to 31 characters. A name starts with a letter or an underscore (_), followed by letters, digits, and underscores.

```
abc
Abc
A1b2c3
s_S_123_S
```

All these names are legal. Note that abc and Abc are two different names for two different variables. Key words must not be used as names for variables.

Data types

A variable must be declared before you can use it.

A variable of type BYTE is stored in one byte of memory and treats unsigned values

BYTE	0 to 255

A variable of type INTEGER is stored in two bytes of memory and treats signed integer values

INTEGER	-32768 to 32767

Note, that the 68HC11 is a small 8 bit microprocessor. The CPU of the 68HC11 treats easily BYTE variables, while there are few machine instructions for 16 bit INTEGER data. Thus you should use BYTE variables wherever possible, and use INTEGER variables only for data that may not fit into a single byte.

The following lines declare the variables a, ab, Ab, and ab_b as BYTE variables and i, j, k, et k33 as INTEGER. All variables are placed into RAM memory.

```
byte  a
byte  ab, Ab, ab_b
int   i, j, k, k33
```

Arrays

Like simple variables, arrays must be declared before they can be used. An array keeps several data items. All arrays have one dimension.

```
byte abc(5)
```

This is a declaration of an array of five RAM bytes. The first item is addressed by abc(0) and the last one by abc(4).

Note, that when using the file START.BAS, all arrays like all simple variables are initialized to 0 after RESET. Else there contents is undefined.

Declarations in [[[E]E]P]ROM

```
<declaration>   =    <type> <variable>() = <constant expression>
                                       [,<constant expression>]*
                |    <type> <variable>() = <string>
<string>        =    "<asciistring>"
```

Until now, all declarations placed the variables into the RAM. You may also declare a variable which resides in the read only memory. Your program cannot write into such a variable, but these variables can be initialized to a value.

```
int def() = 1, 3, 7, 9
byte title() = "Controlboy"
```

The first line declares an array of 4 integers, the second line a byte character string. title(0) keeps the ASCII character ‘C’, title(1) ‘o’. After the last character ‘y’ there is a byte of 0 that terminates the string.

Declarations of input, output ports

```
<declaration>   =    <type> <variable> AT <constant expression>
```

After the RAM and ROM declarations we have to declare variables that have already a specific address. These are mainly the input and output ports.

```
byte PORTA at $1000
```

The item PORTA is located at the address $1000.

Declarations of external variables

To have access to a variable that is declared within an other program, this variable must be declared as EXTERN.

```
extern byte second
```

3.7 Assignments and expressions

```
<instruction>   =   <lexpression> = <expression>
<lexpression>   =   <variable> | <variable> ( <expression> )
<expression>    =   <primary>
                |   - <primary> | NOT <primary>
                |   <expression> * <primary> | <expression> / <primary>
                |   <expression> MOD <primary>
                |   <expression> + <primary> | <expression> - <primary>
                |   <expression> <compare> <expression>
                |   <expression> AND <primary>
                |   <expression> OR <primary> | <expression> XOR <primary>
<primary>       =   <variable>
                |   <variable> ( <expression> )
                |   <constant>
                |   ( <expression> )
<compare>       =   = | == | <> | != | > | >= | < | <=
```

You may assign a value to a variable by using the equal sign. You may assign a constant, the value of another variable or an expression of these.

```
byte a,b,c(10)
int  i, j(10)

a = 7                       the variable a is set to 7
i = -100                    the variable i is set to -100
b = a + 1                   b = 8
c(3) = b + a * 2            c(3) = 8 + 7 * 2 = 22
c(a) = ( b + a ) * 2        c(7) = ( 8 + 7 )* 2 = 30
b = c(3) / a                b = 22 / 7 = 3
```

Expressions consist of constants, variables and operators. Operators are executed according to their precedence. They are classified in four groups.

The first group consist of the brackets. They have the highest priority. Brackets allow you to change the priority within an expression.

The second group is the group of arithmetic operators. Multiplication and division have higher priority than addition and subtraction.

The relational operators form the third group. They are used in conditional statements. They compare two expressions. The result of such a comparison is a logical value. This value is non zero if the result is true, and zero if the result is wrong.

The logical operators in the forth group are applied to each bit of the operands. They are used in conditional statements but also for bit manipulations on variables and more frequently on input output ports.

```
PORTA = PORTA or $08        set bit 3 of port A to 1
PORTA = PORTA and not $08   set bit 3 of port A to 0
PORTA = PORTA xor $08       toggle bit 3 of port A
if PORTA and $08 then       examine bit 3 of port A
```

You may directly address a bit of a variable or of a port.

`PORTA.3 = 1`	set bit 3 of port A to 1
`PORTA.3 = 0`	set bit 3 of port A to 0
`if PORTA.3 = 1 then`	examine bit 3 of port A

Priority	Operator	Description
1 (highest)	**( )**	Brackets
2	**-**	Negation
	NOT	Logical complement
3	*****	Multiplication
	/	Division
	MOD	Remainder of a division
4	**+**	Addition
	-	Subtraction
5	**=**	Equal (also ==)
	<>	Not equal (also !=)
	>	Greater than
	>=	Greater or equal
	<	Less
	<=	Less or equal
6	**AND**	Logical and
7 (lowest)	**OR**	Logical or
	XOR	Logical exclusive or

Constants

```
<constant>      =    [0-9]+
                |    $[0-9|A-F|a-f]+
                |    0x[0-9|A-F|a-f]+
                |    %[0|1]+
                |    `<asciicharacter>`
```

Constants may be entered the following ways

`123`	decimal number
`$001f`	hexadecimal number
`0x001F`	hexadecimal number
`%0001100`	binary number
`` `y` ``	ASCII character.

3.8 FOR loop

```
<instruction>    =    FOR <variable> = <expression> TO <expression>
                                 [ STEP <constant expression> ]
                 |    NEXT [<variable>]
                 |    EXIT FOR
```

The FOR loop uses a byte or an integer counter. The loop defines a starting value, an end value and an increment. If no increment is specified, the default increment 1 is taken.

The loop is terminated by a NEXT statement. You may add for clarity the name of the loop counter to this statement.

The following loop initializes the elements of an array.

```
byte  a, b(10)
int   i, j
for a = 0 to 9
      b(a) = 0
next
```

The loop increment may be positive or negative. A loop may contain another loop.

```
for i = 2 to -30 step -2
      ...
      for j=i to i+4
            ...
      next j
      ...
next i
```

The EXIT FOR statement forces an immediate exit from a loop. The program continues after the NEXT statement.

```
for a = 0 to 9
      ...
      if b(a) = 3 then exit for
      ...
next a
```

The following loop is an endless loop. A program on an embedded system never stops. You will need at least one endless loop in your program.

```
for a = 0 to 1 step 0
      ...
next a
```

3.9 DO LOOP WHILE UNTIL

```
<instruction>    =    DO [ WHILE | UNTIL <expression> ]
                 |    EXIT DO
                 |    LOOP [ WHILE | UNTIL <expression> ]
```

The DO statement declares a loop without a counter. The loop ends with the LOOP statement.

The following loop never ends.

```
do
      ...
loop
```

The following loop uses a condition. If the condition is false when the program reaches the loop, the loop will not be executed and the program jumps over the loop and continues after the LOOP statement. When the conditions becomes false, the loop will terminate and the program continues after the LOOP statement.

```
do while PORTA.3 = 0
      ...
loop
```

The following loop also uses a condition, but the condition is inverted. The loop terminates when the condition becomes true.

```
do until PORTA.3 = 1
      ...
loop
```

The following uses a condition at the end of the loop. Thus the loop will be executed at least once.

```
do
      ...
loop until PORTA.3 = 1
```

The EXIT DO statement forces an immediate exit from a loop. The program continues after the LOOP statement.

3.10 IF

<instruction>	=	IF <expression> THEN <instruction> [ELSE <instruction>]
	|	IF <expression> THEN
	|	ELSE
	|	END IF

IF is a control-flow statement to express a decision. This statement uses often a comparison and executes the THEN part, if the comparison is true, and the ELSE part otherwise. The ELSE part is optional and often missing.

```
byte a,b,c
int j(5)
if a<5 then c=0 else c=1
```

If the variable a is less than 5, c is set to 0, else c is set to 1. If there are several statements to be executed following the decision, another syntax can be used. The ELSE part is still optional.

```
if a<5 then
      c = 0
      ...
else
      c = 1
      ...
end if
```

A comparison can be complex...

```
if j(a)+j(b) < a+b*c then ...
```

IF may combine several expressions by logical operators like AND and OR. The following two lines give the same result.

```
if a>1 and a <4 then ...
if a=2 or a=3 then ...
```

A comparison uses the following operators.

=	Equal (also ==)
<>	Not equal (also !=)
>	Greater than
>=	Greater or equal
<	Less
<=	Less or equal

3.11 GOTO, GOSUB

```
<instruction>   =     GOTO <label>
                |     GOSUB <label>
                |     RETURN [<expression>]
```

GOTO and GOSUB allow you to jump directly to another statement in the program.

```
if a<5 then c=0 else c=1
```

This statement may be replaced by the following programme using GOTO statements.

```
if a>=5 then goto here1
c = 0
goto here2
here1: c=1
here2:
```

GOSUB jumps into a subroutine. The subroutine must terminate with a RETURN statement. The program will then continue after the GOSUB statement. GOSUB cannot pass parameters to the subroutine, and cannot receive a result. The functions as explained in one of the following chapters should replace the GOSUB.

```
gosub here
...         ‘ the program continues here after RETURN

here: ...   ‘ subroutine
...
return      ‘ end of the subroutine
```

3.12 ASM

```
<instruction>   =     ASM <textstring>
```

The ASM directive includes a line of assembly code into the program. The syntax is just the keyword ASM followed by the assembly instruction. But the keyword is not even necessary. The compiler automatically recognizes assembly code within a Basic program and passes these lines untreated to the assembler.

The following two line in a Basic program give the same result.

```
asm   cli
cli
```

The assembly instruction CLI allows interrupts to the CPU.

An assembly instruction may access global variables and functions of the Basic program by their names. The parameters and the local variables of a function are placed on the stack. You must compile your Basic program and look into the assembly program generated by the compiler. You will find the stack addresses of these variables as comments after the declaration section of the function.

3.13 PRINT

<instruction>	=	PRINT <printelement> [,<printelement>]*
<printelement>	=	<expression>
	|	<string>

PRINT displays data on a LCD, on the RS232 serial line, or on any other peripheral.

The PRINT statement is followed by a comma separated list of items to display. You may display strings, variables, and expressions of variables. The variables are displayed as signed decimal numbers. A byte array will be displayed as a character string.

```
int i, j
i = 5
j = -3
print i," / ", j, " = ", i/j, " R ", i MOD 3
```

will display

```
5 / -3 = -1 R 2
```

Putchar

The PRINT statement is based on the putchar function. This function must be supplied by the application. If your program shall print the output of the PRINT statement on a LCD, the putchar function must display exactly one character on the display. You find an example for a LCD in the file LCD.BAS. You may include this file into your program. The main program has to call the function lcdinit() to initialize the display once and may then use the PRINT on the LCD.

Here is another implementation of the putchar function. The PRINT will write out the data on the RS232 line.

```
function putchar(x)
byte x2
for x2=0 to 1 step 0
      if SCSR AND $80 then exit for
next x2
SCDR = x
end function
```

3.14 Functions

<declaration>	=	FUNCTION <function> ([<variable> [,<variable>]*])
	|	END FUNCTION
<instruction>	=	RETURN [<expression>]
<primary>	=	<function> ([<expression> [,<expression>]*])

Functions or procedures give you the best tools to structure your program. A function is a subroutine. The main program calls this function and passes parameters to it. The function executes the subroutine using these parameters and may return a result. The calling program receives the result and continues execution with it. There are functions with no parameters. And there are functions that do not return a result. These are sometimes called procedures.

Following is an example of a function to calculate the maximum of two numbers.

```
function max(a, b)
if a>b then return a else return b
end function
```

The declaration of a function starts with the keyword FUNCTION followed by the name of the function and a list of formal parameters. You may use the keyword SUB instead of FUNCTION. The statement RETURN allows you to return to the calling program, and to pass back the result of the function. The statement END FUNCTION finishes the declaration of the function. This statement automatically forces a RETURN.

```
int i, j, k
i = 8
j = 17
k = max(i, j)                  ‘ k will be 17
```

The calling program calls the function by its name followed by the list of actual parameters, which are passed to the function. In this case, the function max works on the parameter a, which is a copy of the variable i, thus 8, and on the parameter b, a copy of j, thus 17. Max will hopefully return 17 as result.

A function may call another function. Expressions may combine functions and other elements.

```
k = 100 - max(max(7+i,j-3),max(x,max(i,k)))
```

Parameters

The parameters and the result of a function are of type INTEGER. If the program passes a BYTE variable as parameter, the argument will automatically converted to INTEGER. Parameters may be simple variables, or elements of an array. You cannot pass a whole array as parameter. Parameters are passed by value. The function works on copies of the variables that are passed as parameters and thus does not change them.

```
int i, j
i = -6
j = abs(i)
...

function abs(x)         ‘ x is a copy of i
if x < 0 then x = -x    ‘ changes x, does not change i
return x
end function
```

Recursive functions

A function may call itself. All Basic11 functions are recursive by nature. There are few applications that use this feature. The following example demonstrates the usage by a function that prints a number using the putchar function. If the variable (e.g. 1234) is higher than 9, the function treats the upper portion (1234 / 10 = 123) first, and then prints the leading digit (1234 MOD 10 = 4). It is difficult to find an easier solution without recursion. But it is probably also difficult to find a more confusing solution.

```
function printnum(x)
if x <0 then
      putchar(`-`)
      x = -x
end if
if x > 9 then printnum(x/10)
putchar (x MOD 10 + `0`)
end function
```

Extern functions

To access a function already declared in another program, the function must be declared extern.

```
extern function lcdinit()
...
lcdinit()
```

3.15 Scope of global and local variables

Variables declared at the beginning of a Basic program are called global. They can be accessed from everywhere in the program.

You may declare variables at the beginning of a function. These are called the local variables. They are initialized to zero at each call of the function. The parameters and the local variables of a function are placed on the stack. They can thus only be accessed during the lifetime of the subroutine. The place on the stack will be freed when the function ends.

A Basic source program consists of the following parts.

```
Declaration of the global variables
        can be accessed from the main program and from all functions
...
Statements of the main program
...


FUNCTION func1()
Declaration of local variables
        can only be accessed from function func1
...
Statements of the function func1
...
END FUNCTION


FUNCTION func2()
Declaration of local variables
        can only be accessed from function func2
...
Statements of the function func2
...
END FUNCTION
```

The following example shows the usage of local variables

```
int i,j                 ‘ global data
i = abs(j)
...
function abs(x)         ‘ x on the stack
int r                   ‘ r on the stack
if x < 0 then r = -x else r =x
return r                ‘ place on the stack is freed
end function            ‘ x and r disappeared
```

3.16 Example: keypad input, LCD output

This program was written for a Controlboy 3 target board with a one line 16 character liquid crystal display and a 12 keys keypad.

The function putchar which writes out a character to the LCD is placed in the file lcd.bas. This file is included at the end of the program. The function keyget which examines the keypad is also in this file. If you press a key, the function returns the ASCII code of the key.

The main program initializes the ports, it will use, and calls the function lcdinit which initializes the display. Afterwards it writes a message to the display and enters into an endless loop. When you press the T1 button on the target, the relays toggle in a pseudo random way. The function tempo reduces the speed of this action. This function also reads the keypad. If you press a key, the program prints the pressed key on the display.

```
' program to test LCD and keypad on Controlboy 3

#include "start.bas"

      byte count
      DDRD  = 0                ' PD5=PD4=PD3=PD2=input
      SCONF = 0x4C             ' B,C = outputs
      lcdinit()
      print " Controlboy 3  "

      for count=0 to 1 step 0        ' for ever
      if (PORTD and 0x20)<>0 then
            PORTB = 0                ' if T1 = 1
      else
            PORTB = PORTB + 37       ' if T1 = 0
      end if
      tempo(5)
      next count

function tempo(cnt)
      int i, k
      for cnt=cnt to 0 step -1
            for i=0 to 100
                  k = keyget()
                  if k <> 0 then putchar(k)
            next
      next
      end function

#include "lcd.bas"
```

3.17 Interrupt functions

```
<declaration>    =  INTERRUPT FUNCTION <function> AT <constant expression>
```

The declaration of an interrupt function allows you to treat interrupts of a microprocessor.

The following example uses an interrupt function for the real time clock of the 68HC11. The main program enables the clock, enables interrupts and finishes in an endless loop. Rtiint is the interrupt function. The program has to load the address of this function as interrupt vector. The interrupt vector of the real time clock is at the address $FFF0 as indicated in the declaration of the function. The real time clock calls the function regularly 244 times per second. An interrupt functions has neither parameters nor a result. The program counts the elapsed time in two variables minute and second and displays the time each second.

```
' demonstration of an interrupt function

#include "start.bas"

      byte w, s, second, tictac
      int   minute

      lcdinit()                 ' enable LCD
      PACTL = PACTL AND $FC     ' select the speed
      TMSK2 = TMSK2 OR  $40     ' start the timer
      cli                       ' enable ints
      for w=0 to 1 step 0       ' forever
      if second <> s then       ' second changed ...
            if second >= 60 then
                  second = 0
                  minute = minute + 1
            end if
            print "   ", minute,":",second
            s = second
      end if
      next w

interrupt function rtiint at $FFF0
      tictac = tictac + 1
      if tictac >= 244 then     ' 8Mhz: 244, 4,9Mhz: 150
            tictac = 0
            second = second + 1
      end if
      TFLG2 = TFLG2 or $040     ' enable ints again
      end function

#include "lcd.bas"
```

3.18 Syntax

```
statement        =   <prologue>
                 |   <declaration>
                 |   [ <label> : ] <instruction>

<prologue>       =   OPTION <string>
                 |   PROGRAMPOINTER <constant expression>
                 |   DATAPOINTER <constant expression>
                 |   STACKPOINTER <constant expression>

<declaration>    =   <type> <variable> [ ,<variable> ]*
                 |   <type> <variable>(<constant expression>)
                 |   <type> <variable>() = <constant expression>
                                           [,<constant expression>]*
                 |   <type> <variable>() = <string>
                 |   <type> <variable> AT <constant expression>
                 |   FUNCTION <function> ( [<variable> [,<variable>]* ] )
                 |   INTERRUPT FUNCTION <function> AT <constant expression>
                 |   END FUNCTION

<type>           =   BYTE | INTEGER | INT

<instruction>    =   <lexpression> = <expression>
                 |   FOR <variable> = <expression> TO <expression>
                                          [ STEP <constant expression> ]
                 |   NEXT [<variable>]
                 |   EXIT FOR
                 |   DO [ WHILE | UNTIL <expression> ]
                 |   EXIT DO
                 |   LOOP [ WHILE | UNTIL <expression> ]
                 |   IF <expression> THEN <instruction> [ ELSE <instruction> ]
                 |   IF <expression> THEN
                 |   ELSE
                 |   END IF
                 |   GOTO <label>
                 |   GOSUB <label>
                 |   RETURN [<expression>]
                 |   PRINT <printelement> [,<printelement>]*
                 |   ASM <textstring>
                 |   REM <textstring>

<lexpression>    =   <variable> | <variable> ( <expression> )

<expression>     =   <primary>
                 |   - <primary> | NOT <primary>
                 |   <expression> * <primary> | <expression> / <primary>
                 |   <expression> MOD <primary>
                 |   <expression> + <primary> | <expression> - <primary>
                 |   <expression> <compare> <expression>
                 |   <expression> AND <primary>
                 |   <expression> OR <primary> | <expression> XOR <primary>

<primary>        =   <variable>
                 |   <variable> ( <expression> )
                 |   <function> ( [<expression> [ ,<expression>]* ] )
                 |   <constant>
                 |   ( <expression> )

<compare>        =   = | == | <> | != | > | >= | < | <=

<constant>       =   [0-9]+
                 |   $[0-9|A-F|a-f]+
                  |   0x[0-9|A-F|a-f]+
                  |   %[0|1]+
                  |   `<asciicharacter>`

<printelement>    =   <expression>
                  |   <string>

<string>          =   "<asciistring>"
<variable>        =   <name>
<label>           =   <name>
<function>        =   <name>
<name>            =   [A-Z|a-z|_] [A-Z|a-z|0-9|_]*
```

Assembler

The assembler handles several files

<filename>**.a11**	Assembly source file
< filename>**.bak**	Backup of the source file
< filename>**.s19**	Object file. Motorola S-records
< filename>**.lst**	Assembly listing file

The assembly listing contains the original source line, the machine address and the generated data by the assembler. The object file contains the compiled program in Motorola S-records format followed by information for the debugger.

The assembler displays the segments and the first and the last used address for each segment.

The assembler accepts options on the command line. You may enter these options also by the option directive.

-r	Read only. Do not change the source file.
-n	No debug information in the object file
-b	Do not transform branches into jumps.
-j	Treat jumps like branches
-g	Symbols are local by default
-l	Line numbers in the listing
-c	Cycle count of the 68HC11 in the listing
-2..-9	Number of passes, default is 3.

Without the -b option, the assembler automatically replaces branches with an address out of range by jumps. The assembler executes several passes to replace jumps successively by branches.

Without the -g option, all symbols are global. With this option, symbols are valid within the file where they are declared. They may be declared global by the **.globl** directive. A label declared with two colons is also global.

Syntax

Following directives are instructions that do not correspond to machine instruction. These instruction are directives for the assembler.

	option	[r|n|b|j|g|l|c|2..9]*	sets the option
	org|.org	<value>	sets the working address
	fcb|.byte	<value>[,<value>]*	generates bytes in memory
	fcb|.ascii	'<ASCII string>'	generates bytes in memory
	fdb|.word	<value>[,<value>]*	generates 16 bit words in memory
	rmb|.blkb	<value>	reserves bytes in memory
<name>	equ|=	<value>	declares a symbol
	sect|.area	data|text|none	change the section
	.globl	<nom>[,<nom>]*	declare symbol global
	end		ignored

The sect instruction allows you to keep several working addresses e.g. for the RAM (data) and for the EEPROM (text).

A symbol has up to 15 characters. Allowed characters are

a..z A..Z 0..9 _ . $

The first character must not be a digit nor $.

Arithmetical expressions include symbols, constants and the *, which presents the current program address. They can be combined by

+	Addition
-	Subtraction

Expressions are evaluated from the left to the right in 16 bit arithmetic.

Constants may start with one of the following characters, or else they are interpreted as decimal constants.

'	ASCII characters
$	hexadecimal constant
%	binary constant

An assembler source line is composed of an optional label followed by spaces, the opcode followed by spaces and the operands separated by spaces. Finally there can be comments. A source line that starts with an * is treated as a comment line.

CPU

This chapter describes the central processing unit (CPU) of the 68HC11. The CPU executes the machine instructions of the program and controls thus the 68HC11. Other elements of the 68HC11 like memory and I/O ports will be discussed in the following chapters.

Programming Model

The basic data element of the 68HC11 is the byte. It has 8 bits. The least significant bit is called the bit 0, the most significant bit 7.

Beside the byte the 68HC11 also knows the word of 16 bits. The least significant bit is also the bit 0, and the most significant the bit 15. When the CPU stores a word in the memory it stores the most significant byte before the least significant byte.

The 68HC11 uses three different arithmetical presentations:

Unsigned Numbers. A byte may contain values from 0 to 255 ($FF), a word from 0 to 65535 ($FFFF).

Twos complement signed numbers. A byte may contain values from -128 ($80) to +127 ($7F), a word from -32768 ($8000) to +32767 ($7FFF). The most significant bit keeps the sign. When it is 1, the number is negative. A byte or a word with all bits at 0 represents a number 0. This holds true for all the three presentations. Note that there are 128 negative but only 127 positive numbers and -128 ($80) has no positive equivalent. The same holds true for the word -32768 ($8000).

Binary Coded Decimal (BCD). This is a rarely used arithmetical presentation. A byte contains two decimal digits: The bits 7,6,5,4 keep the most significant, and the bits 3,2,1,0 the least significant digit. A byte may contain values from 0 ($00) to 99 ($99). One machine instruction (DAA) supports BCD numbers.

Nearly all arithmetical instructions can be used for unsigned and signed numbers. But the interpretation of the condition codes is completely different. Only some of the arithmetical instructions can be used for BCD numbers and the program must correct the result after each operation by the DAA instruction.

Addresses are 16 bits long and they address bytes. The address space of the 68HC11 CPU thus spans over 2^{16} = 65536 bytes. Addresses are usually expressed in hexadecimal ($0000 to $FFFF). The address space includes all kinds of memory like RAM, ROM, and EEPROM, but also the registers for the input and output ports. There are no special machine instructions for input and output. All instructions that read or write memory can also be used for peripheral devices.

Registers

```
7      ACCA     0 7      ACCB     0  ACCA,ACCB

15                                0  D


15                                0  IX


15                                0  IY


15                                0  SP


15                                0  PC



                S X H I N Z V C  CCR

                                 Carry
                                 Overflow
                                 Zero
                                 Negativ
                                 Interrupt mask
                                 Half Carry
                                 X Interrupt mask
                                 Stop disable
```

ACCA and **ACCB** are the two 8-bit accumulators of the CPU. They are also called registers A and B. All major operations are done within these registers. These two registers combined form the 16-bit register **D.**

```
ldaa    #$12        load $12 into register A
ldab    #$34        load $34 into register B
ldd     #$5678      load $56 into register A and $78 into B
```

IX and **IY** are 16 bit index registers. They are used to address the memory. There are few machine instructions that change these registers, but nearly all instructions that access the memory, can use them as address base. Usage of the register IX may make programs shorter and faster for example when there are lots of operations in the I/O registers. Instead of

```
ldaa    $1033       Load from the I/O register at address $1033
stab    $1034       Store into the I/O register $1034
```

you may write also, using IX

```
ldx     #$1000      Let IX point to the I/O registers
ldaa    $33,X       Load from address $1033 (= $33 + $1000)
stab    $34,X       e.t.c.
```

The 16 bit stack pointer **SP** points to the first free byte on the stack.

The 16 bit program counter **PC** points to the machine instruction that will be executed next.

Condition Codes

The Condition Code Register **CCR** contains eight flag bits. Some of them are used and set by arithmetical operations. Once set they may be used by a following arithmetical operation or by a conditional branch instruction. For example

decb	decrement B
	sets the CCR bits N, Z and V according to the contents of B
beq	branch if Z=1, i.e. if register B is 0

The following discusses the CCR flags in detail. For the beginning only the flags Z and I are necessary.

The **Z Zero** flag indicates that the result of an operation is zero. A compare instruction (CMP) subtracts the second operand from the first (without storing the result). The Z bit thus indicates that the operands are equal.

The **N Negative** flag is set by the most significant bit of an operation. The bit indicates for signed numbers that the result of the operation is negative. This flag is meaningless for unsigned or BCD numbers.

The **V Overflow** flag is set when an operation causes an overflow and the result is not valid. This flag is only meaningful for operations on signed numbers.

The **C Carry** flag is used for arithmetical and shift operations and indicates an overflow. This bit allows operations on operands longer than bytes or 16 bit words. The carry flag, set by an operation on a part of the operand, will be used afterwards by the same operation on another part of the operand.

The **H Half Carry** flag is used for BCD arithmetic only. The ADD, ADC, ABA instructions set this bit for a following DAA instruction.

The **I Interrupt Mask** flag disables all maskable interrupts. When this bit is set, the CPU cannot be interrupted by such an interrupt. The CPU must reset this bit to allow interrupts. When an interrupt arrives, the CPU automatically sets the I flag to 1 to avoid recursive interrupts. The last instruction of an interrupt service routine (RTI) restores the flag to 0.

The **X Interrupt Mask** flag disables interrupts from the XIRQ pin.

The **S Stop Disable** flag disables the STOP instruction. When this flag is set, the CPU ignores the STOP instruction.

Addressing Modes

The 68HC11 has five different modes to address the memory. None of these modes is available for all instructions. You have to look into the machine instruction details to find the available addressing modes.

Immediate. This is the easiest addressing mode. The operand is a constant in the program and is placed behind the operating code. This addressing mode is indicated by a # in the assembly language. The constant may be between $0 and $FF for byte operations and between $0 and $FFFF for word operations.

```
86  7F        ldaa   #127         load 127 into ACCA
8B 10         adda   #$10         add $10 to ACCA
CE 10 00      ldx    #$1000       set IX to the base of the I/O
```

Direct 16 bit. The two bytes following the opcode contain the address of the operand. The operand may be anywhere in the address space of the 68HC11. The operand itself may be a byte or a word according to the operation.

```
B6 10 33      ldaa   $1033        load from address $1033 a byte
FF 11 00      stx    $1100        store the word in IX at $1100:1101
BD F8 77      jsr    $F877        jump to the subroutine at $F877
```

Direct 8 bit. This addressing mode is like the previous one, but the address is stored in one byte. This mode can only address the RAM from $00 to $FF. There is no difference between these addressing modes in the assembly source. The assembler will take the shorter and faster 8 bit mode whenever possible. Compare the following lines to the lines above.

```
96  33        ldaa   $0033        load from address $0033 a byte
DF 80         stx    $0080        store the word in IX at $0080:0081
9D B0         jsr    $00B0        jump to the subroutine at $00B0
```

PC-Relative. Only branch instructions use this addressing mode. The branch address is calculated by the address of the instruction following the branch instruction (the PC), incremented or decremented by a constant from -128 to +127 that is stored as the last byte of the instruction in the program. A branch can thus jump forward and backward but the distance is limited. In the assembly program you specify the absolute address. The assembler will calculate the relative distance. The following branch instruction has its own address as destination. This instruction will be executed again and again until the condition is no longer satisfied.

```
1F 08 20 FC   brclr  $08,x  $20  *   wait until the bit $20 becomes 1
```

Since the destination address of branch instructions are limited, the following assembler source line

```
2C ??         bge    toofar       branch if greater or equal
```

may cause an error during assembly. The assembler will automatically replace the instruction by the following sequence, since a JMP can access the whole address space.

```
2D 03         blt    L            branch if less
7E FA 00      jmp    tofar        Jump if greater or equal
          L   equ    *
```

Indexed (IX, IY). The operand of the instruction may be anywhere in the 68HC11 address space as for the direct addressing mode. The address is calculated from the contents of the index register IX or IY incremented by an unsigned constant $00 to $FF that is stored in the program behind the opcode. The resulting address has 16 bits. This addressing mode is variable and each subprogram that is written to access any memory data, must use it. The following program clears the RAM from $40 to $4F.

```
CE 00 40         ldx     #$0040         start at address $0040
86 10            ldaa    #16            ACCA counter: 16 bytes
6F 00       L:   clr     0,x            clear RAM address = $40 .. $4F
08               inx                    increment address in IX
4A               deca                   decrement counter
26 FA            bne     L              return to loop when not yet finished
```

An other advantage of the indexed addressing mode is, that instructions using this mode are shorter and faster than those using the full 16 bit direct addressing mode. Compare the following two instructions.

```
B6 10 03         ldaa    $1003          load from address $1003
A6 03            ldaa    $03,x          same, when IX continues $1000
```

When you set the register IX to $1000 you can always address the input / output registers using the indexed addressing mode. The bit manipulation instructions (BCLR, BSET, BRCLR, BRSET) do not have the 16 bit direct addressing mode. They can only address the RAM ($00 to $FF) directly. To use these instructions elsewhere you must use an index register.

The index registers IX and IY seem to be interchangeable. This is true for the assembly source. All instructions that exist for the register IX have their equivalent for IY and vice versa. But this is not true on machine language level. Compare the following lines

```
08               inx                    increment IX
18 08            iny                    increment IY
6F 03            clr     3,x            clear memory at address  IX+3
18 6F 03         clr     3,y            clear memory at address  IY+3
```

Note that Motorola added the register IY later to the instruction set. Since there were no free opcodes left, Motorola had to add pre-bytes to extend the instruction set. All instructions using the register IY have a pre-byte (mainly $18). The usage of IY costs thus more place in the program and is slower than IX.

Instruction Set (Overview)

Mnemonic	Operands	Flags	Operation
Load and Store			
LDAA/B	#, dir, ext, ind	NZV	ACCx = M
LDD/S/X/Y	##, dir, ext, ind	NZV	ACCD/SP/IX/IY = M:M+1
PSHA/B/X/Y		-	push A/B/IX/IY
PULA/B/X/Y		-	pop A/B/IX/IY
STAA/B	dir, ext, ind	NZV	M = ACCx
STD/S/X/Y	dir, ext, ind	NZV	M:M+1 = ACCD/SP/IX/IY
TAB		NZV	ACCB = ACCA
TAP		all	CondCodes = ACCA
TBA		NZV	ACCA = ACCB
TPA		-	ACCA = CondCodes
TSX/Y		-	IX/IY = SP+1
TXS/TYS		-	SP = IX/IY - 1
XGDX/Y		-	ACCD <=> IX/IY
Arithmetical Instructions			
ABA		HNZVC	ACCA += ACCB
ADCA/B	#, dir, ext, ind	HNZVC	ACCx += M + C
ADDA/B	#, dir, ext, ind	HNZVC	ACCx += M
ADDD	##, dir, ext, ind	NZVC	ACCD += M:M+1
ANDA/B	#, dir, ext, ind	NZV	ACCx &= M
CBA		NZVC	Compare: ACCA-ACCB
CLRA/B/m	ext, ind	NZVC	ACCx/M = 0
CMPA/B	#, dir, ext, ind	NZVC	ACCx - M
COMA/B/m	ext, ind	NZVC	ACCx/M = ~ ACCx/M
CPD	##, dir, ext, ind	NZVC	ACCD - M:M+1
DAA		NZVC	decimal adjust ACCA
DECA/B/m	ext, ind	NZV	ACCx/M --
EORA/B	#, dir, ext, ind	NZV	ACCx ^= M exclusiv or
FDIV		ZVC	IX,ACCD = ACCD/IX fractional
IDIV		ZVC	IX,ACCD = ACCD/IX integer unsigned
INCA/B/m	ext, ind	NZV	ACCx/M ++
MUL		C	ACCD = ACCA * ACCB
NEGA/B/m	ext, ind	NZVC	ACCx/M = 0 - ACCx/M
ORAA/B	#, dir, ext, ind	NZV	ACCx |= M
SBA		NZVC	ACCA -= ACCB
SBCA/B	#, dir, ext, ind	NZVC	ACCx -= M + C
SUBA/B	#, dir, ext, ind	NZVC	ACCx -= M
SUBD	##, dir, ext, ind	NZVD	ACCD -= M
TSTA/B/m	ext, ind	NZVC	ACCx/M - 0
Shift, Rotate			
ASLA/B/m	ext, ind	NZVC	ACCx/M arithm. shift left 1 bit
ASLD		NZVC	ACCD arithm. shift left double 1 bit
ASRA/B/m	ext, ind	NZVC	ACCx/M arithm. shift right 1 bit
LSLA/B/m	ext, ind	NZVC	ACCx/M shift left 1 bit
LSLD		NZVC	ACCD shift left double 1 bit
LSRA/B/m	ext, ind	NZVC	ACCx/M shift right 1 bit; bit7=0
LSRD		NZVC	ACCD shift right 1 bit; bit15=0
ROLA/B/m	ext, ind	NZVC	ACCx/M rotate left 1 bit thru Carry
RORA/B/m	ext, ind	NZVC	ACCx/M rotate right 1 bit thru Carry

Bit operations			
BCLR	dir, ind mask	NZV	M &= ~mask
BSET	dir, ind mask	NZV	M |= mask
BITA/B	#, dir, ext, ind	NZV	Bit test: ACCx & M
Index register oerations			
ABX/Y		-	IX/Y += ACCB
CPX/Y	##, dir, ext, ind	NZVC	Compare IX/Y - M:M+1
DES		-	SP --
DEX/Y		Z	IX/Y --
INS		-	SP++
INX/Y		Z	IX/Y++
Branches, Control			
BCC,BCS,BEQ,BGE, BGT,BHI,BHS,BLE, BLO,BLS,BLT,BMI, BNE,BPL,BVC,BVS	rel	-	Branch conditional
BRA	rel	-	Branch
BSR	rel	-	Branch to subroutine
BRCLR	dir, ind mask rel	-	Branch if all bits clear
BRSET	dir, ind mask rel	-	Branch if all bits ones
JMP	ext, ind	-	Jump
JSR	dir, ext, ind	-	Jump to subroutine
RTI		alle	Return from Interrupt
RTS		-	Return from Subroutine
STOP		-	
SWI		I	Software Interrupt
WAI		-	Wait for Interrupt
Condition Code ops			
CLC/I/V		CIV	Clear Carry / Int mask / Overflow
SEC/I/V		CIV	Set Carry / Int mask / Overflow

Operands	**Syntax**	
#	#<$00..$FF>	immediate 8 bit
##	#<$00..$FFFF>	immediate 16 bit
dir	<$00..$FF>	direct addressing 8 bit for the RAM
ext	<$0000..$FFFF>	extended direct adressing 16 bit
ind	<$00..$FF>,X	indexed with IX register
	<$00..$FF>,Y	indexed with IY register
rel	<-128..+127>	PC-relative 3. operand in BRCLR, BRSET instructions seperated by a space from the 2. operand
mask	<$00..$FF>	mask for bit instruction, 2. operand seperated by a space from the 1. operand.

ABA

Add ACCB to ACCA

Add the accumulator B to the accumulator A and store the result in the accumulator A.

CCR	H	will be set for a following DAA instruction (BCD)
	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is 0
	V	1, if the operation causes an overflow (signed numbers)
	C	1, if the operation causes a carry

Modes	1B	ABA

ABX/Y

Add ACCB to IX/IY

Add accumulator B to the index register IX or IY. The contents of ACCB is considered as an unsigned number ($00 to $FF).

CCR -

Modes	3A	ABX
	18 3A	ABY

ADC

Add with Carry

Add the accumulator ACCx, the 2. operand, and the carry bit of the condition code register and store the sum in the accumulator ACCx.

CCR	H	will be set for a following DAA instruction (BCD)
	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if the operation causes an overflow (signed numbers)
	C	1, if the operation causes a carry

Modes	89 xx	ADCA	#<8 bit constant>
	C9 xx	ADCB	#<8 bit constant>
	99 xx	ADCA	<8 bit address>
	D9 xx	ADCB	<8 bit address>
	B9 xx xx	ADCA	<16 bit address>
	F9 xx xx	ADCB	<16 bit address>
	A9 xx	ADCA	<8 bit displacement>,X
	E9 xx	ADCB	<8 bit displacement>,X
	18 A9 xx	ADCA	<8 bit displacement>,Y
	18 E9 xx	ADCB	<8 bit displacement>,Y

ADD

Add without Carry

Add the accumulator ACCx and the 2. operand and store the sum in the accumulator ACCx.

CCR	H	will be set for a following DAA instruction (BCD)
	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if the operation causes an overflow (signed numbers)
	C	1, if the operation causes a carry

Modes	8B xx	ADDA	#<8 bit constant>
	CB xx	ADDB	#<8 bit constant>
	9B xx	ADDA	<8 bit address>
	DB xx	ADDB	<8 bit address>
	BB xx xx	ADDA	<16 bit address>
	FB xx xx	ADDB	<16 bit address>
	AB xx	ADDA	<8 bit displacement>,X
	EB xx	ADDB	<8 bit displacement>,X
	18 AB xx	ADDA	<8 bit displacement>,Y
	18 EB xx	ADDB	<8 bit displacement>,Y

ADDD

Add Double Accumulator

Add the accumulator ACCD and the 2. operand and store the sum in the accumulator ACCD.

CCR	N	1, if bit 15 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if the operation causes an overflow (signed numbers)
	C	1, if the operation causes a carry

Modes	C3 xx xx	ADDD	#<16 bit constant>
	D3 xx	ADDD	<8 bit address>
	F3 xx xx	ADDD	<16 bit address>
	E3 xx	ADDD	<8 bit displacement>,X
	18 E3 xx	ADDD	<8 bit displacement>,Y

AND

Logical AND

Perform the logical AND of the accumulator ACCx and the 2. operand and store the result in ACCx. Each bit of the result is 1 if the corresponding bits of both operands are 1.

CCR	N	1, if bit 7 of the result is 1
	Z	1, if the result is zero
	V	0

Modes	84 xx	ANDA	#<8 bit constant>
	C4 xx	ANDB	#<8 bit constant>
	94 xx	ANDA	<8 bit address>
	D4 xx	ANDB	<8 bit address>
	B4 xx xx	ANDA	<16 bit address>
	F4 xx xx	ANDB	<16 bit address>
	A4 xx	ANDA	<8 bit displacement>,X
	E4 xx	ANDB	<8 bit displacement>,X
	18 A4 xx	ANDA	<8 bit displacement>,Y
	18 E4 xx	ANDB	<8 bit displacement>,Y

ASL

Arithmetic Shift Left

Shift the accumulator or the memory location to the left. The least significant bit will be replaced by a 0. The most significant bit will be transferred into the carry. This instruction performs a multiplication by 2.

CCR	N	1, if bit 7 (or bit 15) of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if the operation causes an overflow (signed numbers)
	C	1, if the bit 7 (bit 15) is 1 before the operation

Modes	48	ASLA	
	58	ASLB	
	78 xx xx	ASL	<16 bit address>
	68 xx	ASL	<8 bit displacement>,X
	18 68 xx	ASL	<8 bit displacement>,Y
	05	ASLD	

ASR

Arithmetic Shift Right

Shift the accumulator or the memory location to the right. The most significant bit stays unchanged. The least significant bit is transferred into the carry flag. This instruction performs a division by 2.

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if C=1 and N=0 or C=0 and N=1
	C	1, if the bit 0 is 1 before the operation

Modes	47	ASRA	
	57	ASRB	
	77 xx xx	ASR	<16 bit address>
	67 xx	ASR	<8 bit displacement>,X
	18 67 xx	ASR	<8 bit displacement>,Y

Bcc

Branch conditional

Branch if the condition codes matche the branch condition.

CCR -

				Branch, if ...
Modes	20 xx	BRA	<-128..+127>	always
	21 xx	BRN	<-128..+127>	never
	24 xx	BCC	<-128..+127>	carry is 0
	25 xx	BCS	<-128..+127>	carry is 1
	27 xx	BEQ	<-128..+127>	zero is 1
	2B xx	BMI	<-128..+127>	negative is 1
	26 xx	BNE	<-128..+127>	zero is 0
	2A xx	BPL	<-128..+127>	negative is 0

After comparing signed numbers

2C xx	BGE	<-128..+127>	1. operand >= 2. operand
2E xx	BGT	<-128..+127>	1. operand > 2. operand
2F xx	BLE	<-128..+127>	1. operand <= 2. operand
2D xx	BLT	<-128..+127>	1. operand < 2. operand

After comparing unsigned numbers

22 xx	BHI	<-128..+127>	1. operand > 2. operand
24 xx	BHS	<-128..+127>	1. operand >= 2. operand
25 xx	BLO	<-128..+127>	1. operand < 2. operand
23 xx	BLS	<-128..+127>	1. operand <= 2. operand

After arithmetical operations on signed numbers

28 xx	BVC	<-128..+127>	operation caused no overflow
29 xx	BVS	<-128..+127>	operation caused an overflow

BCLR

Clear Bits in Memory

Clear one or several bits in the memory location. The bits to be cleared are specified in a mask. The other bits of the memory location are left unchanged.

CCR	N	1, if bit 7 of the result is 1
	Z	1, if the result is zero
	V	0

Modes	15 xx xx	BCLR	<8 bit address> <8 bit mask>
	1D xx xx	BCLR	<8 bit displacement>,X <8 bit mask>
	18 1D xx xx	BCLR	<8 bit displacement>,Y <8 bit mask>

BIT

Bit Test

Perform the logical AND of the accumulator ACCx and the memory location without storing the result. The condition codes are set according to the result of the operation. Each bit of the result is 1 if the corresponding bits of both operands are 1.

CCR	N	1, if bit 7 of the result is 1
	Z	1, if the result is zero
	V	0

Modes	85 xx	BITA	#<8 bit constant>
	C5 xx	BITB	#<8 bit constant>
	95 xx	BITA	<8 bit address>
	D5 xx	BITB	<8 bit address>
	B5 xx xx	BITA	<16 bit address>
	F5 xx xx	BITB	<16 bit address>
	A5 xx	BITA	<8 bit displacement>,X
	E5 xx	BITB	<8 bit displacement>,X
	18 A5 xx	BITA	<8 bit displacement>,Y
	18 E5 xx	BITB	<8 bit displacement>,Y

BRCLR

Branch if Bits Clear

Perform the logical AND of the mask and the memory location without storing the result. Branch if the result is 0, i. e. all 1 bits in the mask correspond to 0 bits in the memory location.

CCR -

Modes	13 xx xx xx	BRCLR	<8 bit address> <8 bit mask> <-128..+127>
	1F xx xx xx	BRCLR	<8 bit displacement>,X <8 bit mask> <-128..+127>
	18 1F xx xx xx	BRCLR	<8 bit displacement>,Y <8 bit mask> <-128..+127>

BRSET

Branch if Bits Set

Perform the logical AND of the mask and the inverted memory location without storing the result. Branch if the result is 0, i. e. all 1 bits in the mask correspond to 1 bits in the memory location.

CCR -

Modes	12 xx xx xx	BRSET	<8 bit address> <8 bit mask> <-128..+127>
	1E xx xx xx	BRSET	<8 bit displacement>,X <8 bit mask> <-128..+127>
	18 1E xx xx xx	BRSET	<8 bit displacement>,Y <8 bit mask> <-128..+127>

BSET

Set Bits in Memory

Set one or several bits in the memory location to 1. The bits to be set are specified in the mask. The other bits in the memory location are left unchanged.

CCR	N	1, if bit 7 of the result is 1	
	Z	1, if the result is zero	
	V	0	
Modes	14 xx xx	BSET	<8 bit address> <8 bit mask>
	1C xx xx	BSET	<8 bit displacement>,X <8 bit mask>
	18 1C xx xx	BSET	<8 bit displacement>,Y <8 bit mask>

BSR

Branch to Subroutine

Push the address of the following instruction on the stack and decrement the SP respectively by 2. Then branch to the given address.

CCR	**-**		
Modes	8D xx	BSR	<-128..+127>

CBA

Compare Accumalators

Compare the accumulators ACCA and ACCB and set the condition codes according to the result.

CCR	N	1, if bit 7 of the subtraction is 1: the result is negative (signed numbers)
	Z	1, if the operands are equal
	V	1, if a subtraction caused an overflow (signed numbers)
	C	1, if the unsigned value of ACCA is smaller than ACCB
Modes	11	CBA

CLC/I/V

Clear Condition Code Bits

Clear a flag in the Condition Code Register. The instruction CLC clears the Carry bit, CLI the Interrupt mask bit, and CLV the Overflow bit.

CCR	C	0 for the instruction CLC
	I	0 for the instruction CLI
	V	0 for the instruction CLV
Modes	0C	CLC
	0E	CLI
	0A	CLV

CLR

Clear

Clear the accumulator or the memory location.

CCR	N	0
	Z	1
	V	0
	C	0

Modes	4F	CLRA	
	5F	CLRB	
	7F xx xx	CLR	<16 bit address>
	6F xx	CLR	<8 bit displacement>,X
	18 6F xx	CLR	<8 bit displacement>,Y

CMP

Compare

Compare the accumulator ACCx with the memory location and set the Condition Codes according to the result. The comparison is done by subtracting the 2. operand from the accumulator without storing the result. The Condition Codes are set for a following conditional branch instruction.

CCR	N	1, if bit 7 of the subtraction is 1: the result is negative (signed numbers)
	Z	1, if the operands are equal
	V	1, if a subtraction caused an overflow (signed numbers)
	C	1, if the unsigned value of ACCx is smaller than the 2. operand

Modes	81 xx	CMPA	#<8 bit constant>
	C1 xx	CMPB	#<8 bit constant>
	91 xx	CMPA	<8 bit address>
	D1 xx	CMPB	<8 bit address>
	B1 xx xx	CMPA	<16 bit address>
	F1 xx xx	CMPB	<16 bit address>
	A1 xx	CMPA	<8 bit displacement>,X
	E1 xx	CMPB	<8 bit displacement>,X
	18 A1 xx	CMPA	<8 bit displacement>,Y
	18 E1 xx	CMPB	<8 bit displacement>,Y

COM

Complement

Replace the accumulator or the memory location by ist ones complement. Each 1 is replaced by a 0, and each 0 is replaced by a 1.

CCR	N	1, if bit 7 of the result is 1
	Z	1, if the result is zero
	V	0
	C	1

Modes	43	COMA	
	53	COMB	
	73 xx xx	COM	<16 bit address>
	63 xx	COM	<8 bit displacement>,X
	18 63 xx	COM	<8 bit displacement>,Y

CPD/X/Y

Compare Double Register

Compare the double register with the memory location and set the Condition Codes according to the result. The comparison is done by subtracting the 2. operand from the double register without storing the result. The Condition Codes are set for a following conditional branch instruction.

CCR	N	1, if bit 15 of the subtraction is 1: the result is negative (signed numbers)
	Z	1, if the operands are equal
	V	1, if a subtraction caused an overflow (signed numbers)
	C	1, if the unsigned value of the register is smaller than the 2. operand

Modes	1A 83 xx xx	CPD	#<16 bit constant>
	8C xx xx	CPX	#<16 bit constant>
	18 8C xx xx	CPY	#<16 bit constant>
	1A 93 xx	CPD	<8 bit address>
	9C xx	CPX	<8 bit address>
	18 9C xx	CPY	<8 bit address>
	1A B3 xx xx	CPD	<16 bit address>
	BC xx xx	CPX	<16 bit address>
	18 BC xx xx	CPY	<16 bit address>
	1A A3 xx	CPD	<8 bit displacement>,X
	AC xx	CPX	<8 bit displacement>,X
	1A AC xx	CPY	<8 bit displacement>,X
	CD A3 xx	CPD	<8 bit displacement>,Y
	CD AC xx	CPX	<8 bit displacement>,Y
	18 AC xx	CPY	<8 bit displacement>,Y

DAA

Decimal Adjust ACCA

Adjust the accumulator A after a ABA, ADD, or ADC operation. If both operands of the add operation were in BCD notation, the accumulator will be adjusted to BCD notation.

CCR	N	1, if bit 7 of the result is 1
	Z	1, if the result is zero
	V	?
	C	1, if the operation causes a carry

Modes	19	DAA

DEC

Decrement

Decrement the accumulator or the memory location by 1.

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if the operation causes an overflow (signed numbers), i.e. the operand is $80

Modes	4A	DECA	
	5A	DECB	
	7A xx xx	DEC	<16 bit address>
	6A xx	DEC	<8 bit displacement>,X
	18 6A xx	DEC	<8 bit displacement>,Y

DES/X/Y

Decrement Double Register

Decrement the double register SP, IX, or IY by 1.

CCR	Z	1, if the result is zero
		The instruction DES does not change the CCR

Modes	34	DES
	09	DEX
	18 09	DEY

EOR

Exclusive OR

Perform the EXCLUSIVE OR of the accumulator ACCx and the 2. operand and store the result in ACCx. Each bit of the result is 1 if exactly one of the corresponding bits of both operands is 1.

CCR	N	1, if bit 7 of the result is 1
	Z	1, if the result is zero
	V	0

Modes	88 xx	EORA	#<8 bit constant>
	C8 xx	EORB	#<8 bit constant>
	98 xx	EORA	<8 bit address>
	D8 xx	EORB	<8 bit address>
	B8 xx xx	EORA	<16 bit address>
	F8 xx xx	EORB	<16 bit address>
	A8 xx	EORA	<8 bit displacement>,X
	E8 xx	EORB	<8 bit displacement>,X
	18 A8 xx	EORA	<8 bit displacement>,Y
	18 E8 xx	EORB	<8 bit displacement>,Y

FDIV

Fractional Divide

Divide the 16 bit numerator in the register D by the 16 bit denominator in IX and store the quotient in IX and the remainder in D. The numerator must be smaller than the denominator. The denominator must not be 0. The quotient in IX represents a value between 0,000015 ($0001) and 0,99998 ($FFFF). This instruction is used to resolve the remainder of an IDIV instruction. The remainder of the FDIV instruction can be resolved by an other FDIV instruction.

CCR	Z	1, if the result is zero
	V	1, if $IX \leq D$
	C	1, if the denominator is 0

Modes	03	FDIV

IDIV

Integer Divide

Divide the unsigned 16 bit numerator in D by the unsigned 16 bit denominator in IX and store the quotient in IX and the remainder in D. The denominator must not be 0.

CCR	Z	1, if the result is zero
	V	0
	C	1, if the denominator is 0

Modes	02	IDIV	

INC

Increment

Increment the accumulator or the memory location by 1.

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if the operation causes an overflow (signed numbers) i.e. the operand is $7F

Modes	4C	INCA	
	5C	INCB	
	7C xx xx	INC	<16 bit address>
	6C xx	INC	<8 bit displacement>,X
	18 6C xx	INC	<8 bit displacement>,Y

INS/X/Y

Increment Double Register

Increment the double register SP, IX, or IY by 1.

CCR	Z	1, if the result is zero The instruction INS does not change the CCR.

Modes	31	INS
	08	INX
	18 08	INY

JMP

Jump

Jump to the specified address.

CCR -

Modes	7E xx xx	JMP	<16 bit address>
	6E xx	JMP	<8 bit displacement>,X
	18 6E xx	JMP	<8 bit displacement>,Y

JSR

Jump to Subroutine

Push the address of the following instruction on the stack and decrement the SP respectively by 2. Then jump to the given address.

CCR -

Modes	9D xx	JSR	<8 bit address>
	BD xx xx	JSR	<16 bit address>
	AD xx	JSR	<8 bit displacement>,X
	18 AD xx	JSR	<8 bit displacement>,Y

LDA

Load Accumulator

Load the accumulator by the memory location.

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	0

Modes	86 xx	LDAA	#<8 bit constant>
	C6 xx	LDAB	#<8 bit constant>
	96 xx	LDAA	<8 bit address>
	D6 xx	LDAB	<8 bit address>
	B6 xx xx	LDAA	<16 bit address>
	F6 xx xx	LDAB	<16 bit address>
	A6 xx	LDAA	<8 bit displacement>,X
	E6 xx	LDAB	<8 bit displacement>,X
	18 A6 xx	LDAA	<8 bit displacement>,Y
	18 E6 xx	LDAB	<8 bit displacement>,Y

LDD/S/X/Y

Load Double Register

Load the 16 bit double register by two consecutive bytes. The address specifies the address of the most significant byte. The least significant byte is loaded from the following address.

CCR	N	1, if bit 15 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	0

Modes	CC xx xx	LDD	#<16 bit constant>
	DC xx	LDD	<8 bit address>
	FC xx xx	LDD	<16 bit address>
	EC xx	LDD	<8 bit displacement>,X
	18 EC xx	LDD	<8 bit displacement>,Y
	8E xx xx	LDS	#<16 bit constant>
	9E xx	LDS	<8 bit address>
	BE xx xx	LDS	<16 bit address>
	AE xx	LDS	<8 bit displacement>,X
	18 AE xx	LDS	<8 bit displacement>,Y
	CE xx xx	LDX	#<16 bit constant>
	DE xx	LDX	<8 bit address>
	FE xx xx	LDX	<16 bit address>
	EE xx	LDX	<8 bit displacement>,X
	CD EE xx	LDX	<8 bit displacement>,Y
	18 CE xx xx	LDY	#<16 bit constant>
	18 DE xx	LDY	<8 bit address>
	18 FE xx xx	LDY	<16 bit address>
	1A EE xx	LDY	<8 bit displacement>,X
	18 EE xx	LDY	<8 bit displacement>,Y

LSL

Logical Shift Left

Shift the accumulator or the memory location to the left. The least significant bit will be replaced by a 0. The most significant bit will be transferred into the carry flag.

CCR	N	1, if bit 7 (LSLD: bit 15) of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if the operation causes an overflow (signed numbers)
	C	1, if the bit 7 (LSLD: bit 15) is 1 before the operation

Modes	48	LSLA	
	58	LSLB	
	78 xx xx	LSL	<16 bit address>
	68 xx	LSL	<8 bit displacement>,X
	18 68 xx	LSL	<8 bit displacement>,Y
	05	LSLD	

LSR

Logical Shift Right

Shift the accumulator or the memory location to the right. The most significant bit will be replaced by a 0. The least significant bit will be transferred into the carry flag.

CCR	N	0
	Z	1, if the result is zero
	V	1, if the bit 0 was 1 before the operation
	C	1, if the bit 0 was 1 before the operation

Modes	44	LSRA	
	54	LSRB	
	74 xx xx	LSR	<16 bit address>
	64 xx	LSR	<8 bit displacement>,X
	18 64 xx	LSR	<8 bit displacement>,Y
	04	LSRD	

MUL

Multiply

Multiply the 8 bit unsigned number in accumulator A by the 8 bit unsigned number in accumulator B and store the 16 bit result in the double register D.

CCR	C	1, if the bit 7 of the result (bit 7 of ACCB) is 1

Modes	3D	MUL

NEG

Negate

Replace the accumulator or the memory location by its twos complement.

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if the operation causes an overflow (signed numbers), i.e. the oprand is $80
	C	1, if the operation causes a carry, i. e. the operand is not zero.

Modes	40	NEGA	
	50	NEGB	
	70 xx xx	NEG	<16 bit address>
	60 xx	NEG	<8 bit displacement>,X
	18 60 xx	NEG	<8 bit displacement>,Y

NOP

No Operation

CCR	-

Modes	01	NOP

ORA

Inclusive OR

Perform the INCLUSIVE OR of the accumulator ACCx and the 2. operand and store the result in ACCx. Each bit of the result is 1 if at least one of the corresponding bits of both operands is 1.

CCR	N	1, if bit 7 of the result is 1
	Z	1, if the result is zero
	V	0

Modes	8A xx	ORA	#<8 bit constant>
	CA xx	ORB	#<8 bit constant>
	9A xx	ORA	<8 bit address>
	DA xx	ORB	<8 bit address>
	BA xx xx	ORA	<16 bit address>
	FA xx xx	ORB	<16 bit address>
	AA xx	ORA	<8 bit displacement>,X
	EA xx	ORB	<8 bit displacement>,X
	18 AA xx	ORA	<8 bit displacement>,Y
	18 EA xx	ORB	<8 bit displacement>,Y

PSH

Push Register onto Stack

Push the accumulator ACCA or ACCB or the double register IX or IY on the stack and decrement the stack pointer respectively by 1 or 2.

CCR -

Modes	36	PSHA
	37	PSHB
	3C	PSHX
	18 3C	PSHY

PUL

Pull Register from Stack

Pull the accumulator ACCA or ACCB or the double register IX or IY from the stack after having incremented the stack pointer respectively by 1 or 2.

CCR -

Modes	32	PULA
	33	PULB
	38	PULX
	18 38	PULY

ROL

Rotate Left

Rotate the accumulator or the memory location left. The least significant bit will be replaced by the Carry flag. The most significant bit is transferred into the Carry flag.

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if C=1 and N=0 or C=0 and N=1
	C	1, if the bit 7 is 1 before the operation

Modes	49	ROLA	
	59	ROLB	
	79 xx xx	ROL	<16 bit address>
	69 xx	ROL	<8 bit displacement>,X
	18 69 xx	ROL	<8 bit displacement>,Y

ROR

Rotate Right

Rotate the accumulator or the memory location right. The most significant bit will be replaced by the Carry flag. The least significant bit is transferred into the Carry flag.

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, wenn C=1 und N=0 oder C=0 und N=1
	C	1, if the bit 7 is 1 before the operation

Modes	46	RORA	
	56	RORB	
	76 xx xx	ROR	<16 bit address>
	66 xx	ROR	<8 bit displacement>,X
	18 66 xx	ROR	<8 bit displacement>,Y

RTI

Return from Interrupt

Pop the registers CCR, ACCB, ACCA, IX, IY, and PC from the stack and increment the stack pointer SP by 9 respectively. Since the instruction reads the PC from the stack, it performs a jump. This instruction allows a complete recovery of a program that was suspended by an interrupt

CCR	The CCR is read from the stack. The X bit cannot be set to 1 by this instruction.

Modes	3B	RTI

RTS

Return from Subroutine

Pop a 16 bit address from the stack and increment the stack pointer respectively by 2. Jump to the address. This instruction returns to a program that has executed a BSR or a JSR.

CCR	-	
Modes	39	RTS

SBA

Subtract ACCB from ACCA

Subtract the accumulator B from the accumulator A and store the result in the accumulator A.

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if the operation causes an overflow (signed numbers)
	C	1, if the operation causes a carry

Modes	10	SBA

SBC

Subtract with Carry

Subtract the accumulator B and the Carry flag of the CCR from the accumulator A and store the result in the accumulator A.

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if the operation causes an overflow (signed numbers)
	C	1, if the operation causes a carry

Modes	82 xx	SBCA	#<8 bit constant>
	C2 xx	SBCB	#<8 bit constant>
	92 xx	SBCA	<8 bit address>
	D2 xx	SBCB	<8 bit address>
	B2 xx xx	SBCA	<16 bit address>
	F2 xx xx	SBCB	<16 bit address>
	A2 xx	SBCA	<8 bit displacement>,X
	E2 xx	SBCB	<8 bit displacement>,X
	18 A2 xx	SBCA	<8 bit displacement>,Y
	18 E2 xx	SBCB	<8 bit displacement>,Y

SEC/I/V

Set Condition Code Bits

Set a flag in the Condition Code Register. The instruction CLC sets the Carry, CLI the Interrupt mask, and CLV the Overflow bit.

CCR	C	1 for the instruction CLC
	I	1 beim Befehl CLI
	V	1 beim Befehl CLV

Modes	0D	SEC
	0F	SEI
	0B	SEV

STA

Store Accumulator

Store the accumulator ACCx at the memory location.

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	0

Modes	97 xx	STAA	<8 bit address>
	D7 xx	STAB	<8 bit address>
	B7 xx xx	STAA	<16 bit address>
	F7 xx xx	STAB	<16 bit address>
	A7 xx	STAA	<8 bit displacement>,X
	E7 xx	STAB	<8 bit displacement>,X
	18 A7 xx	STAA	<8 bit displacement>,Y
	18 E7 xx	STAB	<8 bit displacement>,Y

STD/S/X/Y

Store Double Register

Store the most significant byte of the 16 bit double register at the memory location, and the least significant byte at the next memory location.

CCR	N	1, if bit 15 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	0

Modes	DD xx	STD	<8 bit address>
	FD xx xx	STD	<16 bit address>
	ED xx	STD	<8 bit displacement>,X
	18 ED xx	STD	<8 bit displacement>,Y
	9F xx	STS	<8 bit address>
	BF xx xx	STS	<16 bit address>
	AF xx	STS	<8 bit displacement>,X
	18 AF xx	STS	<8 bit displacement>,Y
	DF xx	STX	<8 bit address>
	FF xx xx	STX	<16 bit address>
	EF xx	STX	<8 bit displacement>,X
	CD EF xx	STX	<8 bit displacement>,Y
	18 DF xx	STY	<8 bit address>
	18 FF xx xx	STY	<16 bit address>
	1A EF xx	STY	<8 bit displacement>,X
	18 EF xx	STY	<8 bit displacement>,Y

STOP

Stop Processing

Stop all internal processing and enter the STOP mode to reduce power consumption. All registers and outputs stay unchanged. Only a XIRQ or an unmasked IRQ can force the CPU to recover from this state. If the S bit is set in the CCR, this instruction is ignored.

CCR **-**

Modes	CF	STOP

SUB

Subtract

Subtract the memory location from the accumulator ACCx and store the result in the accumulator.

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if the operation causes an overflow (signed numbers)
	C	1, if the operation causes a carry

Modes	80 xx	SUBA	#<8 bit constant>
	C0 xx	SUBB	#<8 bit constant>
	90 xx	SUBA	<8 bit address>
	D0 xx	SUBB	<8 bit address>
	B0 xx xx	SUBA	<16 bit address>
	F0 xx xx	SUBB	<16 bit address>
	A0 xx	SUBA	<8 bit displacement>,X
	E0 xx	SUBB	<8 bit displacement>,X
	18 A0 xx	SUBA	<8 bit displacement>,Y
	18 E0 xx	SUBB	<8 bit displacement>,Y

SUBD

Subtract Double Accumulator

Subtract the memory location from the double accumulator D and store the result in D.

CCR	N	1, if bit 15 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	1, if the operation causes an overflow (signed numbers)
	C	1, if the operation causes a carry

Modes	83 xx xx	SUBD	#<16 bit constant>
	93 xx	SUBD	<8 bit address>
	B3 xx xx	SUBD	<16 bit address>
	A3 xx	SUBD	<8 bit displacement>,X
	18 A3 xx	SUBD	<8 bit displacement>,Y

SWI

Software Interrupt

Interrupt the program and execute the interrupt service routine for the SWI instruction. This instruction behaves nearly like an external interrupt. The registers PC, IY, IX ACCA, ACCB, and CCR are pushed on the stack and the stack pointer is decremented respectively by 9. The CPU fetches the address of the interrupt service routine from address $FFF6 and jumps to the routine. This instruction is used by the talker and therefore not available.

CCR	I	1

Modes	3F	SWI

TAB/TBA — Transfer Accumulator

Transfer the accumulator ACCA to ACCB (TAB), or ACCB to ACCA (TBA).

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	0

Modes	16	TAB
	17	TBA

TAP/TPA — Transfer Condition Codes

Transfer the accumulator ACCA to the Condition Code Register CCR (TAP), or the CCR to ACCA (TPA).

CCR TPA does not change the CCR.
TAP changes all bits of the CCR. The X flag can not be set to 1.

Modes	06	TAP
	07	TPA

TST — Test

Subtract 0 from the accumulator or from the memory location and set the CCR according to the result.

CCR	N	1, if bit 7 of the result is 1: the result is negative (signed numbers)
	Z	1, if the result is zero
	V	0
	C	0

Modes	4D	TSTA	
	5D	TSTB	
	7D xx xx	TST	<16 bit address>
	6D xx	TST	<8 bit displacement>,X
	18 6D xx	TST	<8 bit displacement>,Y

TSX/Y TX/YS — Transfer Stack Pointer

Store the stack pointer SP incremented by 1 to the index register IX (TSX) or IY (TSY). The stack pointer stays unchanged and the index register points to the last byte written on the stack. Load the stack pointer from IX (TXS) or IY (TYS) decremented by 1.

CCR -

Modes	30	TSX
	18 30	TSY
	35	TXS
	18 35	TYS

WAI

Wait for Interrupt

Enter the WAIT mode to reduce power consumption and wait until an unmasked interrupt interrupts the CPU. In the WAIT mode all input / output units continue to work normally.

CCR	**-**	
Modes	3E	WAI

XGDX/Y

Exchange Double Accumalator and Index Register

Exchange the double register D with the index register IX or IY.

CCR	**-**	
Modes	8F	XGDX
	18 8F	XGDY

Interrupts

Interrupts arrive asynchronously since they are caused by external events. For example the program that is generated by the program generator uses a timer that interrupts the program regularly 150 times per second. You may use other interrupts in your program, you may disable them or enable them by the following instructions:

sei	disable all maskable interrupts
cli	enable all interrupts

An interrupt suspends the running program and calls an interrupt service routine. This routine must terminate with the instruction

rti	Return from Interrupt

If you want to use interrupts, it is wise to follow the following rules: Keep your interrupt service routines as short as possible. Errors in interrupt service routine often provoke inaccuracies, that do not appear directly, but some time later and only occasionally. Interrupts are fast, efficient and dangerous, and they are difficult to debug. It is usual to allow interrupts normally and to disable them only in critical phases of the program. Do you know all the critical phases of your program? It is much more trustworthy to disallow them normally and only enable them in certain phases.

When an interrupt occurs the CPU fetches from a table in the memory the address of the interrupt service routine. When your program wants to handle an interrupt, it must declare the interrupt service routine in this table.

	Address	Interrupt source
*	FFD6	SCI Asynchronous Serial Interface (RS232)
	FFD8	SPI Synchronous Peripheral Interface
	FFDA	Counter / Timer PA Input Edge
	FFDC	Counter / Timer PA Overflow
	FFDE	Timer Overflow
	FFE0	Timer Output Compare 5
	FFE2	Timer Output Compare 4
	FFE4	Timer Output Compare 3
	FFE6	Timer Output Compare 2
	FFE8	Timer Output Compare 1
	FFEA	Timer Input Capture 3
	FFEC	Timer Input Capture 2
	FFEE	Timer Input Capture 1
	FFF0	Real Time Interrupt
	FFF2	IRQ Pin
	FFF4	XIRQ Pin
*	FFF6	SWI Software interrupt
*	FFF8	Illegal opcode
	FFFA	COP Fail
	FFFC	Clock Monitor
*	FFFE	RESET

Interrupts marked with an * are reserved by the talker.

For example: The real time interrupt which is used by the real timer. When this interrupt occurs, the CPU fetches from the address $FFF0 the address of the interrupt service routine. In order to handle this interrupt, you must write in your program

```
org     $FFF0
fdb     rtiint
```

and somewhere else the interrupt service routine

```
rtiint:   ...
          rti
```

The 68HC11 saves all registers on the stack before executing an interrupt service routine. The instruction RTI restores all registers before continuing the interrupted program.

```
                        Interrupt               RTI


SP =  E8 ->                   E8                SP =  E8 ->
                                      PC
                              E7

                              E6
                                      IY
                              E5

                              E4
                                      IX
                              E3

                              E2     ACCA

                              E1     ACCB

                              E0     CCR

                        SP =  DF ->
```

Preparing the software and the target

Standard Parameters

Click on FILE, then on CONFIGURATION.

Port: The port of the PC to communicate with the target: COM1, COM2, COM3, or COM4 or SIM to work with the vsimulator.

Hardware: You may choose from several built-in models. When using other hardware, you may leave this field empty. All configuration data will then be stored in the file ENV.TXT. When you enter a name like TGT1, only a line BOARD=TGT1 will be stored in ENV.TXT, and all other data will be stored in a file TGT1.CNF.

Crystal: You may choose the crystal for the 68HC11 from 1 to 24 MHz.

Baud rate: Choose the communication rate with the target. The possible rates are calculated from the crystal frequency from 300 to 57600 baud.

Communication timeout: Time in ms, the host program waits for a response from the target. If the target does not respond within this time, the debugger will give up the communication. 2000 ms is a fair time for normal communication speed.

RAM: specifies the size and the address of the RAM.

EEPROM: specifies the size and the address of the program memory.

Changing these parameters may require the replacement of the talker in the target. See the command INITTALKER of the debugger.

The Talkers

The debugger uses a small program called talker which is running on the target.

File	internal name	resides	Address	works on
talkboy.a11	cboy	EEPROM	FE80	EEPROM of the 68HC811E2 (Controlboy 1)
talkram.a11	ram	RAM	0	EEPROM of the 68HC11A1,E1,F1,..
talkxico.a11	xicor	EEPROM	FE80	EEPROM compatible XICOR in protected mode
talkxram.a11	xram	RAM/EEPROM	FE80	RAM or EEPROM compatible XICOR in unprotected mode
talkslic.a11	slic	EEPROM	FE80	EEPROM Xicor X68C75 (Controlboy 2, 3)
talkcf1.a11	cboyf1	EEPROM	FE00	EEPROM compatible Xicor(Controlboy F1)

The talker may be adapted to the needs of the target. The talker includes the procedure to write into the EEPROM. You may either use an internal talker or specify a talker in an assembly file. You have to compile this file. The debugger command INITTALKER loads the talker into the target. The talkers of Basic11 have the extension .A11, those of CC11 have the extension .S.

The program uses the asynchronous serial interface of the 68HC11 for communication. The pins PD1 and PD0 are used as Transmit Data and Receive Data and must be translated to RS232 compatible lines.

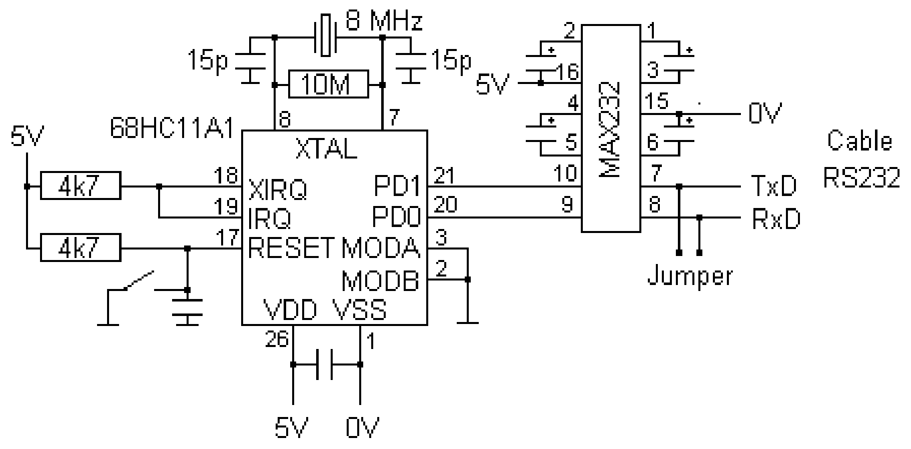
The pointer declarations are essential for the target program.

Using Basic11 these declarations are at the beginning of the program. You will often use the file START.BAS which includes the declarations. These declarations must be adapted to the target board. You should also verify the declarations of the B and C ports which depend on the material.

Using CC11 you find the pointer declarations in under OPTIONS, file.MAK and in the makefile and the port declarations in the file HC11.H.

Finally it must be assured that your program will start on the target after a RESET.

You find here a minimum 68HC11 based target board.



The target - P.C. cable.

68HC11	Max232	PC COM port	
		9 pins	25 pins
PD1 (TxD)	10 - 7	2 (RxD)	3 (RxD)
PD0 (RxD)	9 - 8	3 (TxD)	2 (TxD)
0V		5	7

Once you have changed your configuration you must assure the communication between the P.C. and the target. Connect the target to the serial port of the P.C., supply power to the board, choose the window of the debugger and follow the instructions below according to your target configuration.

Controlboy 1, 2k EEPROM, 256 RAM

You only have to select the correct board during the installation of the software. The talker CBOY is already in the EEPROM of the 68HC11 from FE80 to FFFF, using the RAM from 00E9 to 00FF. After a RESET the talker examines the input D5 (button T1). If it is zero (button pressed), the talker will wait for commands from the serial line. If it is one, the talker jumps to F800 to execute the application program.

Basic11	**ProgramPointer**	**$F800**	**DataPointer**	**$0002**	**StackPointer**	**$00E8**
CC11	**text**	**0xF800**	**data**	**0x0002**	**init_sp**	**0x00E8**

Controlboy 2, Controlboy 3, 8k EEPROM, 512 RAM

You only have to select the correct board during the installation of the software. The talker SLIC is already in the EEPROM from FE80 to FFFF, using the RAM from 00E9 to 00FF. After a RESET the talker examines the input D5 (button T1). If it is zero (button pressed), the talker will wait for commands from the serial line. If it is one, the talker jumps to E000 to execute the application program.

Basic11	**ProgramPointer**	**$E000**	**DataPointer**	**$0002**	**StackPointer**	**$01FF**
CC11	**text**	**0xE000**	**data**	**0x0002**	**init_sp**	**0x01FF**

Controlboy F1, 32k EEPROM, 32k RAM

You only have to select the correct board during the installation of the software. The talker CBOYF1 is already in the EEPROM from FE00 to FFFF, using the RAM from 00E9 to 00FF. After a RESET the talker examines the input G1 (button T1). If it is zero (button pressed), the talker will wait for commands from the serial line. If it is one, the talker jumps to 8000 to execute the application program.

Basic11	**ProgramPointer**	**$8000**	**DataPointer**	**$2000**	**StackPointer**	**$7FFF**
CC11	**text**	**0x8000**	**data**	**0x2000**	**init_sp**	**0x7FFF**

68HC11A1, 512 EEPROM, 256 RAM

You have to download the talker RAM after each RESET into the target using the debugger command INITTALKER.

Basic11	ProgramPointer	$B600	DataPointer	$0002	StackPointer	$00EA
CC11	text	0xB600	data	0x0002	init_sp	0x00EA

The microprocessor always runs in BOOTSTRAP mode. To let your program loaded at B600 start automatically after a RESET, you have to connect Transmit Data and Receive Data of the 68HC11. The memory is used by both, the talker, and the application program.

0000 - 0021	RAM may be used by the application
0022 - 00CB	Talker
00CC - 00EA	Stack 31 bytes
00EB - 00FF	Talker

68HC11E1, 512 EEPROM, 512 RAM

Like the 68HC11A1. But with 512 bytes of RAM, the first 256 bytes may be reserved to the talker.

Basic11	ProgramPointer	$B600	DataPointer	$0100	StackPointer	$01FF
CC11	text	0xB600	data	0x0100	init_sp	0x01FF

68HC11F1, 512 EEPROM, 1k RAM

You have to download the talker RAM after each RESET into the target using the debugger command INITTALKER. The first 256 bytes of RAM are reserved for the talker.

Basic11	ProgramPointer	$FE00	DataPointer	$0100	StackPointer	$03FF
CC11	text	0xFE00	data	0x0100	init_sp	0x03FF

The microprocessor runs in BOOTSTRAP mode to download and to debug the program. To let your program loaded at FE00 start automatically after a RESET, you have to initialise the reset interrupt vector at FFFE in the EEPROM. You write once at the debugger prompt:

```
mset -w FFFE = FE00
```

The 68HC11 will run the program at FE00 after a RESET in SINGLE CHIP mode.

68HC811E2, 2k EEPROM, 256 RAM

You have to load the talker CBOY once using the debugger command INITTALKER into the EEPROM of the target. The talker uses the EEPROM from FE80 to FFFF and the RAM from 00E9 to 00FF. After a RESET the talker examines the input D5. If it is zero, the talker will wait for commands from the serial line. If it is one, the talker jumps to F800 to execute the application program.

Basic11	ProgramPointer	$F800	DataPointer	$0002	StackPointer	$00E8
CC11	text	0xF800	data	0x0002	init_sp	0x00E8

68HC11 using external memory, the talker

Make a copy of an existing talker to adapt it to your target.

Using Basic11, you find the sources of the talkers in the directory TALKER11. The files have the extension .A11.

Using CC11, you find the sources of the talkers in the directory TALKERS. The files have the extension .S.

The talker talkxico.a11 (or talkxico.s) is the most frequently used talker for another target.

The first part contains a $100 long preloader. The debugger command INITTALKER will guide you later on to set the 68HC11 into bootstrap mode. The 68HC11 will load the preloader into the RAM from $0000 to $00FF. The 68HC11 will then jump to address $0000 to execute the preloader. The preloader sets the microcontroller into expanded mode to have access to the external memory. It will then load the talker by the serial line and program it at the end of the EEPROM. This talker works with an EEPROM compatible Xicor at the end of the memory space. Note that nearly all parallel EEPROM are compatible with this type.

The preloader and the talker are in the same source file. You may have to modify the preloader due to your hardware so that external memory can be accessed after the label PRESTART, before the preloader loads the talker. You may also have to modify the talker that way. After a reset, the talker jumps to TALKSTART. All hardware specific is in the first twenty lines after this label. This talker flashes a LED at output PD3. This talker examines a pushbutton at input PD5. If it is pressed, zero, the talker awaits a command from the serial line, If the input is 1, the talker jumps to the beginning of the EEPROM to execute the application program.

Once modified you have to recompile the program.
Using Basic11 click on COMPILE.

Using CC11, there is a file TALKERS.MAK to compile all talkers. You must edit the file pour add your talker. The talker must be compiled like this:

```
talkxxx.s19: talkxxx.s
        icc11w -m -R -btext:0 -bdata:0 -dinit_sp:0 -dheap_size:0 talkxxx.s
```

The preloader must be exactly $100 bytes long. You can add or subtracts some bytes after the label PREFREEBYTES. The talker must exactly end at the address $FFFF. You can add or subtract some bytes after the label TALKFREEBYTES. Note also, that the host will modify lines marked POKE before downloading. If the size of the preloader or the talker is not correct, Basic11 displays an error. Using CC11 you have to verify all lines in the LIS file containing #assert.

The configuration of the debugger proposes all talkers found in the talker directory. Choose your talker. You have to execute the INITTALKER command to download the talker. You can follow the download in the communication field of the debugger.
When the procedure stops when accessing external memory:

- Verify that the ROM is not on. Register CONFIG, bit 1 ROMON must be 0. Load the talker RAM and type in >d CONFIG. You can change CONFIG by >CONFIG=$13, but you must press reset again, to have the new value in the register.
- Verify that you memory is accessible. Load the talker RAM and type in >HPRIO=$25 to change to expanded mode. Verify the access to the external memory using the MEM command of the debugger.

Finally the debugger must show you TARGET STOP.

All targets

Your program has to execute the CLI assembler instruction to allow the communication of the P.C. with the talker on the target.

Your program can overwrite the program or the data of the talker. You can download such a program but you cannot debug it.

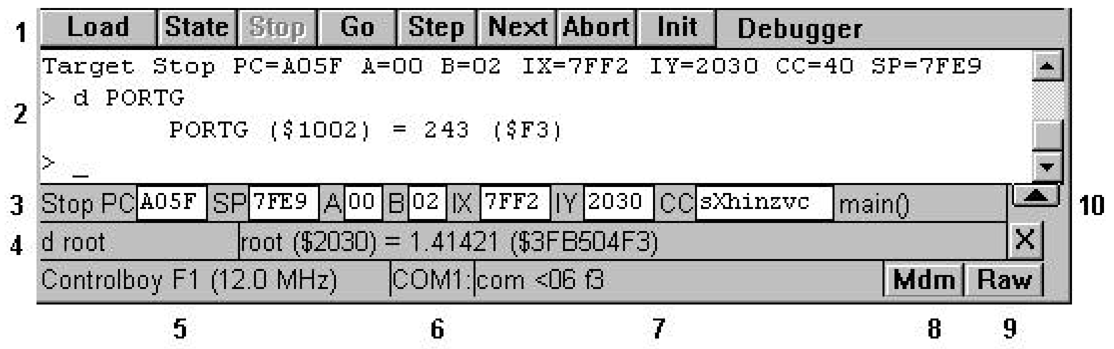
Finally before using the debugger to load and debug the program, the debugger must signal TARGET STOP or TARGET RUNS.

Base Frequency

The crystal chosen for the microprocessor influences lots of other parameters, as you will see below.

Quartz	Bus Frequency	Baud rate	Baud Inittalker
1.000.000	250.000	1200,600,300,150	150
2.000.000	500.000	2400, 1200, 600, 300, 150	300
2.457.600	614.400	19200, 9600, 4800, 2400, 1200, 600, 300, 150	2400
4.000.000	1.000.000	4800, 2400, 1200, 600, 300, 150	600
4.915.200	1.228.800	19200, 9600, 4800, 2400, 1200, 600, 300, 150	4800
8.000.000	2.000.000	9600, 4800, 2400, 1200, 600, 300, 150	1200
9.830.400	2.457.600	19200, 9600, 4800, 2400, 1200, 600, 300	9600
16.000.000	4.000.000	19200, 9600, 4800, 2400, 1200, 600, 300, 150	2400

Debugger



```
Target Stop PC=A05F A=00 B=02 IX=7FF2 IY=2030 CC=40 SP=7FE9
> d PORTG
        PORTG ($1002) = 243 ($F3)
> _
```

```
Stop PC A05F SP 7FE9 A 00 B 02 IX 7FF2 IY 2030 CC sXhinzvc main()
d root                    root ($2030) = 1.41421 ($3FB504F3)
Controlboy F1 (12.0 MHz)  COM1: com <06 f3
```

The debugger establishes the connection from the PC to the target. The debugger runs mainly on the host (PC), but a little part of it, the talker runs on the target.

1. You can enter the most common commands by buttons
2. In this window you enter commands and get the response
3. The state of the target, the registers of the cpu
4. One or several watch line to follow the target memory
5. The configuration of the target. Click on it to change it
6. The communication port. SIM for simulator
7. The communication with the target
8. Modem for remote debugging
9. Raw window for direct communication with the target
10. Two buttons, when available to walk within the stack

Start and Command Line Input

The debugger runs in the bottom window. After the start of the program, it prints the following message

Target does not respond

When you receive this message the debugger could not reach the talker on the board. When you receive the following message

Target runs

there is a program running on the target. While a program is running on the board some commands of the debugger are permitted like display of the memory while others like loading a program are forbidden. By clicking on STOP in the menu or typing STOP in the command line you may force the target into the stop state. The debugger now prints

Target stop

The debugger and the talker on the target are now ready to execute any command. If the program does not stop, you have to start the talker without the application program. Push A and RESET at the same time, release RESET first and A after. If you can not stop the target, read Preparing the software and the target.

The debugger signals by its prompt > in the bottom line that it is ready to execute a command. The debugger keeps the last entered commands. You may access them by using the arrow keys. Some commands continue when entering an empty line. Arguments are hexadecimal numbers without leading $. The debugger keeps the symbols of the program. Instead of a hexadecimal number you may also enter a symbol of the assembly program.

Load and Compare

The following command loads a successfully compiled program into the EEPROM on the board:

```
> load
```

Instead of typing the command on the command line, you may also press the LOAD button in the menu line. The debugger will load the program and also verify the data in the target memory. The debugger loads also the symbol table into its own memory. To verify a program that is already loaded, type

```
> ver
```

To debug a program, that is already loaded, the following command loads only the symbol table

```
> loads
```

Data Output and Input

To display the current state of the target including the registers, press STATE or enter

```
> reg
```

You may change the contents of a register, for example to change the PC to $F800, type

```
> reg PC F800
```

The command D allows you to display variables as they are defined in a high language program. This command accepts wildcards. D * displays local variables of a function. D ** displays all variables. The File "debug.def" keeps names that are always known by the debugger. This file resides either in the current directory or if not found in the directory of the executing program.

```
> d myvar, myvar2
> d my*
> d PORTA
> d *
> d **
```

The following command changes the value of a variable.

```
> PORTA = 11
```

The command M allows you to display the memory in hexadecimal and in ASCII. This command can be used over the total address range including all types of memory and the I/O registers.

```
> m 0
```

displays 64 byte of the RAM at address 0. When entering an empty line, the debugger displays the next 64 bytes of memory.

```
> m 1000 30
```

displays the I/O registers from $1000 to $102F. To examine your program the commands

```
> dis F800              or
> dis start
```

disassemble the memory at address $F800. You see some lines of code in hexadecimal and in assembly language.

The command MS allows you to set memory locations and I/O registers. The following example changes the memory locations at $1000, $1001, and $1002

```
> ms 1000 1 2 3
```

Watch Lines

The Watch command opens a new watch line between the state line and the configuration line. A watch line displays 16 bytes of target memory or a variable in the target.

```
> w 2000
> w TCNT
> w minute
```

These three commands open three new watch lines. In each line, you have at left the command to be executed, in the middle the data output, and at the right a button to delete the line. You may click on the command part to get the most recent data from the target. When the target program stops, by a breakpoint, finishing a single step or due to any other reason, all watch lines will be updated.

Program Start, Breakpoints, and Trace

The command GO starts the program. An optional address may be specified, to set the PC before the program starts.

```
> go F800
```

To debug a program you may set one or more breakpoints.

```
> br temp
```

inserts a breakpoint at the address TEMP. You may even insert a breakpoint in a running program. When the program reaches the breakpoint, it stops execution and passes control to the talker. A machine instruction may have several bytes. You must always set the breakpoint on the first byte. To continue a stopped program, enter GO. The source windows displays breakpoints in the left margin. Clicking on the margin sets or resets a breakpoint. To display all breakpoints in the program, enter

```
> br
```

The following command clears all breakpoints.

```
> brdel
```

The trace mode allows the execution of your program instruction by instruction. The program must be in the stop state. The following commands execute the next machine instruction of your program.

```
> step                    or
> next
```

Both commands are nearly identical except in the case of a subroutine call (BSR or JSR). STEP treats a subroutine call like all other instructions and enters into the subroutine. NEXT executes the subroutine and stops when the subroutine is completed.
When you press SHFT and click on the STEP or the NEXT button, the single step will be repeated until reaching a new high-level source line. When you press ALT and click on the STEP or the NEXT button, the single step will be repeated infinitely. Release SHFT or ALT to stop the repetition.

Theory of Operations of the Debugger and the Talker

How do the debugger main program on the host (PC) and the little talker on the target work together? The debugger does not do an emulation nor a simulation. The application program runs in its own original environment. This requires some co-operation of the program to debug. The talker in the EEPROM of the target works together with the debugger on the host. For example when you enter a command like M to display the memory of the target, the debugger sends a message to the talker. The talker will read out the memory and send the result back to the debugger on the host that displays the data on the screen. All commands work more or less the same, it's the talker that does the real work on the spot.

The talker and the application program therefore have to share the CPU. When the target starts after a RESET, the talker controls the CPU, but passes the control to the application program. The talker now is inactive. If the program reaches a breakpoint, it passes the control by a trap again to the talker. If the host sends a command by the serial line to the board, the talker gains control of the CPU by an interrupt and keeps the control until the command is completely terminated. The talker can therefore work virtually in parallel with the application program. The application may not destroy the resources of the talker like its memory. An application that disables interrupts or changes the configuration of the serial line prevents the talker and therefore the debugger from working.

Breakpoints are implemented using the SWI instruction. This instruction causes a trap that passes control to the talker. The debugger thus changes the application program. The commands NEXT and STEP are likewise implemented.

Interrupts of the Talker

The talker uses the following interrupts:

RESET (FFFE) to start the talker after a RESET.

Illegal Opcode (FFF8).

SWI (FFF6) for breakpoints and traces.

SCI asynchronous serial interface RS232 (FFD6).

Sharing the RS232 between the talker and your program

The talker uses the serial line to communicate with the debugger main program on the host. Your application may nevertheless use also the serial line. When the CPU is interrupted by the serial line, it passes control to the talker. The talker jumps to the RAM address $00FA. An application that wants to treat serial line interrupts must write the address of the interrupt service routine into the RAM at address $00FB, thereby replacing the routine of the talker. The talker thus is completely disabled. There is another possibility to share the RS232 with the talker. When the talker receives a character from the line that it does not recognise as a command, it jumps to the address $00FD with the received character in accumulator A. By writing the address of an interrupt service routine to $00FE your application treats only those data that are not debugger commands.

```
FFD6:           00FA                      EEPROM: Interrupt vector SCI

00FA:           jmp      TalkerInt        RAM: jump to the talker

FE..  Talkerint: Read character           EEPROM Talker
                If character >$06
                         jmp 00FD
                else treated by the talker

00FD            jmp      Talkerend        RAM: jump to the talker

FE.. Takerend:  rti                       EEPROM Talker
```

Raw Window

The raw window allows you to communicate directly with your application program on the board, if your application receives or sends data over the serial line. You may choose among ASCII and HEXADECIMAL character presentation.

Replacement of the Talker in the EEPROM

In the rare case that you have destroyed the talker, you can replace it. In the window of the debugger type:

```
> inittalker
```

and follow the instructions. If you are very limited in EEPROM space, you may use the talker TALKRAM, which resides in the RAM. Then you have to use this procedure regularly.

BRDEL — Delete Breakpoint

BRD[EL]
BRD[EL] <Address>

Delete one or all breakpoints. If no address is specified, all breakpoints will be deleted.

This command can only be executed when the target is in the Stop state.

> brd temp	Delete the breakpoint at address *temp*
> brdel	Delete all breakpoints

BREAK — Insert Breakpoint

BR[EAK]
BR[EAK] <Address>

The command without arguments displays all breakpoints. The command with an address inserts a new breakpoint in the program. When the program reaches this address, it passes automatically the control to the talker. A breakpoint must always be set on the first byte of a machine instruction.

This command can also be executed, while the program is running.

> br temp	Insert breakpoint
> br	Display all breakpoints

D — Display Variables

D	**<Variable> [, <Variable>]**
D	**<pattern>***
D	*****
D	******

Read the memory of the target and display the variables as they are defined.
D * displays all local symbols inside the current function. D ** displays all global and local symbols.

This command can also be executed, while the program is running.

> d minute, second	Display the variables

DIS — Disassemble

D[IS]	
D[IS]	**<Address>**
D[IS]	**<Address> <Length>**

Read the memory of the target and display the data as machine instructions. When entering an empty line, the command continues the display.

This command can also be executed, while the program is running.

> dis loop	Disassemble the program some lines

GO — Start the Program

G[O]	
G[O]	**<Address>**

Start or continue the program. If an address is specified, the PC will be set to the address.

This command can only be executed when the target is in the Stop state.

> go start	Start the program
> g	Continue the program after a breakpoint

INITTALKER — Initialise the Talker

> **INIT**
> **INITTALKER**

Initialise the talker. This command erases the EEPROM totally. This command is only necessary in the rare case that you have destroyed the talker.

> inittalker	Initialise the talker

LOAD — Load Program

L[OAD]

Download the program into the EEPROM on the target. The program must be assembled without errors. The command reads the file <name>.S19, transfers the data into the EEPROM, and verifies the data. The debugger loads the symbol table into its own memory. The PC is set to $F800, the SP to $00E8. All breakpoints are deleted. The command will only load to legal addresses from $F800 to $FE7F and from $FFD6 to $FFFF.

This command can only be executed when the target is in the Stop state.

> load	Load the program
> ver	Compare the program with the file .S19

LOADS

Load Symbols

LOADS
LOADS

Load the symbols of the program into the debugger memory. This command is useful to debug a program that is already running on the target.

This command can be executed, while the program is running. If the target is in the Stop state the command VER should be used instead.

```
> loads                Load the symbol table into the debugger
> m  tbuf  tbufp       Display the data of the program
```

LOG

Logging

LOG	
LOG	**ON**
LOG	**<File name>**
LOG	**ON <File name>**
LOG	**OFF**

Enable or disable the logging of the commands. The log file contains all commands entered at the command line and all displayed data and messages. If no name is specified, the log file is named CBOY.LOG.

This command will be executed independently of the state of the target.

```
> log                  Log into the file CBOY.LOG
> log  off             Stop logging
```

MEM

Display Memory

M[EM]	
M[EM]	**<Address>**
M[EM]	**<Address> <Length>**

Read the memory on the target and display the data in hexadecimal and in ASCII. When entering an empty line, the command continues the display.

This command can also be executed, while the program is running.

```
> m  0                 Display $40 bytes at address 0
>                      And the following $40 bytes
> m  1000  20          Display the I/O registers
```

MFILL

Fill Memory

MF[ILL] <Address> <Length> [=] <Byte>

Fill the memory with data. The second parameter will be interpreted as length or as end address

This command can also be executed, while the program is running.

```
> mf  0  E8 = 00                    Erase the RAM to 0
> mf  F800  FA00  FF                Erase the program to FF
```

MSET

Set memory

MS[ET] <Address> [=] <Byte> [<Byte>]*
MS[ET] -W <Address> [=] <Word> [<Word>]*

Write data in bytes or in 16 bit words into the memory of the target.

This command can also be executed, while the program is running.

```
> ms  102C  33                      Write into the I/O register
> ms  -w  tbufp = tbuf              Initialise the 16 bit pointer tbufp with the value tbuf
```

NEXT

Single Step

NEXT
N

Execute the next machine instruction of the program and stop the execution thereafter. If the instruction is a subroutine call (BSR or JSR), the subroutine will be completely executed and the program will stop at the instruction following the call. When entering an empty line, the command continues the operation. When you press SHFT and click on the the NEXT button, the single step will be repeated until reaching a new high-level source line. When you press ALT and click on the NEXT button, the single step will be repeated infinitely. Release SHFT or ALT to stop the repetition.

This command can only be executed when the target is in the Stop state.

```
> n                                 Execute the next machine instruction
>                                   And the next
```

REG Register

R[EG]
R[EG] A | B | CC | PC | SP | IX | IY [=] <Value>

Display the state and all registers or change the contents of a register. In the Stop state the saved registers will be displayed, else the debugger demands the actual state from the talker.

Changing a register can only be executed when the target is in the Stop state.

> reg	Display the state of the target
> reg PC = start	Change the program counter

STEP Single Step

S
STEP

Execute the next machine instruction of the program and stop the execution thereafter. If the instruction is a subroutine call (BSR or JSR), enter into the subroutine. When entering an empty line, the command continues the operation. When you press SHFT and click on the STEP button, the single step will be repeated until reaching a new high-level source line. When you press ALT and click on the STEP button, the single step will be repeated infinitely. Release SHFT or ALT to stop the repetition.

This command can only be executed when the target is in the Stop state.

> s	Execute the next machine instruction
>	And the next

STOP Stop the Program

STOP

Stop the running program. The debugger sends a command to the talker to stop the running application. The program will be interrupted and can be continued later on by a GO, NEXT, or STEP command.

This command can only be executed, while the program is running.

> stop	Stop the running program

UPLOAD Upload Program

UPLOAD **<Address> <Length> [<filename>]**

Read the target memory and write the data into a file in Motorola S-record format. The second parameter will be interpreted as length or as end address. The filename must be specified the first time using this command. Consecutive uploads without filename will be appended to the file.

This command can also be executed, while the program is running.

```
> upload F800 F900 up.s19          Load the program into the file up.s19
```

VER Verify Program

V[ER]

Compare the program in the EEPROM with the program in the file <name>.S19. This command is useful when debugging a program that is already loaded in the target. The debugger loads the symbol table into its own memory.

This command can only be executed when the target is in the Stop state.

```
> ver                    Compare the program with the file on the disk.
```

WATCH open a Watch Line

W[ATCH] **<Address>**
W[ATCH] **<Variable>**

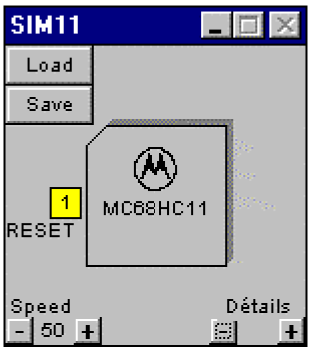
Open a new watch line to display target memory.

```
> w 2000
> w minute
```

In each line, you have at left the command to be executed, in the middle the data output, and at the right a button to delete the line. You may click on the command part to get the most recent data from the target. When the target program stops, by a breakpoint, finishing a single step or due to any other reason, all watch lines will be updated.

Simulator

The simulator allows you to execute your program on a virtual 68HC11 microcontroller.
To replace the real target system by the virtual target system, you have to select the SIM port instead of a COM port in the configuration.
The debugger will then communicate with the simulator instead of communicating with a real target.
The virtual microcontroller will appear like this on your screen:



The virtual microcontroller acts like a real one.
You have to download a talker into its memory, before the debugger can communicate with it.
Fortunately the virtual micro is already delivered with a talker loaded into memory at factory.
You may replace this talker at any time by another talker using the INITTALKER command of the debugger.
Once the talker is loaded, the debugger can communicate with the virtual micro by the virtual serial interface like with any real microcontroller.

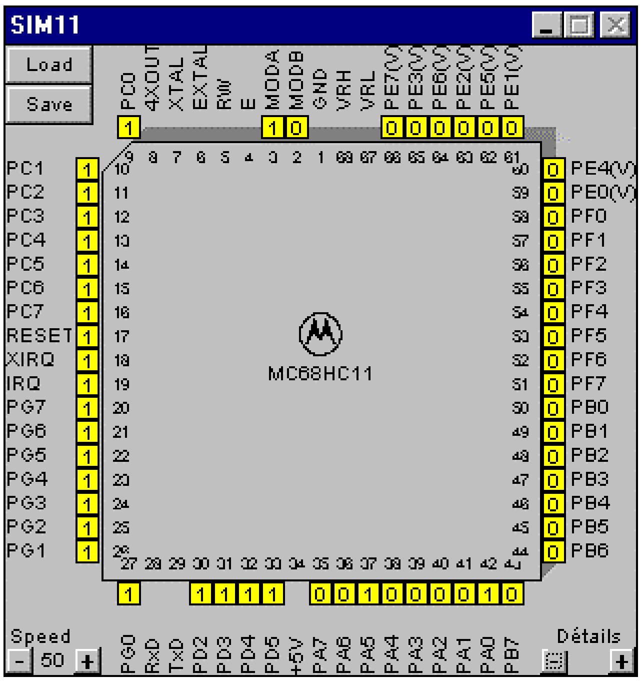
The **yellow button** RESET is the RESET input of the micro. You may click on it to change its state.

The virtual micro runs all the time like a real micro does.
The **Speed** button allows you to select the relative speed of the microcontroller.
At 50%, the simulator takes the half of the cpu time of your P.C., leaving the other half to all other applications running on your P.C.

You can store the state of the virtual micro into a file using the **Save** button. The state includes the memory, the cpu, and the inputs and outputs. Use the **Load** button to reload a saved state.

When launching the simulator, it looks for a file **defaut.sim** in the current working directory and in the BIN directory where the simulator resides. If it finds the file, the simulator loads the state from this file. The installer places a file default.sim into the BIN directory, which includes a talker for the virtual microcontroller.

The Details buttons allow you to see more or less details of the microcontroller.

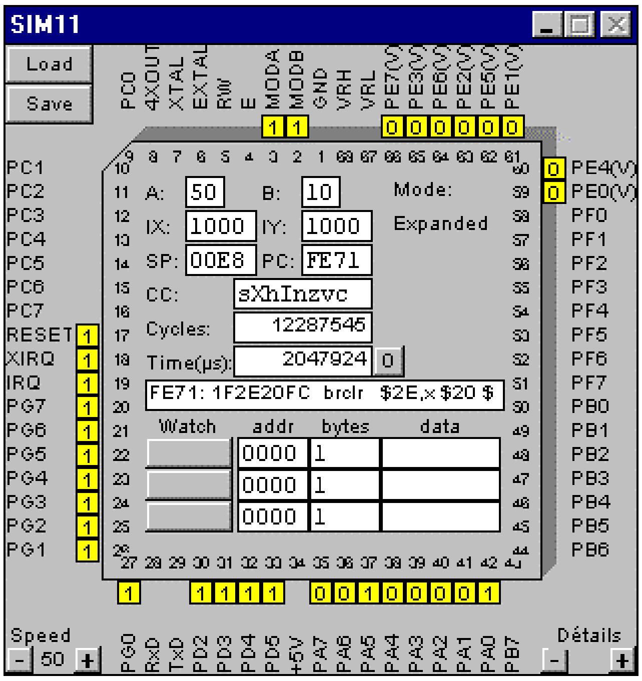


Here you have access to the inputs and outputs of the chip.
Clicking on a digital input toggles its state.
You may also change the state of an analogue input from 0 V to 5 V.
When the program writes to an output, the state changes in the window.
Ports B, C, and F are not available in expanded mode.
You may choose the operating mode by the pins MODA, and MODB and reset the chip using the RESET pin.
The pins IRQ and XIRQ may cause an interrupt.
The pins PD0 and PD1 are used for the communication with the debugger. Your application can not use them, but may send and receive characters by the serial interface (See Debugger, Interrupts of the talker)

Example

Select the port SIM and a 68HC11F1 in the configuration. Open the file flash.c.
Compile the file. Use the debugger to load the file into the virtual chip. Click on GO to start the program.
The program toggles regularly a LED on output port PG0.
If you set the input PG1 to 0, the LED toggles faster.

You may also use the Details button, to open the virtual microcontroller.


SIM11
Load
Save
PC0 4XOUT XTAL EXTAL RW E MODA MODB GND VRH VRL PE7(V) PE3(V) PE6(V) PE2(V) PE5(V) PE1(V)
PC1 PC2 PC3 PC4 PC5 PC6 PC7 RESET XIRQ IRQ PG7 PG6 PG5 PG4 PG3 PG2 PG1
PE4(V) PE0(V) PF0 PF1 PF2 PF3 PF4 PF5 PF6 PF7 PB0 PB1 PB2 PB3 PB4 PB5 PB6
PG0 RxD TxD PD2 PD3 PD4 PD5 +5V PA7 PA6 PA5 PA4 PA3 PA2 PA1 PA0 PB7
Speed
Détails

A: 50 B: 10 Mode: Expanded
IX: 1000 IY: 1000
SP: 00E8 PC: FE71
CC: sXhInzvc
Cycles: 12287546
Time(µs): 2047924 0
FE71: 1F2E20FC brclr $2E,x $20 $

Watch	addr	bytes	data
	0000	1	
	0000	1	
	0000	1	

Speed - 50 +

If the micro is not in the RESET state, and the speed is not at 0, you will see that the cpu runs all the time. When it does not execute an application program, it executes the talker, which is waiting for command on the serial line.
You can the registers of the cpu, the actual mode, and the machine instruction to be executed next.
You see also the cycles, the cpu has executed and the elapsed time. You may set these values to zero.
If you select speed 0, you may execute the program in single step mode.
Do not confuse this single step mode with the single steps of the debugger. Opening and halting a microcontroller is only possible with a virtual chip.

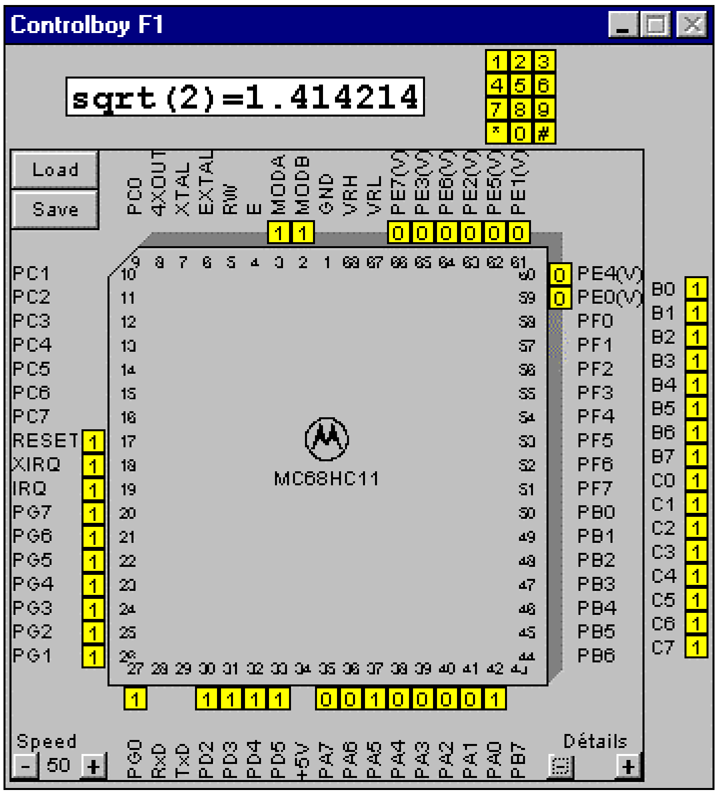
The watch lines allow to display memory and to stop the chip under certain conditions.
Enter the starting address and the size of a memory region.
Select one of the following modes with the left button.

Watch	Display memory (4 bytes maximum)
Break	Stop the cpu, when the PC enters into the region.
Stop Rd	Stop the cpu, after the cpu has read from this region.
Stop Wr	Stop the cpu, after the cpu has written into this region, or in case of an input, when it has changed.
Stop RW	StopRd and StopWr.

If you want to use other circuits around the microcontroller, you must put the chip on a **printed circuit board**. The software calls functions from **board.dll** in the BIN directory.
There are two boards available after the installation:

board0.dll	68HC11 only
boardcf1.dll	Controlboy F1 including a LCD display and a keyboard

If you want to change the board, just copy another file to replace board.dll and restart the software.
Here is the example of CONTROLORD's most popular Controlboy F1 board including a LCD display and a small keyboard:



You may want to write your own board.dll. Sources of the mentioned boards can be found in the BOARDSRC directory. You will obviously need a PC tool to create the dll from your source file.
The source file must include the following functions:

LibMain	Open the DLL
WEP	Close the DLL
BoardOpen()	Open or close the simulator
BoardRead()	Read a byte from the target memory
BoardWrite()	Write a byte to the target memory

Board.dll sends a SIM_PORTCHANGED message to the simulator when an input changed to update the target memory and the watch data.
When you have realized your own board, please send a copy to controlord@controlord.com. We like it.

Realisation

Carriers
MC68HC11A0, A1, E0, E1, MC68HC811E2 all PLCC.
MC68HC11F1FN PLCC.
DIL not realised.

Modes
Boot, single chip, expanded.
Mode Test not realised.
Registers OPTION, BPROT, HPRIO, INIT, CONFIG nor realised.

Memory
64k RAM, unprotected in all modes.
The ROM of the 68HC11A8 without security appears from BF40 to BFFF on mode Boot.
(Motorola M68HC11 Reference Manual Rev 3, B-2)

Resets and Interrupts

Reset	external
XIRQ	realised
IRQ	realised, low level
Illegal Opcode	realised
SWI	realised
SCI	realised for receive data
Real Timer	realised
others	not realised

CPU
Registers and instructions realised as in Motorola M68HC11 Reference Manual Rev 3, chapters 6 and A.

Parallel Input/ Output
Realised for A, B, C, D, E, F, G with their data direction registers.
Not realised: Port B, C handshake, port G chip selects.

Serial Communications Interface SCI
Realised to communicate with the debugger.

Analog-to-Digital Converter
Realised for the register ADR1 only.

Real Timer
Realised.

Serial Peripheral Interface SPI, Timing System, Pulse Accumulator
Not realised.