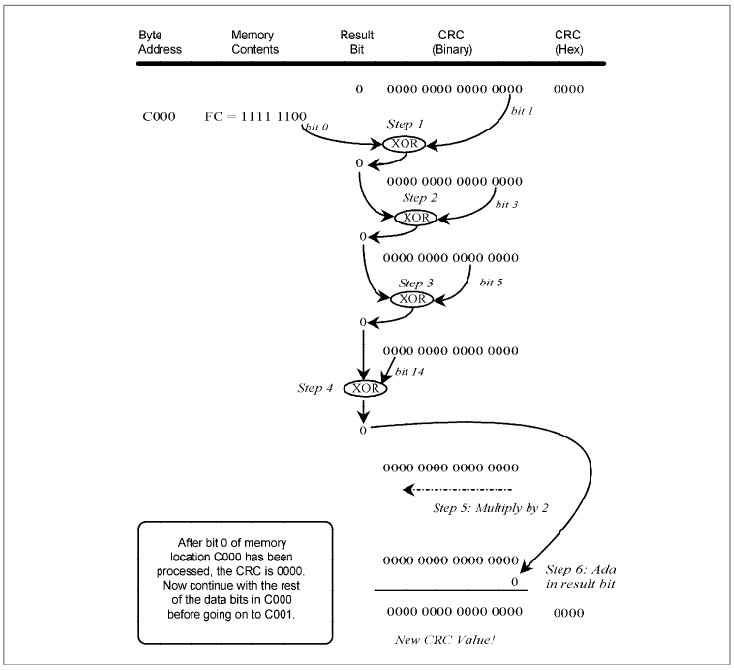
Micro Design 2

Lab #1



Programming with the ICC11 C Compiler

Introduction

Microcomputer Design 1 introduced you to the basic principles of design, construction and programming for an embedded microcomputer. You have successfully designed and built the kernel and you have written programs which exercise the digital I/O (parallel and serial). Among the many new challenges that await you in Micro Design 2, the first hardware topics will be analog interfacing (A/D and D/A). Adding new devices to the 68HC11 prototype board will also require a greater understanding of how to interface non-Motorola devices to the 68HC11. Following the A/D and D/A will be liquid crystal displays. The addition of an LCD will greatly enhance the information available to the user of an "embedded" computer system. Following the LCD is the HC11 Timer Subsystem. The Timer section also includes in-depth coverage of interrupts. Thus the ground-work has been set to investigate real-time programming concepts and (real) multi-tasking.

Before we add all of this new hardware to our micro boards we must work on our software. Lab #1 is an introduction to C programming for microcontrollers. While the intent in using C is to make your programming job easier, you may be surprised at the challenge this type of C programming presents. Hopefully your HC11 programs will fall together faster and your C and assembly language programming skills, in general, will improve.

As always, because we are building dedicated microprocessor systems, our objective is to write ROMable code. Fortunately this is a relatively easy matter given the straightforwardness of both the HC11 and the ICC11 C compiler. ICC11 is provided courtesy of ImageCraft Ltd. of Sunnyvale, California and like many of our development tools it has be provided free of charge.

So here's hoping that you enjoy the term and that you learn a great deal more about embedded systems.

Lab Outline

PRE-LAB: Endian Test Program

A short test program has been provided that should work under MSDOS on your PC as well as on your HC11 board. Obviously the program will need to be compiled for the appropriate target. The prelab is designed to give you a feel for the ICC11 compiler and the program development process.

SECTION 1: CRC Algorithm under Turbo C

You must create an algorithm which performs a Kontron CRC on a series of numbers. This program is to run under Turbo C and will be "ported" to the HC11 later.

SECTION 2: HC11 User Interface (UI)

Right off the bat it will be necessary to create a handful of routines for gathering input from a human and displaying that information on the terminal and PIA Test Board. Two basic families of routines will be developed: I/O routines and data conversion routines. There are no provided libraries with ICC11. If you need a routine, you must roll up your sleeves and build it. The UI must be completely bullet-proof.

SECTION 3: Combined Program

The previous two programs are merged to produce a working program on the HC11 which prompts the user for the addresses and then performs the CRC over the specified range.

SECTION 4: Program Execution from EEPROM

Making the code from the previous section ROMable should be a trivial matter. This will give you an opportunity to modify your own CRT.S start-up file and having gone from paper to EEPROM should give you a good understanding of the entire program development process using ICC11.

Pre-Lab

This prelab will require work outside of the lab as well as a checkoff at the start of your first lab period. You should copy the files indicated below to your own floppy then install ICC11 at home. Run the program below on both the PC and the HC11 board. At the start of your first lab class you will be required to demonstrate this program for your instructor.

( ) Step 1: Copy ICC11 Files

The first thing you must do is copy the executable files from the lab C:\ICC11 directory to your floppy so that they may copied to your computer at home. As with AS6811, you should create a new directory on your harddrive at home (C:\ICC11) and then copy all of the files to it. Also, add the new directory to your path. ICC11 can be run from the DOS command line or from within the Turbo IDE. You will be expected to do both.

( ) Step 2: Copy LAB1 files

On the lab computers you will notice a C:\ICC11\LAB1 subdirectory that contains a set of source files. You should take these home too and place them in a directory that will contain all of your Micro2 labs. These provided files are to give you a starting point. They contain all of the Micro 1 library routines converted to C. You should see that there is a PIA library in C which contains the equates and a delay( ) function. There should also be an SCI library which contains some equates, a puts( ) function and a cgets( ) function. The character functions getch( ) and putchar( ) are actually located in the CRT.S run-time assembly file, but the prototypes appear in the SCI library. Every time you compile a program it must be done in a directory that contains YOUR CRT.S file. After you copy the lab files to your floppy, edit each of the files to place your name in the header! What's missing to tie everything together and run a program is a C file with a main( ). That comes next.

( ) Step 3: Endian Program (MSDOS)

While it will not always be practical to move C programs back and forth from the PC world to the HC11 one, it is useful to understand that various pieces of code (such as algorithms) can easily be developed and tested in the familiar Borland IDE. Once the code is sound it can be moved to the HC11. The biggest drawback with this technique is that the I/O is vastly different. While both can support text based I/O, the PC does not have the control based I/O found on the HC11 (LED, D/A, A/D, ...).

The program in question is shown on the next page. Type the program in exactly, compile it under Turbo C and run it on the PC. This program must be in the same directory as your lab files.

```
#include "sci_lib.c"

typedef union {  unsigned char byte;
                 unsigned int  word; } TWOBYTES;

int main(void)
{
TWOBYTES MyWord;

MyWord.word = 1;

puts("\x1b[2J\x1b[1m\x1b[33m\x1b[5;26HMicro Design 2 — Lab #1");
puts("\x1b[7;28HEndian Test Program");

if(MyWord.byte == 1)
    puts("\x1b[10;22HThis computer is little endian!");
else
    puts("\x1b[10;24HThis computer is big endian!");

getch( );
return 0;
}
```

The #include will not upset Turbo C because it is being used to provide the various prototypes as well as the actual puts( ) C source. Hence you will notice a complete absence of other #include's.

If your PC has the ANSI.SYS driver installed you should get a colorful message when the program is executed. Without ANSI.SYS you will get a bunch of cursor positioning gibberish, but in either case you should be able to determine whether the PC is big or little endian.

( ) Step 4: Endian Program (HC11)

The same program should run without modification on your HC11 board. Using the information provided in class, compile the program from the DOS command line, download it to your HC11 and run it. Then go through the same steps from within the Turbo IDE. Once you have successfully run the program in all the various ways, answer the questions below and get your pre-lab checked off.

Q1. The IBM PC (Intel) is _____ endian.

Q2. The HC11 (Motorola) is _____ endian.

Have your Pre-Lab checked off at this point. You must demonstrate the program running on the PC and HC11. You must show HC11 compilation using both the IDE and DOS command line.

Procedure

Section 1: CRC Algorithm (Turbo C)

( ) Step 1: Background

The very nature of programmed memory devices (EPROM, OTP, EEPROM) makes them subject to long time reliability concerns by the designers of computer equipment. A typical failure mechanism of an EPROM is for cells to "drop bits". A dropped bit means that an erased state of '1' (high energy) leaks and falls to a '0' (low energy) state some time after being programmed. EEPROMs have other failure modes as well and all semiconductor devices are subject to weakening or destruction by ESD.

We have seen in Micro 1 that parity bits can detect errors in serial data transmission, but that they do a marginal job at best. The additive checksum found in every Motorola S19 record is again a poor method for detecting errors. Hard drive controllers, industrial networks, and other "mission critical" systems need a more useful method of detecting errors. That method is the Cyclic Redundancy Check (CRC).

As a simple comparison, the first three bytes in the Micro11 Monitor EPROM, along with the corresponding checksum and CRC are shown below:

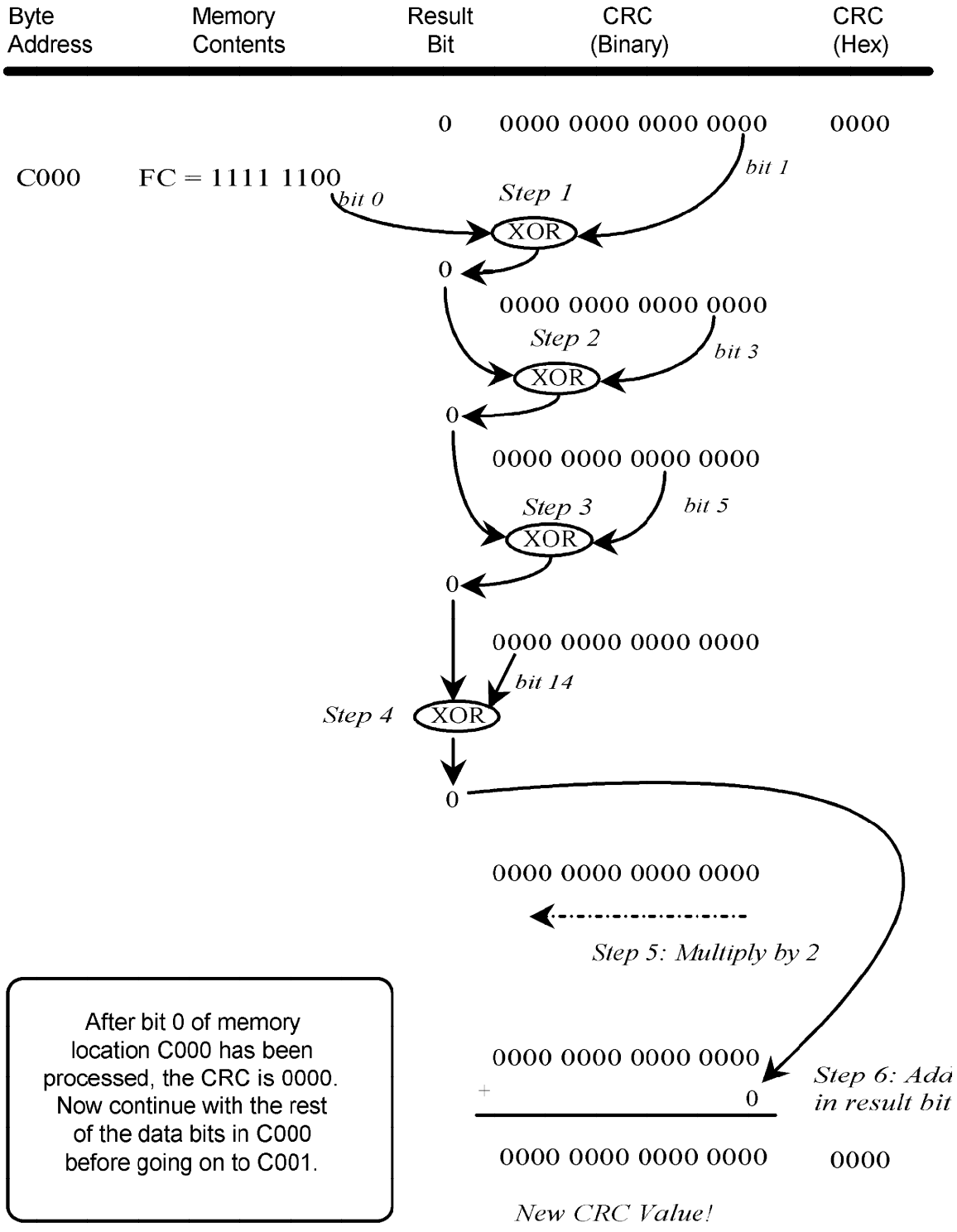
Address	Data	Address	Data
0xC000	0xFC	0xC000	0xFC
0xC001	0x7F	0xC001	0x7F
0xC002	0xFE	0xC002	0xFE
	———		———
Additive Result: 0x0279		CRC Result: 0x7123	

Both the checksum and CRC would detect a simple change in data such as the 0xFC becoming 0xF8 ("dropping" bit 2 in the lowest memory location). Now consider a simple switch in data:

Address	Data	Address	Data
0xC000	0x7F	0xC000	0x7F
0xC001	0xFC	0xC001	0xFC
0xC002	0xFE	0xC002	0xFE
	———		———
Additive Result: 0x0279		CRC Result: 0x30B8	

The CRC has detected the error, where the additive checksum has not. The Xeltek programmer employs only a simple additive checksum. Professional models such as the Kontron employ CRC methods. For this lab you will code, in C, the CRC algorithm used by the Kontron programmer.

Professional programmers often use CRC's to "hack-proof" their final programs. A quick CRC of a program as part of a start-up self-test can determine if any contents (even text strings) have been altered. This is a tool you should keep!



( ) Step 2: Algorithm Specification

The Kontron CRC algorithm performs six operations on each bit of each byte of memory under test to come up with a unique 16 bit signature. This algorithm has a very high probability of detecting any change in the EPROM's data.

Two variables will be required by the CRC (not including loop counters):

ResultBit: The algorithm requires a boolean variable (single bit) to hold intermediate CRC results. Since this value will be exclusive-or'ed against an unsigned int, it might be wise to use an unsigned int. Note that since the HC11 treats zero values so efficiently you may just consider this value zero or nonzero.

CRC: A 16 bit unsigned variable will be needed to store the CRC result.

The algorithm is performed on successive memory locations (eg. C000 to FFFF). Each bit in a byte under test (say, each bit in C000) is processed in the bit sequence 0, 7, 6, 5, 4, 3, 2, 1. Thus, the algorithm is performed on each bit of each byte under test. Therefore, when each bit of C000 has been processed then each bit of C001 must be processed, etc. . . .

The six steps for each bit are:

1. Bit 1 of the 16 bit CRC is exclusive-ORed (XOR) with the bit under test (recall that The order is 0 7 6 5 4 3 2 1). The result of this XOR is saved in ResultBit.

2. Bit 3 of the CRC is XORed with ResultBit. The result of this step is stored back in ResultBit.

3. Bit 5 of the CRC is XORed with ResultBit. The result of this step is stored back in ResultBit.

4. Bit 14 of the CRC is XORed with ResultBit. The result of this step is stored back in ResultBit.

5. Ignoring ResultBit for a moment, the CRC is multiplied by 2 and it's carry discarded.

6. Step 5 will result in a "vacancy" in bit 0 of the CRC (ie. a zero was shifted into the LSB). ResultBit is added to the CRC at bit 0.

These six operations are performed on each bit for C000 and then for each bit in C001, etc. The final result is obtained after all bits in all memory locations have been processed. The diagram on the previous page illustrates the six steps for the first bit under test for location C000. Note that at the top of the page the CRC has been initialized to 0000 before the program begins (same as the additive checksum).

The following table provides sample data for the first 3 bytes under test:

Location	Contents	Bit #	ResultBit	CRC (after all 6 steps)
				0000 0000 0000 0000 = 0x0000
0xC000	0xFC	b0 = 0	0	0000 0000 0000 0000 = 0x0000
		b7 = 1	1	0000 0000 0000 0001 = 0x0001
		b6 = 1	1	0000 0000 0000 0011 = 0x0003
		b5 = 1	0	0000 0000 0000 0110 = 0x0006
		b4 = 1	0	0000 0000 0000 1100 = 0x000C
		b3 = 1	0	0000 0000 0001 1000 = 0x0018
		b2 = 1	0	0000 0000 0011 0000 = 0x0030
		b1 = 0	1	0000 0000 0110 0001 = 0x0061
0xC001	0x7F	b0 = 1	0	0000 0000 1100 0010 = 0x00C2
		b7 = 0	1	0000 0001 1000 0101 = 0x0185
		b6 = 1	1	0000 0011 0000 1011 = 0x030B
		b5 = 1	1	0000 0110 0001 0111 = 0x0617
		b4 = 1	0	0000 1100 0010 1110 = 0x0C2E
		b3 = 1	0	0001 1000 0101 1100 = 0x185C
		b2 = 1	0	0011 0000 1011 1000 = 0x30B8
		b1 = 1	1	0110 0001 0111 0001 = 0x6171
0xC002	0xFE	b0 = 0	0	1100 0010 1110 0010 = 0xC2E2
		b7 = 1	0	1000 0101 1100 0100 = 0x85C4
		b6 = 1	1	0000 1011 1000 1001 = 0x0B89
		b5 = 1	0	0001 0111 0001 0010 = 0x1712
		b4 = 1	0	0010 1110 0010 0100 = 0x2E24
		b3 = 1	0	0101 1100 0100 1000 = 0x5C48
		b2 = 1	1	1011 1000 1001 0001 = 0xB891
		b1 = 1	1	0111 0001 0010 0011 = 0x7123

( ) Step 3: Program Definition

The following main program is to be used for this section of the lab. Your job is to write the CRC( ) function code. The programming is to be done under Turbo C (and/or UNIX).

```
/*************************************************************
* CRC.C    -  Basic Kontron CRC algorithm operating on a set of four  *
*             numbers.  Written in ANSI-C, but requires stdio.h.       *
*                                                                      *
*  by YourName Here                                                    *
*  Today's Date                                                        *
*************************************************************/

#include <stdio.h>

/*
| Prototype(s)
*/
unsigned int CRC(unsigned char *StartAddr, unsigned char *EndAddr);
```

```
/*******************************************************
* int main(void)                                       *
*                                                      *
* Requires: No command line parameters                 *
* Return value: 0 (always)                             *
*******************************************************/
int main(void)
{
unsigned char MemoryArray[3] = { 0xFC, 0x7F, 0xFE };
unsigned int  FinalResult;

FinalResult = CRC(&MemoryArray[0], &MemoryArray[2]);
printf("\nFinal CRC: 0x%04x", FinalResult);

return 0;
}


/*
.-------------------------------------------------------------.
| unsigned int CRC(unsigned char *StartAddr, unsigned char *EndAddr);  |
|                                                             |
| Performs a Kontron CRC algorithm over the address range provided.  |
|                                                             |
| Requires: StartAddr - address of lowest memory location under test.  |
|           EndAddr   - address of highest memory location under test.  |
| Returns:  The 16-bit Kontron CRC result value.              |
'-------------------------------------------------------------'
*/
unsigned int CRC(unsigned char *StartAddr, unsigned char *EndAddr)
{
    happy coding!
}
```

( ) Step 4: Verification

When you have verified that your solution works using the above three test values, check it against some other values courtesy your HC11 board. Do a quick Memory/Dump of the C000 block on the HC11 board and try running Options/Rom Test over some small ranges. Plug some of those Micro11 memory values into the MemoryArray[ ] and rerun the program. Once you are sure that your algorithm is operating properly, have it checked off.

Have a copy of your documented source file(s) available for viewing by your instructor. All files must be completely documented at the time of checkoff.

Section 2: HC11 User Interface

( ) Step 1: Library Routines

Several C-callable assembly language routines have been provided for you in CRT.S. They are the character routines putchar( ) and getch( ). Also, several "library" string routines are found in sci_lib.c. They are puts( ) and cgets( ). Any other routines you need for input, output or data conversion must be coded in C and placed in an appropriate include library file.

The objective for this section of the lab is to create a character-based front end for the CRC algorithm. Therefore, an operator will be entering the starting and ending addresses via the keyboard. The labcheck for Section 2 will be satisfied when you have a working program that does the following:

i. Initialize the SCI.
ii. Prompt the user with a colorful and attractive screen (containing your name and course number). The prompt should ask for a starting address. Using cgets( ), you must get a 4 character address from the user. cgets( ) must be properly used to prevent the user from entering more than 4 characters.
iii. After getting the starting address, convert that value to hex and store it at address 0x6000 (the particular address is of no concern, just place the hex value somewhere so it can be verified using memory dump after the program is finished. If the user entered an improper value, inform the user and return to step ii. The address input code must be bulletproof and flawless. Any input that cannot be properly interpreted as a hex address, must be rejected. Correct input would be an address expressed using the characters 0-9, a-f and A-F.
iv. In the same manner, prompt for and get the ending address. Place that address (in hex, not ASCII) at 0x6002. If the user entered an improper value, again inform Them and reprompt.
v. After pausing for a keypress or a predetermined delay, the program should repeat.

That's it! The "gotcha" with all of this is that you currently have no routines for data conversion or typechecking. Therefore, you must create your own routines for these various activities. Also, there should be no global variables in this program (or at any point in Lab #1).

You may use a global or an address expression like *(unsigned char*)0x6000 to store the test addresses values to memory. If you prefer, you may write the values to the LED a byte at a time, pausing with delay( ) in between.

SUGGESTION: Do not attempt to develop the whole application at once. Use your Turbo C environment to create a routines which perform ASCII-to-hex, ixdigit and/or upcase functions. All of these routines should work for both the PC and the HC11 in the same manner as the endian.c program did.

> Have a copy of your source file(s) available for check-off. All files must be completely documented at the time of checkoff.

Section 3: Combined Program on the HC11

( ) Step 1: Code Merging

1. Take the code from the previous two sections and combine them to create a complete program which prompts the user for starting and ending addresses and then Performs a CRC over the specified range. Watch out for the dreaded “address wrap”! Since you are using the supplied addresses in loops, there is the potential to create an infinite loop if the user enters FFFF for the ending address. You will probably have to treat this as a special condition.

2. As an additional specification, output each byte-under test to the LED display.

3. Once your program is operational, run it over the entire range C000 - FFFF. This should match the value reported by Micro11 for running the Options/Rom Test over the same range (as should any other set of EPROM addresses).

Have a copy of your source file(s) available for check-off. All files must be completely documented at the time of checkoff.

Section 4: Program Execution from EEPROM

( ) Step 1: Modifications to CRT.S

The only changes that need to be made are a few equates at the start of your CRT.S file. There are three equates in particular:

CODE This defines the starting code address. Usually this will be 0x1040 for execution From RAM. Change it to 0xC000 for execution from EEPROM.

GDATA The global data section normally resides somewhere past the end of the code Area (0x5000). There aren't any global variables, so this shouldn't be an issue. For future reference, ICC11 doesn't like global variables in the range 0x0000 - 0x00FF.

STACK For execution from RAM, this is normally set to the same value that Micro11 uses (0x7F6A). For the memory test program running from EEPROM move it to the top of the internal RAM (0x00FF). Note that this will set the amount of space you have for a stack to 256 bytes (which should be more than plenty, but you should keep an Eye on it). Having done this, you should be able to run your program without the 62256 SRAM chip installed.

The reset vector is automatically established for you. Recompile your program, blast your EEPROM from your .S19 file using BOOTLOAD.EXE and run your program over the entire range of 0xC000 to 0xFFFF. Write the value you obtained on the lab check sheet.

( ) Step 2: Kontron CRC Verification

The Kontron does not support EEPROMs, but by picking a similar EPROM, the Kontron's internal CRC test can be run on an EEPROM. The steps below describe how to perform a CRC on a 27128A EPROM. Alternatively you may select a 27256 or 27C256 and run the test over half of the device (the half with your program in it, remember...?):

1. Turn on the Kontron. The display should read SELECT EPROM. Press ENTER to accept this option (other options would be PALs, etc).

2. The display should now read DIAL TYPE 2716. Type in 27128A on the keypad and then press ENTER to accept that device choice.

3. Press the red button marked CHK. The display wil read CH 00. There are several types of checks that can be done. We want Check #3 (CRC) so you must enter 03 and then press ENTER.

4. The display will now read CH SA 0000. The programmer is prompting you for the starting base address (SA). Since the start BASE address for a 27128A is 0000, accept the value in the display (0000) by pressing ENTER.

3. The display should now read CH EA 3FFF. The programmer is prompting you for The ending base address (EA). Since the ending BASE address is 3FFF for a 27128A, accept this value by pressing ENTER. The display will now read CH P>>. The programmer is now prompting you to place your device into the zero insertion force (ZIF) socket. Place your 27128A into the socket to the right of the small Illuminated LED. Make sure that pin 1 is toward the back of the unit. Pull the small green lever toward you to lock the 27128A in place.

5. Press the ENTER key. The display will count out the addresses as the CRC is being calculated. When the result is shown, fill in the number on your LABCHECK sheet and have it initialed by your instructor.

( ) Step 3: C vs Assembly Analysis

Fill in the following information and then get your program checked off by your instructor.

C Version:

Bytes of code: _____

Using your .LST file determine the highest address (not including the reset vector) of your code and then subtract 0xC000 from this.

Execution time: _____ seconds.

Using a stopwatch, time the execution of your CRC test program over the range 0xC000 to 0xFFFF.

Assembly Version:

Using Micro11 as the assembly language comparison, run the Options/Rom Test over the range 0xC000 to 0xFFFF. and time it's execution.

Execution time: _____ seconds.

Have a copy of your source file(s) available for check-off. All files must be completely documented at the time of checkoff.

YOU MUST HAND IN ALL OF YOUR .C SOURCE FILES FOR EVALUATION.
Make sure that the files have been printed with the Cprint utility.

LAB #1: LABCHECK SUMMARY

NAME: _____

PRE-LAB: Endian Test Program

Questions /2
Program execution for PC and HC11. Show your source files with your name to your instructor.

Date:_____ Instructor: _____ /3

LABCHECK #1: CRC Algorithm

Milestone checkoff. Have a copy of your source files on paper or PC screen.

Date:_____ Instructor: _____ /5

LABCHECK #2: User Interface

Milestone checkoff. Have a copy of your source files on paper or PC screen.

Date:_____ Instructor: _____ /5

LABCHECK #3: Combined Program (Final Program in RAM)

Milestone checkoff. Have a copy of your source files on paper or PC screen.

Date:_____ Instructor: _____ /5

LABCHECK #4: EPROM Version

Kontron CRC: _____ Instructor: _____
Questions (C vs Assembly) /6
Operation (You must hand in your source files)

Date:_____ Instructor: _____ /4

PROGRAM DOCUMENTATION /20

Note: The FINAL source files are marked for quality of documentation (commenting, use of identifiers, proper headers, proper equates, etc...). Your instructor may require that you turn in your structure charts or pseudocode for marking also.

TOTAL /50