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 safety, 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.

![Diagram](Attachment)

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](image)

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.
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 than 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 as 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 (SAA), 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 byte. 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^BENNIAM/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 than 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:

0000000000000010100010100011001010100101010010011101001001100101000110100100101001000101001000101000110101
00010101010010111001001011001100101001100101001010000100110101010000110101
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.

<table>
<thead>
<tr>
<th>000...0</th>
<th>SS</th>
<th>Data</th>
<th>ES</th>
<th>ES</th>
<th>000...0</th>
</tr>
</thead>
</table>

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
000000ESES000000

where SS stands for the start signal (1010001 for ALPHA) and ES stands for the end signal (111100). 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


### Parts List

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

<table>
<thead>
<tr>
<th>Data</th>
<th>Parity</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>B2</td>
<td>B3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Alpha**

<table>
<thead>
<tr>
<th>Data</th>
<th>Parity</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>B2</td>
<td>B3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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
TCTRL2 EQU $1021

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

* SCI
SCDR EQU $102F Serial Data Register
BAUD EQU $102B
DDR6 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 *
llda $7E     TIC1 - Rise Time
staa $00E8
ldd #REDGE
std $00E9

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

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

* Variable initialization for main loop
BSR INIT

* SCI Init *
llda $30
staa BAUD
ldaa $500
staa SCCR1
ldaa $50C
staa SCCR2

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

* RTI Init:
llda $503
staa PACTL
ldaa $540
staa TFLG2
ldaa $543
staa TMSK2

* TIC Init:
llda $18
staa TCL2
ldaa $506
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
ldaaSCDR 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
clr
staa STYLE
BRA LOOP

LOOP1 cmpa #‘S’  ‘S’ = Send bytes of data
BNE LOOP2
ldy #SMSSG
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
<table>
<thead>
<tr>
<th>STAA</th>
<th>DATAS</th>
<th>Return</th>
</tr>
</thead>
</table>

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

*RiseEdge Interrupt Handler*

<table>
<thead>
<tr>
<th>REDGE</th>
<th>LDAA</th>
<th>INITC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPA</td>
<td>#$20</td>
<td></td>
</tr>
</tbody>
</table>

Check to see if we’re at the first few pulses

<table>
<thead>
<tr>
<th>BLE</th>
<th>RE_END2</th>
<th>Ignore the first 64 pulses</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAA</td>
<td>ERROR</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>RE_END</td>
<td></td>
</tr>
<tr>
<td>ldaa</td>
<td>DATAS</td>
<td></td>
</tr>
<tr>
<td>ORAA</td>
<td>#$01</td>
<td>Set the reading bit</td>
</tr>
<tr>
<td>ANDA</td>
<td>#$fd</td>
<td>Clear the sent bit</td>
</tr>
<tr>
<td>STAA</td>
<td>DATAS</td>
<td></td>
</tr>
<tr>
<td>LDD</td>
<td>TIC1</td>
<td>Store the event time</td>
</tr>
<tr>
<td>LSRD</td>
<td></td>
<td>No neg nums, 1MHz is good enough</td>
</tr>
<tr>
<td>STD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBD</td>
<td>FALLT</td>
<td>Compute the pulse width</td>
</tr>
<tr>
<td>STD</td>
<td>PULSE</td>
<td></td>
</tr>
<tr>
<td>BSR</td>
<td>HDLPULSE</td>
<td>Deal with the pulse</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RE_END</th>
<th>ldaa</th>
<th>#$04</th>
</tr>
</thead>
<tbody>
<tr>
<td>staa</td>
<td>TFLG1</td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RE_END2</th>
<th>ldaa</th>
<th>#$04</th>
</tr>
</thead>
<tbody>
<tr>
<td>staa</td>
<td>TFLG1</td>
<td></td>
</tr>
<tr>
<td>inc</td>
<td>INITC</td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

*FallEdge Interrupt Handler*

<table>
<thead>
<tr>
<th>FEDGE</th>
<th>LDAA</th>
<th>INITC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPA</td>
<td>#$20</td>
<td></td>
</tr>
</tbody>
</table>

Check to see if we’re at the first few pulses

<table>
<thead>
<tr>
<th>BLE</th>
<th>FE_END2</th>
<th>Ignore the first 64 pulses</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAA</td>
<td>ERROR</td>
<td>Check for errors and compare</td>
</tr>
<tr>
<td>BNE</td>
<td>FE_END</td>
<td></td>
</tr>
<tr>
<td>LDAA</td>
<td>DATAS</td>
<td>Set the status,</td>
</tr>
<tr>
<td>ORAA</td>
<td>#$01</td>
<td>read = 1,</td>
</tr>
<tr>
<td>ANDA</td>
<td>#$fd</td>
<td>data sent = 0</td>
</tr>
<tr>
<td>STAA</td>
<td>DATAS</td>
<td></td>
</tr>
<tr>
<td>LDD</td>
<td>TIC2</td>
<td>Read in the time of the event</td>
</tr>
<tr>
<td>LSRD</td>
<td></td>
<td>No neg numbers, 1MHz is good</td>
</tr>
<tr>
<td>STD</td>
<td>FALLT</td>
<td></td>
</tr>
<tr>
<td>SUBD</td>
<td>RISET</td>
<td>Compute the pulse width</td>
</tr>
</tbody>
</table>

| STD | PULSE   |                             |
| BSR | HDLPULSE| Deal with the pulse         |

<table>
<thead>
<tr>
<th>FE_END</th>
<th>ldaa</th>
<th>#$02</th>
</tr>
</thead>
<tbody>
<tr>
<td>staa</td>
<td>TFLG1</td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FE_END2</th>
<th>ldaa</th>
<th>#$02</th>
</tr>
</thead>
<tbody>
<tr>
<td>staa</td>
<td>TFLG1</td>
<td></td>
</tr>
<tr>
<td>INC</td>
<td>INITC</td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**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_INTR 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.
---

```
HDL_PULSE 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 compliment 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   }
  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 SPLITSET 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)
SPLITSET 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
styx ADDR Advance to the next table entry
inc BYTEC Update the byte count, watching for
BNE TOUT1 low byte overflowing.
inc BYTEC
TOUT1 inc BYTEC
BNE TOUT2
inc BYTEC
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 Move to the next table entry
INX
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 BYTEC
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 (BYTECOUNT + 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 + BYTECOUNT + 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

* 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
ABA
JSR SEND_BYTE

* Bit count
LDAB BITC
ABA
JSR SEND_BYTE

* 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 (transfer 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 occurred
ldab #CR       Advance to the next line
JSR D_OUT
ldab #LF
JSR D_OUT

SND_END LDAA DATAS   Update the data status to indicate
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 subroutine to write to SCI
CNV_END LDAB  SDBUF   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

*******************************************************************************
* Serial Output 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 routine
*******************************************************************************
* 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

SMTP  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

ZERO  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
INIC  RMB  1  Count number of bytes to ignore in the
beginning

CSUM  RMB  1  A checksum used in transmission
SEND  RMB  1  The number of bytes left for transmission
SNDBUF  RMB  1  A temporary storage area for transmission
ADD  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
 ldax  #$7E  TIC1 - Rise Time
 stax  $00E8
* Initialize Registers:
* RTI Init:
  ldaa #$03
  staa PACTL
  ldaa #$40
  staa TFLG2
  ldaa #$43
  staa TMSK2

* TIC Init:
  ldaa #$18
  staa TCBL2
  ldaa #$06
  staa TFLG1
  staa TMSK1

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

  ldx #$STORAGE
  stx ADDR

  ldad #$00
  STAD PULSE

CLI

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

*****************************************************************
* RiseEdge Interrupt Handler
*****************************************************************
REDEGE  ldd TIC1
       std RTIME
       subd FTIME
       ldx ADDR
       std 0,X
       inx
       inx
       stx ADDR
       LDAD PULSE
ADD D #$01
STAD PULSE C
lda #4
staa TFLG1
lda #4
staa RED
staa READ
RTI    Return

*****************************
* FallEdge Interrupt Handler
*****************************
FEDGE    ldd      TIC2
          std      FTIME
          subd     RTIME
          ldX      ADD R
          std      0, X
          inX
          inX
          STX      ADD R
LDAD PULSE C
ADD D #$01
STAD PULSE C
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
PULSE C    RMB 2
RTIME      RMB 2
FTIME      RMB 2
ADDR       RMB 2

      ORG $D00D0
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.

```java
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 failure.");
    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
// AA08240D6
// 6539016DE150113052F02918B8402D804E9FFFC0293046302C5044F02A30485017902E015
// 803FD02BE06F0277063A0040426D00D901E703770232033102200589020E0490033B05CA04
// 6202E001F902BB001F8048601EB025E025304901F98575024D03330481055C04BC0303023B0
// 573024C03104E003B025A05AB01FB0058026602E00251051A022402CB021502C4022C02E5
// 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 failure.");
    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) {}