M68hc11 gcc's tools chain, table of Contents

M68hc11 gcc's tools chain

1. Introduction

This document explains how to use the GNU development chain for the embedded Motorola 68HC11 and 68HC12 microcontrollers. The tools that come with it allow you to create applications written in C, C++ or assembly language.

1.1 Content

The GNU development chain is composed of the following components:

- A C and C++ compiler
- An assembler, a linker and a collection of tools to manipulate object files.
- A source level debugger
- A simulator
- An ANSI C library, a mathematical library and a board support package library.

1.3 Installed Tree

The GNU development chain exports four important directories. These directories contain the compiler binaries, the libraries, the includes and the full documentation set.

`bin`
: This directory contains the executables of the GNU development chain. It contains the compiler driver (gcc), the assembler (gas), the linker (ld) and other tools.

`lib`
: This directory contains the 68HC11 and 68HC12 runtime libraries. It contains the standard C library (libc.a), the math library (libm.a) and board support library (libbsp.a).

`doc`
: This directory contains the documentation of the tool chain. It contains the five documentation sets in HTML form for the preprocessor, compiler, assembler, linker, debugger and the utilities.

`include`
: This directory contains the standard C header files.

1.4 Starting

Once you have installed the tool chain, it is recommended to read and play with the 2. Hello World Example tutorial. It explains how you can build a simple application and run it on the simulator, in the debugger or on your board.

2. Hello World Example

This chapter illustrates through an example how you can build a 68HC11 bootstrap program, compile it, link it and have it run on the target board, in the GNU debugger or in the simulator.

2.1 Overview

This example is a simple hello program written in C that prints `Hello World!` on the 68HC11 serial line. It implements a print function which controls the serial line to send the message and it defines the main to print the hello message.

The source comes with the GNU Embedded Libraries package in the `` `hello/hello.c' `` file.

To compile this program, you can use the following command:

```
m6811-elf-gcc -g -Os -mshort -Wl,-m,m68hc11elfb -o hello.elf hello.c
```

This will produce the `` `hello.elf' `` file. This file in a binary file in the ELF/DWARF-2 standard format. It contains the 68HC11 binary code, the symbols and the debugging information suitable for the GNU debugger. To obtain the Motorola S19 file corresponding to the program, you can use the following command:

```
m6811-elf-objcopy --only-section=.text --only-section=.rodata \
                  --only-section=.vectors --only-section=.data \
                  --output-target=srec hello.elf hello.s19
```

and this will produce `` `hello.s19' `` file in Motorola S19 format.

To run or test this program, you can use the simulator or gdb:

```
m6811-elf-run hello.elf
```

The program will print the following message:

```
Hello world!
```

2.2 Source Description

This section describes step by step the different parts of the `` `hello.c' `` example.

2.2.1 68HC11 Definitions

The hello example writes a message on the serial line. To do this we have to control the 68HC11 SCI and we must access the I/O ports and control them.

A first part defines several flags and values that represent the 68HC11 SIO registers. The complete list of flags are available when you include the `` `<sys/ports.h>' `` file and it is recommended to use the following in your own programs:

```
#include <sys/ports.h>
```

The flags are defined in `<asm-m68hc11/ports_def_F.h>', `<asm-m68hc11/ports_def_E.h>' or `<asm-m68hc12/ports_def.h>' depending on the microcontroller.

Below is the list of flags used by the hello example (several SCI flags have been removed for clarity):

```
#define M6811_BAUD      0x2B    /* SCI Baud register */
#define M6811_SCCR1     0x2C    /* SCI Control register 1 */
#define M6811_SCCR2     0x2D    /* SCI Control register 2 */
#define M6811_SCSR      0x2E    /* SCI Status register */
#define M6811_SCDR      0x2F    /* SCI Data (Read => RDR, Write => TDR) */

/* Flags of the SCCR2 register.  */
#define M6811_TE        0x08    /* Transmit Enable */
#define M6811_RE        0x04    /* Receive Enable */

/* Flags of the SCSR register.  */
#define M6811_TDRE      0x80    /* Transmit Data Register Empty */

/* Flags of the BAUD register.  */
#define M6811_SCP1      0x20    /* SCI Baud rate prescaler select */
#define M6811_SCP0      0x10
#define M6811_SCR2      0x04    /* SCI Baud rate select */
#define M6811_SCR1      0x02
#define M6811_SCR0      0x01

#define M6811_BAUD_DIV_1        (0)
#define M6811_BAUD_DIV_3        (M6811_SCP0)
#define M6811_BAUD_DIV_4        (M6811_SCP1)
#define M6811_BAUD_DIV_13       (M6811_SCP1|M6811_SCP0)

#define M6811_DEF_BAUD M6811_BAUD_DIV_4 /* 1200 baud */
```

The 68HC11 SCI registers can be accessed in C by reading and writing the memory. In the program, the I/O ports are represented by an array of bytes which is mapped at a given fixed address. This array starts at beginning of the I/O register map. It is declared as extern so that the address is defined at link time.

```
extern volatile unsigned char _io_ports[];
```

The volatile keyword is important as it tells GCC to avoid any optimisation when reading the memory. This is necessary otherwise GCC would simply remove the busy wait loop to detect that the transmitter is ready.

For example to write the `SCCR2` SCI register we will do the following:

```
_io_ports[M6811_SCCR2] = M6811_TE;
```

whereas to read the `SCCR1` SCI register we will do the following:

```
unsigned char flags = _io_ports[M6811_SCSR];
```

2.2.2 68HC11 SIO Operations

A second part defines some functions to write characters on the SIO. They access the IO register through the `_io_ports' global variable.

These operations can be used in your program by using the following include:

```
#include <sys/sio.h>
```

A first operation named `serial_send' is defined to send the character on the serial line. This function is defined as follows:

```
static inline void
serial_send (char c)
{
  /* Wait until the SIO has finished to send the character.  */
  while (!(_io_ports[M6811_SCSR] & M6811_TDRE))
    continue;

  _io_ports[M6811_SCDR] = c;
  _io_ports[M6811_SCCR2] |= M6811_TE;
}
```

It sends the character `c' on the serial line. Before sending, it waits for the transmitter to be ready. Once the function returns, the character is in the SCI queue and it may not be sent completely over the serial line.

A second function called `serial_print' uses `serial_send' to send a complete string over the serial line. It iterates over the character string and send each of them until the end of the string represented by a `0' is reached.

```
void
serial_print (const char *msg)
{
  while (*msg != 0)
    serial_send (*msg++);
}
```

2.2.3 Hello World

The last part represents the main function. In C and C++, the main function must be called `main'. When this `main' function is entered, the stack pointer is initialized, the global variables are initialized but the SIO as well as other 68HC11 devices are not yet initialized. In this example, we first configure the SIO by setting the baud rate and the character format. The SIO must then be started by enabling the transmitter and receiver.

The SIO initialization is inlined here for clarity but it is implemented by the `serial_init' operation available when the `<sys/sio.h>' include file is used.

Once the SIO is initialized, we can write messages using `serial_print'.

```
int
main ()
{
  /* Configure the SCI to send at M6811_DEF_BAUD baud.  */
```

```
  _io_ports[M6811_BAUD] = M6811_DEF_BAUD;

  /* Setup character format 1 start, 8-bits, 1 stop.  */
  _io_ports[M6811_SCCR1] = 0;

  /* Enable receiver and transmitter.  */
  _io_ports[M6811_SCCR2] = M6811_TE | M6811_RE;

  serial_print ("Hello world!\n");
  return 0;
}
```

In this example, the `main' function returns and this will give back control to the startup code which will in turn call `exit'. The `exit' will loop forever arround a `wai` instruction. Indeed, a real 68HC11 hardware has no way to exit!

2.3 Compile and Link

Now that we have the C source file, it's necessary to compile, assemble and link it. The compilation is the process by which the 68HC11 instructions are generated from the `hello.c' source file. The assembling is the process which transforms the 68HC11 instructions into the 68HC11 binary code. The link is the final process which collects together all binary files, organizes them in memory and assigns them final addresses. These processes can be separated or run in a single command.

To compile the program, you will use the `m6811-elf-gcc' driver. This driver invokes the pre-processor, the compiler, the assembler and the linker depending on options and files that you give as arguments. For the example, you will issue the following command:

```
m6811-elf-gcc -g -Os -mshort -Wl,-m,m68hc11elfb -o hello.elf hello.c
```

The driver uses the file extension to decide what must be done for a given file. For `hello.c' file it will use the C compiler. We use the following options to control the compilation and assembling passes:

`-g`
: Generate symbolic debugging information

`-Os`
: Optimize for speed and space

`-mshort`
: Use 16-bit for integers (int type)

Once the `hello.c' file is compiled and assembled, the driver will run the linker. The link pass is a non-trivial process although it has been simplified. One important role of the linker is to map the program in memory and assign each symbol an address. To tell the linker where it must put the program we use the `memory.x' file to describe the memory. This file is used by the linker when we use the `-Wl,-m,m68hc11elfb' option. It contains the following definitions:

```
MEMORY
{
  page0 (rwx) : ORIGIN = 0x0, LENGTH = 256
  text  (rx)  : ORIGIN = 0x0, LENGTH = 256
  data        : ORIGIN = 0x0, LENGTH = 0
}
```

The `MEMORY' part describes the memory regions that the board provides. The example uses the 68HC11 bootstrap mode as this is available for most target boards. It does not depend on the RAM and ROM characteristics of the target. In the bootstrap mode, we can rely only on the 68HC11 internal RAM mapped at address 0. The important definition for this is the `text' region. It indicates that it starts at 0 and can contain 256 bytes. The `rx' flags indicate that this `text' region is both readable (`r') and executable (`x'). The `text' region is used by the linker to put the program code in it. The `data' region is used by the linker to put the global and static variables. In this example, we defined this region as empty so that we can detect problems in using the bootstrap mode.

The `memory.x' file also defines the following part:

```
PROVIDE (_stack = 0x0100-1);
PROVIDE (_io_ports = 0x1000);
```

These two definitions create two symbols named `_stack' and `_io_ports' and assign them a value. The `_stack' symbol represents the top of the stack and the `_io_ports' symbol represents the 68HC11 I/O registers that we used in the program. The I/O registers are mapped at address `0x1000' by default.

For the link pass, the following options are important:

`-Wl,-m,m68hc11elfb`
: Tell the linker to use a link configuration based on the user specific `memory.x' file.

`-o hello.elf`
: Tell the linker to use `hello.elf' as the output file name.

`-mshort`
: Use 16-bit for integers (int type). This is not really used by the linker but necessary for a final link so that the `m6811-elf-gcc' gives to the linker the correct startup code and the libraries compatible with the program.

2.4 Running The Example

Now that the hello program is compiled and linked, you want to test and have it run somewhere. There are several ways to run the example:

- By using the simulator
- By using GNU Debugger GDB
- By downloading the program to some 68HC11 board in bootstrap mode.

2.4.1 Simulator

The simulator is the quickest and easiest way to test the example. You don't need to have a target board. Just type the following command:

```
m6811-elf-run hello.elf
```

The simulator creates a virtual 68HC11 running at 8Mhz and simulates all its instructions with the internal devices. The program prints the following:

```
Hello world!
```

2.4.2 GNU Debugger

The GNU Debugger can be used to test and debug the example. It provides the simulator and allows debugging at source level with it. The process is more complex but remains quite easy. Start the debugger as follows:

```
m6811-elf-gdb hello.elf
```

Then, you must tell GDB to connect to the simulator. This is done by using the `target sim' command:

```
(gdb) target sim
```

The simulator is ready and the program must be loaded in memory. This is done by using the `load' command as follows:

```
(gdb) load
```

We can now execute the program by starting the simulator:

```
(gdb) run
```

The simulator print the hello message in the GDB output console.

2.4.3 68HC11 Board

Using a real 68HC11 board is a little bit more complex. First you must configure your board to start with the 68HC11 bootstrap mode.

Now, you must connect the 68HC11 board and your host computer with a serial cable. The serial line must be configured at 1200 baud, no parity and one stop bit. It must not use the RTS/CTS control flow.

To upload the program you will use a program loader on the host computer. Some loaders are able to use the `hello.elf' ELF file directly while some others can only read the `hello.s19' file.

```
loader -d /dev/ttyS0 hello.elf
```

3. Compiler

The GNU Compiler Collection integrates several compilers for many architectures. The C and C++ compilers have been ported for 68HC11 and 68HC12 micro-controllers.

This chapter gives some information about how the compiler implements several aspects of the C and C++ languages. The reader is supposed to be familiar with these programming languages (see section 7. References to obtain information or books about those languages).

The complete GCC documentation is available in The GNU Compiler Collection.

3.1 Supported Data Types

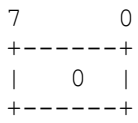
The compiler supports all the data types defined in ISO/IEC 9899:1990. The table below gives information about each type and how they are passed in function calls.

char
unsigned char
signed char

Characters are 8-bit entities. They are unsigned by default. The `char` type can be signed when you use the `` `-fsigned-char' `` option.

They are passed as parameters on the stack as 16-bit values. They may be stored in the `B`, `X`, `Y` registers. Most arithmetic operations are generated inline.

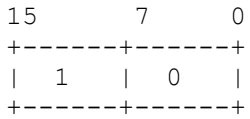
A function returns it in register `B`.



short
unsigned short

Shorts are 16-bit entities. They are passed as parameters on the stack as 16-bit values. They may be stored in the `D`, `X`, `Y` registers. Most arithmetic operations are generated inline (except mult, div and mod).

A function returns it in register `D`.



int
unsigned int

Integers are either 16 or 32-bit entities. They are 32-bit by default. The `` `-mshort' `` option turns on the 16-bit integer mode. The parameter passing rule applies to either short or long depending on its size.

void*
pointer

A pointer is a 16-bit entity. The parameter passing rules are the same as an unsigned short.

A pointer can point to a far function (that is a function which is in a paged memory). The pointer will in fact point to a trampoline code whose address is not in paged memory.

long
unsigned long

Long integers are 32-bit entities. They are passed as parameters on the stack as 32-bit values. They may be stored in the `D+X` register: the low-part in `D` and the high-part in `X`. Logical operations are generated inline, some addition and subtraction are inline. Other arithmetic operations are made by a library call. Comparison are inline.

A function returns it in register `D` and `X`.

```
long
31      23      15      7     0
+------+------+------+------+
|  3   |  2   |  1   |  0   |
+------+------+------+------+
```

float

Floats are IEEE 32-bit floating point numbers. They are treated like longs for copies, parameter passing rule and register allocation. Most/all of the operations are made by a library call.

A function returns it in register `D` and `X`.

long long
unsigned long long

Long long are 64-bit entities. They are never allocated in a hard register. They are passed on the stack for operations. They are returned like a struct (in memory). Logical operations (and, or, xor) are generated inline. Other operations are made by a library call. Some shift operations are generated inline.

```
63      55      47      39      31      23      15      7     0
+------+------+------+------+------+------+------+------+
|  7   |  6   |  5   |  4   |  3   |  2   |  1   |  0   |
+------+------+------+------+------+------+------+------+
```

double

Double are IEEE 64-bit floating point numbers. They are treated like `long long` for copies and parameter passing rule. All operations are made by a library call.

The `` `-fshort-double' `` option changes the definition of `` `double' `` to use IEEE 32-bit floating point numbers.

3.2 Register Allocation

The 68HC11/68HC12 registers (`D`, `X` and `Y`) are completely managed by GCC. Local variables can be stored in these registers. However, since the 68HC11 and 68HC12 have so few registers, soft-registers are used to simulate hard registers for GCC (this avoids register spill failure in the reload pass). There are two kinds of soft-registers:

- A set of soft registers which are replaced during the machine specific reorganization into a real hard register. At present, this set of soft registers is only composed of a single register called `Z`. This register is treated like `X` or `Y` register by GCC. It is replaced by `X` or `Y` and sometimes `D`. When it is replaced, the old `X/Y/D` value is saved and restored if this is necessary. In general, the `Z` register is allocated during the reload pass for the reload of an address. This `Z` register helps significantly the reload pass and contributes in the generation of better code. It is possible to disable the `Z` register by using the `` `-ffixed-z' `` option (in general and depending on the program, this will result in register spill failure...).
- A set of general soft registers which are located in `` `.softregs' `` section (mapped in `` `page0' `` or `` `data' `` memory bank). These registers are not replaced during the machine reorganization. There are 4 soft registers by default for 68HC11 and none for 68HC12. This is configured by using the `` `-msoft-reg-count=`*`cnt`*`' ``. The minimum depends on your program, and the maximum is 32.

  Note: It is not clear whether having a lot of soft-registers produces optimal code. For small functions, reducing the number of soft registers is a good idea. For large functions, increasing to 8 or 12 helps in generating a smaller and faster code.

3.3 Register Usage

The table below indicates how the compiler uses the given registers.

Register	Description
`A`	The A and B register are not directly used by GCC. Some patterns generate instructions that use these registers to generate specific instructions such as add with a carry (used in 32-bit add).
`B`	
`D`	This register is used for most arithmetic operation. It holds 8 or 16-bit values. It is also used to hold the low-part of a 32-bit value (either a long or a float). In a function call, it is used to pass the first argument of that function. If the function returns a struct or a 64-bit value, the result will be returned in memory. The caller passes a pointer to that memory location in this register. This register is also used for the result of functions.
`X`	This register is used for indexed addressing. It holds 8 or 16-bit values. It is also used to hold the high-part of a 32-bit value (either a long or a float). When the first function parameter is 32-bit wide, it is used to pass the high-part of the 32-bit value. It is used for the result of functions when the result does not fit in a 16-bit value. This includes long, float as well as small structures (`` `-fno-pcc-struct-return' ``).
`Y`	This register is used for indexed addressing. It holds 8 or 16-bit values.
`Z`	This register is used for indexed addressing. It is treated like `X` or `Y` by GCC. It is replaced by either `X` or `Y` during the machine reorganization. When it must be saved, it is saved in

`_.z' (see below).

`_.z` — This is a 16-bit soft-register in `.page0' that is used during the `Z` register replacement to save the `Z` register. It is possible to eliminate the use of that register by passing the `-ffixed-z' option. The program may however not compile in some cases.

`_.xy` — This is a 16-bit soft-register in `.page0' that is used during the `Z` register replacement to save the old content of either `X` or `Y`. It is possible to eliminate the use of that register by passing the `-ffixed-z' option. The program may however not compile in some cases.

`_.frame` — This is a 16-bit soft-register in `.page0' that represents the frame pointer. It is possible to eliminate the use of that register by passing the `-fomit-frame-pointer' flag. But in that case, debugging becomes difficult.

`_.tmp` — This is a 16-bit soft-register in `.page0' that is used internally by the machine description. It is not available to GCC. It is used in some cases where the machine description needs a memory location to copy some hard registers (reg <-> reg copy). It is not possible to eliminate the use of that register.

`_.d1`..`_.d32` — These are the 32 soft-registers in `.page0'. Each of them is 16-bit wide. Consecutive registers may be allocated to store long as well as long long values. The use of these registers is controlled by the `-msoft-reg-count=*n*' option.

GCC assumes that the following registers are clobbered by any function call:

```
D, X, Y, Z
```

The soft registers in `.page0' have a name which cannot be specified in C and C++. No conflict can therefore arise with a program global variable or function. However, if you want to access those registers from a C or C++ function, do the following declaration:

```
extern unsigned short d1 __asm__("_.d1");
```

Such declaration tells GCC that the external variable `d1` corresponds to the assembly symbol `_.d1`.

3.4 Traps and Interrupts

GCC for 68HC11 and 68HC12 supports the generation of trap and interrupt handlers. The trap handler correspond to either the `swi` exception handler or to the invalid opcode handler. The difference between the trap and interrupt handlers are that the former is a synchronous exception while the later is asynchronous. Trap and interrupt handlers are specified by using the GNU extension `__attribute__'.

3.4.1 Interrupts

To define an interrupt handler, you must declare the prototype of the interrupt handler as follows:

```
void my_interrupt_handler(void) __attribute__((interrupt));
```

Then, you must define your interrupt handler as follows:

```
void my_interrupt_handler(void)
{
}
```

The prologue of the interrupt handler saves the GCC soft registers _.tmp, _.xy and _.z. Other soft

registers need not to be saved (unless they are used in the interrupt handler). The epilogue restores these registers and uses `rti` to return from interrupt.

Note: You are responsible for installing the interrupt handler in the 68HC11/68HC12 vectors table.

Bugs: You can define an interrupt handler which has parameters and which returns some value. Such invalid specification will be forbidden later.

3.4.2 Traps

The trap handler is defined in the same manner except that you can pass parameters and it can return a value. The trap handler follows exactly the same parameter passing rules as a normal function. To define some generic system call handler, you can define for example:

```
int syscall(int op, ...) __attribute__((trap));
int syscall(int op, ...)
{
  int result;
  ...
  return result;
}
```

The prologue of the trap does not save the GCC soft registers _.tmp, _.xy and _.x as the interrupt handler does. This is because the trap is synchronous and its invocation is treated like a function call by GCC.

The epilogue saves the result on the stack so that the `rti` instruction used to return pops the correct result.

To invoke a trap handler, just call the trap handler, for example:

```
int result = syscall(1, "Hello World!", 12);
```

A `swi` instruction will be generated instead of a `bsr`.

Note: You are responsible for installing the trap handler in the 68HC11 or 68HC12 vector's table. If you define several trap handlers, you must be careful in switching the swi vector.

Limitation: You can define a trap handler for the illegal opcode but there is no way (yet) to tell GCC to generate the illegal opcode to invoke the trap handler.

3.5 GCC CPU Target

GCC has three options `` `-m68hc11' ``, `` `-m68hc12' `` and `` `-m68hcs12' `` which control what is the target micro-controller. These options are passed to the assembler and they also control the linker. When none of these options are specified, GCC uses the default CPU target that you specified during the configuration. For the binary installations (RPMs and Windows) the default CPU is the 68HC11.

When compiling, assembling and linking you must make sure to pass the same cpu target option. If you fail, you will get a linker error since 68HC11 and 68HC12 object files have different ELF magic numbers

4. Linker

The linker comes with the GNU binutils package. It is responsible for collecting the object files together, map the code in memory and create the final executable. This chapter presents some details about using the GNU linker for 68HC11 and 68HC12. The GNU ld documentation is probably more complete. The GNU binutils contain the assembler, linker, archiver and many other utilities. These utilities use various format for object files including ELF-32, Motorola S-records, Intel HEX records, and others.

4.1 Object Files

Gas generates object files in the ELF format. The specific tags `EM_68HC11` (70) and `EM_68HC12` (53) are used to identify the machine. These tags are defined by the `` `Motorola Embedded Working Group'. `` Gas generates the following relocation types:

- 8-bit PCREL
- 8-bit absolute
- 8-bit low address part (no complain on overflow)
- 8-bit high address part
- 16-bit PCREL
- 16-bit absolute
- branch instruction marks

4.2 Sections

This section presents the important ELF sections that an object file can contain. Each of them play an important role for building the final application.

`.text`
: This is the standard text section for code program. It contains the 68HC11 or 68HC12 binary code. By default, the linker places the text at `0x8000`. This can be overridden with `` `-Ttext' `` option or better by specifying a `text` memory region.

`.rodata`
: This is the standard read-only data section. It is placed after the text section by the linker. The compiler uses this section to store constant values like strings, jump tables.

`.data`
: This is the standard data section. It is used for initialized global and static variables.

`.bss`
: This is the standard bss section. It appears after the data section. It is used for non-initialized global and static variables.

`.page0`
: This is a specific section for the 68HC11 and 68HC12 to represent the page 0. It is intended to be used to help in specifying data which are defined in the first 256-bytes (page 0) of the address space. It is placed at address 0 by the linker. It is treated as a .data section.

`.eeprom`
: This is a specific section for the 68HC11 and 68HC12 to represent the internal EEPROM.

`.install[0-4]`
: These sections represent some initialization sections. They are used by the GCC startup code. Some of them are for used by applications. During the final link, these sections are merged with the `` `.text' `` section, and are put at beginning of the program. Therefore, they appear at beginning of ROM area. The `` `.install0' `` is reserved. It initializes the stack pointer. The `` `.install1' `` is a placeholder for applications. The `` `.install2' `` is reserved. It initializes the bss and data sections. The `` `.install3' `` is a placeholder for applications. The `` `.install4' `` is reserved and invokes the main.

`.fini[0-4]`
: These sections represent some termination sections. They are used by the GCC finish code (`` `exit' ``). Some of them are for use by applications. During the final link, these sections are merged with the `` `.text' `` section. `` `.fini0' `` and `` `.fini4' `` are reserved. `` `.fini1' `` is a placeholder for applications. It can contain code which is executed during exit and before the C++ static destructors. `` `.fini2' `` is reserved and corresponds to the C++ static destructors. `` `.fini3' `` is a placeholder for applications. This code is executed after the C++ static destructors are called. The `` `.fini4' `` is reserved and corresponds to the runtime exit.

`.vectors`
: This is a specific section for the 68HC1x to represent the vectors table. By default the vectors table is located at `0xffc0` and placed at that address by the linker. By using the `` `-defsym vectors_addr=*addr*' `` linker option, it is possible to set another address for the vectors table. For example, `` `-defsym vectors_addr=0xbfc0' `` sets the vectors table at `0xbfc0`.

`.softregs`
: This is a specific section for the 68HC11 and 68HC12 to hold the gcc soft registers. For 68HC11, this section is put during the final link in the `` `.page0' `` section. For 68HC12, it is put in the `` `.bss' `` section.

`others`
: Several standard sections exists for DWARF-2 debugging info, stack frame unwinding description and so on.

4.3 Linker Memory Banks

The Linker contains built-in linker scripts which indicate how to perform the link and where to put the different sections. You can use the provided linker scripts or write yours. In general, you will want to give the linker some information about the ROM and RAM you have on your board. The Linker

defines two emulations which allow you to use one

The `m68hc11elf' and `m68hc12elf' emulations define the memory as follows:

```
MEMORY
{
  page0 (rwx) : ORIGIN = 0x0, LENGTH = 256
  text  (rx)  : ORIGIN = 0x08000, LENGTH = 0x8000
  data        : ORIGIN = 0x01100, LENGTH = 0x6F00
  eeprom      : ORIGIN = 0xb600, LENGTH = 512
}
PROVIDE (_stack = 0x1100 + 0x6f00 - 1);
```

which means that there are two RAM banks, one at `[0x0..0xff]` and one at `[0x1100..0x7fff]`. There is also a ROM bank at `[0x8000..0xffff]`. The `PROVIDE` instruction defines the address of the top of the stack to the top of the RAM. This memory configuration is suitable for the 68HC11/68HC12 simulator. It was defined like this to be able to run the GCC test-suite.

The `m68hc11elfb' and `m68hc12elfb' emulations are the same except that they include a file `memory.x' that you must provide in the current directory or in one of the `-L' directories. The file you have to write must contain the above `MEMORY` definition. You can change the address and sizes of the different memory banks. You must also define the address of the top of the stack.

To select a particular linker emulation, use the `-m emulation' option. If you use `m6811-elf-gcc` or `m6812-elf-gcc`, you must pass the emulation through the `-Xlinker' or `-Wl' option. For example:

```
m6811-elf-gcc -Wl,-m,m68hc11elfb -o tst.out tst.o
```

When you specify a linker emulation through the `-Wl,-m' option, you must be careful to use the emulation that corresponds to your CPU target. If you compiled your program with `-m68hc12` or `-m68hcs12` you should use the `m68hc12elfb' emulation.

4.4 Ld Symbols

The Linker uses and provides some symbols that you can use in your program.

`_start`
: This symbol represents the entry point of the program. It is provided by the GCC startup file and it is used by the linker to know the entry point of the program. The linker script is configured to try to put this symbol at beginning of the ROM area.

`_stack`
: This symbol represents the top of the stack. It is computed by the linker to refer to the top of the data memory bank. The GCC startup file sets the stack pointer to this value at beginning.

`etext`
: This symbol represents the end address of the `.text' section.

`edata`
: This symbol represents the end address of the `.data' section.

`__data_image`
: This symbol represents the starting address of the ROM area which contains the copy of initialized data. These data must be copied from ROM to RAM during initialization. This is done by the GCC startup file.

`__data_image_end`
: This symbol is: `` `__data_image + __data_section_size' ``.

`__data_section_start`
: This symbol represents the starting address of the data section in RAM.

  This symbol should rather be named: `` `__data_start' ``.

`__data_section_size`
: This symbol represents the size in bytes of the data section that must be copied from ROM to RAM.

  This symbol should rather be named: `` `__data_size' ``.

`__bss_start`
: This symbol represents the starting address of the BSS section in RAM. This is also the end of the data section.

`__bss_size`
: This symbol represents the size in bytes of the BSS section. Together with `` `__bss_start' `` it is used to initialize the BSS section at startup time.

`_end`
: This symbol represents the ending address of the BSS section in RAM.

`_vectors_addr`
: This symbol represents the starting address of the 68HC11/68HC12 vector's table. The default value is `0xffc0`. If you have provided a `` `-defsym vectors_addr=addr' `` option to the linker, this symbol will have the value you have specified.

  Note: If you provide a `` `vectors_addr' `` symbol, there will be two symbols: one with _ and one without.

5. Optimization Hints

This chapter gives advices and hints to optimize your application.

5.1 Using `inline`

The GNU compiler can inline functions and methods. When a function is inlined, its code is inserted directly at the place where it is used. This optimization saves a function call and return. In some cases,

the compiler can even optimize further the inlined function just because it knows what are the parameters of the function. This optimization is done automatically when the compiler is invoked with `-O3' option. However the use of `-O3' may result in larger code as the compiler can inline small and larger functions.

In many cases a better result is obtained by specifying which function can be inline. The inline optimization can still be performed at `-O1' or `-Os'. You should follow the following rules:

- The function should be declared with the `inline' keyword.
- It is sometimes good practice to also declare the function `extern' or `static'. Even if a function is inline, the compiler will emit the code for that function even if it has inline it somewhere. When a function is `static' or `extern', the compiler has more information as to where and how the function can be used and it will not emit a body for it. You should use `static' when your function is used and declared in a single `.c` or `.C` file. You should use `extern' when you declared and implemented your function in some header file.
- The function body must be known to the compiler at the place it is used. If the function is defined in a first file and used in a second file, the compiler will not be able to inline the function. In most cases, an inline function should be defined in some include file that is included whenever the function is used.
- The compiler will inline functions only when it is optimizing the code.

Below is an example of a function that can be inlined by the compiler.

```
static int inline mean(int a, int b)
{
   return (a + b) / 2;
}
int main()
{
   extern int a, b;

   int result = mean(10, 20);
   printf("Result = %d\n", result);
   result = mean(a, b);
   printf("Result = %d\n", result);
   return 0;
}
```

The first call to `mean' will generate no code: the compiler knows the arguments and since they are constant it will compute itself the result (15). The second call depends on global variables whose values are not known at compile time. However, the compiler will directly use those variables and does not need to push them on the stack, call the function and so on.

5.2 Frame pointer

The frame pointer is used to access local variables as well as function arguments. For 68HC11 and 68HC12, the frame pointer is the address of the beginning of local variables on the stack. When a function is called, a new frame pointer is computed at beginning of the function. The old frame pointer is saved on the stack.

The frame pointer can be eliminated by the compiler with the `-fomit-frame-pointer' option. When it is eliminated it is replaced by the stack pointer plus an offset that depends on the current stack depth.

Elimination of the frame pointer provides better performance, reduces the generated code but it makes debugging with Gdb much harder. Without the frame pointer, Gdb is unable to compute the caller frame and print the stack backtrace.

The frame pointer is not eliminable in some conditions. In general it can't be eliminated when the function uses the builtin alloca or when it allocates a local variable whose size is not known at compile time.

6. Using Memory Bank Window

This chapter presents the memory bank switching support that the GNU Development Tools support for both 68HC11 and 68HC12.

6.1 Why Using Memory Bank Window

The 68HC11 and 68HC12 use 16-bit addresses to access the memory, thus limiting the address space to 64K. To access to more than 64K of memory, it is necessary to use a mechanism that will map the memory above 64K in the 16-bit address space. Such mechanism is provided by the 68HC12 through its memory bank switching mechanism.

As far as the 68HC11 is concerned, an external hardware circuitry is necessary to provide such memory bank switching. This mechanism is therefore board-specific and the GNU compiler uses a few support routines to help in generation of calls accross memory banks.

The memory bank switching is supported in a transparent manner only for the code. It can be used for the data but requires application support for this.

6.2 Using Memory Bank Window in GCC

To use the memory bank switching mechanism there are several things to take care of:

- It is necessary to use a specific calling convention which takes care of saving, setting and restoring the memory page. For 68HC12, this calling convention is based on the `call' and `rtc' instructions. For 68HC11, this must be implemented by board specific operations which more or less simulate the `call' and `rtc'. This step is handled by the compiler.
- Functions that use the memory bank switching calling convention should be dispatched in one or several memory regions to actually put them in different pages. This step is handled by the linker.

6.2.1 Function Declaration

To control the calling convention of a function or method, the GNU compiler uses function attributes. There are at least two function attributes which are important:

- The `far' and `near' function attributes specify which calling convention the function is using. A `far' function uses the `call' and `rtc' calling conventions which means that it will take care of memory bank switching during the call.
- The `section' function attributes is recommended (but not mandatory) to control in which memory bank the function will be put at link time.

The following declaration:

```
extern void __attribute__((far)) foo (int);
```

indicates that the operation `foo' uses the memory bank switching mechanism and uses `rtc` to return.

The following declaration:

```
extern void __attribute__((near)) bar (int);
```

indicates that the operation `bar' uses the normal calling convention.

6.2.2 Function Call

When a `far' function is called, the compiler uses a `call` instruction on 68HC12 and for 68HC11 it generates a sequence of instructions to invoke an external handler.

For 68HC11, this external handler must:

- push the current page on the stack (1 byte)
- install the new page passed in register B
- jump to the function whose address is passed in register Y.

6.2.3 Function Return

The `far' function uses `rtc` on 68HC12 to return (instead of `rts`).

On 68HC11, since `rtc` does not exist, the compiler generates a jump to an external handler which must:

- pop the old page from the stack
- restore the old page from the value poped
- do the return (`rts`)

The 68HC11 far-call frame is similar to the 68HC12 far-call frame.

6.3 Example

For example, consider the following code extract:

```
void __attribute__((near)) foo () {};
void __attribute__((far)) bar ()
{
  foo ();
}
void __attribute__((near)) main ()
{
  bar ();
}
```

In the example the `foo' and `main' function are declared as `near' functions meaning they will use the jsr/rts calling convention. The `bar' function is however declared `far' meaning that it will use the call/rtc convention. In the main, the compiler will generate the following instructions:

```
68HC11                          68HC12
------                          ------
pshb
ldy  #%addr(bar)
ldab #%page(bar)
jsr  __call_a32                 call bar
```

For 68HC11, the `__call_a32' support handler will save the current page on stack (thus creating a 68HC12 call frame), it will switch to the page where `bar' is defined and finaly will jump (with jmp) to the function entry point. For 68HC12 the call instruction performs the same actions in hardware. Within `bar' the call to `foo' is implemented using a jsr because `bar' is a near function. For this to work, the `bar' and `foo' functions must be in the **same** page. This is checked by the linker and if they are in two different pages an error is reported.

For the return of the `bar' function the compiler generates the following instruction:

```
68HC11                          68HC12
------                          ------
jmp __return_void               rtc
```

For 68HC11 the `__return_void' support handler must simulate the 68HC12 rtc instruction. It must pop the previous page number from the stack, install it and then return using rts.

6.4 Far Function Addresses

It is possible to get the address of a `far' function as well as declare a C++ virtual method as `far'. In both cases, since the address managed by the compiler is a 16-bit address, and because the compiler will always use a jsr to call the resulting function, a small trampoline code is used. The address of the `far' function will always be the address of its trampoline code. The trampoline is generated automatically by the linker when an address of a far function is used. There is one trampoline for each `far' function.

The trampoline is always called by a jsr instruction. It will switch the jsr calling convention to a call calling convention. Once the stack layout corresponds to a call, the trampoline jumps to the real far function.

6.5 68HC11 Board Specific Support

The support of memory bank switching for 68HC11 is based on board specific functions which must be provided by the BSP. This section explains how to write these functions.

The `__call_a32' is used to call a far function.

The `__return_void' is used to return from a function returning no value.

The `__return_16' is used to return from a function returning a 8 or 16-bit value.

The `__return_32' is used to return from a function returning a 32-bit value.

Warning: This section is incomplete and will be provided soon.

6.6 Page Calculation

The linker uses the following formulas to compute the physical address from the symbol/linker virtual address:

```
if sym_addr >= sym_bank_base then
  %addr(sym_addr) = ((sym_addr - sym_bank_base) % page_bank_size)
                                + page_bank_base
  %page(sym_addr) = ((sym_addr - sym_bank_base) / page_bank_size)
else
  %addr(sym_addr) = sym_addr
  %page(sym_addr) = 0
end if;
```

The page_bank_size must be a power of two.

```
1/ sym_base = 0x8000, page_bank_size = 0x1C00 (7k), page_bank_base = 0x8000

   sym_addr = 0x8000 => addr = 0x8000, page = 0
   sym_addr = 0x9000 => addr = 0x9000, page = 0
   sym_addr = 0xF000 => addr = 0x8FFF, page = 8 => Error

2/ sym_base = 0x10000

   memory { b1 : 0x10000..0x11c00; b2 : 0x12000..0x13c00 }

   sym_addr = 0x10000 => addr = 0x8000, page = 0
   sym_addr = 0x11000 => addr = 0x9000, page = 0
   sym_addr = 0x12000 => addr =
```

7. References

This appendix gives references to books and standard documents that are useful in developping an application:

`ISO/IEC 9899:1990`
: Programming languages -- C

  This is the ANSI C standard available at http://www.iso.org.

`ISO/IEC 14882:1998`
: Programming languages -- C++

  This is the ANSI C++ standard available at http://www.iso.org.

`Debugging with GDB: the GNU Source-Level Debugger`
`ISBN 1-882114-77-9`
: The GDB documentation book published by the Free Software Foundation.

`Understanding and Porting GNU CC`
`ISBN 1-882114-37-X`
: The GCC documentation book published by the Free Software Foundation.

`M68HC11 Reference Manual`
: The 68HC11 reference manual

`M68HC12 Reference Manual`
: The 68HC12 reference manual