Scanning and Decoding Information From Magnetic Strips

Final Project Report
December 9, 1999
E157

Peter Scheidler and James Benham

Abstract:

Magnetic stripes are used in daily life to store digital information. Possibly the most common application is on cards, such as ATM, credit cards, and Harvey Mudd College meal cards. Normally, these cards are swiped through readers so that the information can be collected and sent to a central database for processing. This system could be particularly useful in the use of a drink-dispensing machine, where it may be necessary to keep track of the age and account of the user, for a variety of safeties, commercial, and legal reasons. This project documents efforts to create a system capable of reading the information encoded on the cards, decoding it, and sending it to a host computer. The magnetic information on the stripe is translated into digital pulses by hardware reverse-engineered from a Discover Card™ scanner, and the pulses are then sent to an HC11 for processing and transmission to the host computer.

Introduction

Using information stored on magnetic cards to identify people through machinery has become a part of daily life for most of America. From the point of curiosity alone, decoding and examining this information is an interesting task. When combined with the possibility for use in simple devices, reading and decoding the information is a very attractive problem.

The hardware to turn the magnetic information into on a train of pulses was found on a Discover Card™ scanner. Originally, the plan was to reverse engineer the card scanner, and build one of our own, for ease of integration. As time became short, and the possibility of getting the hardware with sufficient time became less likely, we decided to simply take the data directly off of the card scanner board, to minimize possible problem areas.

The data coming off of the card scanner board is encoded in a pulse train, where the value of a bit depends on the relative length of the pulses. This data is then read into the HC11, which measures the length of the pulses, and the output of the HC11 will be sent to a computer by an RS-232 connection. To successfully communicate with the computer, a Maxim 232 chip will need to change the voltage levels of the serial data coming out of the HC11. To facilitate debugging and hopefully minimize misconceptions about encoding, the HC11 will have the ability to choose between sending raw pulse widths to the host computer, and sending bytes of data made from the pulses. The choice between these two options will be made by simply pressing a key on the host computer, which will get sent via RS-232 to the HC11.

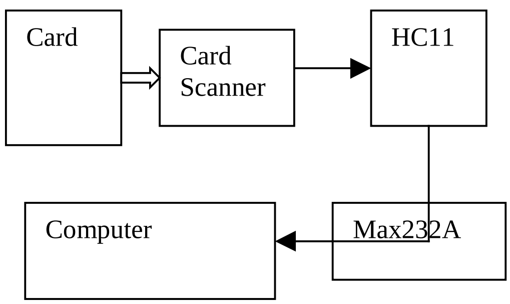


Figure 1. Flow of information from the card, to the scanner, to the HC11, and to the computer through the Max232 chip.

New Hardware

The information on the magnetic stripe is stored as magnetic blocks of alternating poles. When moved past a solenoid, or scanner, the moving magnetic field becomes a small current coming out of the solenoid. As the polarity of the magnet moving by the scanner changes, so does the polarity of the current. The current coming out of the card scanner is very small and noisy, so it must be amplified and cleaned using some fairly involved and mysterious analog circuitry. While schematics and a PCB file were generated to copy the analog circuitry, we decided to take the fast and easy route by using the filter on the board directly. This decision was made easier by the fear that we could not get the parts in time, and would need to remove some components from the board to measure them.

The information coming out of the card scanner board is simply a stream of pulses. It is the pulse width of these pulses that determines if the card holds a one or zero. Zero pulses are twice as long as one pulses, and one pulses always come in pairs, so a one and a zero take the same space, total.

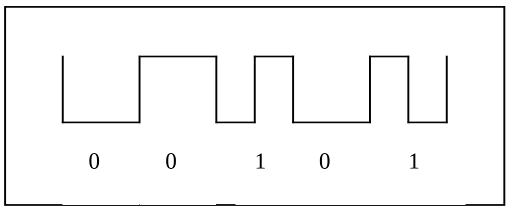
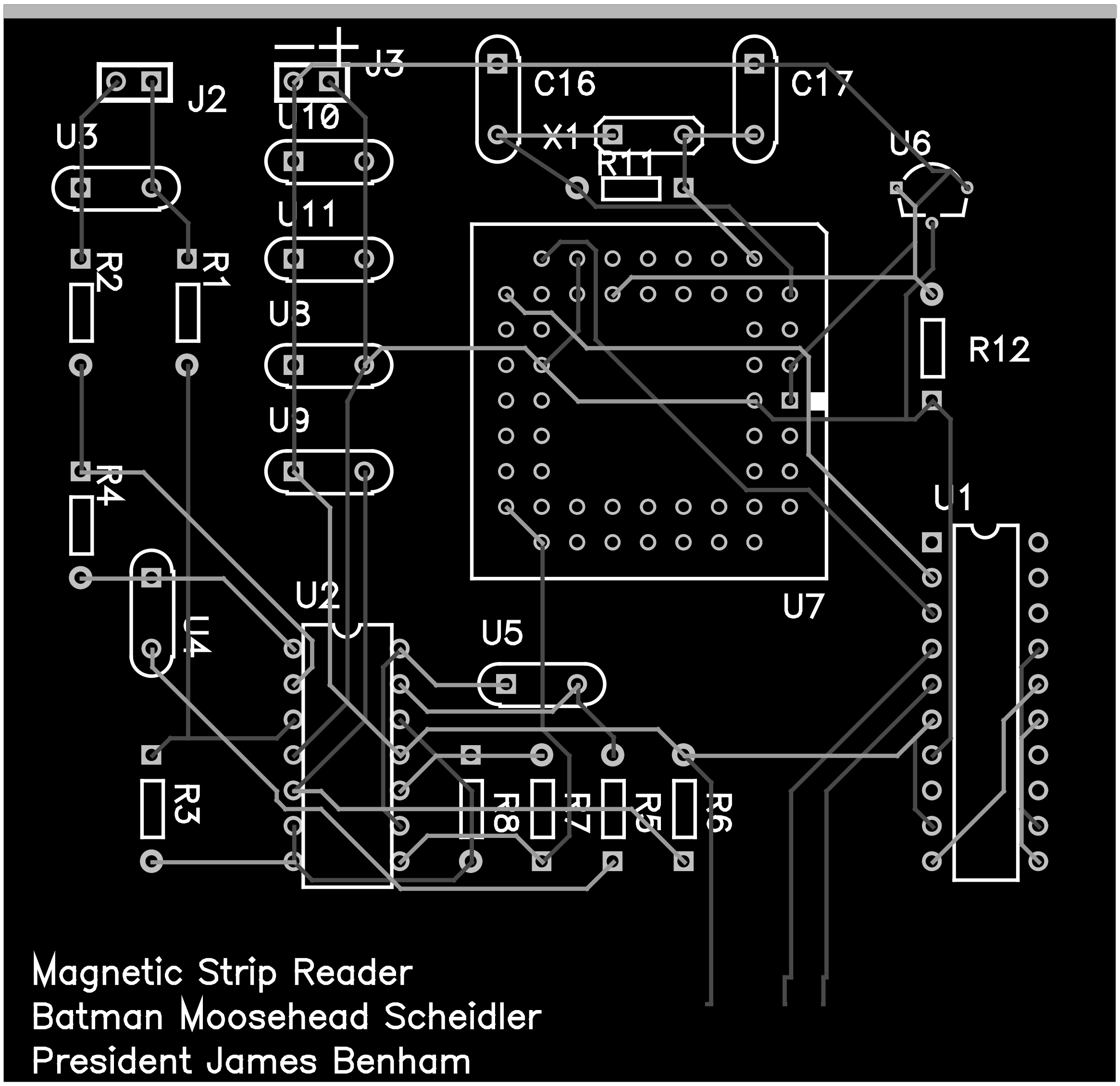


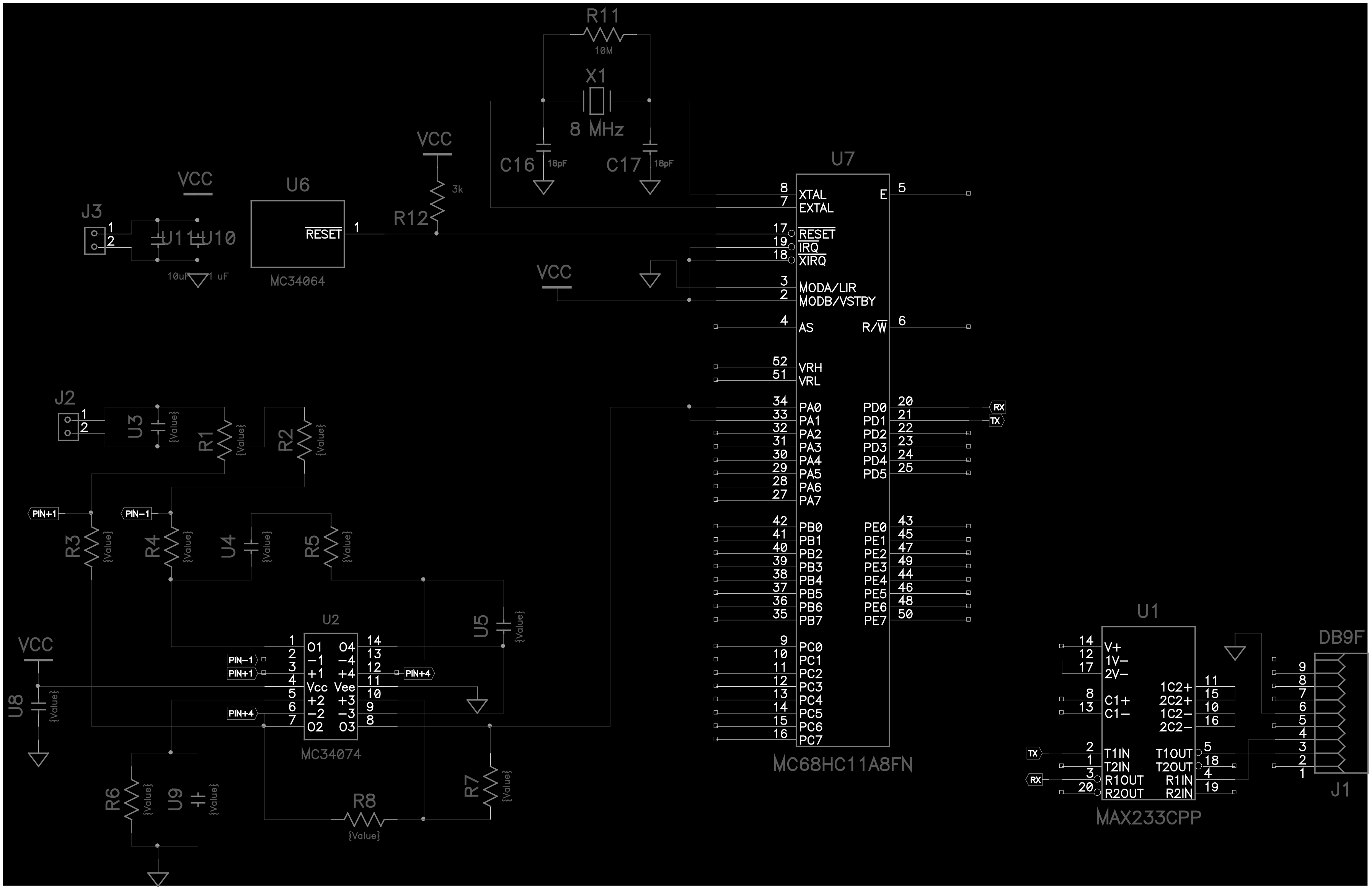
Figure 2. A stream of data from the card scanner board, and the corresponding binary values

The polarity of the pulse is irrelevant, so a decoder fed a given pulse and its inverse will translate them as the same data stream. The reason for this encoding scheme is that the card will be scanned in at an extremely variable speed, which can change in the middle of the transmission. This means that the encoding cannot have any absolute roots, but must be relative to itself. There are two standard methods for encoding the data, both of which are included in appendix A.

Schematics

The schematics and PCB image are listed on the following pages. The resistors and capacitors attached to the MC34074 were not measured, because in order to do so, they would have to have been removed from the board that we used, hence destroying the board.





U7
MC68HC11A8FN
XTAL
EXTAL
RESET
IRQ
XIRQ
MODA/LIR
MODB/VSTBY
AS
VRH
VRL
E
R/W
PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
PB0
PB1
PB2
PB3
PB4
PB5
PB6
PB7
PC0
PC1
PC2
PC3
PC4
PC5
PC6
PC7
PD0
PD1
PD2
PD3
PD4
PD5
PE0
PE1
PE2
PE3
PE4
PE5
PE6
PE7
U1
MAX233CPP
V+
1V-
2V-
C1+
C1-
T1IN
T2IN
R1OUT
R2OUT
1C2+
2C2+
1C2-
2C2-
T1OUT
T2OUT
R1IN
R2IN
J1
DB9F
TX
RX
R11
10M
X1
8 MHz
C16
18pF
C17
18pF
VCC
R12
3k
U6
MC34064
RESET
U10
U11
10uF
1 uF
J3
J2
U2
MC34074
R1
R2
R3
R4
R5
R6
R7
R8
U3
U4
U5
U8
U9
{Value}
PIN+1
PIN-1
PIN+4

Microcontroller Design

The HC11's three main functions were to measure the pulses coming in from the card scanner board, translate these pulses into useable data, and relay this information to the computer. The first and last objectives were fairly straightforwardly accomplished by using the HC11's hardware. Translating the pulses into data, however, was a more difficult task, given the limited and unintuitive programming language of the HC11. Because of this issue, the chip has the option of translating the data into bits, or simply passing the width of the pulses on to the computer for analysis there.

Measuring the pulses from the scanner board is accomplished by taking the pulse in at two input capture pins (IC1 and IC2). An input capture pin captures the time, as determined by the HC11s internal timer, of a specific event occurring on the pin. This event can be configured to be a rising edge, a falling edge, or either. In our setup, one pin, IC1, is set to capture the time of the rising edge of the incoming pulse, and IC2 is set to capture data at the falling edge. Using two pins rather then one, sets up an automatic buffer of information, allowing data to be processed with less chance of loosing any information. When a pin captures a time, it also generates an interrupt. In this interrupt, the program subtracts the time of the current pulse from the time of the previous pulse to get the pulse width, which is then processed and stored to be sent later.

Processing each pulse can happen in two modes. In the "Standard" mode, the HC11 examines the width of the pulse, and compares it to an already stored "split" value, which is halfway between the time of the last one pulse and zero pulse. When the chip is initialized, the one, zero, and split time are all set to zero. Since the first bits on a card are normally zero, this guarantees that the zero time will be loaded correctly. Since the time is considered an absolute number, which is never negative, the captured times are shifted to the right one bit as soon as they are read. This prevents any twos-compliment errors when the times are being subtracted or compared. The shift reduces the resolution of the timer to eight microseconds, which is very sufficient for measuring pulses that are milliseconds long.

Currently, the HC11 reads every pulse greater then the split value as a zero, and every pulse less then the split value is a one. These bits were shifted into an empty byte until eight bits were in, and then stored the byte and began a new one. Ideally, the program would toggle a "short" flag for every pulse less then the split value. That way, a one would only be assigned when two short pulses arrived when, adjacent to each other, and an error would be generated, resetting the short flag when there was a lone short pulse. This system was initially implemented, but there were numerous errors in the readings that caused us to doubt whether or not we had the correct decoding scheme. Because of this issue, ones were counted as simply one short pulse rather then two. It was because of this confusion over decoding the pulses that a second "Timer" processing mode was included. In this mode, the HC11 never translates the pulses into ones and zeros, but simply keeps the data as pulse width times, and sends these times out.

After the pulses were decoded the bytes were stored in a large memory block until the card scanning was done. When a pulse was read in, the HC11 cleared the "sent" flag, and set the "read" flag. When a real time interrupt was encountered, approximately every half-second, the read and sent flags were checked, and if both flags are clear, the HC11 sends out the data and sets the sent flag. If the read flag is set, the HC11 simply clears it.

The actual sending of the data is done through a structured protocol. Normally, the HC11 sends out a header byte ($AA), followed by the number of bytes of data and leftover bits to expect, not counting any bytes in the header or trailer, followed by a single checksum byte for the previous four bytes, to check for errors. The checksum is calculated by summing together the values of the previous bytes, and keeping only the low eight bits. Next, the HC11 goes to the data gathered from the recent card scan, and sends it out, starting at the first byte received, and sending high byte first, then low bye. Finally, the HC11 sends out the same style of one byte checksum, and ends the transmission with $00. If an error was detected, the HC11 simply sends out a different error message, starting with the header $02, followed by the error byte, and closing with the checksum. The actual sending of the information is done by loading checking the SCI condition register to make sure the port is not busy sending other information, and then loading the data to be sent into the register. There is also a similar system that allows text messages written into the HC11 memory to be sent to the host computer. These messages can be arbitrarily long, but they must be terminated with a special end character ($00).

A system peripheral to the main purpose of the project, but still very useful, is the ability to change the state of the chip. Since RS-232 communication is two way, the HC11 checks the SCI Condition Register and retrieves information sent to it when it is not otherwise occupied with incoming pulses or sending data. If the information is one of the specialized codes, “S”, “T”, or the unimplemented “F”, the HC11 changes its style register to 1, 2, or 0 respectively. Before handling the pulse, the HC11 checks this style register to see if it should process the incoming pulses for bytes or for time information.

Results

The final results can best be summarized by an example: “`B4217661232959528^BENNHAM/JAMES`” which represents data obtained from the magnetic strip on my ATM card. The following paragraphs will describe how the result was obtained and some of the problems with the project.

The final version of the code that we demonstrated did not work as well as it should have. The code attempted to convert the pulse widths into binary. However, the code could not do this because it did not correctly measure the pulse widths. The most likely cause of this problem is that our interrupt handlers took too much time. However, using a program other then our main program, we obtained accurate pulse width values. This program is a stripped down version that was used during the debugging phase of our project. The code for this test program is given in Appendix C. The test program simply measured the pulse width and stored the value in memory.

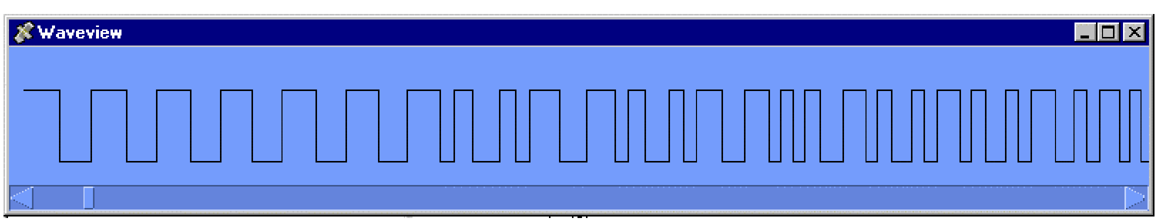
The pulses generated by the final code would frequently contain very long pulses at somewhat regular intervals. Since the test program generated results that did not display these long pulses on the same hardware, there is something wrong with the final code.

The most difficult part of the project has the lack of a known output. We did not know what our output should have been until a very late stage. It would have been sensible to spend time trying to obtain output that we could verify as the project progressed.

We originally proposed to send only the pulse widths to a host computer. The final version of our software did employ serial communication successfully. While it did not produce the desired signal, the test program we developed did.

To verify that the pulses we recorded were correct, we manually decoded it. The pulse widths were captured from the memory of the test program and moved to a file on the PC. Since the hexadecimal values were difficult to visualize, we created a program that allowed us to view the signal. The code for this program is given in Appendix D. A screenshot of the program is given below. The first image shows the wave display incorporating a large time period. One can see the broad patterns of ones and zeros, the ones being the darker regions.

The next view shows the waveform expanded over a longer time scale. The shorter pulses represent a one, while the longer pulses represent zeros, as described above. We decoded this waveform by hand to verify that we 1) understood the way the data was encoded and 2) were reading the pulse widths correctly.



Interpreting the signal yielded the following results:

```
000000000000000010100010100011001010101001011000101111010101101000110100
100010101001011100100010010110011001010100100110010101000100101000110101
```

```
111100100011101001001110110001011100001110110111111001010101010000111011
011101001011001110000001000000100000010000001000000100000010000001000000
100000010000001000000100000010000001011111000000100110010000000100111011
000101000010010001011010100100110011101011110101100010100001000000100000
000100000001000000100000010000001000000100001000000100111010111101010010
010000010000001000000100000010000001000000100111110011111000000000
```

As pretty as the above data is, one must know how to interpret the data. The general format of the card consists of some leading zeros, a start signal (sentinel), the data, an end signal(s) and some trailing zeros.

000…0	SS	*Data*	ES	ES	000…0

The format for the data and the values for the start signal (SS) and end signal (ES) depend on the encoding selected for the data. There are two common encoding formats. The first is BCD, which consists of four bits for data and one for parity per number. The other common format is the ALPHA format, which uses six bits for the data and one for parity. The ALPHA format can encode both letters and numbers, although it is limited to uppercase letters. See Appendix A, Data Encoding for more information.

Decoding the bits above, which were produced by my ATM card, leads to the following result:

SSB4217661232959528^BENNHAM/JAMES_____________^03071015977100______00774
000000***ESES***0000000

where *SS* stands for the start signal (1010001 for ALPHA) and *ES* stands for the end signal (1111100). The underscore character was used in place of a space to make the spaces more visible. This data shows that the code on the HC11 successfully recorded the pulses and that we can interpret the binary data on the card.

All of the data on the card was encoded with the ALPHA format. The first field B4217661232959528 matches the number on the front of the card, with the exception of the leading B, which seems to be a common feature of ATM/Credit cards. The next field is very obviously my name. It is also a fixed width field, but since my name is not very long, the remainder of the field is filled with spaces. The third filed starts out with the expiration date, July 3rd. The rest of the information in the field does not seem to match anything printed on the card, but it may contain information about the bank, allowed ATM networks, etc.

Since the data we received matches the data on the card, we have successfully produced a hardware and software solution that can read the pulse-widths from a magnetic strip.

References

[1] Card-o-Matic by Count Zero.
[2] MC34074 data sheet from Motorola (http://scgproducts.motorola.com/ProdSum.asp?base=MC34074)

Parts List

Part	**Source**	**Vendor Part #**	**Price**
Maxim 232A serial transceiver	Scrap (available from Maxim)	MAX232ACPE-ND	$4.88
Discover Card ™ Scanning Terminal	Andy Cosand (Current availability unknown)		
25 pin female D-sub connector	Stock Room		

Appendix A: Data Encoding

Data can be encoded in any format on the magnetic strip, but there are some commonly used formats, BCD and ALPHA. Both of these formats have some similarities. The both have some data bits and a parity bit for every symbol. Both encode the least significant bit first and both set the parity bit so that the total sum is odd. Tables describing the format are given below.

BCD

Data				Parity	Value
B1	B2	B3	B4		
0	0	0	0	1	0
1	0	0	0	0	1
0	1	0	0	0	2
1	1	0	0	1	3
0	0	1	0	0	4
1	0	1	0	1	5
0	1	1	0	1	6
1	1	1	0	0	7
0	0	0	1	0	8
1	0	0	1	1	9
0	1	0	1	1	Control (:)
1	1	0	1	0	Start Signal(;)
0	0	1	1	1	Control (<)
1	0	1	1	0	Field Separator (=)
0	1	1	1	0	Control (>)
1	1	1	1	1	End Signal (?)

Alpha

Data						Parity	Value
B1	B2	B3	B4	B5	B6		
0	0	0	0	0	0	1	Special (space)
1	0	0	0	0	0	0	Special (!)
0	1	0	0	0	0	0	Special (“)
1	1	0	0	0	0	1	Special (#)
0	0	1	0	0	0	0	Special ($)
1	0	1	0	0	0	1	Start Signal (%)
0	1	1	0	0	0	1	Special (&)
1	1	1	0	0	0	0	Special (`)
0	0	0	1	0	0	0	Special (( )
1	0	0	1	0	0	1	Special ( ))
0	1	0	1	0	0	1	Special (*)
1	1	0	1	0	0	0	Special (+)
0	0	1	1	0	0	1	Special (‘)
1	0	1	1	0	0	0	Special (-)
0	1	1	1	0	0	0	Special (.)
1	1	1	1	0	0	1	Special (/)
0	0	0	0	1	0	0	0
1	0	0	0	1	0	1	1
0	1	0	0	1	0	1	2
1	1	0	0	1	0	0	3

0	0	1	0	1	0	1	4
1	0	1	0	1	0	0	5
0	1	1	0	1	0	0	6
1	1	1	0	1	0	1	7
0	0	0	1	1	0	1	8
1	0	0	1	1	0	0	9
0	1	0	1	1	0	0	Special (:)
1	1	0	1	1	0	1	Special (;)
0	0	1	1	1	0	0	Special (<)
1	0	1	1	1	0	1	Special (=)
0	1	1	1	1	0	1	Special (>)
1	1	1	1	1	0	0	End Signal (?)
0	0	0	0	0	1	0	Special (@)
1	0	0	0	0	1	1	A
0	1	0	0	0	1	1	B
1	1	0	0	0	1	0	C
0	0	1	0	0	1	1	D
1	0	1	0	0	1	0	E
0	1	1	0	0	1	0	F
1	1	1	0	0	1	1	G
0	0	0	1	0	1	1	H
1	0	0	1	0	1	0	I
0	1	0	1	0	1	0	J
1	1	0	1	0	1	1	K
0	0	1	1	0	1	0	L
1	0	1	1	0	1	1	M
0	1	1	1	0	1	1	N
1	1	1	1	0	1	0	O
0	0	0	0	1	1	1	P
1	0	0	0	1	1	0	Q
0	1	0	0	1	1	0	R
1	1	0	0	1	1	1	S
0	0	1	0	1	1	0	T
1	0	1	0	1	1	1	U
0	1	1	0	1	1	1	Z
1	1	1	0	1	1	0	W
0	0	0	1	1	1	0	X
1	0	0	1	1	1	1	Y
0	1	0	1	1	1	1	Z
1	1	0	1	1	1	0	Special ([ )
0	0	1	1	1	1	1	Special (\ )
1	0	1	1	1	1	0	Special ( ])
0	1	1	1	1	1	0	Field Separator (^)
1	1	1	1	1	1	1	Special (_)

Appendix B: Cardscan HC11 Code

```
*
 Card Scan HC11
*
* Read raw bits from a magnetic card strip and sends the raw bits
* up to a host PC via a serial connection
*
* James J. Benham
* Peter Schiedler
*
* Changelog
*  1999-11-28 23:45
*     Initial Creation
*  1999-12-*
*     Working on it
*  1999-12-07 16:45
*     Successful full test :)

*********************************
* Constants
*********************************
* Input Capture Constants
TIC1    EQU     $1010          Rise Time
TIC2    EQU     $1012          Fall Time
TMSK1   EQU     $1022
TFLG1   EQU     $1023
TCTL2   EQU     $1021

* RTI Constants
PACTL   EQU     $1026
TMSK2   EQU     $1024
TFLG2   EQU     $1025

* SCI
SCDR    EQU     $102F          Serial Data Register
BAUD    EQU     $102B
DDRD    EQU     $1009
SPCR    EQU     $1028
SCCR1   EQU     $102C
SCCR2   EQU     $102D
SCSR2   EQU     $102E

* Characters
EOT     EQU     $00
CR      EQU     $0D
LF      EQU     $0A

*******************************************************************
* Program
*******************************************************************

********************************
* Initialization
********************************
        ORG     $D000
```

```
*        * Set up interrupts *
         ldaa    #$7E    TIC1 - Rise Time
         staa    $00E8
         ldd     #REDGE
         std     $00E9

         ldaa    #$7E     TIC2 - Fall Time
         staa    $00E5
         ldd     #FEDGE
         std     $00E6

         ldaa    #$7E      RTI
         staa    $00EB
         ldd     #TMR_ITR
         std     $00EC

*        * Variable initialization for main loop
         BSR     INIT

*        * SCI Init *
         ldaa    #$30
         staa    BAUD
         ldaa    #$00
         staa    SCCR1
         ldaa    #$0C
         staa    SCCR2

*        * Output a startup message
         ldy     #MSG           Love Me
         JSR     SENDMSG

*        * RTI Init:
         ldaa    #$03
         staa    PACTL
         ldaa    #$40
         staa    TFLG2
         ldaa    #$43
         staa    TMSK2

*        * TIC Init:
         ldaa    #$18
         staa    TCTL2
         ldaa    #$06
         staa    TFLG1
         staa    TMSK1

         CLI                    Enable All Interrupts

* Begin program loop. Program is interrupt driven
* This loops polls SCI, checking for input.
* Pressing a key changes the mode by changing STYLE reg
LOOP     ldaa    SCSR2
         BITA    #$20
         BEQ     LOOP
         ldaa    SCDR           Get data from SCI

         ldab    #CR            Send a line feed
```

```
        JSR     D_OUT
        ldab    #LF
        JSR     D_OUT

        cmpa    #'F'            'F' = used in the Future
        BNE     LOOP1
        clra
        staa    STYLE
        BRA     LOOP

LOOP1   cmpa    #'S'            'S' = Send bytes of data
        BNE     LOOP2
        ldy     #SMSG
        JSR     SENDMSG
        ldaa    #$01
        staa    STYLE
        BRA     LOOP

LOOP2   cmpa    #'T'            'T' = Send Times
        BNE     LOOP
        ldy     #TMSG
        JSR     SENDMSG
        ldaa    #$02
        staa    STYLE
        bra     LOOP


********************************
* Init Subroutine
*   Modifies A, D, and X registers
********************************
* Initialize time counters
INIT    LDD     #$0000          Zero the:
        STD     RISET             rise time
        STD     FALLT             fall time
        STD     SPLIT             split value
        std     ONET            one time
        std     ZEROT           zero time

        ldaa    #$01            Set the style to send bytes, not times
        staa    STYLE

* Initialize data array

        LDAA    #$00
        STAA    BITC
        staa    BYTECH
        STAA    BYTEC
        STAA    INITC
* Set the first entry to 0
        LDX     #RAW
        STAA    0,X
        stx     ADDR

* Initialize error and data status
        STAA    ERROR
        LDAA    #$02
```

```
        STAA    DATAS
        RTS                     Return

********************************
* RiseEdge Interrupt Handler
********************************
REDGE   LDAA   INITC
        CMPA   #$20           Check to see if we're at the first few
pulses
        BLE    RE_END2        Ignore the first 64 pulses
        LDAA    ERROR
        BNE     RE_END
        ldaa    DATAS
        ORAA    #$01            Set the reading bit
        ANDA    #$fd            Clear the sent bit
        STAA    DATAS
        LDD     TIC1            Store the event time
        LSRD                    No neg nums, 1MHZ is good enough
        STD     RISET
        SUBD    FALLT           Compute the pulse width
        STD     PULSE
        BSR     HDLPULSE        Deal with the pulse
RE_END  ldaa    #$04
        staa    TFLG1
        RTI                     Return
RE_END2 ldaa    #$04
        staa    TFLG1
        inc    INITC
        RTI                     Return

********************************
* FallEdge Interrupt Handler
********************************
FEDGE   LDAA   INITC
        CMPA   #$20           Check to see if we're at the first few
pulses
        BLE    FE_END2        Ignore the first 64 pulses
        LDAA    ERROR           Check for errors and compare
        BNE     FE_END
        LDAA    DATAS           Set the status,
        ORAA    #$01            read = 1,
        ANDA    #$fd            data sent = 0
        STAA    DATAS
        LDD     TIC2            Read in the time of the event
        LSRD                    No neg numbers, 1MHz is good
        STD     FALLT
        SUBD    RISET           Compute the pulse width

        STD     PULSE
        BSR     HDLPULSE        Deal with the pulse
FE_END  ldaa    #$02
        staa    TFLG1
        RTI                     Return
FE_END2 ldaa    #$02
        staa    TFLG1
        INC    INITC
        RTI                     Return
```

```
********************************
* Timer Interrupt Handler
*   Used to detect when a card has been swiped. It does this by setting
*   A read flag to zero. This flag will be set by the edge interrupts,
*   but if it detects that the read flag is still zero, then we are
*   no longer reading a card. If there is data to send, it will do so
********************************
TMR_ITR LDAA    DATAS
        ANDA    #$03            See if read = 0 and dataSent = 0
        BNE     TMR_END         Go to the end if != 0
        JSR     SEND            Send the data
        JSR     INIT            Re-initialize the data
TMR_END LDAA    DATAS
        ANDA    #$fe            Clear the read bit
        staa    DATAS
        ldaa    #$40
        staa    TFLG2
        RTI                     Return

********************************
* Handle Pulse Subroutine
*  Responsible for taking a pulse width, converting it to a one or
zero,
*  updating the split point between a one and a zero, and for storing
*  the bit.
********************************
HDLPULSE ldaa   STYLE
        cmpa    #$02          Check the style
        BEQ     TOUT          Jump to time stle
        LDD     PULSE           Load the pulse width
        SUBD    SPLIT           Compare to split point. NOTE:
*                               beware 2's complimant and counter
*                               roll-over. We don't deal with it.
        BLE     S_ELSE          if(pulseWidth > split) {
        LDAA    #$00               bit = 0;
        STAA    BIT
        LDD     PULSE
        STD     ZEROT              zerotime = pulseWidth;
        BSR     STORE              store();
        BRA     S_END           } else {
S_ELSE  LDD     PULSE
        STD     ONET               onetime = pulseWidth;
        LDAA    #$01                bit = 1;
        STAA    BIT
        BSR     STORE
        BRA     S_END
S_END   LDD     ONET
        ADDD    ZEROT           Add onetime and zerotime
        LSRD                    Divide by two
        STD     SPLIT           split = (onetime + zerotime) / 2;

*       Do some adjustments for the first couple of pulses
*       which must be zeros. Use these to set up our split
*       point correctly. See if we are still reading the first
*       two pulses
```

```
        ldaa    BYTECH
        BNE     SPLIT1
        LDAA     BYTEC
        LSLA
        ADDA     BITC
        SUBA     #$03
        BGE      SPLTSET          if((rawBitCount+(rawByteCount<<1))<3)
SPLIT1  LDD      PULSE
        LSRD                      set onetime = pulse/2;
        STD      ONET              1 is 1/2 width of 0
        LDD      #$0000
        STD      SPLIT            set split to 0 (every pulse is a 0)
SPLTSET RTS                      Return

*       Stores the pulse in two bytes in the table and then returns
*       from the routine.

TOUT    ldd     PULSE
        ldy     ADDR
        std     0,Y            Write the pulse
        iny
        iny
        sty     ADDR           Advance to the next table entry
        inc     BYTEC          Update the byte count, watching for
        BNE     TOUT1          low byte overflowing.
        inc     BYTECH
TOUT1   inc     BYTEC
        BNE     TOUT2
        inc     BYTECH
TOUT2   RTS

********************************
* Store data Subroutine
*  Stores each bit. Each bit is added into the LSB of a byte until the
*  byte is full. Then a new byte is addressed.
********************************
STORE   ldx      ADDR          Load the table address
        LDAA     BITC
        SUBA     #$08            Check to see if bitcount >= 8
        BLT      STR_CNT
        INX                      Move to the next table entry
        stx      ADDR
        LDAA     #$00            Set the data area to zero to
        STAA     BITC            Set the number of bits to 0
        STAA     0,X              avoid random values
        INC      BYTEC           Increment the number of bytes
        BNE     STR_CNT
        inc     BYTECH
STR_CNT LDAA     0,X            Read in the current byte
        LSLA                     Move the previous values up one
        ORAA     BIT             Add the last bit
        staa     0,X
        INC      BITC            Update the count
        RTS                      return

********************************
* Send Subroutine
```

```
*
*
*  Sends the raw data over the serial port. The format is as follows
*  A Header Byte: 1=raw data ok 2=error encountered (HEAD)
*  If an error was encountered, no further bytes will be sent
*  Otherwise, TWO bytes will be sent containing the number of WHOLE
*   bytes read by the HC11 (BYTECOUNTH + BYTECOUNT).
*  After that, a byte will be sent containing the number of remaining
*   bits (BITCOUNT).
*  These bytes will be followed by a checksum:
*    CHKSUM = HEAD + BYTECOUNTH + BYTECOUNT + BITCOUNT
*    This value could wrap, but it should still provide an ok check
*  Next, BYTECOUNT + 1 bytes will be sent containing the raw data.
*    the last byte in this sequence will contain BITCOUNT bits of
*    real data in the LSBs of the byte.
*  These bytes will be followed by another checksum:
*    CHKSUM = RAW0 + RAW1 + ... + RAW(BYTECOUNT) + RAW(BYTECOUNT + 1)
*  Finally, a NULL bit will be sent. This byte will be all zeros
********************************
SEND    LDAA    ERROR
        BEQ     BITEME
        JMP     SEND_ERR        Decide which HEAD to send
* Send beginning info
BITEME  LDAA    #$00            Set the CHKSUM to 0
        LDAB    #$AA            Load header
        ABA                     Increment the checksum
        JSR     SEND_BYTE       Send header
* Byte count
        ldab    BYTECH
        aba
        JSR     SEND_BYTE
        LDAB    BYTEC           Load the byte count
        ABA                     Increment the checksum
        JSR     SEND_BYTE       Send the byte count
* Bit count
        LDAB    BITC            Load the bit count
        ABA                     Increment the checksum
        JSR     SEND_BYTE       Send the bit count
* Checksum
        STAA    CSUM            Save the checksum
        LDAB    CSUM            Load into B
        JSR     SEND_BYTE       Send the checksum
        ldab    #CR           Move to a new line
        JSR     D_OUT
        ldab    #LF
        JSR     D_OUT
        LDAA    #$00            Reset the checksum
* Send the raw data
        LDAB    BYTEC
        LDY     #RAW            Set Y to point to the top of the table
        STAB    SENDC           Set the count for the number of bytes
SENDTOP BNE     SEND1         End if we’ve sent everything (Count=0)
        tst     BYTECH
        BEQ     SENTRAW
        dec     BYTECH
SEND1   LDAB    0,Y             Read in the next byte
        ABA                     Update the checksum
```

```
        JSR     SEND_BYTE       Send the byte in B
        INY                     Move to the next entry
        DEC     SENDC           Dec num to send and SET COND CODES
        BRA     SENDTOP
        BRA     SND_END
SENTRAW LDAB    0,Y             Read in the next byte w/ left-over bits
        ABA                     Update the checksum
        JSR     SEND_BYTE       Send it
* Send the checksum
        ldab    #CR            Send newline
        JSR     D_OUT
        ldab    #LF
        JSR     D_OUT
        STAA    CSUM            Store the checksum (transter to B)
        LDAB    CSUM            Load checksum into B
        JSR     SEND_BYTE       Send B
        LDAA    #$00
* Send the terminating NULL
        LDAB    #$00
        JSR     SEND_BYTE
        ldab    #CR            Advance to the next line
        JSR     D_OUT
        ldab    #LF
        JSR     D_OUT
        BRA     SND_END
SEND_ERR LDAB   #$02            The header command
        JSR     SEND_BYTE
        LDAB    ERROR
        JSR     SEND_BYTE       Send the error that occured
        ldab    #CR            Advance to the next line
        JSR     D_OUT
        ldab    #LF
        JSR     D_OUT
SND_END LDAA    DATAS           Update the data status to inidcate
        ORAA    #$02              that the data has been sent
        STAA    DATAS
        RTS                     Return

********************************
* Send Byte Subroutine
*
* Input: B
*
* Unaffected: A, X, Y
*
* This subroutine takes a byte and sends it over the serial line as
* two bytes representing the value of the input byte in hex. For
example,
* given the value of 0x2F (stored in B), the subroutine will send the
* values 0x32 ('2' in ASCII) and 0x46 ('F') over the serial line.
********************************
SEND_BYTE STAB  SNDBUF          Store the value so we can modify it
        LSRB                    Get the top four bits
        LSRB
        LSRB
        LSRB
        CMPB    #$0A            See if it's a number or letter
```

```
        BGE     ALPHA
        ADDB    #$30            It’s a number, add ’0’
        JSR     D_OUT
        BRA     CNV_END         Skip to end
ALPHA   ADDB    #$37            It’s a letter, add ’A’ - 10(10)
        JSR     D_OUT           Use subrountine to write to SCI
CNV_END LDAB    SNDBUF          Reload the value for the low-order bits
        ANDB    #$0F            Mask off the top bits
* Repeat above code to select number or letter
        CMPB    #$0A            See if it’s a number or letter
        BGE     ALPHAB
        ADDB    #$30            It’s a number, add ’0’
        JSR     D_OUT           Write to serial
        BRA     CNV_ENDB        Skip to end
ALPHAB  ADDB    #$37            It’s a letter, add ’A’ - 10(10)
        JSR     D_OUT           Write to serial
CNV_ENDB RTS                    Return


*****************************************
* Serail Ouput Routine
*
* Input: B
*
* Outputs the value given in accumulator B to the serial port.
* It will loop if the serial port is not yet ready to send.
*****************************************
D_OUT   pshb
D_OUT1  ldab    SCSR2
        BITB    #$80
        BEQ     D_OUT1          Loop until we can send data
        pulb
        stab    SCDR
        RTS


*************************************
* Message rounine
*
* Input: Y
*
* Dirty: B
*
* Takes the value referenced by Y and prints out the string until
* a NULL (0x00) is encountered.
*************************************
SENDMSG ldab    0,Y
        BEQ     MSG1
        JSR     D_OUT
        INY
        BRA     SENDMSG
MSG1    RTS

*******************************************************************
*********************************
* Data Area
*********************************
```

```
MSG      FCB     CR,LF
         FCC     'Love Me'
         FCB     CR,LF,EOT

TMSG     FCC     'Timer Enabled'
         FCB     CR,LF,EOT
SMSG     FCC     'S-Mode Enabled'
         FCB     CR,LF,EOT

STYLE    RMB     1       0 = Full on try, 1 = Short Pulse is 1, 2 =
Times Only
RISET    RMB     2       the rise time
FALLT    RMB     2       the fall time
PULSE    RMB     2       the width of the pulse
SPLIT    RMB     2       the time between a 0 and a 1 pulse

ZEROT    RMB     2       the last pulse width for a 0 bit
ONET     RMB     2       the last pulse width for a 1 bit
ERROR    RMB     1       error status
* 2 = buffer overflow
* 4 = unexpected long pulse
BIT      RMB     1       hold a single bit for processing

DATAS    RMB     1       1 = reading data 2 = data sent 3 = last pulse
*                        was short
BITC     RMB     1       Count the number of raw bits 0..7
BYTECH   RMB     1       High byte (Used for sending time)
BYTEC    RMB     1       The number of whole raw-data bytes
INITC    RMB     1       Count number of bytes to ignore in the
beginning

CSUM     RMB     1       A checksum used in transmission
SENDC    RMB     1       The number of bytes left for transmission
SNDBUF   RMB     1       A temporary storage area for transmission
ADDR     RMB     2       Address of current byte (X)
RAW      RMB     64      area to hold the raw data


                 END
```

Appendix C: Test HC11 Code

This program only reads and then stores the pulse widths.

```
* Card Scan HC11

*

* Read raw bits from a magnetic card strip and sends the raw bits

* up to a host PC via a serial connection

*

* James J. Benham

*

* Changelog

*  1999-11-28 23:45

*     Initial Creation

*********************************
* Constants
*********************************
TIC1    EQU     $1010           Rise Time
TIC2    EQU     $1012           Fall Time
TMSK1   EQU     $1022
TFLG1   EQU     $1023

TCTL2   EQU     $1021
PACTL   EQU     $1026
TMSK2   EQU     $1024
TFLG2   EQU     $1025

SCDR    EQU     $102F           Serial Data Register
BAUD    EQU     $102B
DDRD    EQU     $1009
SPCR    EQU     $1028
SCCR1   EQU     $102C
SCCR2   EQU     $102D
SCSR2   EQU     $102E
EOT     EQU     $00
*******************************************************************
* Program
*******************************************************************

********************************
* Initialization
********************************
            ORG     $D000
* Set up interrupts
            ldaa    #$7E    TIC1 - Rise Time
            staa    $00E8
```

```
            ldd     #REDGE
            std     $00E9
            ldaa    #$7E      TIC2 - Fall Time
            staa    $00E5
            ldd     #FEDGE
            std     $00E6
            ldaa    #$7E       RTI
            staa    $00EB
            ldd     #TMR_ITR
            std     $00EC

* Initialize Registers:
* RTI Init:
      ldaa    #$03
      staa    PACTL
      ldaa    #$40
      staa    TFLG2
      ldaa    #$43
      staa    TMSK2

* TIC Init:
      ldaa    #$18
      staa    TCTL2
      ldaa    #$06
      staa    TFLG1
      staa    TMSK1

      ldd     #00
      std     RTIME
      std     FTIME
      staa    RED
      inca
      staa    READ

      ldx     #STORAGE
      stx     ADDR

      ldad  #$00
      STAD  PULSEC

      CLI

* Begin program loop. Program is interrupt driven
LOOP    BRA     LOOP

********************************
* RiseEdge Interrupt Handler
********************************
REDGE   ldd     TIC1
      std     RTIME
      subd    FTIME
      ldx     ADDR
      std     0,X
      inx
      inx
      stx     ADDR
      LDAD  PULSEC
```

```
      ADDD  #$01
      STAD  PULSEC
      ldaa    #$04
      staa    TFLG1
      ldaa    #$01
      staa    RED
      staa    READ
      RTI                    Return

********************************
* FallEdge Interrupt Handler
********************************
FEDGE   ldd     TIC2
      std     FTIME
      subd    RTIME
      ldx     ADDR
      std     0,X
      inx
      inx
      STX     ADDR
      LDAD  PULSEC
      ADDD  #$01
      STAD  PULSEC
FE_END  ldaa    #$02
      staa    TFLG1
      ldaa    #$01
      staa    RED
      staa    READ
      RTI                    Return

********************************
* Timer Interrupt Handler
*   Used to detect when a card has been swiped. It does this by setting
*   A read flag to zero. This flag will be set by the edge interrupts,
*   but if it detects that the read flag is still zero, then we are
*   no longer reading a card. If there is data to send, it will do so
********************************
TMR_ITR LDAA    READ
      BNE     TMR_END        Go to the end if != 0
      ldaa    RED
      BEQ     TMR_END
      nop
TMR_END clr     READ
      ldaa    #$40
      staa    TFLG2
      RTI                    Return

READ    RMB 1
RED     RMB 1
PULSEC      RMB 2
RTIME   RMB 2
FTIME   RMB 2
ADDR    RMB 2

      ORG   $D0D0
STORAGE RMB 1
```

```
END
```

Appendix D: Signal Viewing Source code

The following source code is for the Waveview program used to display the signals read by the HC11. It was written in Java. The program wv reads input in one format, while wv2 reads a slightly different format. Both programs accept one command line argument, which is a number between 0 and 1 that controls the scaling of the pulse into pixels on the screen. The input for the program is read from standard input.

```
// Author:  James Benham
// File:    wv.java
//
// Reads pulse widths and displays them graphically
//
// The input file format consists of a number of pulses in hex
// as the first line. Each subsequent line starts with a
// memory address which is ignored. The following characters
// are the hexadecimal values of each byte. A pulse is recorded
// as a two-byte value. After the bytes displayed in hex, an
// ASCII representation is given, which is also ignored.
//
// ex:
// 038A
// D0D0 94 45 02 E7 15 06 04 41 12 BD 05 72 0D 2C 09 0D  E     A   r ,
// D0E0 0C FC 08 B0 0B 95 09 46 0C A9 09 2E 0B D2 09 16        F   .
// D0F0 0C 1E 09 B9 0C 4A 0A 3A 0D 10 09 E2 0C 5E 09 9E      J :     ^
// D100 0B 07 09 AD 0B 48 09 50 0B 42 09 6C 0B 21 09 9B      H P B l !
// D110 0B 38 09 36 0A F6 09 4B 0A 65 08 D8 0A 21 08 C8  8 6   K e   !
// D120 0A 24 08 D6 0A 22 08 84 09 B1 08 39 08 D7 08 54  $   "     9   T

import java.io.*;

class wv {
public static void main(String argv[]) {

  // Parse the command line arguments
  double scale = 0.01;
  int start = 0;

  if(argv.length >= 1) {
    scale = Double.parseDouble(argv[0]);
  }

  System.out.println("Start Position is pulse " + start);
  System.out.println("Scaling pulse by factor of " + scale);


  // Read in stuff from standard in
  int numPulses = 0;
  int pls[] = null;
  try {
    numPulses = Character.digit((char)System.in.read(), 16);
    numPulses = 16*numPulses + Character.digit((char)System.in.read(), 16);
    numPulses = 16*numPulses + Character.digit((char)System.in.read(), 16);
    numPulses = 16*numPulses + Character.digit((char)System.in.read(), 16);

    pls = new int[numPulses];

    System.out.println("NumPulses: " + numPulses);

    int nextChar = System.in.read();

    while(nextChar != '\n' && nextChar != -1)
      nextChar = System.in.read();

    System.in.skip(4);
    nextChar = System.in.read();

    int linepos = 0;
    int value = 0;
    int count = 0;
```

```
    while(nextChar != -1 && count < numPulses) {
      if(linepos == 8) {
       linepos = 0;
       while(nextChar != ’\n’ && nextChar != -1)
         nextChar = System.in.read();
       System.in.skip(5);
      }

      value = 0;
      value = Character.digit((char)System.in.read(), 16);
      value = 16*value + Character.digit((char)System.in.read(), 16);
      System.in.skip(1);
      value = 16*value + Character.digit((char)System.in.read(), 16);
      value = 16*value + Character.digit((char)System.in.read(), 16);
      nextChar = System.in.read();
      pls[count] = value;

      linepos++;
      count++;
    }

  } catch (IOException e) {
    System.err.println("Read failuare.");
    System.exit(0);
  }

  Waveview wv = new Waveview(start, scale, pls);
  wv.setSize(800, 115);
  wv.show();
}
}

// Author:  James Benham
// File:    wv2.java
//
// Reads pulse widths and displays them graphically
//
// The input file consists of a header containing the number of
// bytes = 2* number of pulses. Values in the file are ASCII
// characters representing hexadecimal values. The first two
// characters in the header are ignored, the next four give the
// number of bytes. In the following line, each pulse is stored as
// two bytes represented in hex.
//
// Example: numPulses = 0x0824
// AA082400D6
// 6539016D0E150113052F029188B402D804E9FFFC0293046302C5044F02A30485017902E2015
// 803FD02BE06F60277063A0004026D00D901E703770232033102200589020E0490033B05CA04
// 6202E001F902B001F8048601E8025E0253004901F98575024D03330481055C04BC0303023B0
// 573024C031004E0030E025A05AB01FB0058026602E00251051A022402CB021502C4022C02E5
// 027C0047021A0619028

import java.io.*;

class wv2 {
public static void main(String argv[]) {

  // Parse the command line arguments
  double scale = 0.01;
  int start = 0;

  if(argv.length >= 1) {
    scale = Double.parseDouble(argv[0]);
  }

  System.out.println("Start Position is pulse " + start);
  System.out.println("Scaling pulse by factor of " + scale);


  // Read in stuff from standard in
  int numPulses = 0;
```

```
  int pls[] = null;
  try {
    // Skip command header
    System.in.skip(2);

    numPulses = Character.digit((char)System.in.read(), 16);
    numPulses = 16*numPulses + Character.digit((char)System.in.read(), 16);
    numPulses = 16*numPulses + Character.digit((char)System.in.read(), 16);
    numPulses = 16*numPulses + Character.digit((char)System.in.read(), 16);

    numPulses = numPulses/2;

    pls = new int[numPulses];

    System.out.println("NumPulses: " + numPulses);

    int nextChar = System.in.read();

    while(nextChar != '\n' && nextChar != -1)
      nextChar = System.in.read();


    nextChar = System.in.read();

    int value = 0;
    int count = 0;

    while(nextChar != -1 && count < numPulses) {
      value = 0;
      value = Character.digit((char) nextChar, 16);
      value = 16*value + Character.digit((char)System.in.read(), 16);
      value = 16*value + Character.digit((char)System.in.read(), 16);
      value = 16*value + Character.digit((char)System.in.read(), 16);
      nextChar = System.in.read();
      pls[count] = value;

      //      System.out.println(count + ": " + value);

      count++;
    }

  } catch (IOException e) {
    System.err.println("Read failuare.");
    System.exit(0);
  }

  Waveview wv = new Waveview(start, scale, pls);
  wv.setSize(800, 115);
  wv.show();
}
}

// Author:  James Benham
// File:    Waveview.java
//
// Displays pulse widths graphically and lets a user scroll through
// the display.

import java.awt.*;
import java.awt.event.*;

public class Waveview extends Frame
  implements WindowListener, AdjustmentListener {

  int pulses[];
  int start;
  double scale;

  Scrollbar scroll;

  int high = 30;
  int low = 80;
```

```
  Waveview(int start, double scale, int pulses[]) {
    super("Waveview");
    this.start = start;
    this.scale = scale;
    this.pulses = pulses;
    addWindowListener(this);

    setLayout(new BorderLayout());

    scroll = new Scrollbar(Scrollbar.HORIZONTAL, 0, 10, 0, pulses.length);
    add(scroll, BorderLayout.SOUTH);
    scroll.addAdjustmentListener(this);
  }

  public void paint(Graphics g) {
    g.setColor(Color.black);

    int prevx = 10;
    int prevy = high;
    int x = 10;
    int y = prevy;

    for(int i=start; i < pulses.length; i++) {
      x += (int) (scale * pulses[i]);
      y = ( (i%2) == 0) ? high : low;
      g.drawLine(prevx, prevy, prevx, y);
      g.drawLine(prevx, y, x, y);
      prevx = x;
      prevy = y;
    }

  }

  public void adjustmentValueChanged(AdjustmentEvent e) {
    start = e.getValue();
    System.out.println("Start: " + start);
    repaint();
  }

  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }

  public void windowOpened(WindowEvent e) {}
  public void windowClosed(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}

}
```