Designing and Programming a Complete HiSeC™-based RKE System

Fairchild
Application Note 985

INTRODUCTION

This application note explains how to design, program and implement a complete high security RKE (remote keyless entry) system based on the Fairchild Semiconductor NM95HS01/02 HiSeC Rolling Code Generator and the MM57HS HiSeC Rolling Code Decoder.

It is broken down into several sections which provide detailed information for developing constituent subsystems such as the transmitter, receiver and decoder. The last section provides information for programming the system microcontroller and EEPROM, and integrating system components. This application note also provides all the necessary circuit schematics, PC board layouts, and code listings to assist the designer in developing a complete RKE system.

Transmitter Design

This section briefly discusses a few important design points to consider when implementing an RF transmitter based on the NM95HS01 HiSec Generator.

BASIC HARDWARE

Figure 1 shows a simple RF transmitter based on the NM95HS01 HiSeC Rolling Code Generator chip. This version of the HiSeC device is clocked with an RC network; the NM95HS02 version is clocked with a crystal oscillator. The two key switch inputs are connected directly to grounded, single pole switches. These device inputs have internal pull-up resistors to reduce the external component count.

An indicator LED can be used in the transmitter design to provide a visual cue the device is transmitting. The TX output pin controls the base of an NPN transistor, which forms a tuned RF amplifier based on a SAW filter. The RFEN output is used as the ground for the tuned RF circuit.

BIT CODING TRANSMISSION FORMATS

The HiSeC device has eleven bit coding formats available for transmitting key data—seven for RF applications and four for IR applications. Complete waveform details for all bit coding formats are given in the data sheet for the NM95HS01/02 HiSeC Rolling Code Generator. A few details of their particular usage in transmitter applications is discussed here.

RF bit coding format 0 has a narrow bandwidth, and may require a clock recovery circuit to ease signal decoding; however, once the clock signal has been recovered, data decoding can be achieved by exclusive-ORing the data and clock streams. This bit format—like RF bit format 2—has a DC level that is independent of the data transmitted.

RF bit coding format 4, and all the IR formats, provide a constant transmit energy per message, assuming signal transmission during logic HIGH. The user should note that the high and low bit times for these formats are different, which is an important consideration when designing the transmission preamble and NRZ sync timing.

Windows® is a registered trademark of Microsoft Corporation.

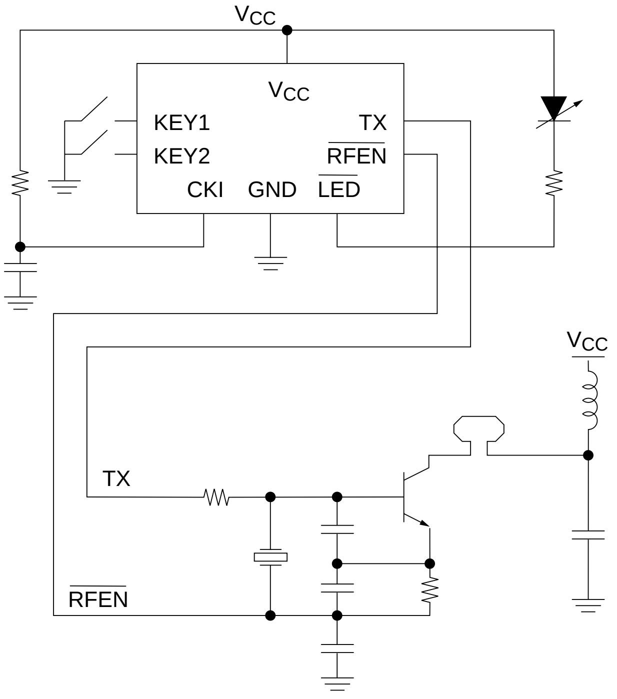


FIGURE 1. RF Transmitter using NM95HS01 HiSeC Rolling Code Generator

RF coding formats 1, 3, 5 and 7 are PWM type formats which are relatively easy to decode. RF format 7 has a low duty cycle, which allows the RF transmitter to achieve a higher peak power. The IR formats are modulated versions of RF coding format 4, and are more suitable for IR transmitter applications. The duty cycle and number of pulses in these modes allow the user to refine IR circuit power usage.

BIT CODING TIMING BLOCK

The prescalers in the HiSeC generator timing block are configurable, and allow the user to set the time base for all bit coding formats. Either prescaler output can be used as the time base for any bit coding format if the external clock has a sufficiently low frequency, and can be scaled properly. However, the output of the first prescaler is generally intended for use with an IR transmitter, while the second prescaler divides the signal further for use with an RF transmitter.

A complete explanation of the prescalers, and examples for choosing scaling factors, can be found in the data sheet for the NM95HS01/02 HiSeC Generator.

DATA FIELD USAGE IN TRANSMISSION FRAMES

To implement efficient transmitter and decoder designs, it is important to understand the purpose and use of the individual fields in a transmission frame. These are discussed briefly below.

The preamble field, if enabled, is transmitted once with the first frame to provide a known, recognizable signal to “wake up” the microcontroller in the receiver-decoder circuit. It has a fixed format of two bit times at logic HIGH, one bit time at logic LOW, and eight zeroes encoded in the user-selected bit format. If desired, this field can be separated completely from a frame by eight LOW bit times. This is achieved by enabling the sync field in NRZ mode with all zeroes (byte 0h).

The sync field, if enabled, is sent in every frame. It provides a known bit timing reference pattern for the rest of the frame. The eight bits sent in the sync frame are fully programmable, and can be encoded in either a standard bit coding format or NRZ bit coding. If desired, part of the sync field could be used to send extra key identifier data to replace or extend the fixed identifier field.

The key ID field, if enabled, is sent in every frame. Its length can be set to 0, 20, or 24 bits. The field is sent in the selected bit coding format. Its contents are fully programmable, and it provides a unique identifier for each key to facilitate decoding, and to identify a particular key in applications where one decoder is used with several keys. It can also be used as the basis for a fixed code generator application.

The data field is sent in every frame. This 4-bit field is transmitted using the selected bit coding format. It indicates which keys have been pressed, whether a sync frame is being sent, and whether the battery level in the transmitter is low.

The dynamic code field is sent in every frame. Its length can be set to 24 or 36 bits. Increasing the field length provides additional security. The field is sent in the selected bit coding format, and provides a secure rolling code which changes with each new transmission. In a sync frame, this field is replaced by a 40 bit initialization field.

The parity field, if enabled, is an 8-bit field sent with each frame using the selected bit coding format. It is a bytewise exclusive OR-ing of all bytes in the frame from the sync field to the dynamic code field, and serves as a check on data integrity.

The stop bit is present in all frames. It is used to delimit the end of a frame for bit coding formats that require a definite end. All IR formats need this delimiter to distinguish between a “1” and a “0” in the penultimate bit of a frame. For bit coding modes where delimiting is not required, the stop bit is read as a “1”.

TRANSMISSION OUTPUT POLARITY

The TxPol bit in EEPROM determines the quiescent state of the TX output line. If this bit is set to “0”, the quiescent output level will be a logic LOW. If it is set to “1”, the quiescent output level will be a logic HIGH, and the bit coding formats will be inverted. TxPol = 0 should be used to drive the base of an NPN transistor, as shown in the example transmitter circuit in Figure 1. TxPol = 1 should be used to drive an IR diode in series with a current limiting resistor.

TRANSMITTER TIME-OUT

Transmitters based on a HiSeC device can be protected against premature battery drain caused by a key held low for a period of time causing continuous transmission of data frames. If the timeout bit in EEPROM is enabled, the device will enter HALT mode after approximately 80 seconds.

TRANSMISSION INDICATION

Either the LED or RFEN outputs of the NM95HS01/02 can be used to indicate device transmission. The LED output is active during a pause, whereas the RFEN output is active during frame transmission.

Receiver and Decoder Design

This section describes how to design a HiSeC receiver-decoder system using an 8-bit microcontroller and an RF receiver. We begin by considering general IR and RF receiver designs, then use this discussion as a starting point for a more specific HiSeC RKE receiver-decoder system. The receiver-decoder system discussed here is designed with a Fairchild Semiconductor COP888CG microcontroller and a MM57HS HiSeC Rolling Code Decoder.

System design will be considered in four major sections: general RF/IR receiver design, decoder logic design, decoder software design, and a detailed explanation of how to implement the HiSeC rolling code algorithm with the sample software code provided.

GENERAL RF/IR RECEIVER DESIGN

This section discusses several basic RF and IR design issues, and example receiver system designs are described in some detail. Since most RKE systems utilize RF transmission, most of the design focus here is on RF systems.

• RF Receivers

Some of the design parameters engineers encounter with RF receiver circuit design are discussed below. RF receivers for RKE applications are typically high frequency circuits that operate over the range of 240 MHz to 434 MHz. These circuits generally have low power consumption requirements (<2 mA to 4 mA), and must operate over a wide temperature range (-40°C to +85°C).

The modulation scheme most commonly used for RKE RF applications is amplitude modulation, usually in the form of pulse width modulation (PWM). This is because transmitters that provide bursts of RF pulses, instead of transmitting RF energy continuously, can be designed to radiate more power while still meeting FCC requirements. RKE receivers also tend to be high-Q circuits which helps improve bit error rates.

The RF receiver example presented in Figure 2 is a super-regenerative type receiver. This means the oscillator turns itself on and off at a predetermined rate, allowing power consumption to be reduced. This super-regenerative receiver uses a grounded-base RF amplifier (Q1) to increase sensitivity and reduce detector radiation. Q2 functions as the detector, which is basically a UHF oscillator that turns itself on and off at a 200 kHz rate. The detected signal is conditioned by a dual operational amplifier, one half of which is used to amplify the low-level RF signal, the other half is used as a comparator to drive the decoder IC.

In this circuit,a 4 µVpeak RF input signal yields approximately 0.5 mV of signal at the output of A2. With these levels, the peak signal-to-RMS noise figure at the output of A1 is approximately 12 dB, which allows satisfactory decoding.

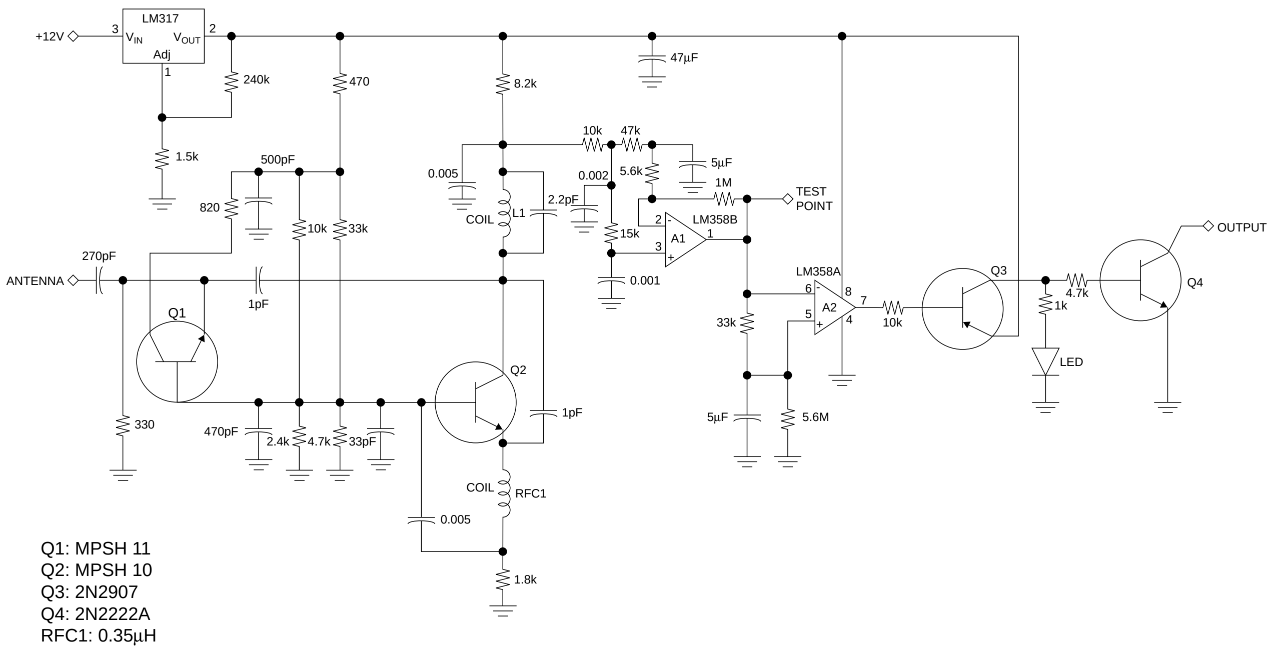


FIGURE 2. Super-Regenerative RF Receiver

The center frequency (f_C) of the circuit may be varied by changing C8, with little effect on receiver sensitivity. The output of A2 produces appropriate logic level pulses for the COP888CG microcontroller.

A voltage regulator is required in this circuit since the detector circuit has no power supply rejection capability, and small irregularities in the power supply voltage, due to ripple or load variations, could cause a loss of data.

A properly operating receiver should have very narrow pulses (6V peak, 200 kHz–400 kHz rate) across R_G. Receiver operation may be checked using pin 1 of A1 as a test point. Here, with no input signal, there should be approximately 0.2V peak-to-peak of noise. This test point may be used to tune receivers and transmitters together for maximum response.

• IR Receivers

So far, the discussion has focused on RF receivers, IR applications have similar requirements.

There are basically six stages involved in implementing an IR receiver system. The first stage amplifies the incoming signal, followed by a limiter stage that limits the signal. The third stage provides a bandpass filter, and the fourth stage is used to demodulate the signal. Next, an integrator circuit is used to sharpen the demodulated pulse, and finally, the last stage uses a comparator to ensure an appropriate logic level output.

Figure 3 shows the six basic design stages for an IR receiver stage.

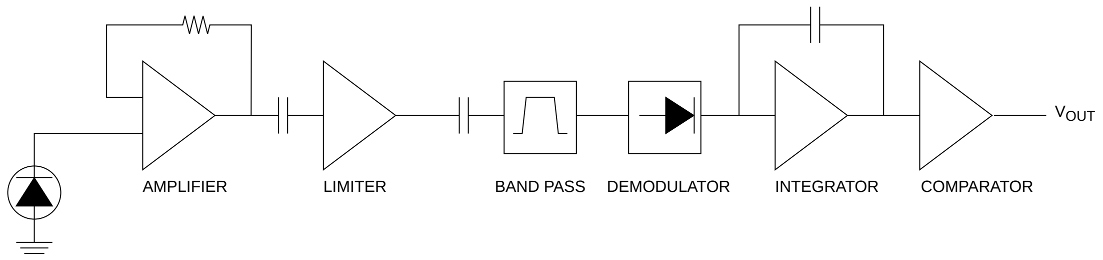


FIGURE 3. Block Diagram of IR Receiver

DECODER SYNCHRONIZATION

There are two primary methods for ensuring synchronization between a HiSeC generator and decoder—using a sync frame, and performing a forward calculation of the rolling code. The first method establishesinitial synchronization, or resynchronization, between the devices. The second method maintains synchronization between the devices.

In forward calculation, the decoder “forward-calculates” a predetermined number of codes ahead (known as the code window) searching for a code match. This procedure allows a decoder that was previously synchronized with a generator to miss one or more transmitted codes, and still find a match. This could occur, for example, if the transmitter was out of range, or if the key was activated in the user’s pocket.

The other method of synchronization is required to initialize a new decoder, to resynchronize a transmitter and decoder after a battery change, to add or replace a new key, or if the code window is exceeded. This method requires the use of a sync frame.

A sync frame contains enough information for the decoder to learn a key completely. The decoder software can decide whether or not it should accept sync frames, and under what conditions. In a sync frame, the data field is set to all zeros, implying that no key has been pressed, a condition that is impossible when sending a normal data frame. This allows the software to detect a sync frame easily.

DECODER LOGIC DESIGN

The focus here is on designing a decoder system using a COP888CG microcontroller to implement the functions of signal capture, frame decoding, and program execution.

When a HiSeC transmission frame is received, the decoder system can process the frame in two different ways.

The first method is to route the output of the receiver circuit directly to one of the microcontroller's input ports, where software polls the input port continuously in search of a valid frame. However, there are two problems with this approach. The first problem is with power consumption. If the microcontroller polls the input port continuously, it must remain powered up, and most CMOS microcontrollers consume between 10 mA to 25 mA in normal operating mode. In automotive applications where continous battery drain is a problem, this is unacceptable. The second problem is that the decoder is constantly trying to decode unwanted transmissions, which reduces the time available for other tasks performed by the microcontroller. In applications where power consumption is not critical, and processor multitasking is not used, such as garage door openers, this method is a

valid design option.

The second method takes advantage of the capture ability of the COP888CG, and requires much less operating current. This microcontroller operates at clock speeds up to 10 MHz, and contains six timing capture registers that can measure external frequencies or time events precisely. These attributes make the COP888CG microcontroller a good choice for this RKE design.

The COP888CG capture registers can be used to capture an incoming data frame, and allows the microcontroller to be powered down to wait for a frame to be sent. The COP888CG uses approximately 3.5 mA in idle mode with a 10 MHz clock, which is far less of a current drain. Capturing also allows the microcontroller to ignore a frame that is received at a frequency other than that expected.

This capturing capability also allows a potential secondary input from the ignition system. This secondary input could be used as a safety measure to allow the decoder system to recognize and accept a sync frame only when a key is physically placedin the ignition system. This design option would allow the user the possibility of resynchronizing the key and decoder if synchronization is lost.

Another consideration of good decoder design is key information storage. If multiple keys are used, the decoder must have a method of storing information to recognize these multiple keys. Information storage is also required for resynchronization in cases where a data frame transmission has been missed.

A serial EEPROM, interfaced to the microcontroller, solves these problems. Each key's seed code can be stored in this additional memory, so that each time a key is activated, the next several valid codes are calculated and stored. The EEPROM can also be used to provide additional depth to the "code window" by establishing a forward code look-up table (with any number of future valid codes) for the keys in case of accidental key activation, or missed code transmission.

The last design consideration is the number and type of receiver-decoder outputs necessary to utilize the transmitted information. A complete receiver-decoder system implemented with the COP888CG can easily be designed to accomodate several outputs that each sink 20 mA loads. These outputs might be used to transfer data from the receiver-decoder to other systems of the automobile. These outputs could be RS-232C, SPI, Microwire™ , Class B, or SAE J1850 protocol outputs. Please refer to the COP8™ selection guide for details of our configurable controller methodology.

Figure 4 shows a complete receiver-decoder example circuit which uses a super-regenerative RF receiver module, the input capture capability of the COP888CG, a serial EEPROM to store multiple key information and count-ahead table, two 20 mA outputs, and a Microwire port for multi-system communication.

DECODER SOFTWARE DESIGN

The decoder system software can be broken down into four major routines: Main Control, PWM Decode, Rolling Code Generation, and Output Control.

The Main Control routine begins by initializing the microcontroller variables, registers, and port states. It then places the microcontroller into idle mode. When the microcontroller awakens from idle mode by sensing a rising edge on one of the port L pins, the Main Control routine reads the captured data, then passes control to the PWM Decode routine.

The PWM Decode routine times out the data frame and restores the original transmission reception capability. It then returns control back to the Main Control routine, which stores the unencoded frame in RAM. Program control is then passed to the Rolling Code Generation routine which compares the received data frame to a table of frames previously calculated for each key. If a match is found, it writes a logic "1" into a "match register". If a match is not found, the routine writes a logic "0" into the match register. Control is again handed back to the Main Control routine, which checks the match register and passes the appropriate commands to the output register. At this point, program control is passed to the Output Control routine which executes the appropriate output response. Finally, the Main Control routine powers the microcontroller down and waits for the next data frame input capture.

THE HISEC ROLLING CODE ALGORITHM

The section above described a general approach to writing software code for an RKE decoder system. For more detailed information about the HiSeC rolling code generation algorithm, and how to implement the functions in software, please contact your local Fairchild Sales office.

Programming Information

This section briefly discusses how to program and verify the non-volatile EEPROM configuration memory on-board the NM95HS01/02 rolling code generator. It also discusses how to program and verify a NM93C86A Serial EEPROM which is used in the receiver-decoder system. A programmer, built around the Fairchild Semiconductor COP888CG microcontroller, has been developed to program both the NM95HS01/02 HiSeC Generator and the NM93C86A. Schematics, PC board layouts, and software listings for the programmer are provided, along with additional contact information.

NM95HS01/02 PROGRAMMING OVERVIEW

The NM95HS01/02 HiSeC Rolling Code Generator has four pins that are used for programming. These are the K1, K2, TX, and CKI pins. The K1 pin functions as the chip select line which times the write cycle for the 104-bit non-volatile memory. The K2 pin functions as the data strobe. The CKI pin serves as the serial clock input, and the TX pin acts as the data out pin.

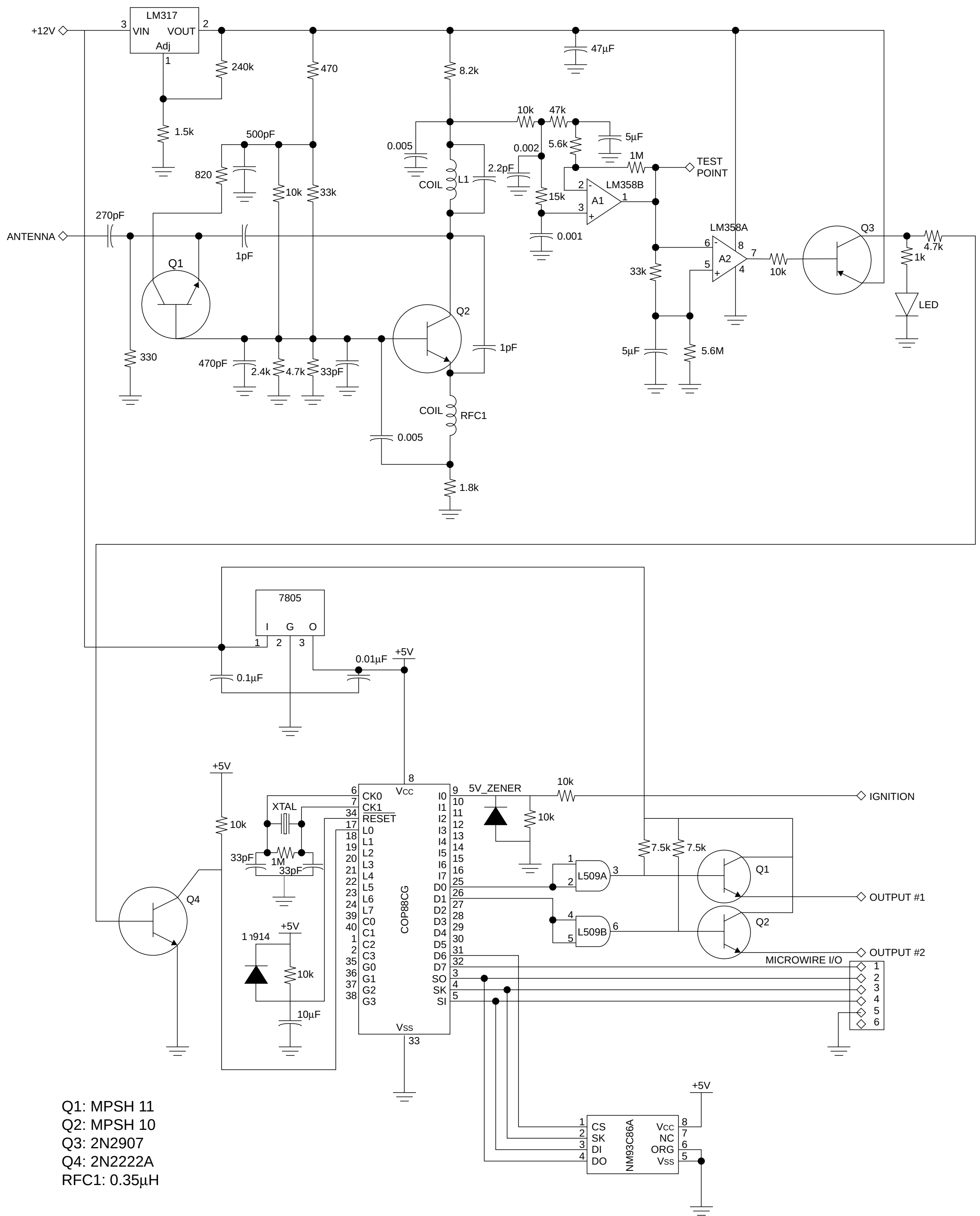


FIGURE 4. Complete RKE Receiver

Three voltages are required to program the device. These are Supervoltage (nominal 12V), Read/Write Voltage (V_{RW}), and Ground (0V). Read and Write modes can only be entered by applying a supervoltage to the device in a specific sequence. This precludes any risk of the device entering these modes during normal operation. The use of a supervoltage is for mode select only. The NM95HS01/02 generator has an on-board charge pump to supply the necessary programming voltage to the cells.

The programming protocol for the EEPROM array on-board the NM95HS01/02 matches the Microwire format closely;

however, there are some important differences. The first difference is the need for a supervoltage to select programming mode. Another difference is the requirement that the CKI clock input be clocked a minimum of 1500 times upon power-up to ready the device for programming. This is to allow the internal state machines and registers to complete their power up sequences.

COP888CG DESCRIPTION

The COP888CG is a fully static, low power, 8-bit CMOS microcontroller that contains 4,096 bytes of ROM to store program code, and 192 bytes of RAM to store register data. It has an 8-bit input, an 8-bit output, and two 8-bit bidirectional ports. The microcontroller uses a Microwire interface that allows it to communicate with a variety of serial EEPROMs.

Figure 5 shows the pin arrangement for the COP888CG micrcocontroller.

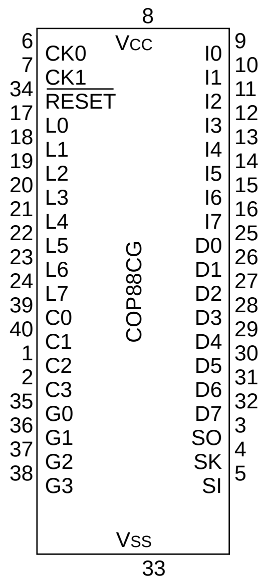


FIGURE 5. COP888CG Microcontroller Pin Arrangement

NM93C86A DESCRIPTION

The NM93C86A is a 16,384-bit non-volatile serial EEPROM that can be configured for either a 1,024 by 16, or a 2,048 by 8 architecture. The configuration is determined by the state of the ORG pin. If the ORG pin is tied low, the NM93C86A is configured as a 2,048 byte-wide memory; if the ORG pin is tied to V_{CC}, or left floating, the 1,024 word-wide configuration is enabled. An internal pull-up resistor assures that a floating ORG pin will be pulled high.

Figure 6 shows the pin arrangement for the NM93C86A.

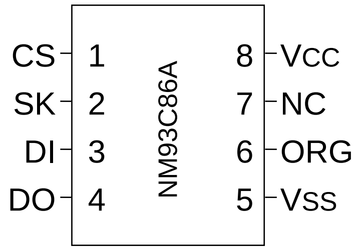


FIGURE 6. NM93C86A Pin Arrangement

PROGRAMMER DESCRIPTION

The programmer described here can be built using the schematics, PC board layouts, and software listings provided in this application note. It was designed to interface with an IBM-compatible PC through the RS-232C serial port. Any standard terminal communication software package (e.g., Windows®, PC-Talk, Kermit, etc.) may be used to communicate with the programmer.

The programmer was designed to accept specific 2-byte command sequences that tell the programmer which function to perform. For example, a 05h command tells the programmer to write the next 13 bytes of data to the EEPROM array in the NM95HS01/02. A 04h command tells the programmer to read the 13-bytes of EEPROM memory in the HiSeC device. This allows verification of the write operation.

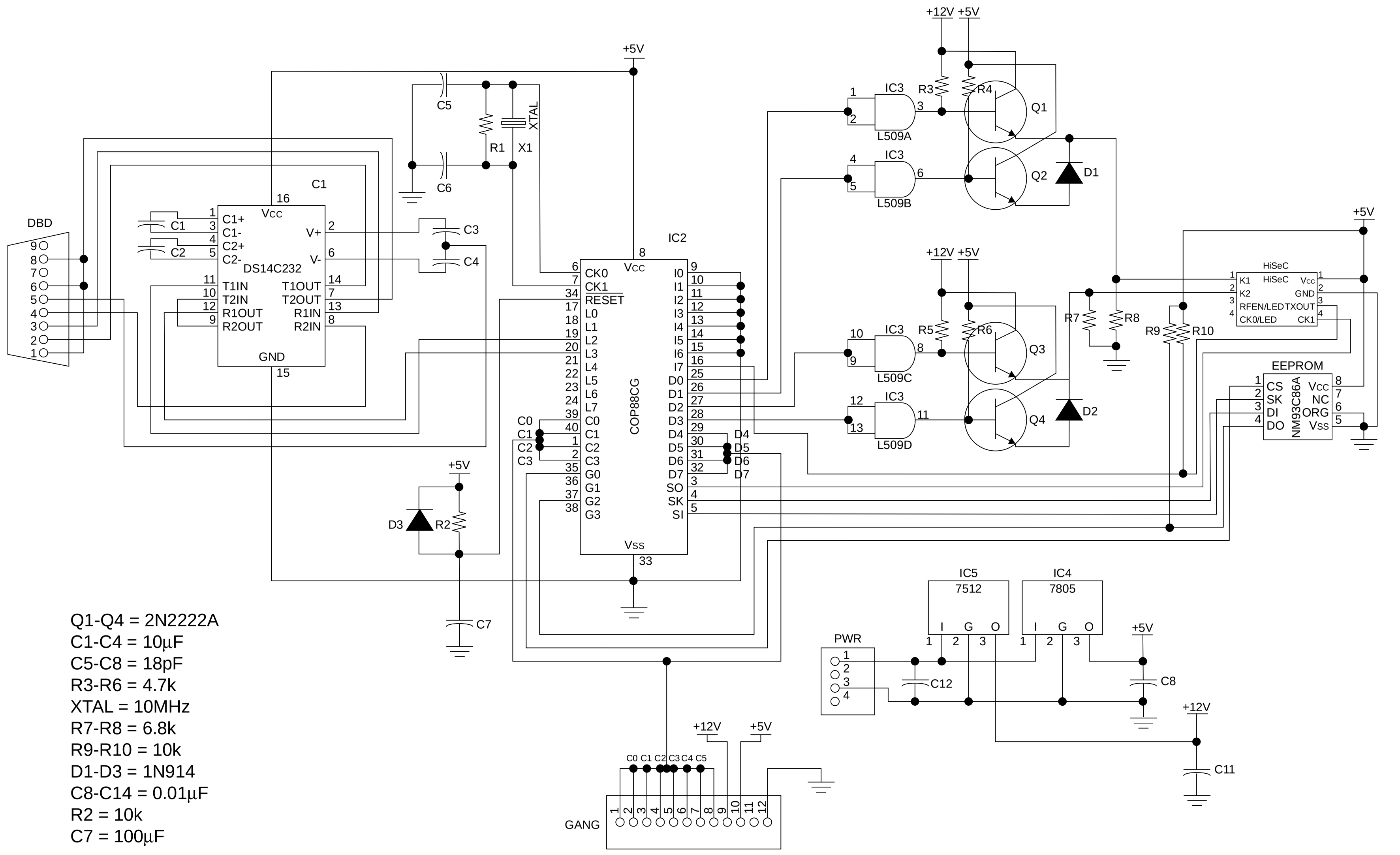
Similarly, a 13h command causes the programmer to write the next 2,048 bytes of data to the NM93C86A EEPROM array, and a 12h command causes the programmer to read the 2,048 bytes of NM93C86A EEPROM memory into the PC, allowing the write operation to be verified.

DEVICE PROGRAMMING

The information given in this application note provides the basic tools and instructions needed to program and verify the EEPROM array contained on-board the HiSeC rolling code generator, and the external EEPROM used to provide memory capability to the HiSeC decoder.

When programming these devices, care should be taken to obey all timing requirements and programming procedures for each part. Requirements, procedures and waveforms are given in detail in the individual data sheets.

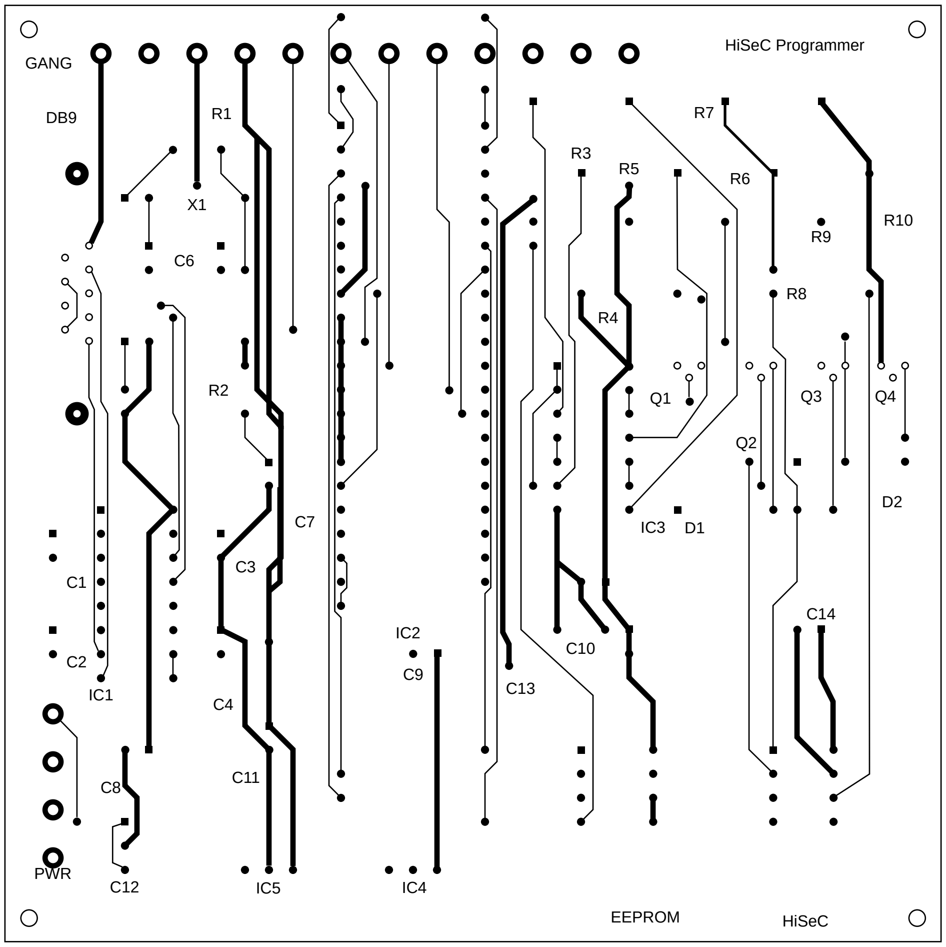
Schematic Listing



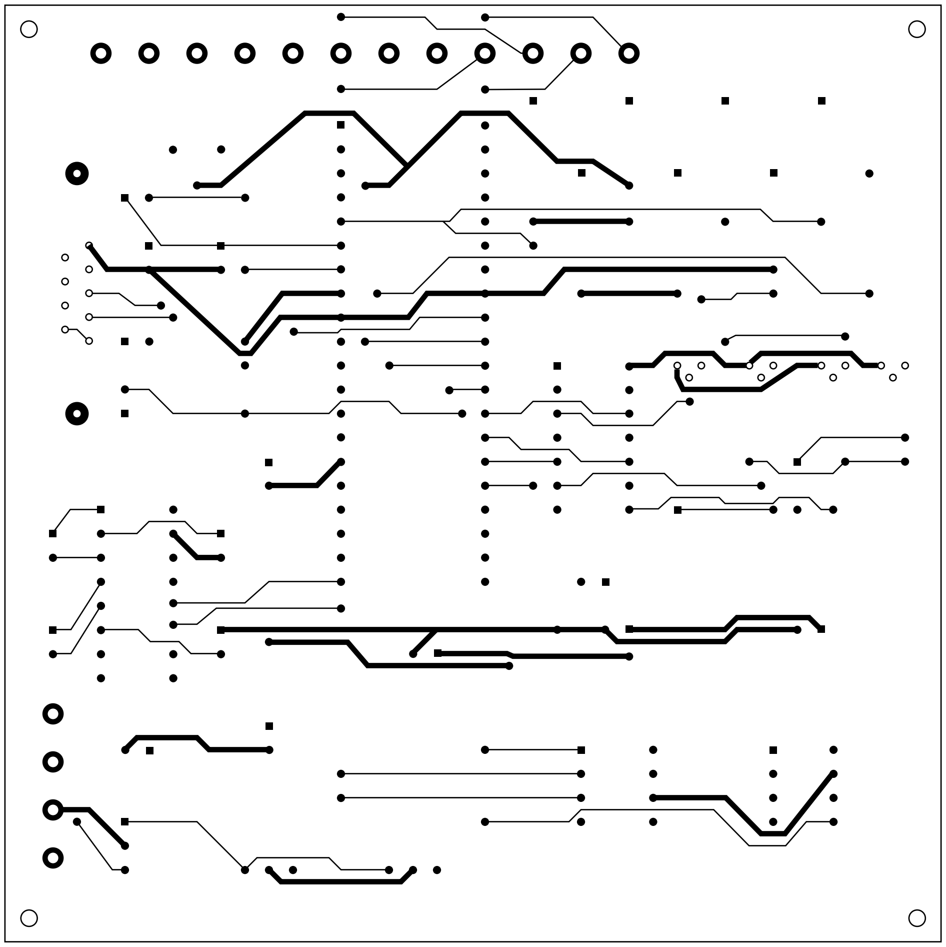
Q1-Q4 = 2N2222A
C1-C4 = 10µF
C5-C8 = 18pF
R3-R6 = 4.7k
XTAL = 10MHz
R7-R8 = 6.8k
R9-R10 = 10k
D1-D3 = 1N914
C8-C14 = 0.01µF
R2 = 10k
C7 = 100µF

FIGURE 7. Programmer Schematic

Top Layer of PCB



Bottom Layer of PCB



```
;---------------------------------------------------------------------
;                        Fairchild Semiconductor
;---------------------------------------------------------------------
;File:     Prog.asm
;Date      5-15-94
;Author:   Charles Watts
;Description:
;Assemble code for the HiSEC transmitter and external EEPROM programmer
;The programmer uses D0, D1, D2, D3, D7 and I7 pins for HiSEC programming.
;The programmer uses the Microwire interface and portG pin G0 for external
;EEPROM programming. The programmer uses the L2, AL3 pins of portL for UART
;communication.
;
;             Char1      Char2     from UART
;Functions    0          4         Read HiSEC data
;             0          5         Write HiSEC data
;             1          2         Read 98C86A EE
;             1          3         Write 93C86A EE
;
.incld cop888cg.inc
.sect one, ram
addl:   .dsb 1
addh:   .dsb 1
byt:    .dsb 1
pole:   .dsb 1
cnt:    .dsb 1
tmp:    .dsb 2
.endsect
.sect code, rom, abs=0
;------------------ Initialize port & register data ------------------
        delay1 = 0f0
        delay2 = 0f1
        page   = 0f2
;
start:   ld      PORTD, #00     ;Reset HiSEC I/O
         ld      PORTFC, #031   ;Set G0 as output
         ld      PORTCD, #00    ;Reset G port
         sbit    MSEL, CNTRL    ;Enable Microwire
         sbit    S0, CNTRL      ;Set Microwire strobe
         ld      PSR, #050      :Set UART
         ld      PAUD, #00b     ;Set UART for 9600, n, 8, 1
         sbit    2, PORTLC      ;Enable UART
         sbit    5, ENUI        ;
         rbit    3, PORTLC      ;
         rbit    3, PORTLC      ;
         ld      addh, #00      ;Clear Variables
         ld      addl, #00      ;
         ld      pole, #00      ;
         ld      PORTD, #0A     ;Set K1 and K2 to 5V
;-------------------------- Main Routine --------------------------
main:    jsr     getchar        ;Get byte from UART store byte in byt
         ifeq    byt, #030      ;if byt=0 then jump to subroutine HiSEC
         jp      HiSEC          ;
         ifeq    byt, #031      ;if byt=1 then jump to subroutine EE
         jp      EE             ;
         jp      main           ;loop until branch
;
EE:      jsr     getchar        ;Get byte from UART store byte in byt
         ifeq    byte, #032     ;if byt=2 read eeprom contents
         jsr     eerd           ;
         ifeq    byt, #033      ;if byt=3 write data to eeprom
         jsr     eewt           ;
         ld      pole, #00      ;pole=0
         jp      main           ;
;
HiSEC:   jsr     init           ;clock CKI 1,536 times
         jsr     getchar        ;Get byte from UART store byte in byt
         IFEQ    byt, #034      ;if byt=4 read hisec NVM
         jsr     hird           ;
         ifeq    byt, #035      ;if byt=5 write data into hisec NVM
         jsr     hiwt           ;
         jp      main           ;
;----------------------- End of main routine -----------------------
;
;--------------------- Subroutines will follow ---------------------
;-------------------------------------------------------------------
;Subroutine:   eewt
;Description:  The purpose of this routine is to write 2048 bytes
;              of data into a NM93C86A EE. The data is received
;              from the UART. The EE address is then incremented.
;-------------------------------------------------------------------
eewt:    jsr     ewen           ;Enable write mode
         ld      b, #tmp        ;load pointer
top:     jsr     getchar        ;Get byte from UART store byte in byt
         ld      a, byt         ;get first char
         x       a, [b+]        ;store at pointer location
         jsr     getchar        ;get second char
         ld      a, byt         ;
         x       a, [b-]        ;store at pointer location
         jsr     ascbin         ;convert ascii to binary
         jsr     put            ;store byt in eeprom
         jsr     count          ;advance address
         ifeq    pole, #01      ;if pole=1 return to main
         ret                    ;
         jp      top            ;
;-------------------------------------------------------------------
;Subroutine:   eerd
;Description:  The purpose of this routine is to read 2048 bytes
;              of data into a NM93C86A EE. The data is then sent
;              to the UART. The EE address is the incremented.
;-------------------------------------------------------------------
eerd:    ld      b, #tmp        ;load pointer
eelp:    jsr     get            ;Get data from eeprom
         jsr     binasc         ;convert binary to ascii
         ld      a, [b+]        ;
         x       a, byt         ‘store first ascii char
         jsr     putchar;       ;send data to PC
         ld      a, [b-]        ;
         x       a, byt         ;store second ascii char
         jsr     putchar        ;send data to PC
         jsr     count          ;advance address
         ifeq    pole, #01      ;if pole=1 return to main
         ret                    ;
         jp      eelp           ;
;-------------------------------------------------------------------
;Subroutine:   hiwt
;Description:  The purpose of this routine is to write 13 bytes
;              of data into the HiSEC NVM. The data is received
;              from the UART.
;-------------------------------------------------------------------
```

```
hiwt:     sbit    2, PORTD       ;set K2 to 12V
          sbit    0, PORTD       ;set K1 to 12B
          rbit    2, PORTD       ;set K2 to 5V
          rbit    0, PORTD       ;set K1 to 5V
          rc                     ;reset carry
          ld      page, #0d      ;set number of bytes to read (13 bytes)
          ld      cnt, #00       ;set cnt to 0 so first bit read is D0
lpt1:     sbit    0, PORTD       ;set k1 to 12V
          ld      b, #tmp
          jsr     getchar        ;Get byte from UART store byte in byt
          ld      a, byt
          x       a, [b+]
          jsr     getchar
          ld      a, byt
          x       a, [b-]
          jsr     ascbin
          rbit    3, PORTD       ;set K2 to 0
lpt:      ifbit   7, byt         ;if bit 0 in byt=1 set K2 to 5V if not skip
          sbit    3, PORTD       ;set K2=5V
          sbit    7, PORTD       ;clock CKI once
          rbit    7, PORTD       ;
          rbit    3, PORTD       ;K2 = 0
          ifeq    cnt, #07       ;has byte been written
          jp      rst1           ;if so jump to reset
          ld      a, cnt         ;increment cnt
          inc     a              ;
          x       a, cnt         ;
          ld      a, byt         ;shift byte once to the right
          rlc     a              ;
          x       a, byt         ;
          jp      lpt            ;loop
rst1:     clr     a              ;set cnt to 0
          x       a, cnt         ;
          rbit    0, PORTD       ;set K1=5V
lpt2:     ifbit   7, PORTI       ;loop until internal programming is complete
          jp      lpt2           ;
          drsz    page           ;decrement page by 1
          jp      lpt1           ;
          sbit    3, PORTD       ;
          rbit    1, PORTD       ;
          rbit    1, PORTD       ;
          rbit    1, PORTD       ;
          rbit    1, PORTD       ;
          rbit    1, PORTD       ;
          sbit    1, PORTD       ;
          ret                    ;
;----------------------------------------------------------------------
;Subroutine:   hird
;Description:  The purpose of this routine is to read 13 bytes
;              of data from the HiSEC NVM. The data is sent
;              to the UART.
;----------------------------------------------------------------------
hird:     sbit    0, PORTD       ;set K1 to 12V
          sbit    0, PORTD       ;
          sbit    0, PORTD       ;
          sbit    0, PORTD       ;
          rbit    0, PORTD       ;set K1 to 5V
          ld      page, #0d      ;read 13 bytes
          ld      cnt, #00       ;set bit read to 0
          rc                     ;reset carry
```

```
         clr     a              ;clear variable byt
         x       a, byt         ;
         sbit    0, PORTD       ;set K1 to 12V
         sbit    0, PORTD       ;set K1 to 12V
         sbit    0, PORTD       ;set K1 to 12V
         sbit    0, PORTD       ;set K1 to 12V
         sbit    7, PORTD       ;dummy bit 1
         nop
         nop
         rbit    7, PORTD       ;
         sbit    7, PORTS       ;dummy bit 2
         nop
         nop
lpt3:    sbit 7, PORTD          ;set CKI high
         ld      a, PORTI       ;Read bit input must be on I7-bit
         rbit    7, PORTD       ;set CKI low
         rlc     a              ;
         ld      a, byt         ;
         rlc     a              ;change rrc to do bit reverse
         x       a, byt         ;
         ifeq    cnt, #07       ;has byte been read
         jp      rst2           ;if so send byte
         ld      a, cnt         ;increment cnt
         inc     a              ;
         x       a, cnt         ;
         jp      lpt3           ;
rst2:    ld      b, #tmp
         jsr     binasc
         ld      a, [b+]
         x       a, byt
         jsr     putchar        ;send data to PC
         ld      a, [b]
         x       a, byt
         jsr     putchar
         clr     a              ;reset cnt
         x       a, cnt         ;
         drsz    page           ;decrement page
         jp      lpt3           ;
         rbit    1, PORTD       ;
         rbit    0, PORTD       ;
         rbit    0, PORTD       ;
         rbit    0, PORTD       ;
         rbit    0, PORTD       ;
         rbit    0, PORTD       ;
         sbit    1, PORTD       ;
         sbit    1, PORTD       ;
         ret
;----------------------------------------------------------------------
;Subroutine:   putchar
;Description:  The purpose of this routine is to send a byte
;              of data to the UART.
;----------------------------------------------------------------------
putchar:                        ;
         x       a, byt         ;load tbuf with byt
         x       a, TBUF        ;
```

```
a1:      ifbit   1, ENUR         ;check flag for end of transfer
         jp      a1              ;
         ld      delay1, #04     ;
brt:     ld      delay2, #0ff    ;
del:     drsz    delay2          ;
         jp      del             ;
         drsz    delay1
         jp      brt
         ret                     ;
;---------------------------------------------------------------------
;Subroutine:   getchar
;Description:  The purpose of this routine is to receive a byte
;              of data and store it into the variable byt.
;---------------------------------------------------------------------
getchar:         ;
         ifbit   1, ENU          ;wait for received byte
         jp      rst             ;
         jp      getchar         ;
rst:     x       a, RBUF         ;get transmitted byte and put into byt
         x       a, byt          ;
         ret
;---------------------------------------------------------------------
;Subroutine:   get
;Description:  The purpose of this routine is to get a byte
;              of data from an NM93C86A EEPROM and store the
;              value into the variable byt.
;---------------------------------------------------------------------
get:     sbit    0, PORTGD       ;set CS high
         ld      a, addh         ;load high address
         or      a, #030         ;and or it with command op-code
         x       a, SIOR         ;put in shift register
         sbit    .BUSY, PSW      ;send to EE
lp4:     ifbit   BUSY, PSW       ;
         jp      lp4             ;
         ld      a, addl         ;load low address
         x       a, SIOR         ;put in shift register
         sbit    BUSY, PSW       ;send to EE
lp5:     ifbit   BUSY, PSW       ;
         jp      lp5             ;
         ld      SIOR, #00       ;
         sbit    BUSY, PSW       ;read in dummy bit
         rbit    BUSY, PSW       ;
         sbit    BUSY, PSW       ;read EEPROM byte
lp6:     ifbit   BUSY, PSW       ;
         jp      lp6             ;
         x       a, SIOR         ;load that byte into byt
         x       a, byt          ;
         rbit    0, PORTGD       ;reset CS
         ret                     ;
;---------------------------------------------------------------------
;Subroutine:   put
;Description:  The purpose of this routine is to write 1 byte
;              of data to the NM93C86A.
;---------------------------------------------------------------------
put:     sbit    0, PORTGD       ;set CS high
         ld      a, addh         ;load high address
         or      a, #028         ;OR it with op-code
         x       a, SIOR         ;load into shift register
         sbit    BUSY, PSW       ;send to EE
lp7:     ifbit   BUSY, PSW       ;
```

```
         jp      lp7            ;
         ld      a, addl        ;load low address
         x       a, SIOR        ;load into shift register
         sbit    BUSY, PSW      ;send to EE
lp8:     ifbit   BUSY, PSW      ;
         jp      lp8            ;
         ld      a, byt         ;put data into shift register
         x       a, SIOR        ;send to EE
         sbit    BUSY, PSW      ;
lp9:     ifbit   BUSY, PSW      ;
         jp      lp9            ;
lp10:    ifbit   SI, PORTGP     ;wait for programming cycle end
         jp      lp10           ;
lp11:    ifbit   SI, PORTGP     ;wait for DO to fall low
         jp      lp12           ;
         jp      lp11           ;
lp12:    rbit    0, PORTGD      ;reset CS
         ret                    ;
;-------------------------------------------------------------------
;Subroutine:   ewen
;Description:  The purpose of this routine is to enable
;              the NM93C86A programming cycles.
;-------------------------------------------------------------------
ewen:    sbit    0, PORTGD      ;set CS high
         ld      SIOR, #026     ;load op-code into shift register
         sbit    BUSY, PSW      ;send to EE
lp13:    ifbit   BUSY, PSW      ;
         jp      lp13           ;
         ld      SIOR, #00      ;send zero’s to EE
         sbit    BUSY, PSW      ;
lp14:    ifbit   BUSY, PSW      ;
         jp      lp14           ;
         rbit    0, PORTGD      ;reset CS
         ret                    ;
;-------------------------------------------------------------------
;Subroutine:   count
;Description:  The purpose of this routine is to keep track of
;              the EEPROM address pointer.
;-------------------------------------------------------------------
count:   ld      a, addl        ;load low address and
         ifeq    a, #0ff        ;increment. if low address
         jp      za             ;equals hex ff then jump to
         inc     a              ;next routine.
         x       a, addl        ;
         ret                    ;
za:      ld      a, addh        ;check to see if high address
         ifeq    am #07         ;is = to hex 07. if not reset low address
         jp      zc             ;and exit. if so reset low and high address
         inc     a              ;
         x       a, addh        ;
         ld      addl, #00      ;
         ret                    ;
zc:      ld      pole, #01      ;set pole to 1 to flag host routine
         ld      addh, #00      ;
         ld      addl, #00      ;
         ld      byt, #06       ;set byt to 6 so accedenal branch will
         ret                    ;not occur.
;-------------------------------------------------------------------
;Subroutine:   init
;Description:  The purpose of this routine is clock the hisec CKI
```

```
;         input 1,536 times.
;--------------------------------------------------------------------
init:     ld      delay1, #06    ;upper limit
loop1     ld      delay2, #0ff   ;lower limit
loop2     sbit    7, PORTD       ;clock CKI 1536
          nop
          nop
          nop
          nop
          nop
          rbit    7, PORTD       ;chip registers
          drsz    delay2         ;set.
          jp      loop2          ;
          drsz    delay1         ;
          jp      loop1          ;
          ret                    ;
;--------------------------------------------------------------------
;Subroutine:   binary to ascii
;Description:  The purpose of this routine is convert a binary
;              byte into two ascii characters.
;--------------------------------------------------------------------
binasc:   x       a, byt
          push    a              ;put accum a on stack
          swap    a              ;Most significant nibble gets printed first
          and     a, #0ff        ;swap upper and lower nibbles
          ifgt    1, #009        ;if a-f add 7 hex
          add     a, #007        ;
          add     a, #030        ;convert to ascii
          x       a, [b+]        ;
          pop     a              ;Least significant nibble gets printed last
          and     a, #009        ;pull a from stack
          ifgt    a, #009        ;if a-f add 7
          add     a, #007        ;
          add     a, #030        ;convert to ascii
          x       a, [b-]        ;
          ret                    ;
;--------------------------------------------------------------------
;Subroutine:   ascii to binary
;Description:  The purpose of this routine is convert a 2 ascii
;              characters into one binary byte.
;--------------------------------------------------------------------
ascbin:   rc                     ;reset carry
          ld      a, [b+]        ;load first character
          ifget   a, #039        ;if not 0 - 9
          jp      h1             ;jump to next condition
          and     a, #00f        ;else mask off high nibble
          jp      h3             ;jump to shift
h1:       ifgt    a, #046        ;if not A - F
          jp      h2             ;jump to next condition
          and     a, #00f        ;else mask off high nibble
          add     a, #009        ;add 9 hex to convert nibble
          jp      h3             ;jump to shift
h2:       ifgt    a, #006        ;if not a - f
          jp      h3             ;jump to next condition
          and     a, #00f        else mask off high nibble
          add     a, #009        ;add 9 hex to convert nibble
h3:       swap    a              ;shift bits
          x       a, byt         ;store into variable
          ld      a, [b-]        ;load first character
          ifgt    a, #039        ;if not 0 - 9
```

```
         jp      h4             ;jump to next condition
         and     a, #00f        ;else mask off high nibble
         jp      h6             ;jump to shift
h4:      ifgt    a, #046        ;if not A - F
         jp      h5             ;jump to next condition
         and     a, #00f        ;else mask off high nibble
         add     a, #009        ;add 9 hex to convert nibble
         jp      h6             ;jump to shift
h5:      ifgt    a, #066        ;if not a - f
         jp      h6             ;jump to next condition
         and     a, #00f        ;else mask off high nibble
         add     a, #009        ;add 9 hex to convert nibble
h6:      or      a, byt         ;logical or a and byt
         x       a, byt         ;store new byte in variable
         ret
         .end start
;--------------------------end of program-----------------------------
```

THIRD PARTY SUPPORT

The following contact information is provided to assist the user in selecting and maintaining COP8xx and NM95HS01/02 development tools.

Manufacturer and Product	Phone Numbers		
Xeltek–Superpro II	U.S.:	(408) 524-1929	
	Europe:	+49-5722-203125	(Germany)
	Asia:	+65-276-6433	(Singapore)
	BBS:	(408) 245-7082	
System General Turpro-1	U.S.:	(408) 263-6667/(800) 967-4776	
Universal Programmer	Europe:	+31-921-7844	(Switzerland)
	Asia:	+886-2-9173005	(Taipei, Taiwan)
	BBS:	(408) 262-6438	

The programmer described in this application note can be ordered fron Fairchild, order information NM95HSPROG.

Fairchild Semiconductor
Americas
Customer Response Center
Tel. 1-888-522-5372

Fairchild Semiconductor
Europe
Fax: +44 (0) 1793-856858
Deutsch Tel: +49 (0) 8141-6102-0
English Tel: +44 (0) 1793-856856
Français Tel: +33 (0) 1-6930-3696
Italiano Tel: +39 (0) 2-249111-1

Fairchild Semiconductor
Hong Kong
8/F, Room 808, Empire Centre
68 Mody Road, Tsimshatsui East
Kowloon. Hong Kong
Tel; +852-2722-8338
Fax: +852-2722-8383

Fairchild Semiconductor
Japan Ltd.
4F, Natsume Bldg.
2-18-6, Yushima, Bunkyo-ku
Tokyo, 113-0034 Japan
Tel: 81-3-3818-8840
Fax: 81-3-3818-8841