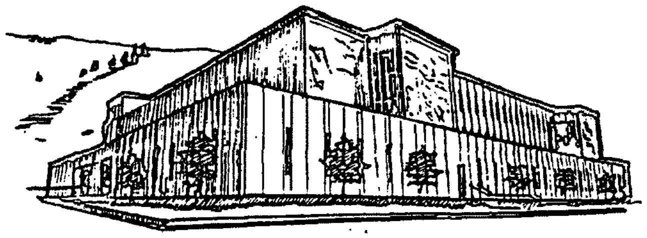
1992

Small C on the Motorola MC68HC11 EVB and EVBU

James Henry Hofmann

The University of Montana



SMALL C ON THE MOTOROLA MC68HC11 EVB AND EVBU

by

James Henry Hofmann

B. A., University of Texas at Arlington, 1966

M. A., University of Texas at Arlington, 1970

Presented in partial fulfillment of the requirements

for the degree of

Master of Science

University of Montana

October 21, 1992

Approved by

[illegible]

Chairman, Project Committee

[illegible]

Dean, Graduate School

November 10, 1992

Date

11-18-92

Table of Contents

Tables

Figures

Chapter 1

Introduction

This paper is a report of a project in the field of embedded systems programming. Embedded systems are computers with the following general characteristics among others:

1. They are often small in size and in computing capacity.
2. They are not general purpose computing devices. They solve very specialized problems, usually in real time. Some examples of embedded systems are computers that operate industrial machinery, fly airplanes, monitor automotive mechanical systems, etc.
3. They do not have the usual off-line storage devices, user interface systems, or input/output devices. Instead, they are likely to have interface to mechanical devices, input from sensors, and so forth. They are not in general designed for human interface.
4. They do not have a general purpose operating system. Most user-created programs operate at a very low level. In the vernacular, this is called "Programming to the metal."

Software on an Embedded System

Some of the difficulties in writing software for embedded systems can be seen from the description. The devices themselves do not support large programming environments of their own. Programs that are to operate in an embedded systems environment are almost always created elsewhere and then transferred to the embedded environment for testing and use. Even the process of downloading the program to the embedded system is complicated by the fact that there is no natural way to move instructions and data into an embedded system. Its design all but precludes programming. Traditionally, programs that are to run on an embedded system are written in the assembly language of the embedded processor and then assembled externally using what is called a cross-assembler to create a binary image containing instructions and data for the embedded processor. This brings with it all the difficulties of the assembly language environment, exacerbated by the remoteness of the target.

An alternative to assembly language that is becoming more popular is the use of higher level languages and cross-compilers to create programs. The language chosen is often C, because of its efficiency and closeness to hardware.

Language Translation

Implementing a program on a computer always involves language translation. It is too tedious to construct binary code for any computer processor directly. The language translation can be from the processor's assembly language, or from a higher level language, supposedly unconstrained by a particular processor. In either case the effect is to convert instructions from human readable form into machine binary code.

Introduction to Small C

A subset of C called Small C has been around for a long time. It has a colorful and exciting history. It was originally developed for very small personal computers when they used 8-bit processors, and could only address 64k of memory. This corresponds rather closely to the situation existing in the embedded systems world. The processors considered here are the same size and the software very often works with data in the form of only bytes or bits. Small C is a subset of C with the following characteristics:

1. The more complicated and difficult to implement language constructs are omitted. The only data types are integers and characters (or WORDS and BYTES).

2. Rules enforcing consistency, safety features, type checking, etc. are abandoned in favor of efficiency.

3. What remains are C structures which can be directly implemented into assembler. The language is not designed to protect the programmer from himself.

The resulting Small C source code is:

1. readable, therefore modifiable, therefore maintainable.

2. compact. It compares with assembler code, and is suitable for use in an embedded environment.

3. dangerous. The programmer is expected to catch his own inconsistencies. For example, a function written with no return statement can be used in an assignment statement as though it did. The value "returned" will be whatever happened to be in the accumulator when the function returns from its call. It is assumed that the programmer knows this, and has his reasons. No error message is generated at compiler time, and the resulting program runs without "error". "User friendly" is not taken into consideration because of its extremely limited resources. Its users are professional hardware-firmware engineers (sic).

The Motorola EVB and EVBU

Some of the most common processors in use today belong to the Motorola M68HC11 family. Each processor in this family contains, on a single semiconductor chip, the central processing unit, all of the storage it requires for operation, all of the input/output devices required for interface with the world, clocks, analog to digital circuitry, and more. In actual use, one could expect such a chip to receive data directly from the world through sensors, and respond with output directly to its own pins, controlling devices directly or through buffering. The amount of support circuitry outside of the chip is purposely held to an absolute minimum to lower cost.

This paper considers two members of the family, and the environment each presents to the software developer. Each member is described, and construction details are given for the one that will not directly support the use of a compiler because of lack of memory.

Memory Considerations

One of the fundamental challenges of the embedded systems environment is the utilization of memory. The embedded system must have several different kinds of storage available. The program itself must be "Permanent". Permanent has several meanings.

Non-Volatile Memory

ROM : Read Only Memory. Data located in ROM can not be changed after the manufacturing process.

PROM : Programmable Read Only Memory. Data located in PROM can be modified exactly once after the chip is manufactured.

EPROM : Erasable Programmable Read Only Memory. Data can be changed by using extraordinary means, beyond the normal operation of the computer. Traditionally, ultra-violet light is used to restore the data bits to a known pattern, and then high voltage (25 volts or thereabouts) is used to reprogram the chip.

EEPROM : Electrically Erasable Programmable Read Only Memory. Data located in EEPROM can be changed, requiring only operating voltages (5 volts). The reprogramming operation typically requires much more time than is permitted by normal program execution, and there is a life expectancy associated with EEPROM expressed in number of times reprogrammed. EEPROM is still permanent in the sense that it retains value when power is removed.

Volatile Memory

RAM : Random Access Memory. A complete misnomer. All of the memory discussed here is randomly accessed. Every data location has its own address, and does not have to be located serially. RAM is further divided into:

DRAM : Dynamic Random Access Memory. Data located in DRAM will retain its value for only a very short period of time, and electrical refreshing is constantly required. The writing operation is quite fast, and is suitable for real time program operation. This type of memory requires complicated refreshing circuitry, and more power than is usually available in a possibly battery operated embedded system.

SRAM : Static Random Access Memory. Reading and writing are acceptably fast, and this memory retains its value under the influence of a constant battery source of system value (5.0 Volts) and low current, permitting memory retention during sleep periods.

The M68HC11 family contains most of the above types in varying amounts on each chip. The amount and location of each is different. Different versions are designated by a suffix on their microprocessor unit part number. For example, the microprocessor found on the EVB is the M68HC11A1, while the microprocessor on the EVBU is the M68HC11E9. The presence of ROM or EPROM is indicated by an infix. The M68HC11E9 has ROM while the M68HC711E9 has EPROM. The addressing of each type of memory is often controlled by the programmer. Typically, a user application will contain the executable instructions in ROM (or EPROM), initialization and configuration information which could vary from installation to installation (the combination of a lock, the address of a device) in EEPROM, and temporary data that changes while the program is running in SRAM or DRAM.

Description of the EVB

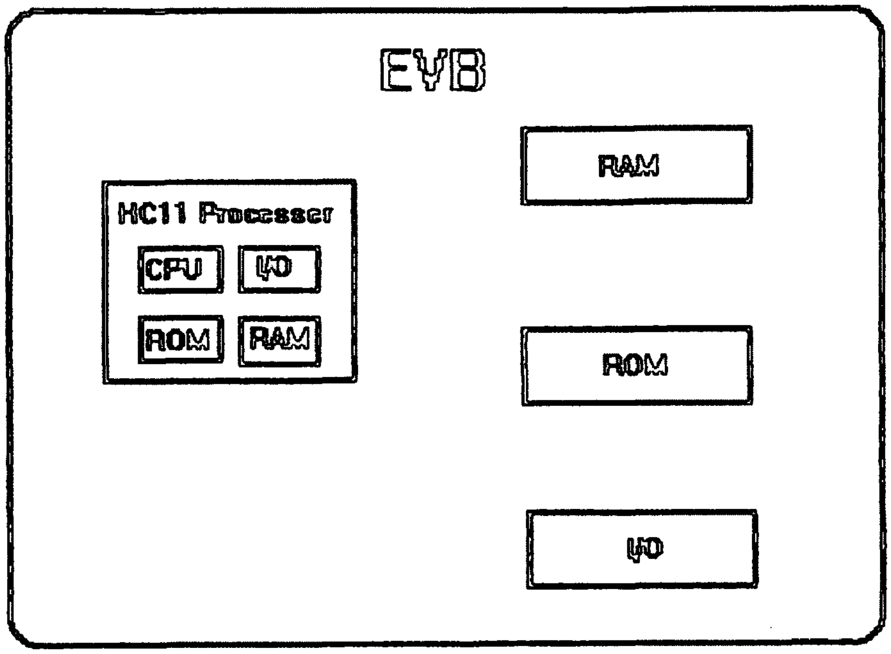


Figure 1. Simplified Block Diagram of the EVB

A complete description the EVB will be found in Motorola literature. The appropriate references are found in the Bibliography. There are some comments that can be made here, however.

The EVB has 8 or 16 kilobytes of external SRAM, used to contain the user's developmental program, and 8k of ROM, which holds the Buffalo There is no ROM present on the MCU itself.

The EVB has two serial ports, but they are not completely configurable, and will not support any hardware control circuitry.

The EVB is the older design, and is somewhat removed from a final embedded environment. User programs are not stored on the chip itself, but in external SRAM provided on the EVB. The HC11 series includes several I/O ports, which are used to communicate with the world outside the microprocessor. There is a dilemma in attempting to use the HC11 for both system development and to represent the finished self-contained embedded system: I/O which must be supported between the host computer and the processor during development has no place in the finished system. Furthermore, and even more seriously, HC11 I/O ports are used to communicate with the EVB's external memory, required during the development of software: access to the HC11's on chip memory in the finished product does not use these ports. The EVB design addresses this by using a Port Replacement Unit which simulates the functionality of the ports used for external memory access. As with all simulation in a real-time environment, there are risks that the execution of the finished system, accessing on-chip memory and accessing all I/O ports directly will not exactly duplicate that of the development environment.

Description of the EVBU

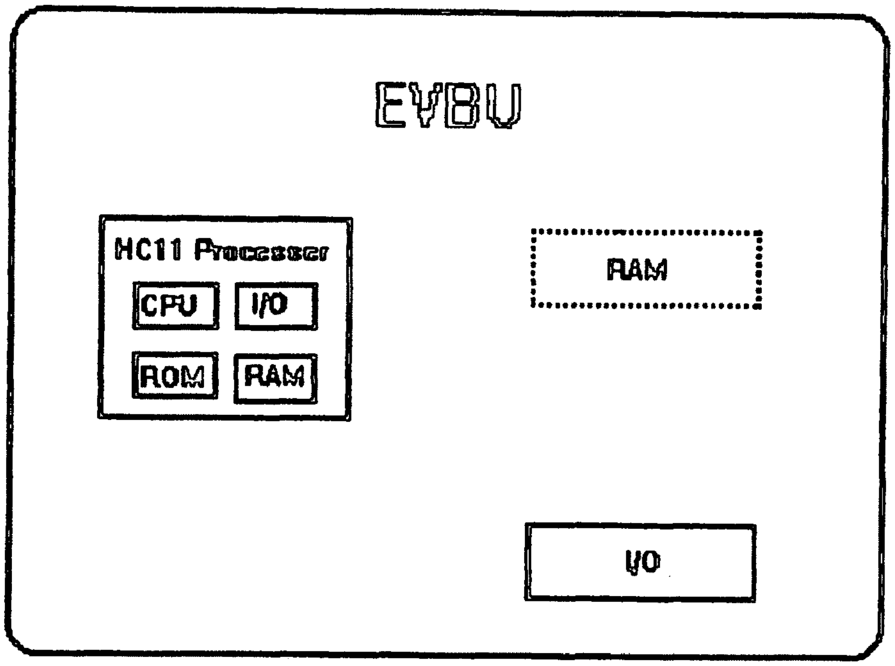


Figure 2. Simple Block Diagram of the EVBU

The EVBU is a newer, simpler design than the EVB. It has no port replacement unit, no external SRAM, and a single serial port. The EVBU has many fewer parts than the EVB, and thus is easier to describe to beginners and is much less costly than the EVB. Containing fewer chips, the EVBU in developmental mode is also closer to the final configuration, which is assumed to be a single chip configuration.

The EVBU has the MCU68HC711E9 MCU, which contains 12k of EPROM. The Motorola Buffalo monitor loaded into 8k of this EPROM provides a testing / operation / communications / debugging interface for development without additional microchips or external PROMs.

The EVBU includes a wire-wrap area that can be used to install circuitry external to the MCU.

Comparison of EVB and EVBU

The EVB was designed to be used for software prototyping using the M68HC11 MCU. It contains 8k or 16k of SRAM, external to the CPU chip, to store software under development. SRAM is much more useful for software prototyping than EPROM or even EEPROM.

There is no area provided for wire-wrapping external hardware to the EVB.

The Buffalo monitor is provided in a PROM external to the chip. Chip ROM is not used. It is assumed that programs developed external to the chip will be eventually coded into the chip during manufacturing.

In contrast, the EVBU as shipped has no external memory, but provides a prototyping area where memory or other chips can be added. If external SRAM is added to the EVBU, the EVB and the EVBU are virtually identical, and programs written for one invariably operate on the other.

The EVBU does not contain the PRU, however, so the two ports simulated by the EVB can not be used during development of software on the EVBU. In the EVB they can be used, but only via hardware simulation through the PRU.

Apparently, the EVBU is targeted at a designer who has software already developed for the 68HC11 MCU, and wishes to prototype a hardware interface to this chip. No RAM beyond what will be present in the commercial version of the chip is provided on the board. The lack of RAM would suggest that the EVBU is not designed for software development.

Buffalo and the EVB/EVBU Development Environment

Buffalo is a combination monitor program and user interface, written and supplied by Motorola in assembly language form with the EVBU and EVB. Buffalo permits the user to observe operation of the MCU, tracing instructions and viewing memory through a dumb terminal or terminal emulator. Buffalo also provides routines to download software through the EVB/EVBU serial port, translating it from Motorola S-19 ASCII to M68HC11 binary form. All of the major routines used by Buffalo are also available to the developer through published addresses guaranteed by Motorola to operate as specified throughout the developmental life of the MCU.

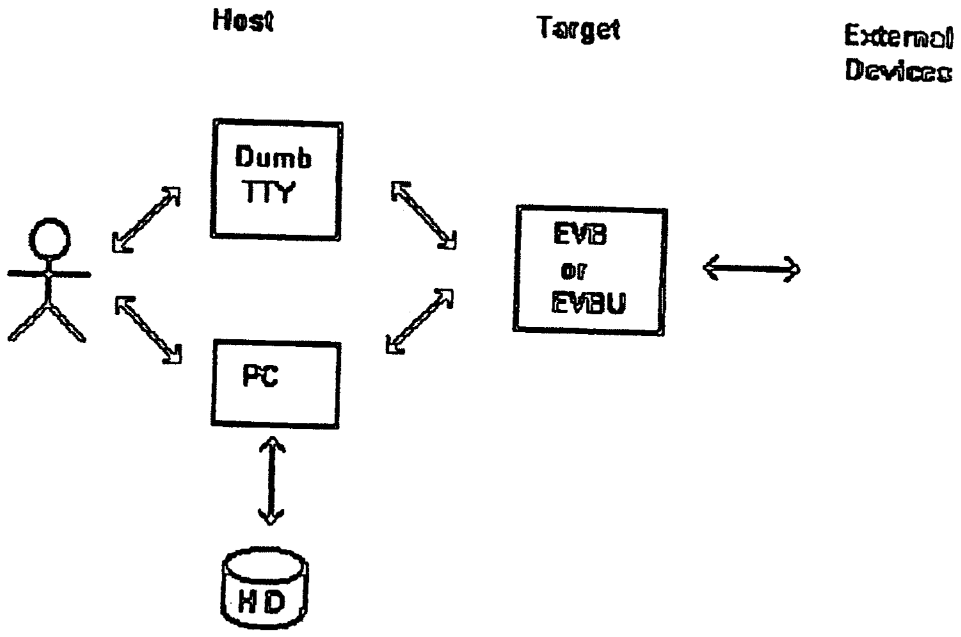


Figure 3. User Interface with the EVB/EVBU

Since Buffalo includes a very limited assembler, the simplest developmental environment consists of a dumb terminal connected to the EVB/EVBU through it's serial port, and operating Buffalo. See figure 4.

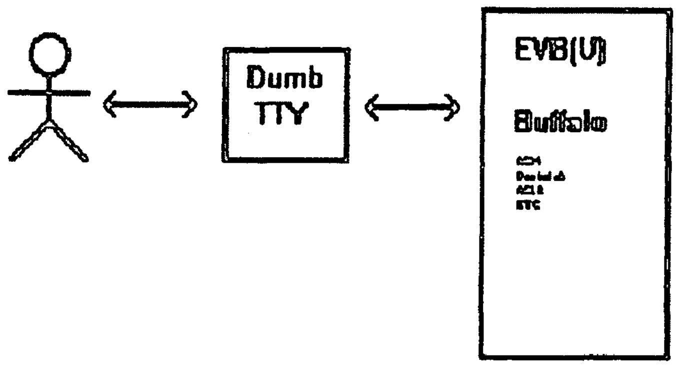


Figure 4. Simple interface with Buffalo

The actual usefulness of this arrangement is very limited. The assembler is extremely limited, and in practice is more likely to be used to alter single instructions of an existing program for testing and debugging purposes than to actually develop software.

A much more viable alternative for software development is described in Figure 5.

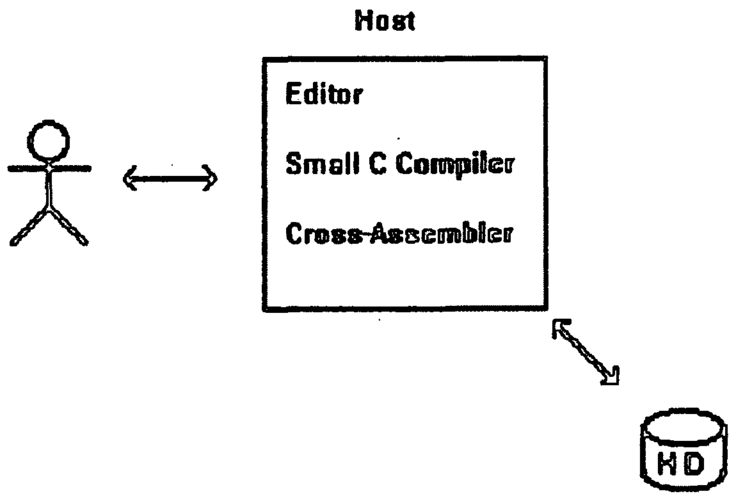


Figure 5. The Host Developmental Environment

The host computer supplies support for the development of the program, providing editors, cross-compilers, cross-assemblers and a terminal emulation program to take the place of the dumb terminal. Motorola S-19 ASCII code produced by the cross-assembler is then transferred to the EVB/EVBU using services provided in the Buffalo, as illustrated in Figure 6.

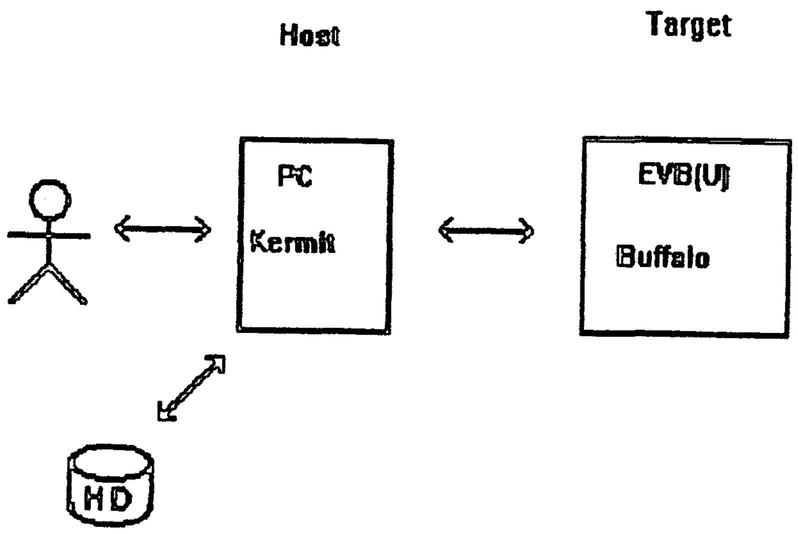


Figure 6. The PC Terminal Emulation Environment

<u>Project Definition</u>

In the sections that follow, a project is described that investigates in detail the use of the EVB/EVBU in software development. The project has the following key components:

1. Evaluate the feasibility of using Small C for embedded systems development using the 68HC11. Contrast the use of Small C with a cross-assembler. Provide a User's Manual for Small C on the EVB(U). Compare and contrast C, Small C, and Small C for the EVB(U). Discuss the inclusion of the Motorola monitor in embedded system development.

2. Provide a Construction Manual for the addition of hardware

necessary to use the EVBU as a platform for Small C. Compare the costs of a Small C environment with other environments for studying assembly language and embedded systems on the EVB(U).

3. Compare hand-assembled code on the Motorola M68HC11 with assembly code created by a Small C compiler. Discuss the modification vs. compact code tradeoff.

4. Provide a small real-time multitasking operating system which might be useful as a test bed for further real time systems development.

Chapter 2

Cross Compilers and Cross Assemblers

```
+------------+
| C Source   |
+------------+
      |
      |  Compiled by Small C cross compiler on host system
      |
+------------+
|  Target    |
| Assembler  |
+------------+
      |
      |  Assembled by Motorola cross assembler on host system
      |
+------------+
| Motorola   |  (An ASCII text file stored on the host)
| S Records  |
+------------+
      |
      |  Downloaded and converted into target binary
      |  by a program existing on the target
      |
+------------+
|  Target    |
|  Binary    |
+------------+
```

Figure 7. The Compiled Code Generation Process

A cross compiler is a compiler that generates code for a processor other than the one that supports execution of the compiler. A cross assembler assembles code for a processor other than the processor present on the host system.

Whether the translator executes on the same processor as the binary code it generates executes on is not significant. The essential differences are between assemblers and compilers. Both assemblers and compilers are processor specific, in that they generate code for a particular processor. A compiler that generates assembly code for one processor can not generate assembly code that can be cross-assembled to run on another.

It occurred to me, as perhaps it has occurred to others, to simply use the high-level language and environment of a host computer to create a program for the target computer. One could employ the vast resources of the host environment, and then translate from the host assembler language into the target assembler language.

This fails because the host language compiler is written for the host computer. The output, written in assembler for the host, is not generic. It is specific to that host. Translating from one assembler language to another assembler language is equivalent to changing the alphabet from Roman to Cyrilic to produce a natural language translation. Assembler is almost word for word equivalent to binary. It is simply written out for ease of human comprehension. Instructions for one computer will not translate into instructions for another.

When a programmer uses a cross-assembler, he is writing (and thinking) in the _target_ language. He is absolutely uninterested in the assembly language of the machine he is using for a host. The cross-assembler he then uses _does not perform a language translation_. The language is and always was the language of the target processor. Therefore, the use of a high level language for an embedded environment requires a cross-compiler and a cross-assembler, or a cross-compiler that translates directly into the binary code of the target processor.

It is widely supposed that programmers produce efficient code when working in assembler language, and that compilers produce less efficient assembler code. It is my belief that compilers and languages chosen with embedded systems in mind will produce code that compares favorably with hand-assembled code. Possibly programs can be created in a subset of C that will prove adequate for an embedded systems environment.

Chapter 3

Small C

History

Ron Cain's Small C Compiler v1.0, which debuted in the May 1980 issue of Dr. Dobbs Journal, was originally a very small subset of the C language[1]. Small C is a one-pass compiler which generates assembly language from a C input file. The subset of data types which the original Small C recognized consisted only of characters, integers, and one-dimensional arrays of either type. Additionally, the only control statements were while and if. Small C was also restricted to bitwise, logical (&, |) operators since Boolean (&&, ||) operators were not supported.

In 1982 James E. Hendrix assumed trusteeship of Small C. Hendirx published numerous upgrades through Dr. Dobbs Journal culminating in the release of Small C v2.1 for DP/M in 1984. New features added along the way include code optimization, data initializing, conditional compiling, extern storage, for, while, switch/case, and goto statements, and a plethora of operators. To complete the system, James E. Hendrix and Ernest Payne developed a CP/M compatible version of the UNIX C standard I/O library. The internal design of Small C v2.1 was the subject of Hendrix's The Small C Handbook.

The first published 8086 PC-DOS implementation of Small C v2.1 appeared in 1985. Along the way, code optimization techniques were

----

1.Volkman, Victor A Survey of CUG C Compilers. (Contained in The C Users Group Library, Volume II, Robert Ward and Kenji Hino, Eds.) R&D Publications, Inc., Lawerence, KS, 1989.

refined even more. The present incarnation from .OPHendrix, Small C v2.2, is available for 8086 PC-DOS only. Small C v2.2 was released simultaneously with Hendrix's definitive reference work _A Small C Compiler: Language, Usage, Theory, and Design_ in 1988.

The differences between Small C and full implementations of the language are important to programmers who anticipate using a full C compiler in the future. No one wants to write programs that will be hard to convert. This upward compatibility is made possible by the fact that Small C is a subset compiler. For the same reason, however, downward conversions can be expected to be difficult or even out of the question.

Small C supports integers, characters, pointers, and single-dimensioned arrays of integers or characters.

Small C does not support structures or unions. A structure is a collection of objects of any type; other languages often call them records. A union is an object that has multiple declarations; that is, a single piece of memory that can contain any one of several types of data at different times.

During expression evaluation, the Small C compiler assumes that any undeclared name is a function, and automatically declares it as such. If the reference is followed by parentheses, a call is generated; otherwise, the function's address is generated by reference to the label bearing its name. If the same function is defined later in the program, the label for the function is generated. If, on the other hand, it is not defined, then Small C automatically declares the name as an external reference, to be resolved at link time. This arrangement makes it unnecessary to declare a function before referring to it.

Small C functions return only integer types. This can be a converted character, an integer, or a pointer. Small C doesn't care. One of the early versions of Small C was ported to the Motorola 6800 chip, which contains the instruction set of the HC11. It has also been ported and expanded to the Z80, 8080, 6809 (Tandy Color Computer) and the 8086 family.

Small C does not support post-incrementation. "q++" and "++q" evaluate to the same thing, a pre-increment. Expressions such as

"while(i++)" will produce the same code as "while(++i)."

Small C on the HC11

Programming on the HC11 is unlike programming on a larger system. There are fundamental differences in memory, runtime environment, and interrupts among others.

Memory Differences

The HC11 is an 8-bitprocesser, and addresses 65536 locations between 0000H and FFFFH. Not all of the memory addressed is actually on the chip. The chip itself may contain ROM, RAM, EPROM and EEPROM. Additionally, many addresses are special and are completely controlled by the processor. Such special addresses are used as I/O ports and configuration registers. These addresses can not be programmed directly or normally. Some addresses can be changed only within a few machine cycles of startup, after which they resemble ROM addresses. See the bibliography for complete descriptions of those addresses.

What all this means to the programmer is that he can not trust a separate "Linker" or "Loader" program to choose an address space. He must explicitely map data into RAM areas and programs into permanent storage. For example, ROM, EPROM, and EEPROM are all permanent to various degrees, so are suitable areas for mapping programs; however, all three require too much time to be effective holding program variables.

The programmer will have no access to off-line storage for locating his program during development. Both program instructions and data must all be located in target system RAM in order to permit the edit--compile--download--execute--debug--edit cycle. In the final version, or even the prototype stage, the program must be loaded into EPROM or EEPROM. It cannot be copied into RAM from an external file-storage device because there isn't one on the target system. Small C for the HC11 contains several special pre-processor commands to separate data and instructions in the assembly code generated so that eventually program and data can be correctly located.

During the development stage, the program will start from reset or from the Buffalo monitor using a CALL instruction. The final version must start from reset, or power-up.

The compiler must be capable of setting addresses. There is no linker, and the addresses will only be known to the programmer.

Runtime Utilities

There is no run-time support for programs that are to execute on the EVB(U). There are no utilities to be called from a program, because there is no operating system. The program operates the CPU directly. No run-time initialization takes place when a program is to execute. Any initialization must be done within the program. The program must initialize and control the stack.

<u>Interrupts</u>

All interrupt routines must be defined by the program, and only those interrupts supported by the HC11 are permitted. Since there is no operating system, there are no default interrupt routines. If an interrupt occurs during execution of a program, the program must contain code to handle it.

<u>ROM vs. RAM</u>

Development of programs will most likely occur in RAM. The difficulties and time expense of downloading into EPROM or EEPROM are too great. The general idea is to:

1. Develop the program in RAM external to the 68HC11, employ the Buffalo in EVBU ROM.
2. Locate the final version in ROM to run without the Buffalo or external RAM.

It makes no difference to the compiler where the code is to be located. The programmer is responsible for locating the code.

Chapter 4

Writing Programs in Small C on the M68HC11

Introduction and Disclaimer

This chapter will not attempt instruction in C or in Small C. The bibliography lists several fine works on these subjects. This chapter will attempt to point out the differences between Small C and Small C for the 68HC11, and describe fully the additional features of Small C for the 68HC11 required to write embedded systems programs.

ROM vs. RAM

The nature of the 68HC11 environment requires the Small C compiler to be able to create two different versions of a program--one to run in the presence of the Buffalo monitor, from RAM; and one to run alone, from ROM. Attempting to develop the ROM version without first prototyping it with a version running in RAM is out of the question. In this discussion, RAM will refer to volatile memory, and ROM will refer to any type of non-volatile memory. A command-line compiler switch -R chooses the ROM version. Omitting the switch chooses the RAM version.

The RAM Version

The compiler does not initialize the stack pointer. The user program uses the Buffalo stack, or the programmer sets the stack manually using Buffalo RM commands. The programmer starts the program manually from the Buffalo monitor using CALL or GO commands, giving the address used in the #code directive in the Small C source code. A #stack directive in the RAM version is ignored.

The ROM Version

The system stack is set to the address given in the #stack directive. In this version, the programmer must be certain that control is passed to the starting address of the executable code on reset or power-on. This is the way completed self-contained program will operate. Usually, the designer will use the prototype version until he or she is ready to commit the program to ROM, and then switch to the ROM version.

The Stack

The Buffalo stack is very small, only 55 bytes, and carefully designed for the needs of the Buffalo. Usually, the programmer sets the stack to the top of the extended RAM area, and it works down towards the top of his code. The programmer must generally reset this stack to an area of RAM chosen so that there is enough space, working downwards, for the needs of the program. The most obvious program need is for the printf() function, which stores all character strings on the stack and then pops the characters off one by one for display. The stack must be at least the size of the longest character string that is to be displayed. There will be no warning when the stack space is exceeded. The program just stops, with nothing displayed on the screen.

Interrupts

Interrupt programming is very common in embedded systems, and the 68HC11 Small C compiler includes features for programming with interrupts.

For a program to use an interrupt service routine, three things must happen.

1. Interrupts must be enabled.

2. A function must be written to service the interrupt.

3. The function must be called when the interrupt occurs.

Taking these three things in order, the first task is to enable the chosen interrupt. Suppose that the interrupt to be used is the Timer Overflow interrupt, as defined in Motorola literature. The following lines of Small C code will enable this interrupt.

```
main ()
{
#asm
      LDX   #$1024      /* Timer Interrupt Mask Register 2 */
      LDAB #$03         /* Set bits 0 and 1                */
      STAB 0,X
#endasm
  .
  .
  .
}
```

This interrupt will now occur when the timer register overflows, and will be serviced by the default interrupt service routine in the Buffalo, or will stop the processor if it is not defined in the Buffalo. Again, see the Motorola literature for complete details.

To accomplish step 2, a function named toi is written:

```
interrupt toi() {
    fputs("The timer overflowed.", stdout);
}
```

This sample response illustrates how to use the interrupt keyword to define an interrupt service routine. From the Small C programmer's point of view, an interrupt service routine (ISR) is a function without a call. When such a function is defined using the Small C keyword "interrupt", the function will be called whenever that interrupt occurs. Since the MCU saves and restores registers as a part of the interrupt procedure, code to save and restore the stack on entry and exit of the interrupt service routine is omitted when the keyword interrupt is used.

The third thing that must be done to use an interrupt is to re-vector the interrupt. A small digression is in order here.

Each interrupt that can be generated by the 68HC11 must be supported by an interrupt service routine. The default interrupt routines used by Buffalo are stored in Buffalo ROM. The one for Timer Overflow is located at ROM location $E365. The 68HC11 MCU will jump to the address located at $FFDE, $FFDF. The Buffalo monitor puts an address in RAM ($00D0) into locations $FFDE, $FFDF. (See Figure 8.)

There is a table of jumps located in RAM to this and all the other interrupt routines. It is located in RAM so that it can be changed to jump to a routine of the user's choice, rather than the default ROM routine. This is called "Hooking" the vector.

Each vector in the table consists of three bytes. The first byte is a $7E (JMP instruction), and the second and third form the address for the jump. The RAM interrupt vector table is located from $00FF downward to $00C4. The contents of locations $00D0, $00D1, $00D2 will be $7E, $E3, $65, until changed to point to the new routine.

The addresses can be changed "by hand", using Buffalo MM commands, or "automatically" by using Small C for the HC11's poke() function:

```
{
      poke(0xD1, toi);
}
```

Since a function name is no more than an address, this substitutes the address of toi() for the default Buffalo Timer Overflow routine in

ROM at addresses $00D1 and $00D2. The JMP instruction itself remains at $00D0. As an alternative, the two bytes at $FFDE, DF could have been changed to point to the user's toi() function, if these locations were not located in ROM, and were not part of the Buffalo monitor. They could be so vectored in the final version, once the Buffalo is no longer part of the system.

Now when the timer register overflows during execution of the program, the new ISR, toi(), will be called to respond.

```
+-------------+   +------------+    +------------+   +-----------+
| Interrupt   |---|7E |00 |D0 |----|7E |E3 |65 |---| toi ISR   |
+-------------+   +------------+    +------------+   +-----------+
                   FFDE              00D0
```

Figure 8.

The Interrupt Sequence

There are a couple of additional caveats, however.

1. The interrupt must be one of those supported by the 68HC11, and must have a specific name. The names and descriptions are shown in Table 1.

Table 1

M68HC11 Family Interrupt Functions

Interrupt Name	Interrupt Function
sci	SCI system
spi	SPI system
pac_in	Pulse Accumulator Input Edge
pac_of	Pulse Accumulator Overflow
toi	Timer Overflow Interrupt
tme_oc5	Timer Output Compare 5
tme_oc4	Timer Output Compare 4
tme_oc3	Timer Output Compare 3
tme_oc2	Timer Output Compare 2
tme_oc1	Timer Output Compare 1
tme_ic3	Timer Input Capture 3

tme_ic2	Timer Input Capture 2
tme_ic1	Timer Input Capture 1
rti	Real Time Interrupt
irq	External Pin Service Routine
xirq	Pseudo Non-Maskable Interrupt
swi	Software Interrupt
illegal	Illegal Opcode Trap
cop	Computer Operating Properly Failure
clk_mon	Clock Failure

The mnemonics here are copied from Motorola designations of the interrupts and the reader is referred to the M68HC11 handbook for more information about the interrupts themselves.

2. Changes to the Timer Interrupt Mask Register 2, or any other time-protected location between $1000 and $103F, must be done "Quickly" on reset, before the time protect timer expires. This probably precludes including this function and support code into a separate library function. The time protect timer will surely expire while the function calling code is being executed. The sample Small C program named CLOCK.C is a prototype version, meant to be started from the Buffalo. The Small C code in this program that appears to set the protected register actually does nothing, since it executes much too late to set $1024 to anything. (Buffalo may have been running for hours before you started your program). The ROM version will begin quickly enough, however. It may in fact be best to change protected registers from assembler code located in EEPROM, at $B600, at least in the preliminary runtime version.

3. In the case of the EVB, the user must re-vector the interrupts manually.

Study the sample programs CLOCK.C and CLOCK2.C for completed examples, and welcome to the world of interrupt programming.

Code and Data Areas

Memory address spaces for both code and data are under programmer control in the Small C compiler. The address space for the EVB/EVBU is not homogeneous. Different parts of it are designed for different needs. The ultimate user's program will doubtless reside in some kind of permanent memory, either on the chip or possibly in external ROM or EEPROM. Variables must be, well, variable, which means that they must be located in RAM.

Small C takes two approaches to this situation.

The rom Keyword

A variable declared with this keyword will be stored as a constant in the code space rather than the data space. A prompt to the user, for example, declared:

```
{
      rom msg[] = "Press any key to continue.";
 .
 .
 .
}
```

as a global variable will exist on program startup.

One of the fundamental differences between programming in a standard operating system supported environment and programming in an embedded systems environment must be addressed here.

The sequence of events for program startup in a standard, operating system supported environment is, roughly,

1. Load the program in RAM from off-line storage.
2. Load the program counter with the first executable address.

3. . . .

The load just before go <u>does not happen</u> when a program is to run on an EVB/EVBU. There is no disk system to "Load" anything. RAM is not initialized. The executable code must be located in some kind of permanent memory. If data that is considered constant, or that is to be already defined at program startup is not in ROM it won't be defined at all when the system wakes up.

If the user downloads the program immediately before running it each time, then the effect will be the same as in the operating-system-supported environment, and constants can be stored in RAM. Usually this isn't the way it is done, and it <u>can not</u> be done that way in the final version, which must start out of reset. The user is not going to download the program immediately before he or she runs the program. Programming for EVB/EVBU is just a wholly different world.

Finally, note that the <u>rom</u> keyword does not mean that the variable will actually be located in ROM. The variable will be located wherever the programmer declared the code area to be with the <u>#code</u> directive. This will probably be RAM for the developmental version and some kind of permanent memory for the final product.

<u>Files Produced by the Small C Compiler</u>

The Small C compiler places assembly code for all globally defined variables into a file called DATAOUT.ASM. All code defined inside functions is located in a file called CODEOUT.ASM. This permits CODEOUT to be located in ROM while DATAOUT is located in RAM. In the case of the prototype versions, probably both sections will be located in RAM somewhere, but they do not have to be contiguous, and eventually the data area must fit within the chosen processor's RAM area. The Small C preprocessor keywords #data and #code determine what the start address of the different sections will be. For example, in the developmental version of the EVBU, the external RAM was all located at $6000 - $7FFF, and usually #data and #code were set as follows:

```
#data 0x6000
#code 0x60F0
```

This reserves the RAM between $6000 and $60F0 for global data variables. The 68HC11 stack pointer was manually set at $7FFF to move down from the high end of the available RAM. During development data usually can not be stored in the ultimate user area on chip RAM, $0000 - $01FF, because the runtime version must co-exist with the Buffalo, which uses a large portion of that area for itself. The final ROM version will probably relocate both code and data into the address space used by the Buffalo, and not use Buffalo at all. The dilemma with the Buffalo is that it will not be present in the final version, but must be present during development. Prototype code can not even be downloaded without the Buffalo!

Compiler Organization

It is very useful for the Small C programmer to understand what assembly code will be generated by his Small C constructions. Knowing how the compiler will treat Small C code enables the programmer to select structures that will be translated with maximum efficiency.

Small C has only two types of data, byte and word. Byte is used for characters and for short integers. Word length data (16 bits) are used for integers, of course, but also for addresses, or pointers. The mixing of integers and pointers is considered very dangerous in standard C, and is to be avoided when the compiler doesn't expressly forbid it. Since all addresses are 16 bits long on the HC11, and since absolute addresses are always used, the use of integers for pointer operations is not quite as confusing in Small C for the HC11.

Efficiency in Subroutine Calls

One of the major ways efficiency is obtained in hand-assembled code is through the custom planning of subroutine calls. Rather than saving all the registers, the astute programmer only saves the ones that the subroutine modifies. Values are passed to subroutines in registers where possible, and in general the overhead involved in calling and returning is reduced to a minimum. These very efficiencies make hand assembled code very difficult to read, therefore difficult to maintain. The compiler generates subroutine calls that always operate the same way. The efficiency of compiler-generated subroutine calls can be improved by eliminating parameters and using global variables where practical.

Since local variables are stored on the stack, instructions can be pared from the compiled code by declaring locals outside the function. However, not much address space will be saved in this way, since the variable on the stack requires the same amount of RAM space as the global variable in the data area. Note that for other reasons, constants <u>must</u> be declared outside of a function block unless the <u>rom</u> keyword is used.

Using chars for short integers when possible will save quite a bit of RAM, but since then the compiler silently extends characters to signed integers in expressions evaluation, the code required to process characters sometimes actually exceeds the code required to process integers.

The 68HC11 has a limited number of registers, and since an integer or a pointer uses two bytes, even fewer arrangements are possible. Again, the reader is referred to Motorola reference manuals for a complete description of the 68HC11 MPU.

The compiler uses the D register as an accumulator, and almost all operations use this register. The X register is used in addressing. The Y register is used as a temporary stack pointer location for subroutine activation records.

Figure 9 is a portion of the assembler code generated by the Small C compiler from the HELLO.C program. Comments have been added to help explain how function calls and returns are translated by Small C.

```
========================================================================
0791                         _fputc          Label with leading underline
0792 66c9 18 3c                  PSHY        Put return address on stack
0793 66cb 18 30                  TSY         and set local activation record
0794 66cd 18 3c                  PSHY        pointer.  Saves Y and S.
0795 66cf 18 30                  TSY
0796 66d1 18 ec 08               LDD 8,Y     See note 1.
0797 66d4 7e 67 16               JMP _59     Start routine.  We are at the
0798                         _60             top of a switch routine here
0799 66d7 18 ec 06               LDD 6,Y     Get address of output routine
0800 66da 37                     PSHB        Push onto stack
0801 66db 36                     PSHA        Amounts to PUSH D.
0802 66dc cc 00 0d               LDD #13     Put carriage return into accum.
0803 66df 38                     PULX        Pop address of output routine
```

```
0804 66e0 8f                        XGDX        Swap parameters
0805 66e1 3c                        PSHX        Exchange position on stack
0806 66e2 ce 00 01                  LDX #1      Number of parameters
0807 66e5 8f                        XGDX        1 now in D
0808 66e6 ad 00                     JSR 0,X     Address of output routine in X
0809 66e8 31                        INS         On return, Adjust stack for
0810 66e9 31                        INS         param 1 and 2.
0811 66ea 18 ec 06                  LDD 6,Y     Repeat above sequence for LF
0812 66ed 37                        PSHB
0813 66ee 36                        PSHA
0814 66ef cc 00 0a                  LDD #10     Line Feed
0815 66f2 38                        PULX
0816 66f3 8f                        XGDX
0817 66f4 3c                        PSHX
0818 66f5 ce 00 01                  LDX #1
0819 66f8 8f                        XGDX
0820 66f9 ad 00                     JSR 0,X
0821 66fb 31                        INS
0822 66fc 31                        INS
0823 66fd 7e 67 22                  JMP _58     Clean up and return
0824                           _61              Default switch operation
0825 6700 18 ec 06                  LDD 6,Y     Get 1st argument (char)
0826 6703 37                        PSHB        Set up for subroutine call
0827 6704 36                        PSHA
0828 6705 18 ec 08                  LDD 8,Y     Get 2nd argument
0829 6708 38                        PULX        First argument in X
0830 6709 8f                        XGDX        Exchange arguments
0831 670a 3c                        PSHX        Push 2nd argument
0832 670b ce 00 01                  LDX #1      Number of arguments
0833 670e 8f                        XGDX        Exchange numargs and 1st arg
0834 670f ad 00                     JSR 0,X     Jump indirect to routine at X
0835 6711 31                        INS         Restore single param
0836 6712 31                        INS
0837 6713 7e 67 22                  JMP _58     Exit
0838                           _59
0839 6716 bd 6e 1d                  JSR __SWITCH      See note 2.
0840 6719 66 d7                     FDB _60     If char is \n.
0841 671b 00 0a                     FDB 10
0842 671d 00 00                     FDB 0
0843 671f 7e 67 00                  JMP _61     If char is not \n.
0844                           _58
0845 6722 18 ec 08                  LDD 8,Y     Get return value in D
0846 6725 18 38                     PULY        Restore stack and exit
0847 6727 18 35                     TYS         Transfer Y to Stack Pointer
0848 6729 18 38                     PULY        Restore Y register
0849 672b 39                        RTS         Return from Subroutine
------------------------------------------------------------------------
```

Figure 9. Partial Output from the HELLO.C Program in Small C

Note 1: Contents of stack at the time of this instruction:

```
+-------------------------------------------------------------
| sp | sp | ra | ra | p1 | p1 | p2 | p2 |....|....|....|...
+-------------------------------------------------------------
  Y
```

Y contains the address of the top of the local stack. "sp" is the stored system stack pointer. "ra" is the stored return address. "p1" and "p2" are the parameters passed to the subroutine. The second parameter is stored at Y + 8 (bytes).

Note 2: The operation of __SWITCH is too complicated and assembler-specific to include here. The assembly code used to implement __SWITCH is an excellent argument for a higher level language, where a small piece of code is written once very carefully for maximum efficiency, then reused as a single module in a higher level language. The jumps are taken indirectly as illustrated.

The number of parameters is in D when a subroutine is called. All parameters are passed on the stack. All functions return integers, whether a return statement is present or not, whether they are called as a function or not, whether they are supposed to return anything or not. If a function is called as a R-Value, whatever is in the accumulator on return will be assigned. It is the programmer's responsibility to make sure that return statements are used in functions that are to return values, and not to use void functions in assignment statements. There will be no error or warning messages.

Push and pop in the 68HC11 refer to bytes when A and B are used, so the compiler must do them in pairs for a word (integer or address in register D). The stack is much more extensively used by the Small C compiler than it is by Buffalo. For this reason the programmer must usually set a new stack manually. If any screen output routines are used a larger stack will surely be required. There is no stack failure routine. The usual symptom of stack overrun is a blank screen.

The compiler does not check for variable declarations. Any variables not declared are assumed by the compiler to be external.

A file listing the Motorola-defined HC11 variables exists with the name 68HC11.H. It is quite effective to refer to those variables from within Small C source code, without a previous #define or #include, and then simply include the 68HC11.H file in the output files going to the assembler. 68HC11.H will supply the definitions for the variables. The compiler, as usual, will not complain about the undeclared identifiers.

For example

```
{
    int x;
    x = TCNT;  /* TCNT not declared anywhere in the program */
  .
  .
}
```

The assembler puts them together using

```
asm myfile.asm 68hc11.h
```

usually automatically in the batch file.

Chapter 5

Error Detection and Debugging Methods

The compiler is both cryptic and cruelly forgiving. Error messages do not include line numbers, but do include a "pointer" to the last thing recognized. The entire line containing the last thing recognized is printed, so that in case the position of the error is not easily determined, comments placed on likely lines will help determine the exact line by establishing context. For example, suppose that the error message refers to a { which the programmer has used on a line by itself, and the programmer can not now be sure which { is being referred to by the pointer. The programmer can edit all the source code, adding comments of the general form:

```
{  /* Open main block in foo.c */
```

to every line containing a single {. The comment will be printed with the { when the error is detected and the programmer can determine which line of which file contains the error.

Included files are included before compilation takes place, so line numbers are not very useful. Sometimes when debugging the most difficult thing to determine is the <u>name</u> of the file containing the error. Again, many comment lines in the library files help determine the line.

Unresolved external references and undeclared variables are not detected by the compiler, but they are by the assembler. Missing Small C source code is indicated in the assembler listing as an unresolved reference. It is not obvious which library source files refer to which other library source files, and sometimes several iterations of the compile--assemble--edit sequence are required to identify all the necessary source code, at least for the first few times. The list of files required by printf() is especially extensive. One bright spot exists, however. It makes absolutely no difference what order the source code is in, just so it's all there.

Exception: The file STARTUP.C must be first in the list of includes. It consists of a jump to the main routine, and any code included before it will be executed, usually with unfortunate results, before the jump to main starts the program officially. If this file is not included, then there is no jump to the main routine. The results, as they say, are undefined, and sometimes quite spectacular.

The edit--compile--assemble--connect--download--execute process is so long that detecting and correcting small oversights become really time consumming. If possible, parts of the program should probably be written in Small C for the host computer, and when most of the bugs are worked out, a version for the EVBU can be edited from the host version and downloaded with at least some of the logic worked out. Of course, the host computer cannot duplicate many of the operations present on the EVBU.

Along the same lines, since debugging is so difficult in the host--client environment and the tools are so primative that small routines and short steps are usually necessary. A large program with several small problems will be so impossible to debug that it will have to be abandoned and reconstructed almost line by line in smaller steps.

For instance, suppose one is constructing a program involving user input and output to the terminal. The first part to get correct is the interface itself. Write a program that will accept a string input at the terminal and write it on the screen. Make sure that that part is as robust and as simple as possible. Then add the computational parts to the input as a second pass at the project. The ability to produce reliable output to the screen will prove invaluable during debugging.

Small C is very primative. It does not warn the programmer of many, many things. If the wrong data type is used in a printf, using %c for %d for instance, no conversion will take place, execution of the program will produce nothing at all on the screen, and no warning or error has occured.

The Buffalo stack is very small, and carefully designed for the needs of the Buffalo. It is located as close to the bottom of the chip's RAM as possible. The programmer must generally reset this stack to an area of RAM choosen so that there is enough room, working downwards, for the needs of the program. The most obvious need is for printf(), which stores all character strings on the stack and then pops the characters off one by one for display. The stack must be at least the size of the longest character string that is to be displayed. The Buffalo stack is only about 55 bytes long. There will be no warning when you exceed the stack space. The program just stops; usually nothing is displayed on the screen.

The programmer can set the stack with the RM command from the Buffalo monitor before starting the run, usually to the top of RAM where it will work downward towards the executable code.

Chapter 6

The Small C Compiler Users Guide

The version of the Small C compiler presented here uses the IBM-PC compatible as a host system. References to DOS are references to MS-DOS commonly used on those systems.

The Small C compiler command syntax is:

```
cc <filename> [-R]
```

The compiler reads an ASCII text source code file written in a subset of C called Small C. The output from the compiler is 68HC11 assembly code written to two standard DOS files named CODEOUT.ASM and DATAOUT.ASM. CODEOUT.ASM contains the executable assembly code and DATAOUT.ASM contains the designated data storage areas. The addresses of these two areas correspond to the use of the special Small C directives #code and #data. The use of these two keywords is explained in the section on the Small C language for the 68HC11.

That is all the compiler actually does. However, other things are necessary to completely translate Small C source code into Motorola S19 code for downloading to the EVBU. The two files created by the compiler are usually combined into a single file and then assembled using the Motorola Freeware Assembler into Motorola S19 code.

To keep the size of the compiler small, external routines are employed for several essential operations, including all of the logical expression code. These must be included or "Linked" into the Small C source code before that code goes to the assembler. Currently, all the external routines the compiler needs reside in a file named 6811_LIB.C

The library could optionally be #included in the source code, and the choice of using an #include or concatenation of source code is one of style. In fact, the entire library could be included by concatenation of the source code into a single Small C source code file rather than using #include if desired.

The complete procedure for creating Motorola S19 code from a user program called MYPROG.C is then:

```
copy MYPROG.C+6811_LIB.C ASPO.C
cc ASPO.C
copy DATAOUT.ASM+CODEOUT.ASM MYPROG.ASM
as11 MYPROG.ASM
```

The copy command combines MYPROG.C and the 6811_LIB.C file into a temporary named randomly ASPO.C. Any legal filename will do as well.

All but the absolute essentials have been left out of this sequence. Since very little error detection or correction exists in the compiler, other files need to be generated for this purpose. The listing file generated by the assembler seems to be particularly useful for error detection. Search such a file for the assembler error "Symbol undefined on pass 2", and one detects source code accidentally omitted from the compilation. This error is especially easy to generate, since almost all code must be specifically included in the form of Small C source code for this compiler.

Using this system for a while certainly makes one familiar with all the small things the modern, hip, slick and cool compiler does for you behind the scenes. The Small C user must know exactly what must be done, and where, and when and why in order to translate a program from Small C to Motorola S19 code.

Code created in modules can be combined into a single program only by combining Small C modules into a single Small C file using #include preprocesser directives, or by combining assembly output files into a single assembly file before assembling.

A batch file can be used to automate the above sequence. The actual batch file used to compile and assemble the sample and test programs looks like this:

```
del codeout.asm
del dataout.asm
del aspo.c
del %1.s19
del %1.lst
del %1.asm
copy %1.c+\include\6811_lib.c aspo.c
cc aspo.c %2 %3
copy dataout.asm+codeout.asm %1.asm
as11 %1.asm -l > %1.lst
```

The delete commands remove old copies of intermediate files so that in case the batch sequence aborts, the programmer will not be left with a confusing set of intermediate files, some current to this attempt and some remaining from a previous compilation.

%1, %2, and %3 are DOS command line parameters of the DOS command to execute the batch file. In this case, %1 is the root name of the user's program, without extension. Including the extension will produce an error. %2 and %3 are optional parameters which, if present, will be passed to the compiler in due course. This would provide for the -R for RUNTIME compilation, for instance.

The copy command does the file concatenation. The assembler created listing file is very useful for program verification.

The environment used for testing the compiler has all the necessary executable files in a \bin directory, and all source code included in a user's program is in \include. There is no library of pre-compiled, pre-assembled code. Everything necessary for the program must be present in source code form when the compiler executes. The program source is in the programmer's working directory. At the end of a compile all the files deleted at the beginning of the batch file execution have been rebuilt in the working directory.

The Small C compiler places assembly code for all globally defined variables into a file called DATAOUT.ASM. All code defined inside functions is located in a file called CODEOUT.ASM This permits CODEOUT to be located in ROM while DATAOUT is located in RAM. In the case of the runtime version, probably both sections will be located in RAM somewhere, but they do not have to be contiguous, and eventually the data area must fit within the ultimate RAM configuration. The keywords #data and #code determine what the start address of the different sections will be. For example, in the developmental version of the EVBU, the external RAM was all located at $6000 - $7FFF, and usually <u>#data</u> and <u>#code</u> were set as follows:

```
#data 0x6000
#code 0x60F0
```

Then the 68HC11 stack pointer was manually set at $7FFF to move down from the high end. Data can not usually be stored in the ultimate user area on chip RAM, $0000 - $01FF, because the runtime version must co-exist with Buffalo, which uses a large portion of that for itself. The ROM version will probably relocate both code and data.

Chapter 7

The Small C Library

Introduction

This chapter discusses the Small C library, as implemented on the M68HC711 Microprocesser.

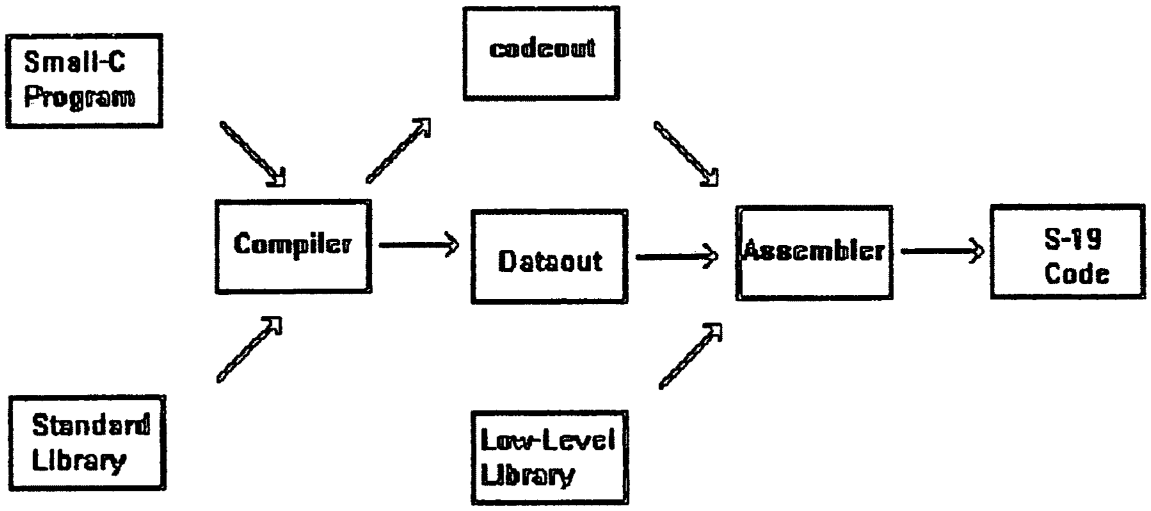


Figure 10. The Compiling and Assembling Sequence

The library consists of any code required for execution of a program that is not generated by the compiler or is not part of the Small C language. Using this definition, there are actually two libraries for Small C on the HC11.

The Low Level Library

The low level library, called 6811_LIB.C, is a collection of routines which are required by the language itself, and must be included in all compilations. For instance, a logical operation such as "<" (greater than) is implemented by the compiler as a call to an external routine. The low level library must supply that routine. Low level library routines are written in assembly code, and implement logical operations, peek and poke, the enabling and disabling of interrupts, multiplication of integers, and so forth. Since they are written in assembler, they are not compiled with the Small C source code. Instead, they are merged with the compiler output using the DOS copy command. Then the merged file is submitted to the assembler.

The user-callable functions present in this low-level library are listed in Table 2.

Table 2

User-Callable Functions in the Low-Level Library

Operation	Function Name
Enable interrupts	`e_int()`
Disable interrupts	`d_int()`

Set the bits	`bit_set(addr, val)`
	`int addr;`
	`char val;`
	(Called with 1's in the position to be set)

Clear the bits	`bit_clear(addr, val)`
	`int addr;`
	`char val;`
	(Called with 1's in the position to be cleared)

Poke a byte	`pokeb(addr,val)`
	`unsigned int addr;`
	`unsigned char val;`

Poke a word	`poke(addr,val)`
	`unsigned int addr;`
	`int val;`

Peek a byte	`peekb(addr)`
	`unsigned int addr;`

Peek a word	`peek(addr)`
	`unsigned int addr;`

The Standard Library

The second library used by Small C is an implementation of a subset of the C library. Small C depends on its library for most of its power, as does standard C. The library for Small C for the HC11 is created by writing low-level code to implement whatever is unique to the HC11 processer and using the Small C library, as defined in Version 2.2 for the IBM-PC for all other functions.

Input and Output

Two fundamental functions in Small C, fputc() and fgetc(), must be defined in terms local to the HC11 environment, and then used by all of the more sophisticated functions. This means that the entire Small C Version 2.1 (IBM-PC) library can be ported without change once these two functions are in place. Each of the two fundamental functions call Motorola-specific routines which in turn call or duplicate Buffalo routines for input and output. The entry points for the Buffalo routines are guaranteed by Motorola to remain valid throughout the supported life of the HC11. If it is not desired to use Buffalo, the required Buffalo routines can easily be extracted and used alone. In this case, the routines must be initialized on start up by some sort of run-time startup utility, but this too is not difficult.

The two Motorola HC11 specific routines which support fputc() and fgetc() are called SER_out and SER_in respectively, and are listed here for reference.

```
/* I/O Device Driver for Motorola's 68HC11 EVB */
/* Written by M. Taylor */

SER_out(ch) char ch; {
      ch=ch;      /* Clears the upper byte of register D */
#asm
      LDD 6,Y     /* Get Param 1 into accumulator */
      TBA         /* Shift low byte up            */
      JSR $FFB8   /* Output register A to Buffalo */
#endasm
}

/* Input Driver for 68HC11 Serial Port */
/* Written by James Hofmann */

SER_out(ch) char ch; {
#asm
JEKS  LDAB  $102E  /* Get control byte */
      BITB  #$80   /* Test for valid data bit set */
      BEQ   JEKS   /* Loop until valid data bit set */
      LDAB  7,Y    /* Put low byte of param 1 into B */
      STAB  $102F  /* [B] --> Serial Output Register  */
#endasm
}
```

A small portability problem exists with the constants $102E and $102F. They aren't absolutly constant. The base address for the registers used by the M68HC11 MCU can be changed. However, use of variables for the constant would require the programmer to define some base register, or the definition would have to exist in some header file. Furthermore, for consistancy, the correct register names would have to be used, requiring inclusion of the header file 68HC11.H. Perhaps further development will necessitate these changes, but for now the increased complexity of every compilation seems unwarrented. As a compromise, comments concerning the assumptions and restrictions are included in the library code.

A Buffalo routine to output a string exists, and nothing prevents using that routine to define fputs(), except that to do so would create one additional interface location between Small C and Small C for the HC11.

The rest of the Small C for the HC11 library is directly copied from the Small C library, which is itself taken in great part from Kernighan and Ritchie C.

Header Files

There are two predefined header files in Small C for the HC11. The header necessary for input and output is stdio.h. It contains links between ser_in()/ser_out() and stdin/stdout, and other modifications to make HELLO.C work in Small C for the HC11 exactly as written in the Kernighan and Ritchie example.

The second predefined header file is called 68HC11.H. It contains all of the registers defined in the Motorola literature with an offset to a base address. These registers are completely defined in Motorola literature and are generally located at address $1000. This base address can be changed, so the header file gives the registers as offsets. The programmer is left to define a base to the register array. Examples of this usage may be found in the sample programs.

Chapter 8

Sample Programs

Introduction

This chapter introduces the use of the Small C compiler and the EVBU through a sequence of sample programs. The programs are not intended to be illustrative of the sample uses for which the board is intended, nor are they intended to be useful at all. They serve merely to demonstrate some of the necessary skills involved in using Small C on the EVBU.

Hello.c

HELLO.C is the program from Kernighan and Ritchie, first edition, which is traditionally used to introduce the compiler and system dependencies. It is assumed that variations on this and all the other original programs will be experimentally compiled and executed. The important things to note for this sample are:

1. Several include files are necessary for even the simplest program.
2. Printf() requires the most include files of any of the library functions.
3. The compiling sequence can be automated quite successfully by a batch file in MS-DOS. This batch file can be used to compile any single-module program.
4. Once past the include files and the preprocessor material necessary for the EVBU, the program itself is completely equivalent to the Kernighan and Ritchie version.

5. The size of the overhead functions, library, and included I/O modules is about 2k. This remains constant for all programs, and is about the lower limit for a program that does I/O using printf(). Considerable savings can be made by using puts() for string output.

```
/* hello.c
 * Small c for the 68HC11
 */

#code 0x60F0
#data 0x6000

#include "startup.c"
#include "stdio.h"
#include "fprintf.c"
#include "atoi.c"
#include "fopen.c"
#include "fputc.c"
#include "itoa.c"
#include "itoab.c"
#include "strlen.c"
#include "reverse.c"
#include "is.c"
#include "ser_out.c"

char msg1[] = "Hello, World!\n";

main()
{
     printf("%s", msg1);
}
```

Note that the message is defined globally, that is, in the data area. The reader may want to redefine it locally and see what changes this causes. Note that the message is now relocated in the code area, after the function code.

Fibonacci Numbers

Most of the standard expressions available in C are supported in Small C. This example shows array definition and manipulation.

Uncomment the global array definition and comment out the local array definition to see how the compiler redefines the array. The programmer must know what the compiler will do, since the programmer must ultimately manage the stack and give the starting address to Buffalo. If the array is declared globally, then the stack can be smaller, but the memory will be allocated permanently. Perhaps in another program, a large stack can be defined and used by different functions for different things, re-using the memory in a very efficient way. Since we measure memory in bytes, rather than kilobytes or megabytes on this level, it can make a very big difference how the programmer uses memory. Note that the compiler will allocate the string "Fibonacci..." three times, clearly not very efficient. A better construction would be:

```
char msg[] = "Fibonacci(%d) = %d\n";
```

and then use it in calls like:

```
printf(msg,i,a[i]);
```

It will be stored in the data area only once, for a clear saving of 40 bytes.

```
/* The Fibonacci Series in Small C for the HC11 */

#code 0x60F0
#data 0x6000

#include "startup.c"  /* "jmp _main" to start the asm output   */
#include "stdio.h"    /* Defines stream in fputc                */
#include "fputc.c"
#include "ser_out.c"  /* Driver for EVBU serial port            */
#include "reverse.c"  /* Needed by itoa()                       */
#include "strlen.c"   /* Needed by reverse()                    */
#include "itoab.c"    /* Needed by printf()                     */
#include "itoa.c"     /* Needed by printf()                     */

#include "is.c"       /* isdigit(), etc.                        */
#include "fprintf.c"    /* Formatted output                     */
#include "atoi.c"       /* Needed by printf()                   */

/* int i, a[15];*/      /* Declared globally in data area       */

main()
{
    int i, a[15];  /* Declared locally and put into the stack  */

    a[0] = 0;
    a[1] = 1;
    printf("Fibonacci(0) = %d\n", a[0]);
    printf("Fibonacci(1) = %d\n", a[1]);
    for(i = 2; i < 13; i++) {
        a[i] = a[i-1] + a[i-2];
        printf("Fibonacci(%d) = %d\n", i, a[i]);
        }
}
```

<u>Sieve of Eratosthenes</u>

At one more level of complexity, we have a standard C test program, easily translated into Small C.

```
/* Eratosthenes Sieve Prime Number Program in C,
 * Byte magazine, January 1983.
 * The code has been corrected to produce primes correctly.
 */

#code 0x6000   /* Code begins execution here */
#data 0x7000   /* Data area begins here */

#include "startup.c"   /* Must come after #code and #data */
#include "stdio.h"
```

```
#include "fprintf.c"
#include "atoi.c"      /* Needed for printf() */
#include "fputc.c"     /* Needed for printf() */
#include "itoa.c"      /* Needed for printf() */
#include "itoab.c"     /* Needed for printf() */
#include "strlen.c"    /* Needed for printf() */
#include "reverse.c"   /* Needed for printf() */
#include "is.c"        /* Needed for printf() */
#include "ser_out.c"   /* Driver for serial port */

#define TRUE    1
#define FALSE   0
#define SIZE    2000

char flags[SIZE+1];

main () {
    int i, k, count;
    count = 0;

/* Set the array to all TRUE */
    for(i = 0; i <= SIZE; i++)
        flags[i] = TRUE;

    for(i = 2; i <= SIZE; i++ ) {
        if ( flags[i] )  {  /* Found a prime number */
            for ( k = i + i; k <= SIZE; k += i )
                /* Mark all multiples NOT PRIME */
                flags[k] = FALSE;
                count++;          /* The number of primes */
            }
        }
        printf("%d primes\n", count);
        display();    /* Display the elements that are prime */
    }

/* display the prime numbers, 16 to the line */
display() {
    int i, line;
    line = 0;
    printf("Primes are:\n");
    for( i = 0; i < SIZE; i++) {
        if(flags[i]) {  /* This element is a prime number */
            printf("%d ", i);
            line++;
            if(line == 16) { /* one line full */
                printf("\n"); /* End the line with linefeed */
                line = 0;
            }
        }
    }
    printf("\n"); /* End the table of primes */
}
```

Interrupt Programming

The following program is from the read.me file for the compiler, and is assumed to have been written by Matthew Taylor. It is a simple illustration of interrupt programming. It must be compiled as a ROM-resident program to work as it is written. The program has been modified to show the improved library, and now uses printf() to the standard output by default rather than fprintf() requiring a named output file. The EVBU has only one serial output port, which may without loss of generality be called standard output. For some explanation of the interrupt code, see Chapter 4.

```
/* PROGRAM NAME: CLOCK.C */
/* PROGRAM TO DEMONSTRATE THE USE OF INTERRUPTS USING SMALL C.         */
/* PROGRAM WRITTEN FOR MOTOROLA 68HC11 BOARD RUNNING BUFFALO MONITOR. */
/* PROGRAM SIZE: 4200 BYTES                                            */
/*  NOTE: fprintf() uses 3500 bytes of those 4200 bytes- instead use  */
/*        fputs() (about 120 bytes) and itoa() (about 100 bytes)       */
/*        if memory is tight.                                          */
/*    ENSURE THERE ARE AT LEAST 0x80 BYTES OF STACK SPACE!!!!          */

/* Define where things should reside... */
#code    0x60f0
#data    0x6000
#include <startup.c>

/* Define where register bank is mapped... */
#define  REG_BASE 0x1000

#include <68hc11.h>
#include <stdio.h>

/* Link in the following libraries (nearly all needed by fprintf() )*/

#include <fopen.c>
#include <fputs.c>
#include <fputc.c>
#include <itoa.c>
```

```
#include <atoi.c>
#include <is.c>
#include <fprintf.c>
#include <itoab.c>
#include <reverse.c>
#include <strlen.c>

/* Link in the device driver for the Motorola 68HC11 EVBU */

#include <SER_out.c>

unsigned int    sys_clk;
char            buf[10];
/* Make the message live in ROM, so it's there on power up!! */
rom char message[] = "ELAPSED TIME: %u minutes and %u seconds.\n";

/* Define the ISR routine... */
/* Called whenever the system timer overflows (about every 0.5 seconds)
*/

interrupt toi() {
    sys_clk++;
 /* Reset the timer overflow interrupt flag so it can happen again */
    bit_set(0x1025,0x80);
    }

main(){
/* Set the PreScale bits before the time protect timer expires */
/* This code will do nothing if Buffalo starts out of reset.   */
#asm
      LDX  #$1024
      LDAB #$03
      STAB 0,X
      LDS  #$7FFE   ;Attempt to set the stack pointer
#endasm
/* change the Timer Overflow Interrupt Vector to point to our routine */
    poke(0xD1,toi);
/* Clear the Timer Overflow Flag */
    bit_set(REG_BASE+TFLG2,0x80);
/*  Enable the Timer Overflow Interrupt */
    bit_set(REG_BASE+TMSK2,0x80);
/* Enable interrupts */
    e_int();
    sys_clk = 0;
    printf("Starting system timer...\n");
    while (1) {
        unsigned int temp;
        temp = sys_clk;
        while (temp/2 == sys_clk/2);
            printf(message,sys_clk/120,(sys_clk/2) % 60);
    }
}
```

Unless this program is present in ROM and started out of reset, the prescaler bits will not be set quickly enough and the clock will run "fast". Note the use of the predefined register and flag names used in the program. These are all defined in 68HC11.H.

Chapter 9

Jim's Multitasking Operating System

Introduction

This chapter describes a small, multi-tasking operating system written in Small C for the EVBU. The program serves to illustrate some real-time capabilities of Small C, and gives directions to further explorations that may be possibe.

Description

The main components of the operating system are:

Initialization

Scheduling

Wait

Signal

Software Interrupt

Hardware (Timer) Interrupt

The Operating System

The operating system is interrupt-driven. That is, each task in the system operates until interrupted. An interrupt can occur one of two ways. Either the time interval for that task expires, or the task is put to sleep by a semaphore to wait for a system resource.

In the case of a timer interrupt, the hardware defined timer interrupt vector causes execution of the timer interrupt function to begin. This function stores the stack pointer of the currently executing task in its task control block for later use. Note that the HC11 MCU registers are saved on the current stack as a result of the interrupt itself. The timer interrupt function then calls the scheduler for a new task to execute. The scheduler may replace the saved stack pointer with another from the task control block array. In any case, when control returns to the timer interrupt function, the stack pointer is reloaded and a normal return from interrupt occurs. The MCU pops all the registers for the new current task from the stack designated by the scheduler, and the new task begins running exactly where it left off when it was interrupted.

In the case of an interrupt caused by a wait on a system resource, the interrupt is generated by the semaphore wait function, which puts the current task to sleep by changing its status in the task control block and then calling the scheduler. The scheduler then selects a task to run from among those awake and another task runs in place of the sleeping task. The task remains asleep until awakened by a signal operation. The status field of the task control block consists of the address of the semaphore that the task is sleeping on, or the null address, indicating that the task is not sleeping.

The semaphores used are technically called blocked set semaphores (Ben-Ari 60), but because of the rotating method that the scheduler uses, they produce the same results as blocked queue semaphores.

The scheduler uses round robin scheduling (Tannenbaum 81), insuring no starvation. Tasks are chosed from those awake. The scheduler does not put tasks to sleep or wake them up. Only the semaphore functions do that.

Initialization is somewhat more devious and not all the code is obvious. The central problem to be solved is the problem of starting a system from a static situation. The tasks to be started have not been interrupted yet. Essentially, an interrupt condition is artificially created from which each task can return from the first time.

Initialization creates an array of task control blocks for the tasks that are to execute, and initializes that array. Then it creates a stack in RAM for each task, and artificially sets the address of the task in the stack where it would be if the task had been interrupted. The address is thus correctly retrieved when the task is selected by the scheduler for execution.

Having set the stage to be such that the current task has just been selected, initialization executes a return from interrupt instruction, the MCU retrieves the stack pointer for the current task, and then retrieves the registers from the current stack, and "Continues" with the execution of the current task.

There is no provision for stopping the system. Pressing the reset causes a non-maskable interrupt to occur, which eventually transfers control to the Buffalo monitor. Unplugging the power supply will also stop the system.

The Tasks

All of the operating system source code is listed together after the task functions are defined.

The tasks to run under this operating system were selected to demonstrate multitasking and interprocess communication. The first two tasks cooperate in a producer/consumer configuration, with the producer producing a time stamp from the free-running timer and the consumer displaying that time in a matrix on the screen of the Heath H-19 terminal. Interprocess communication is via a buffer of timer values, guarded from simultaneous access by a pair of semaphores called Elements and Spaces. This demonstration is coded directly from the pseudo code in Ben-Ari, and serves to demonstrate that such pseudo code can be directly translated into a real program using Small C (Ben-Ari 57).

Task three is a simple counter, which shares the screen with the other tasks through the Screen semaphore. The final task polls a switch on port A. The important things about task 4 is the ease with which hardware events are translated into program code using Small C, and that three tasks can share a common resource.

There is nothing important about having four tasks. Four seemed adequate for demonstration purposes. Tasks could be added by modifying the initialization code in an obvious way, changing MAXTASKS, and so forth. The size of the stack for each task is currently set at 150 bytes. This is adequate for the task with the most demanding requirements. If desired, some tasks could have different sized stacks.

The program could be placed in ROM quite easily.

```
/* tr16.c  The producer consumer problem solved in Small C for the HC11
 * using semaphores.  Taken from M. Ben-Ari Principles of Concurrent and
 * Distributed Programming.
 * Adapted by James Hofmann
 */

#data 0x6000
#code 0x6300

#include "startup.c"
#include "stdio.h"
#include "fputc.c"
```

```
#include "fputs.c"
#include "fprintf.c"
#include "atoi.c"
#include "fgetc.c"
#include "itoa.c"
#include "reverse.c"
#include "itoab.c"
#include "strlen.c"
#include "is.c"
#include "SER_out.c"
#include "SER_in.c"
#include "h19drv.c"       /* Provides direct screen accessing */
#define REG_BASE 0x1000
#include "68HC11.H"

/* The producer consumer problem from Ben-Ari */

#define N 5 /* Number of items in the circular buffer */

unsigned int B[N];      /* the circular buffer */
int In_Ptr, Out_Ptr;    /* pointers into the buffer */
int Elements, Spaces, Screen;  /* Semaphores */

/* Gets a time stamp and converts it to a time string. */
producer()
{
    unsigned int tmstmp; /* The product */

    while(1) {
        tmstmp = *(REG_BASE+TCNT);
        wait(&Spaces);              /* Start critical section */
        B[In_Ptr] = tmstmp;
        In_Ptr = (In_Ptr + 1) % N;
        signal(&Elements);          /* End critical section   */
    }
}

/* Displays a time string on the terminal */
consumer()
{
    unsigned int i;
    int j, row, col;
    while(1) {
        wait(&Elements);             /* Start critical section */
        i = B[Out_Ptr];
        Out_Ptr = (Out_Ptr + 1) % N;
        signal(&Spaces);             /* End critical section   */
        if(Out_Ptr == 0)
            j = (j + 1) % N;
        row = 10 + j * 2;
        col = 10 + 10 * Out_Ptr;
        wait(&Screen);              /* Start critical section  */
        gotoxy(row, col);
        printf("     ");
        gotoxy(row, col);
        printf("%u", i);
        signal(&Screen);            /* End critical section    */
    }
}

/* Counting task */
counter()
{
    int cnt1, cnt2;
    cnt1 = 0;
    cnt2 = 0;
    while(1) {
        cnt1++;
        if(cnt1 % 1000 == 0) {
            cnt2++;
            wait(&Screen);          /* Start critical section */
            gotoxy(1,60);
            printf("%d",cnt2);
            signal(&Screen);        /* End critical section   */
        }
    }
}

/* Read Port A */
button()
{
    int a,b,c;
    c = 0;
    while(1) {
        a = *(REG_BASE + PORTA);
        if(b != a) {
            b = a;
            c++;
            wait(&Screen);        /* Start critical section */
            gotoxy(1,10);
            printf("Switch toggled: %d", c);
            signal(&Screen);      /* End critical section   */
        }
    }
}

main()
{
    Spaces = N;
    Screen = 1;                 /* Only one screen */
    cls();                      /* Clear the screen */
    gotoxy(1, 53);
    puts("Count:");
    cur_off();                  /* Turn off cursor  */
    init();
}


/* Jim's Multitasking Operating System */

#define AWAKE       0
#define STACK       0
#define STATUS      1
#define MAXTASKS    4
#define TCBSIZ      2
#define STKSIZ      150
#define BIT7        0x80
#define TOV         0xD1    /* Vector for timer interrupt */
#define SWI         0xF5    /* Vector for software interrupt */

char stack0[STKSIZ];
char * top0;                /* Marker set at top of stack area */
char stack1[STKSIZ];
char * top1;
char stack2[STKSIZ];
char * top2;
char stack3[STKSIZ];
char * top3;

int TCBTbl[MAXTASKS * TCBSIZ];
/* The table of task control blocks.  Each TCB
 * holds a stack pointer and a status integer.
*/

int curtask;   /* The current task number */
int sp;        /* holds the stack pointer during the swap */

/* Initialize the operating system */
init()
{
    char * p;   /* Used to "Convert" an integer to a pointer */

    curtask = 1;

    /* Set up Task Control Block array */
    TCBTbl[0 * TCBSIZ + STACK] = &top0;
    TCBTbl[1 * TCBSIZ + STACK] = &top1;
    TCBTbl[2 * TCBSIZ + STACK] = &top2;
    TCBTbl[3 * TCBSIZ + STACK] = &top3;
    p = TCBTbl[0 * TCBSIZ + STACK];
    /* Put the program counter into the initial stack /
    poke((p - 1), consumer);
    p = TCBTbl[1 * TCBSIZ + STACK];
    poke((p - 1), producer);
    p = TCBTbl[2 * TCBSIZ + STACK];
    poke((p - 1), counter);
    p = TCBTbl[3 * TCBSIZ + STACK];
    poke((p - 1), button);
    /* Move sp into position for first call */
    TCBTbl[0 * TCBSIZ + STACK] -= 9;
    TCBTbl[1 * TCBSIZ + STACK] -= 9;
    TCBTbl[2 * TCBSIZ + STACK] -= 9;
    TCBTbl[3 * TCBSIZ + STACK] -= 9;

    sp = TCBTbl[1 * TCBSIZ + STACK];

    TCBTbl[0 * TCBSIZ + STATUS] = AWAKE;
    TCBTbl[1 * TCBSIZ + STATUS] = AWAKE;
    TCBTbl[2 * TCBSIZ + STATUS] = AWAKE;
    TCBTbl[3 * TCBSIZ + STATUS] = AWAKE;

    /* Change the Software Interrupt Vector to point to our routine */
    poke(SWI, swi);

    /* Change the timer overflow interrupt vector to point to our routine */
    poke(TOV,toi);

    /* Clear the timer overflow flag */
    bit_set(REG_BASE+TFLG2, BIT7);

    /* Enable the Timer Overflow Interrupt */
    bit_set(REG_BASE+TMSK2, BIT7);

#asm
    LDS _sp       ;Load the stack pointer for this task
    RTI           ;Start the task
#endasm
}

/* Timer interrupt */
interrupt toi()
{
#asm
    SEI           ;Disable interrupts.  No nested interrupts
    STS _sp       ;Store the stack pointer
#endasm
    sched();     /* Call the scheduler for a new process */
/* Reset the timer overflow interrupt flag so it can happen again */
bit_set(REG_BASE+TFLG2, BIT7);
```

```
#asm
    LDS _sp     ;Load the new stack pointer from memory
    CLI         ;Enable interrupts
#endasm
}

/* Software Interrupt */
interrupt swi()
{
#asm
    SEI         ;Disable interrupts.  No nested interrupts
    STS _sp     ;Store the stack pointer
#endasm
    sched();    /* Call the scheduler for a new process */
#asm
    LDS _sp     ;Load the new stack pointer from memory
    CLI         ;Enable interrupts
#endasm
}

/* Scheduler */
sched()
{
    /* Save the previous task's stack pointer */
    TCBTbl[curtask * TCBSIZ + STACK] = sp;

    /* Select a new task from those awake */
    curtask = (curtask + 1) % MAXTASKS;
    while (TCBTbl[curtask * TCBSIZ + STATUS] != AWAKE)
        curtask = (curtask + 1) % MAXTASKS;
    /* Restore the new stack pointer from the Task Control Block */
    sp = TCBTbl[curtask * TCBSIZ + STACK];
}

/* semaphore routines */

/* Also called down or P */
wait(s) int *s;
{
    d_int();            /* Enter critical section */
    if(*s > 0) {
        (*s)--;
        e_int();
        return;
    }
    /* Store the name (address) of the semaphore in the status field */
    TCBTbl[curtask * TCBSIZ + STATUS] = s;
#asm
    SWI     ;Software Interrupt
    CLI     ;Enable interrupts, leave critical section
```

```
#endasm
}

/* Also called up or V */
signal(s) int *s;
{
    int t;

    d_int();             /* Enter critical section               */
    if(*s == 0) {        /* Wake up a task sleeping on this semaphore */
        t = 0;           /* if possible                          */
        while (t < MAXTASKS) {
            if(TCBTbl[t * TCBSIZ + STATUS] == s) {
                TCBTbl[t * TCBSIZ + STATUS] = AWAKE;
#asm
    SWI         ;Software Interrupt
    CLI         ;Enable interrupts, leave critical section
#endasm
                return;
            }
            t++;
        }
    }
    (*s)++;
    e_int();             /* Leave critical section */
}
```

Chapter 10

Conclusion

<u>What was Accomplished?</u>

1. Matthew Taylor's Small C for the EVB was implemented on the EVBU. Small C, originally written for Z80, rewritten for Intel microprocessors of the 80x86 family, rewritten for Motorola 68xx microprocessors, rewritten by Matthew Taylor for the Motorola M68HC11 series embedded systems microprocessors as present in the Motorola EVB environment was re-implemented for the Motorola EVBU.

2. Primitive I/O functions SER_in and SER_out were written so that the Version 2 Small C library could be ported to the EVBU. The library port was accomplished.

3. A Small C Compiler Users Manual created for Small C for the EVBU.

4. Downloading to the EVBU, communicating with the EVBU, and developing Software on the EVBU are discussed in useful detail.

5. A multitasking operating system was written in Small C and demonstrated on the EVBU.

6. 8K External RAM was added to EVBU to facilitate software development.

<u>Why is it Important?</u>

Software Development in a mid level language is now possible on the EVBU. A higher level language than Small C comes at a high cost, and produces executables that are too large for a small, embedded systems environment. Assembler language is too difficult to use for complex programs.

Suggestions for Further Research

1. Tasks could be developed to run under the simple multitasking system presented here. These tasks could duplicate the tasks present in the Motorola freeware multitasking operating system, MCX11, written in assembler language. Comparisons of the code produced, the effort required to modify that code, and the overall efficiency of the Small C system and the assembler-based system could be made.

2. The real-time multitasking operating system presented here could be expanded to include message passing and monitor operation, either directly or by the use of the semaphore construct already present.

3. Stand-alone programs could be written for implementation on the EVBU in Small C. These programs could explore the features of the M68HC11 MCU in greater detail than is possible using programs written in assembler.

4. Features of embedded systems development, such as multi-level interrupts, could be implemented in software using Small C.

Appendix A

Adding External RAM to the EVBU

To provide resources for development using the Small C compiler, 8k of SRAM can be added to the EVBU. This is a simple task in theory, but actual attempts to perform this task resulted in several false starts. These will all be addressed here, so that they need not be repeated.

The easiest part proved to be the design. I simply used the schematic diagram for an EVB, with a few simple modifications, and it worked fine. This procedure was confirmed by Motorola personnel. A schematic diagram is included.

The first attempt to implement this design involved push-in wires onto a prototyping board with spring contacts. This very nearly worked. The circuitry created on the prototype board was coupled, via a prefabricated ribbon cable with plug, to the header provided on the EVBU, and made a neat arrangement. However, the connections from EVBU to prototype board through the ribbon cable proved to be too long, with the result that data was corrupted as described below.

Analysis of Data Problems

Filling the external memory using the Buffalo BF (Buffer Fill) command, then reading the data using Buffalo MD command demonstrated the problem. Corruption occurred during "Rollover" periods, when several address pins which had been 1 rolled over simultaneously to a new subrange involving a 1 bit in a more significant position, and at least 13 bits alternating from 1 to 0 at the same time. This sudden change in voltage levels caused damped oscillation on the address lines, which did not completely stabilize again until some 60 or so address bus cycles had occurred.

The header plug and long ribbon wire combination was removed, and wiring to the EVBU header was made as short as possible. The resulting scheme worked better, but there were still data errors from time to time.

The next attempt to improve the reliability was to abandon the prototype board and attempt wire-wrapping sockets on the EVBU's own prototype area. Again, this very nearly worked. However, using wire several sizes larger than appropriate for wire-wrapping use produced wiring that could not be visually inspected. An ohmmeter was used as a continuity checking device, and the current from the ohmmeter, indiscriminately used to test the pins of P4, destroyed Port B of the EVBU. This was confirmed in a telephone conversation with the project engineer for the EVBU at Motorola. He explained that the reverse threshold on the pins was 0.3VDC. The output pins are designed to operate safely at voltages from 0 to 5 VDC, referenced to ground. "Ringing" the circuit, with no concern for lead polarity, reverse biased those pins, putting a negative voltage on the pins, with respect to ground, causing an unregulated current of some 100 ma to flow. The resulting heat destroyed the output transistors.

The rule is "<u>Never reverse bias any pins on a microchip assembly.</u>" Devices which operate correctly in a range of 0-5 VDC will sometimes show very low impedance when voltages below 0 are applied, and may be easily destroyed if significant currents are allowed to flow.

The third attempt, once the EVBU had been repaired by replacement of the MCU, was once again on the spring-contact prototype board, using _appropriate_ wire-wrap material and wrapped on the EVBU end at P4. This proved completely unsuccessful, because the small gauge wire would not make reliable contact with the spring connections on the prototype board.

Then it was discovered that break-off pin headers and steel wire-wrap pins could be forced into the spring contact prototype board, permitting wire-wrap on _both_ ends of the connecting wire. This was completely successful and proved to be both quick to assemble and economical.

Finally, it is my opinion that wire wrap to the prototype area on the EVBU, using appropriate wire-wrap stock, would also be very reliable and simple to do.

The final result is a small unit, easily fitting on a breadboard 8 x 10 inches, which combines the ease of mechanical interfacing with the MCU with the software development possibilities offered by the more expensive EVB configuration, for very little more than the cost of an EVBU.

Data Bus Termination

Motorola Memories explains that the data bus is nothing other than a transmission line carrying radio frequency energy, and should be treated as such. Accordingly, they recommend terminating the data bus for proper impedance matching, and suggest 10k ohm resistors connected to Vcc for this purpose. These pullup resistors did prove to be necessary for reliable transfer of data.

A problem similar to the buffer fill problem developed when long, compiler generated, S-record files were being transferred from the host to the EVBU. After about a thousand memory locations had been written during the course of a download, a single write would fail, producing a "rom" error. The problem could be solved by slowing the data transfer rate by putting a 5ms delay loop into the transfer program.

Data transfer from host to EVBU is based on simple "Dump and Run" technology. The design of the EVBU assumes that each transaction begins with the EVBU waiting on data from the host. When data is received, the EVBU processes the received data by writing it to a memory location, then reading the same location immediately afterwards to insure that the data was correctly stored.

The first solution provided for a longer wait between attempts to write, giving the data bus enough time to stabilize between bursts of rf energy. When the terminating resistors were added, character pacing was no longer needed. It seems that the data bus is much more stable with the terminating resistors. This is the preferred solution, since otherwise access to RAM without user-imposed delays would not be reliable.

Power Supply RF Bypass

As suggested by Motorola, a 0.01 uFd capacitor was added from the Vcc pin of the memory chip to ground. This capacitor puts Vcc at ground potential at rf frequencies, preventing rf currents from existing on the power supply line. As suggested, this capacitor is placed physically close to the Vcc pin of the RAM chip.

The final prototype consisted of an EVBU, jumpered for Extended-Expanded mode of operation, and connected by wire-wrap to a separate prototype board with spring connectors holding the extra chips. Four integrated circuit chips were required for the test prototype. They are described as follows:

74LS138 - Multiplexer described in The TTL Data Book. This device uses the top three address lines to select one of eight pins to be held low. The pin brought to active low selects the memory chip, and moving the RAM chip select pin from one to another of the eight output pins selects the addresses to be stored in the RAM chip. This can be done in 8k increments throughout the 64k total address space. For prototyping purposes the chip was wired to select $6000-$7FFF, although other address ranges were also found to be valid. If an external memory location conflicts with internal memory locations in the MCU, the MCU takes priority and those locations in external RAM are not accessible. Thus it is a simple matter to configure the entire 64k without fear of interfering with MCU internal RAM. Those locations in external RAM that are duplicated in internal RAM are simply ignored.

74LS05 - Hex Not Gate described in The TTL Data Book. Only one section used, for the inverse of the clock, needed by the RAM memory chip. Note 1k Ohm resistor required for output loading.

74LS373 - Data Latch described in The TTL Data Book. Port C on the MCU alternately provides data and the lower eight bits of the address. The address byte is latched in this chip. Once data appears on Port C, both data and the previously provided address are available on the RAM chip.

6264LP - SRAM described in Motorola Memories. Provides storage and retrieval for 8192 bytes of data with 100 ns access time. Addressing 8k requires 13 address lines. Five come directly from Port B of the MCU, and the lower eight are supplied by the latch from Port C.

Appendix B

Installing the Buffalo Monitor

Introduction

Installation of the Buffalo into the EPROM of an M68HC711 MCU chip involves both hardware and software activities. In general, programming the EPROM requires that 12 volts, rather than the normal system voltage of 5 volts, be present on the XIRQ pin of the MCU. Then, the software in the form of Motorola S-records is downloaded through the bootstrap download procedure. The preliminaries described below can be made in any order, and then the EPROM programming sequence should be followed in order. The complete procedure is described variously in Motorola documentation. The most complete descriptions are to be found in AN1060[1] and a memo available from Motorola Engineering in Austin.[2]

Preparation

According to AN1060, the XIRQ pin must be isolated from the EVBU while the 12VDC programming voltage is applied. This involves cutting a trace and adding circuitry to put 12 VDC on the XIRQ pin of the MCU. According to the memo from Engineering, this trace cutting is not necessary, but **applying 12VDC without cutting the trace has not been tried.**

----

1. Motorola Semiconductor Application Note AN1060, (Motorola Inc., Literature Distribution Center, Phoenix, 1990). This AN is part of the EVBU student package.

2. This unpublished memo may be obtained by request from Motorola Product Engineering, Austin, 515-891-3837.

Once the hardware preparations are made, the Buffalo download can proceed. The software part is not as well documented by Motorola, at least not in a step by step manner. The sequence is as follows:

Preliminary Activities

These may be done in any order:

1. Find and assemble the Buffalo assembler source. The version used here is BUF32.ASM. Use the Motorola freeware assembler to create BUF32.S19. Include the - L switch to produce a useful listing file. Make sure you note the name and directory for the S19 file. You will have to specify this name and directory from inside the PCBUG11 program later.

2. Make the hardware modification described in AN1060, and leave the jumper in the normal mode. (This means that the EVBU is exactly as it came from Austin.)

EPROM Programming Activities

1. Get the EVBU talking to PCBUG11. This is done as follows:

   a. Power up the EVBU.

(The following steps use the BOOTSTRAP mode and are somewhat described in the literature in various places. Best and most complete is in AN1060.)

   b. Remove jumper from J7, so that XIRQ is disabled.

   c. Jumper J3, so that bootstrap mode is selected.

   d. Press <Reset> on EVBU.

   e. Start PCBUG11. See Appendix C for details.

   f. Test for successful communication with PCBUG11 commands.

<u>Once Communication is Established:</u>

Issue the following PCBUG11 commands:

i. Type CONTROL BASE HEX (Or be a lot better at translating from hex to decimal than I am!)

j. Type EPROM D000 FFFF (to set "Programming Mode" in this area).

k. Type EPROM (without arguments to check your work).

l. Move the jumper on your hardware modification to apply 12 VDC to the XIRQ line as specified in AN1060.

m. Type LOADS BUF32.S19.

n. Wait several minutes. There will be some fairly positive messages showing progress.

o. If all goes well, your EPROM now contains BUFFALO.

p. Remove the jumper supplying 12 VDC to restore the EVBU to normal operation.

<u>To Use The Buffalo</u>

1. Put jumper J2 over pins 2-3, enabling MONITOR operation.

2. Start Kermit to communicate with the EVBU at 9600 baud.

3. Reset the EVBU, or just power it up.

(NOT in Bootstrap mode. Put a jumper on J7, NO jumper on J3, 12 VDC <u>not</u> connected to the EVBU.)

4. You should be seeing the Buffalo prompt.

<u>Erasing the EPROM</u>

In case the attempt to program the EPROM fails for some reason, the following may be helpful.

An EPROM is programmed by setting selected bits from 1 to 0. The only way to set a bit to 1 is with ultraviolet light. Erasing an EPROM means putting it under an ultraviolet light for a period of time, to reset all bits set to 1. PCBUG11 has a command for testing erased EPROM.

Erasing the EPROM may be done with any of the garden variety EPROM erasers, according to Motorola. The cheapest did the job in half an hour. Motorola engineers suggested one hour. Follow the directions given with the EPROM eraser, and don't forget to remove and later replace the small cover over the EPROM window of the MCU.

Appendix C

Communicating with the EVBU

Introduction

Communicating with the EVBU requires planning. There are several things that can go wrong, and it is hard to debug communication problems.

Hardware

The hardware provided for communication with the EVBU consists of a UART with only partial functionality. No hardware handshaking is supported, and only three wires are used. Any protocol must be implemented in software.

Software

There are two software configurations which have been used to communicate with the HC11, one employing the Buffalo monitor and a terminal or terminal emulation program and the other employing the Motorola program PCBUG11 running on an IBM-PC. Each of these has advantages and disadvantages.

Buffalo and a Terminal (Emulator)

Buffalo has a simple user interface which can be accessed through any dumb terminal or terminal emulation program. In this configuration, all of the code to support communications is present on the EVBU, most of it in ROM. It would seem that this has the disadvantage of using most of the program space present on the EVBU for the Buffalo, but in actual use as a developmental board, this is not as much a problem as it seems. Since both program and data must reside in RAM during all but the final step of the developmental process, and Buffalo resides in ROM, there is little conflict.

Buffalo does no handshaking with the external terminal. Problems have occurred when the external terminal or terminal emulation can not keep up with the Buffalo's data transfer rate, either because of time sharing or I/O interruptions. Simply reducing the baud rate can sometimes solve missing data problems. If the host computer allows its communication task to wait for indefinite periods of time, either for multitasking or host I/O, however, no setting of the baud rate can guarantee that the host will be ready when the next character is transmitted.

Setting the baud rate can be done manually after Buffalo starts, or automatically by patching and reassembling Buffalo assembly code.

Buffalo requires most of the RAM space present on the chip itself. The program itself resides in either ROM or EPROM. Chips designated M68HC11xxx have Buffalo in ROM, while chips designated M68HC711xxx have EPROM in place of RAM, which can be programmed with Buffalo.

PCBUG11 and a PC

Motorola supplies a program which runs on an IBM-PC compatible computer and operates the HC11. The only software present on the EVBU when PCBUG11 is used is the "Talker". The talker can be located in either RAM or EEPROM, and may be either downloaded each time the program starts, using the bootstrap mode of the MCU, or may be present in EEPROM and started without downloading.

The HC11 is in "Special Test Mode" if PCBUG is used, because of the combination of extended mode to support external RAM, and bootstrap mode to support the talker. The MCU has slightly different capabilities in this mode from the capabilities that it has in the non-special modes (Normal and Extended only). Registers may be changed in Special Test Mode that normally may not be changed, for instance. It is conceivable that software developed in this mode might not operate correctly in the delivered mode.

The talker on PCBUG11 does not support handshaking in communications. PCBUG11 tries to determine the state of communications by counting cycles in the PC to determine if a response from the talker should have been received, or if an error has occurred with the talker. Unfortunately, this software is dependent on the system clock on the IBM-PC, and if the PC has an 80486 CPU running at 33mHz, PCBUG11 always times out and reports a communication error. The problem can be removed by slowing the speed of the PC to 16mHz or less.

The talker resides in MCU RAM, using all of it, or in EEPROM, freeing MCU RAM for developmental software. The talker in EEPROM could be a slight advantage during development over the Buffalo configuration as the RAM to be used in the final version for data storage could be used for that purpose as well during development. However, user programs that use EEPROM would have to share with the talker during development.

Although harder to pin down, it seems that it is slightly more complicated to operate the MCU through the talker than through Buffalo, including the complexity of downloading the talker every time the board is reset. No communication with the external terminal is required to follow a reset if the Buffalo is used. In the case of the talker, communications must be reestablished each time the board is reset. Talker communications have been observed to be less reliable generally than communications using Buffalo, although this could be an individual experience. Continued use of the talker, producing increased familiarity, could remove this objection.

<u>PCBUG Startup Sequence</u>

The following sequence has been followed to start PCBUG11 and establish communications with the EVBU:

1. Jumper removed from J7. (To disable XIRQ interrupts)

2. Jumper placed on J3. (To enable Special Bootstrap Mode)

3. Power to EVBU.

4. Start PCBUG11 using c> PCBUG11.

(The following is a walk-through of the PCBUG11 startup sequence.)

5. "Is the talker installed on your board? (Y/N)" N

6. "Do you wish to use the XIRQ interrupt? (Y/N)" N

7. "Which device are you using? :" F

(Response is -E)

8. "Do you wish to load a macro automatically? (Y/N)" N

9. "Which communications port are you using?" : 2

(to select COM2) (Response is port=2)

10. "Are you using an 8mHz crystal? (Y/N)" Y

11. "Command Line: PCBUG11 -E port=2"

"Press EXC to quit or any key to run PCBUG11" <spacebar>

The Speed Problem on the IBM-PC

Using an IBM-PC with an 80486 microprocessor running at 33 mHz as the host PC, PCBUG11 would not operate correctly until the speed was reduced to 16mHz. The symptoms were:

1. Apparently successful download of talker software. Lights on breakout box indicating properly. The host serial port is configured as DTE, and EVBU is configured as DCE so no null modem adapter required. Only TxD, RxD, and Gnd required or supported.
2. Many errors of the "Communications Fault" variety received. Unable to read or write more than a few bytes correctly. Use of Ctrl-R could, eventually, produce "Communications synchronized" message, and occasionally memory could be read. Occasionally the 68HC11 registers could be read.
3. All symptoms disappeared immediately when host computer slowed down to 16 mHz.

Bibliography

Ben-Ari, M. Principles of Concurrent and Distributed Programming. New York: Prentice Hall, 1990.

Hendrix, James E., A Small C Compiler. 2nd ed. Redwood City: M&T Books, 1990.

Kernighan, Brian W., and Dennis M. Ritchie. The C Programming Language. Englewood Cliffs: Prentice-Hall, Inc., 1978.

Motorola Inc., M68HC11 Reference Manual. rev. 2. Motorola Literature Distribution: Phoenix, 1991.

Motorola Inc,. MC68HC11E9/D HCMOS Single-Chip Microcontroller. Motorola Literature Distribution: Phoenix, 1988.

Motorola Inc., M68HC11EVBU Universal Evaluation Board User's Manual. Motorola Literature Distribution: Phoenix, 1990.

Motorola Inc., M68HC11EVB Evaluation Board User's Manual. Motorola Literature Distribution: Phoenix, 1990.

Motorola Inc., Memory Memories. Motorola Literature Distribution: Phoenix, 1991.
Motorola Memory

Plauger, P. J., The Standard C Library. Englewood Cliffs: Prentice-Hall, Inc., 1992.

Tannenbaum, Andrew S. Operating Systems Design and Implementation. Englewood Cliffs: Prentice-Hall, 1987.

Ward, Robert and Kenji Hino, ed. The C Users' Group Library. Volume II. Lawrence, Kansas: R & D Publications, Inc., 1989.