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
0xC000	0xFC
0xC001	0x7F
0xC002	0xFE

Additive Result: 0x0279

Address	Data
0xC000	0xFC
0xC001	0x7F
0xC002	0xFE

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
0xC000	0x7F
0xC001	0xFC
0xC002	0xFE

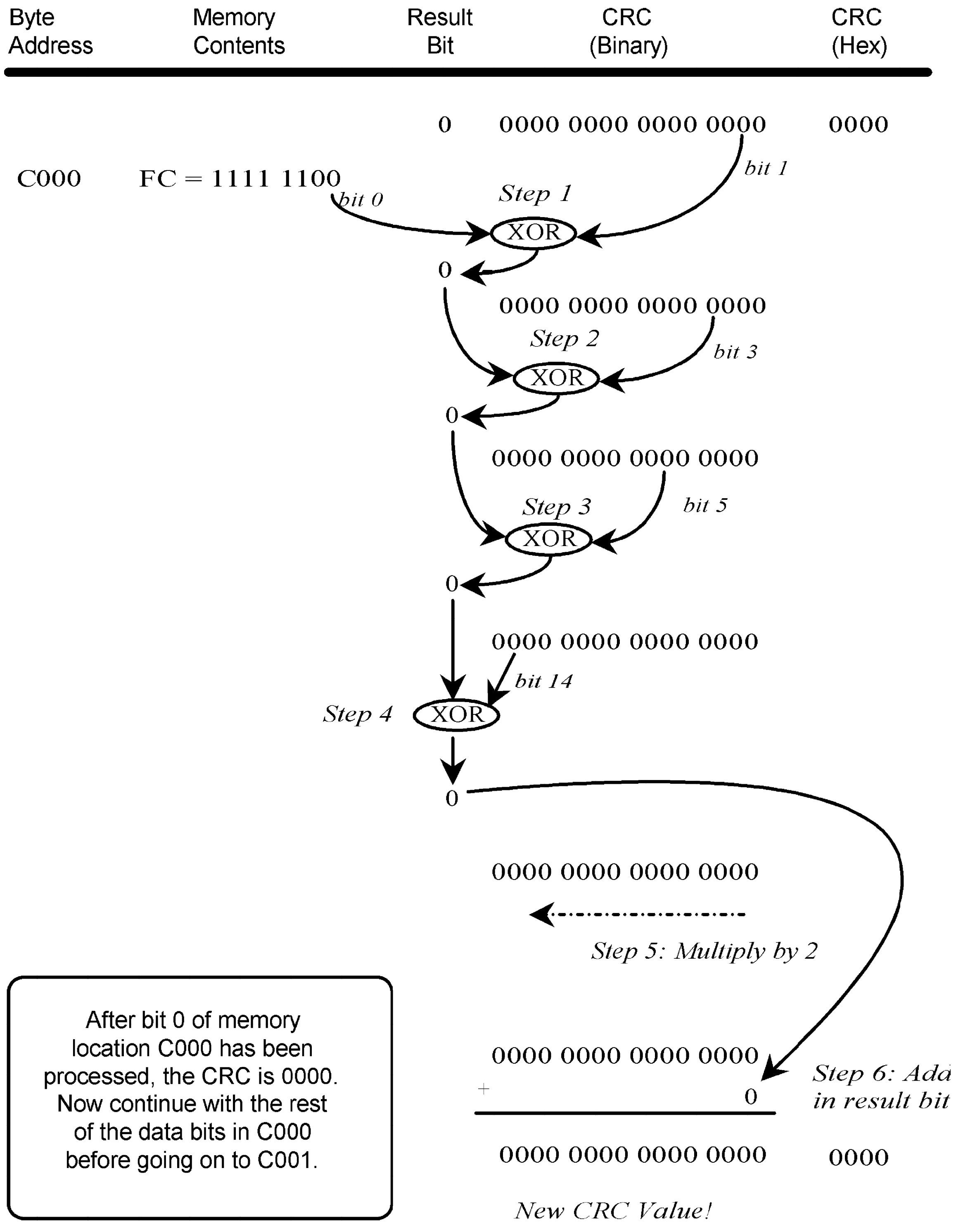
Additive Result: 0x0279

Address	Data
0xC000	0x7F
0xC001	0xFC
0xC002	0xFE

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!



After bit 0 of memory location C000 has been processed, the CRC is 0000. Now continue with the rest of the data bits in C000 before going on to C001.

( ) 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.