Cordless Caller Identification (CCID)

Project Documentation

Submitted to fulfill the requirements of
Philips Semiconductors
Dream Machine Design Contest

by

Derek Matsunaga
September 30, 1994

Contents

(in order of appearance)

Required document

Brief Project Summary

Brief Project Summary

Introduction

The Cordless Caller Identification project, herein after referred to as "CCID", combines the convenience of a cordless phone with the information of caller identification. Most cordless phone users who subscribe to the caller ID service do not have more than one caller ID "box". For this reason, the convenience of a cordless phone is easily nullified. If the cordless handset is away from its base unit (and the caller ID box), the user must return to the caller ID box to see who is calling prior to answering the phone. To illustrate this point, assume that the base unit for the cordless phone and the caller ID box are located in the kitchen and the user is working in the garage (or garden, patio, living room, watching TV, bathroom, etc.). When the phone rings and the user wishes to know who is calling, he must return to the kitchen to see the caller ID box, which defeats the purpose of a cordless phone! The user may as well have a wired phone if he wishes to see the caller ID information prior to answering the phone.

The CCID project eliminates this problem by combining an integrated, full featured caller ID system and a cordless telephone. When the phone rings, the caller ID information is decoded and displayed on the base unit and transmitted to the handset unit. This eliminates the need to return to the caller ID "box" prior to answering the cordless phone. Although the prototype CCID system is built into a consumer-grade cordless telephone system (see photos), the CCID system could be manufactured as a cordless telephone add-on accessory.

Hardware overview

The prototype CCID system utilizes two 8XC75X microcontrollers - the base unit contains a '752 while the handset unit uses a '750. These two microcontrollers were selected for the following reasons:

Base unit (87C752)	Handset unit (87C750)
• IIC support	• small footprint
• PWM output	• low power consumption
• 2K ROM, 64 bytes RAM	• open drain I/O pins
• number of I/O pins	• 1K ROM, 64 bytes RAM
• included in evaluation kit	• included in evaluation kit

Using these two microcontrollers enables the system to obtain the necessary performance while minimizing software overhead and reducing the number of required external support components. In addition to the microcontrollers, each unit in the CCID system (the base and handset) require only two additional ICs and a few discretes to achieve full functionality (see schematics).

Scoring worksheet requirements

The following text addresses each individual section on the scoring worksheet. Due to the magnitude of this project, it is impossible to fully describe it in fewer than 500 words. Refer to the "optional" information for greater detail on the CCID system specifics.

Uniqueness of the Application

Although caller ID is a well known and mature consumer product, the CCID project is unique for the following reasons:

- The system is cordless (see project description).
- The caller ID FSK demodulation is accomplished in software which eliminates the needs for a dedicated FSK demodulator device, thus reducing cost, pin count, etc.
- The system is fully integrated into an existing cordless telephone (see photos).
- The complete system is very inexpensive (see BOM).

As mentioned above, the CCID project utilizes specific features found in the selected 8XC75X microcontrollers. Revision 1.5 of the base unit software uses almost all 2K bytes of ROM and all 64 bytes of RAM. The power requirements and small package size of the handset unit make the 87C752 an obvious choice.

Part count reduction

Since the caller ID FSK demodulation is done in software, the need for a discrete FSK demodulator and its support components is eliminated. Additionally, the internal pull up resistors found on many of the '750 and '752 ports are used to replace discrete resistors, where practical. The major peripheral components (the 24C04 and LCD display on the base unit and the MAX877 and LCD display on the handset unit) require a minimal number of external components.

Cost reduction

The cost of the CCID system has been minimized by eliminating the use of a discrete FSK demodulator (and support hardware) and replacing it with an LM339, which is just about the cheapest IC available. Additional savings is realized by doing the ring signal processing in software. See the BOM and cost estimate for further details.

Power consumption reduction

The need for low power consumption in a battery operated cordless handset is obvious. Power consumption has been minimized by turning-off the display and microprocessor when not in use. Also, the low power version of the LM339, the LP339, was incorporated to reduce the quiescent power consumption of the CCID handset hardware. Large pull up resistors and low power components help reduce the quiescent current drain on the handset battery to less than 100µA. The '752 runs at a "slow" 6MHz to help minimize the power requirements. The "power-down" mode of the '752 is used to further reduce power consumption. Extensive testing has shown no noticeable degradation in battery life.

Increased performance

Raw processor performance is not a major consideration in the CCID system. However, it should be noted that the reliability of the FSK demodulation is a direct function of processor speed. A slower processor would not be able to provide the noise margins which were achieved with the 12MHz 87C752. The first iteration of the CCID prototype was developed with a 6MHz oscillator and the results were found to be unreliable. The ability to simply swap crystals and modify software was most certainly valuable.

Creative use of software

Again, a major feature of the CCID system is that the caller ID FSK demodulation is done in software. Another software feature is the commonality of the base unit architecture with that of the handset unit. Much of the core software (FSK demodulation and display drivers) were directly transported from the base unit software. All software code segments, drivers, and subroutines were developed and tested out of the CCID system (see Software Testing and Validation).

There were no "accidents" in developing the software. Each instruction, code segment, subroutine, memory location, and register was *engineered* to perform a specific task under various constraints such as execution time, ROM space required, and transportability. Each and every line of assembly code is commented with the number of cycles required for execution and an explanation of its function. The error trapping and error handling functions were developed as if CCID were a "medical device".

Optional document

CCID Base Unit Hardware Description & Design Review

CCID Base Unit Hardware Description & Design Review

Introduction & Preface

Schematics were not available for the Sony SPP-75 cordless phone system. All of the signals were found by probing around on the phone's PCBs until the desired signal was found or by carefully injecting known signal sources and observing responses. The electrical characteristics of the signals, in general, are not known. For example, it is not known how much current each signal can source/sink nor are the signal's slew rates known. However, care has been taken to preserve the original functionality of the cordless phone system by minimizing the source/sink currents of the phone's signals. Although the system works quite well as is, modifications will be made if (and when) the signal output characteristics are learned.

There is nothing outstanding about the Sony SPP-75 cordless phone system. This phone was selected as the CCID platform simply because it was on hand at the inception of the CCID project. I have been using this cordless phone for several years and have been satisfied with its performance. The construction of the phone is quite rugged which makes for easy plastics modification. After several years, this phone is still sold in retail stores, which is an indicator of a stable design.

In addition to providing full caller ID functionality, the CCID base hardware was designed under the following constraints (guidelines):

- allow high noise margins and component tolerances
- minimize component count
- use common "off the shelf" devices
- minimize cost
- provide adequate phone line isolation
- the original functionality of the phone must remain completely intact
- all IC pins must be accounted for
- minimize power consumption

The following discussion will show that all of the design criteria were met and that the hardware design is solid and stable.

Hardware Description & Design review

(reference CCID Base Schematic v2.6 and BOM)

The CCID base unit hardware design consists of four ICs. U1 is an LM339 quad voltage comparator which is used to convert the analog phone data into digital PWM format. U1 is also used to provide a hardware timer for the software (to be discussed later). U2 is an 87C752 microcontroller running at 12MHz. U3 is a 24C04 serial EEPROM which communicates using the IIC protocol. This device provides non-volatile storage for call records. Finally, U4 is a four line by 16 character LCD display unit which is controlled by the HD44780 system (see HD44780 data sheet).

The phone line front-end

The phone line is connected to the CCID base hardware via CN1. MOV1 protects the CCID circuitry from high voltage transients which may occur on the phone line. This is "standard practice" for most phone connected devices. The phone line signal is capacitively coupled through C1 and C2 to the inputs of U1. Diodes CR2 and CR4 ensure that the differential voltage at the inputs to U1 does not exceed about .6V. Diodes CR1 and CR3 protect the inputs of U1 from exceeding the power supply rails by more than about .6V.

U1 is configured as a differential comparator which will convert small analog signals into a time varying pulse width modulated signal at the output. Resistor R2 provides a return path for the phone signal while R4 sinks a small amount of current from the inverting input of U1, thus ensuring that U1's output is high when there is no signal on the phone line. R1 and R3 provide a reference voltage at the non-inverting input of U1 about which the input signal can "swing". R5 provides a pull up for U1's open collector output.

The output of U1 (pin 1) is connected to the inverting input of U1 (pin 4). The purpose of this second comparator is to invert the output of the first comparator before the input to the microcontroller. Additionally, the second comparator provides a large hysteresis window which helps to eliminate the effects of noise on the phone line. R6 and R7 provide a reference at the non-inverting input of U1 (pin 5). R8 produces a voltage divider action with the reference depending of the state of the output. R9 is the pull up resistor for the output of the second comparator. The output is then connected to P1.3 of the microcontroller. Although P1.3 has an internal pull up resistor, R9 is added to provide a "stronger" pull up to help maintain waveform symmetry and reduce the effects of open collector capacitance.

In summary, the phone line front-end of the CCID system provides phone line isolation and a PWM representation of the differential analog phone line data.

The EEPROM

The EEPROM (U3) is a 24C04 device capable of storing 512 bytes in two 256 byte pages. The communications protocol is functionally identical to IIC. So, only two signals from the processor are required - namely SDA and SCL. U3 pins 1, 2, and 3 set the address of the EEPROM and are grounded in this application. Additional IIC devices can easily be added if necessary. The WP (write protect) pin of U3 is grounded to allow writes to the device.

The IIC clock and data lines are pulled up to Vdd through R11 and R10, respectively. This is necessary because the SDA and SCL signals are open drain. The 10K resistors provide enough current sourcing capability to ensure relatively fast rise times when the drive transistors are turned-off. In other words, the 10K resistors can quickly discharge any capacitance which may be present on the open drains.

The function buttons

The function buttons are connected to U2 via CN2. U2 port bits P1.4, P1.5, and P1.6 are used to read the status of the function buttons. These port inputs are in an open drain configuration with internal pull up resistors. These internal pull ups eliminate the need for external resistors. Closing SW2 pulls P1.5 and P1.6 low through the "wired-OR" configuration provided by CR5 and CR6. This allows three port bits to read the four function buttons. The button decoding functions are accomplished in software (see software description).

The hardware timer ("watchdog")

One of the spare comparators in U1 is used to implement an independent hardware timer for the '752 to use. The software uses this timer as a "soft watchdog" in that is can clear the signal and continually poll it until it becomes asserted again.

Q1 is a general purpose signal FET which is used to quickly discharge C8 through R17. R17 is a small resistor (10Ω) which limits the drain current of Q1 to an acceptable level. When P0.3 (W_DOG_RST) is set high by the software, the gate of Q1 is pulled high through P0.3's internal pull up resistor. This effectively grounds the drain of Q1 and discharges C8. Conversely, when the software clears P0.3, the gate of Q1 is grounded and its drain is allowed to float, thus allowing C8 to charge through R18.

On the charge cycle, C8 slowly charges through R18. Meanwhile, the voltage at the non-inverting input of U1 (pin 9) is slightly below the Vdd rail since R21 and R11 provide a path to Vdd which is divided by R20. When the charge on C8 exceeds the voltage at the non-inverting input, the output transistor of the comparator is turned-on, thus pulling the output signal low through R22.

With the comparator output low, the new reference at the non-inverting input switched from being slightly below Vdd to being slightly above ground due to the divider action of R22 and R21 with R19. The software can read this output to determine if the time constant has expired. The time constant is about one second and its value is not critical to software operation.

The operation of the "watchdog" is summarized as follows: The software clears the watchdog by asserting P0.3. This causes the comparator output to go high and can be read at P1.7. The software then clears P0.3 to allow C8 to charge. When C8 charges to slightly below Vdd, the output of the comparator goes low and is read at P1.7. The software can repeat this process randomly because the watchdog is "passive" since it does not reset the processor. Hence, the term "soft watchdog" is used to describe this function.

The LCD display interface

U4 is a four line by 16 character LCD display which is controlled by the popular HD44780 display controller. This controller provides a standard interface among many different LCD configurations. It masks the "ugly" tasks of refreshing the display and character generation, thus allowing the software to simply "print" a character or send a command.

The hardware interface is pretty straight forward. The eight bit data interface is connected to port 3 of the '752. All of P3's pins have internal pull up resistors and the display is CMOS compatible so no additional level shifting or buffering is needed. The display control signals are connected to P1.0, P1.1 and P1.2. These port bits also have internal pull up resistors which eliminates the need for external devices. The remaining details of the display interface such as timing and initialization are implemented in software.

Potentiometer R22 is used to adjust the contrast of the LCD display. C9 filters the signal at the CONT pin to prevent noise from corrupting the display.

The base unit interface

The base unit interface is perhaps the most interesting part of the design - primarily because it only requires two port bits and does not affect the original functionality of the phone. It must be reiterated that schematics for the cordless phone system are not available. For this reason, the original function of the base unit signals may not be completely understood.

When the software is not transmitting data to the handset, P0.2 (open drain) is held low which essentially pulls the gate of Q2 to ground and holds the PAGE_BUTTON signal near ground. The "page" function of the phone is active high. Since the gate of Q2 is grounded, Q2's drain is allowed to float. This allows the base unit's normal data signal to pass through the 20K/100K (R53) combination through C53 to the phone's transmitting hardware. When P0.2 is low, the phone operates in its normal ("native") mode.

When the software is ready to transmit data to the cordless handset, P0.2 is set high. Although P0.2 is in an open drain configuration, its signal is pulled up to Vdd through R15 and R16. This also pulls the PAGE_BUTTON signal to Vdd, which enables the phone's "page" feature. When in "page" mode, all of the necessary transmission hardware is powered-up and the transmitter accepts data through C53. Additionally, the gate of Q2 is pulled high through R15 and R16. This effectively grounds the drain of Q2 and inhibits the phone's native data through the 20K

side of R53. In other words, when P0.2 is asserted by the software, the phone's "page" button is pressed (and held) and the phone's native data signal is inhibited.

The output of P0.4 is normally an open drain signal with an internal pull up. However, the software uses P0.4 in the PWM mode, which applies an active pull up (totem pole) to its output. The transmit data appears at P0.4 (XMIT_DATA). This signal is significantly attenuated by the R12 and R13 down to the levels which the phone's transmit hardware normally sees (this value was determined by direct measurement). The PWM data signal is then capacitively coupled though C7 to the transmitter's data input. R14 isolates C7 and R13 from the phone's nromal data signal when the phone is in its normal mode of operation. The data is then transmitted, unbeknownst to the phone, to the cordless handset using the phone's native (and approved) FM transmission frequencies. When the software completes the data transmission, the phone is returned to its native mode by clearing P0.2.

Miscellaneous hardware notes

The CCID base hardware operates from a single +5V power supply, denoted as Vdd on the schematic. This power supply is "borrowed" from the base unit. Although the specifics of this power supply are not known, it is capable of supplying more than enough current to run the CCID hardware. Visual inspection of the power supply indicates that it is pretty clean and free of noise. For good measure, capacitor C3 locally filters the power supply on the CCID prototype board.

Capacitor C6 provides a momentary high signal at U2's reset pin when power is first applied to the system. Then, C6 is gradually charged to ground via the internal resistance of the RST pin. This provides a falling edge which resets the processor on power-up. Crystal X1 provides a 12MHz time base for U2's internal oscillator.

This version of the CCID design leaves one comparator within the LM339 unused. Until a future application finds use for it, its open collector output pin and input pins are grounded to prevent noise from entering the device. To minimize EMI and susceptibility, all IC pins are tied somewhere, whether they are used or not.

Optional document

CCID Handset Unit Hardware Description & Design Review

CCID Handset Unit Hardware Description & Design Review

Introduction & Preface

As with the base unit hardware, schematics were not available for the Sony SPP-75 cordless phone system. All of the signals were found by probing around on the phone's PCBs until the desired signal was found or by carefully injecting known signal sources and observing responses. The electrical characteristics of the signals, in general, are not known. For example, it is not known how much current each signal can source/sink nor are the signal's slew rates known. However, care has been taken to preserve the original functionality of the cordless phone system by minimizing the source/sink currents of the phone's signals. Although the system works quite well as is, modifications will be made if (and when) the signal output characteristics are learned.

The CCID handset hardware was designed under the following constraints (guidelines):

- minimize power consumption because the handset is battery operated
- allow high noise margins and component tolerances
- minimize component count
- use common "off the shelf" devices
- minimize cost
- the original functionality of the phone must remain completely intact
- all IC pins must be accounted for

The following discussion will show that all of the design criteria were met and that the hardware design is solid and stable.

Hardware Description & Design review

(reference CCID Handset Schematic v1.3 and BOM)

Like the base unit, the CCID handset hardware consists of four ICs. IC1, and LP339, is a low power version of the LM339 quad voltage comparator. When this device is in its quiescent state, it draws only 60µA. U2 is a MAX877 voltage converter which converts the handset's 4V battery voltage to the +5V required by U3 and U4. U3 is an 87C750 microcontroller with 1K of program ROM, 64 bytes of RAM, and plenty of I/O pins. Finally, U4 is a two line by 16 character LCD display which is controlled by the HD44780.

The power supply system

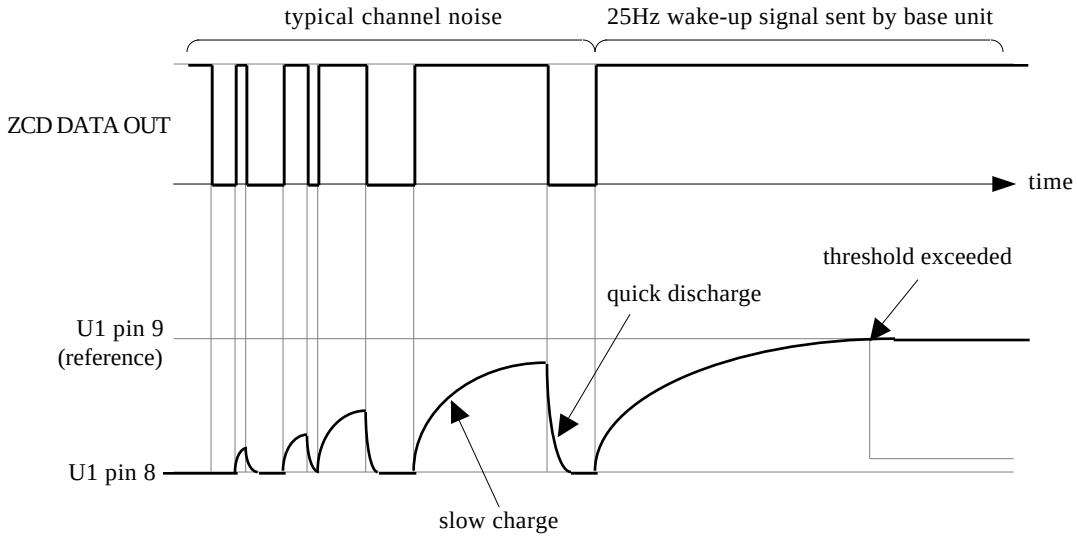
As mentioned earlier, the battery voltage found within the cordless handset is 4V. This is fine for linear parts but the "digital" devices require +5V. So, the MAX877 was selected to boost the 4V battery voltage to +5V. The MAX877 is a versatile converter which is capable of automatically switching between the boost and the buck modes as input voltage varies. The CCID application operates strictly in the boost mode due to the low battery voltage. This device also has a shutdown pin which makes it a perfect choice for this application. When the MAX877 is in the shutdown mode, it draws only 20µA. Additionally, the MAX877 requires only three external components to complete its system - namely and inductor and two capacitors. Internal operational details of the MAX877 can be found in its data sheet. The configuration of the MAX877 shown on the schematic was "lifted" directly from the MAX877 application notes.

The output of U2 is the +5V power supply (Vdd) for the processor and the LCD display. When the CCID system is not in use (i.e. shut down), U2 is put into its low power shutdown mode, thus removing power from the processor and the display.

U1, the LP339 quad voltage comparator, is supplied directly from the handset's 4V battery and therefore cannot be turned-off unless the battery is discharged. This is necessary because U1 is responsible for activating the MAX877, hence the LP339 must always have power.

The non-inverting input of U1 (pin 7) is connected to a zero cross detected output on the handset. This output was found at IC501 on the handset. This signal, termed ZCD DATA OUT on the schematic, is a PWM representation of the data received by the handset. It has a fairly high output impedance, so U1 (pin 7) is used to buffer the signal so it can be used with the rest of the CCID hardware. A Vbat/2 reference is applied to the non-inverting input of the comparator (pin 6). When this reference is exceeded by the non-inverting signal (ZCD DATA OUT), the output of the comparator floats due to its open collector configuration.

The +5V power supply is enabled when the voltage at U1 pin 8 exceeds the reference set by R6 and R8 at U1 pin 9. This can only happen if a long pulse is received at ZCD DATA OUT. Each time ZCD DATA OUT goes below the threshold set by R1 and R2, U1 pin 1 is pulled low and C2 is quickly discharged through R3 and CR2. If C2 is allowed to charge through R4 and R5 to the threshold set by R6 and R8, the +5V power supply will be enabled. This is how the base unit can "wake-up" the handset unit by sending a series of 25Hz pulses. The following signal diagram further illustrates the "wake-up" process.



When the threshold at U1 pin 9 is exceeded, the output of U1 (pin 14) is pulled low. Feedback resistor R7 pulls the reference voltage at U1 pin 9 essentially to ground, thus making it impossible for the power supply to get turned-off by a low signal at the inverting input. In this manner, the +5V power supply is latched on. The output is connected to U1 pin 4 for inversion. This causes U1 pin 2 to get pulled up through R9, thus enabling the +5V power supply and pulling the gate of Q1 high.

When Q1 is turned-on, its drain is pulled essentially to ground, which enables the power supply to the rest of the cordless handset. At this point, it should be noted that the power to the entire handset is not always on. To extend battery life, the handset's native CPU enables the power supply every 3.5 seconds for about one second. This produces a "channel polling" function which conserves battery power while sampling the cordless channel for data from the base unit.

Immediately after U3 powers-up, the software clears PS_LATCH (open drain), which permanently latches-on the +5V supply. Subsequently, the +5V supply can only be disabled when the software releases the PS_LATCH port bit and clears the PS_SHUTDOWN port bit.

The operation of the power supply system is summarized as follows: The +5V power supply is normally in its

shutdown mode, drawing only its 20µA quiescent current. When a 25Hz wake-up pulse is received from the base unit, the power supply is enable and latched-on, both by hardware and software. It can subsequently only be disabled by software or by pressing the handset's "talk" button.

The data signal

The cordless data signal is fed into the processor at P1.3. This port is in an open drain configuration with an internal pull up resistor. The remaining comparator in U1 is used to buffer the high impedance ZCD DATA OUT signal from the '750. The comparator reference is set at Vbat/2 by R1 and R2 and filtered by C1. R10 and C8 form a first order low pass filter which attenuates high frequency noise (fast transitions) on the ZCD DATA OUT signal. This signal is then squared-up by U1 and connected to the '750.

The data at P1.3 (RECEIVE_DATA) is in a PWM format. The software uses an FSK demodulation algorithm to decode the received data and display it on the LCD.

The LCD display interface

U4 is a two line by 16 character LCD display which is controlled by the popular HD44780 display controller. This controller provides a standard interface among many different LCD configurations. It masks the "ugly" tasks of refreshing the display and character generation, thus allowing the software to simply "print" a character or send a command.

The hardware interface is pretty straight forward. In fact, it is identical to that which is used on the base unit. The eight bit data interface is connected to port 3 of the '750. All of P3's pins have internal pull up resistors and the display is CMOS compatible so no additional level shifting or buffering is needed. The display control signals are connected to P1.0, P1.1 and P1.2. These port bits also have internal pull up resistors which eliminates the need for external devices. The remaining details of the display interface such as timing and initialization are implemented in software.

Potentiometer R11 is used to adjust the contrast of the LCD display.

Miscellaneous hardware notes

Because schematics were not available during development of the handset hardware, it is difficult to provide detail regarding the native handset signals. Comments on the left hand side of the CCID handset schematic represent the entire knowledge of the signals.

IC701 pin 64, termed SYS_POWER_ON\ on the schematic, is a signal which is normally high and goes low when the native power systems on the handset are manually enable (i.e. by pressing the "talk" button). The signal is "wire ORed" with the PS_SHUTDOWN signal through CR3. This provides a hardware method by which the +5V power supply can be immediately disabled. The software also detects a manual request for handset power and shuts off the +5V power supply. By using both the hardware and software methods of shutting-down the power supply, it is virtually guaranteed that the CCID system will immediately shut-down when the phone is answered.

The "talk button" signal is connected directly to the gate of Q2. In this capacity, Q2 provides a buffer and inversion between the high impedance talk button signal and the processor. P1.5 has an internal pull up resistor. The software uses this signal to detect a manual request for system power (see software description).

The POWER_ENABLE signal at P1.4 is used by the software to detect handset ringing. This signal goes low if the handset power is under its native CPU control. When the handset rings, its native CPU pulls P1.4 low through CR4, thus indicating to the software that the handset is ringing. P1.4 has an internal pull up resistor.

Optional document

CCID Base Unit Software Description

CCID Base Unit Software Description

Introduction & functional definition

The purpose of the CCID base software is to provide a full function caller ID platform which rivals many commercially available units. In addition, the CCID base software is responsible for transmitting the caller ID information to the cordless handset using the base unit's existing hardware, with some minor modifications (see schematic).

To achieve the functionality of a commercially available caller ID unit, the software must be able to quickly respond to three basic function buttons and display the results on an LCD display. The current CCID system can store up to ten calls in its EEPROM and display each of the ten calls at the user's request.

Pressing the "forward" button advances the display to the next record in the database. A depression of the "back" button scrolls the display to the previous call record in the database. Although all records in the database are stored in chronological order, the user may randomly delete records by pressing the "delete" button. For example, if there were one record from each of four months in the database, the user could delete the middle two records, thus leaving only the calls from the first and the fourth months in the database.

A fourth function, performed when the "xmit" button is pressed, differs from commercially available caller ID systems. When this button is pressed, the call record currently displayed on the LCD is transmitted to the cordless handset. This additional function is useful for testing the cleanliness of the cordless data channel as well as testing the functionality of the CCID system.

In achieving this functionality, the CCID base unit software performs a multitude of tasks. The major tasks performed by the software are listed as follows:

- scan and respond to function buttons
- check the phone line for a ring signal
- acquire the caller ID information by performing the FSK demodulation
- check the validity of the caller ID data
- decode and format the caller ID information
- display the caller ID information
- save the caller ID information to EEPROM
- transmit the name and phone number to the cordless handset
- administer the database contained within the EEPROM

In addition to the aforementioned, the software contains many error traps and error handling routines to help ensure that all failure modes are annunciated and dealt with properly. The external timer (the "soft watchdog") is monitored during open loop operations to ensure that the software doesn't get "stuck" while waiting for an event. Additionally, the IIC bus is monitored for proper function during all IIC operations. Although a failure of the IIC bus is fatal to the CCID software, an IIC error code is annunciated on the LCD display to aid the user in troubleshooting. It should be noted that IIC errors are extremely unlikely and are usually caused by open/short bus wires or the absence of the IIC device (the EEPROM is the only device on the IIC bus in this version of the CCID system). All non-IIC errors are not fatal to the CCID system. In the case of a non-IIC error, the software annunciates the error as if it were a call record. This error record informs the user that a non-fatal error occurred.

Verbal flowchart

The following text describes the operation of the CCID base unit software. The "verbal" flowchart is offered in lieu

of a “traditional” flowchart because such a flowchart would be very large and difficult to interpret. Refer to the software listing for program labels and detailed comments about registers, bits, and timing.

When the system is powered-up, the processor begins execution at ROM address #000h. At this time, the CCID system is initialized by placing the base unit in its “native” (unmodified) mode. The cordless phone operates normally when the phone is in its native mode. Then, the LCD display is initialized by calling the `init_display` subroutine, which allows about 15ms for the display and its subsystems to settle following power-up. The display is cleared and its interface characteristics are set. (see the HD44780 data sheet for more initialization information)

Following display initialization, the EEPROM is initialized using the `init_eeprom` subroutine. This subroutine determines if the EEPROM has been used under the current revision of CCID base software. If not, the `version_string` is written to the EEPROM and the record address offset table is initialized (this has the effect of clearing the EEPROM). If the EEPROM has been used under the current version of software, the number of the most recently stored record is returned to the main program and displayed by calling the `disp_record` subroutine.

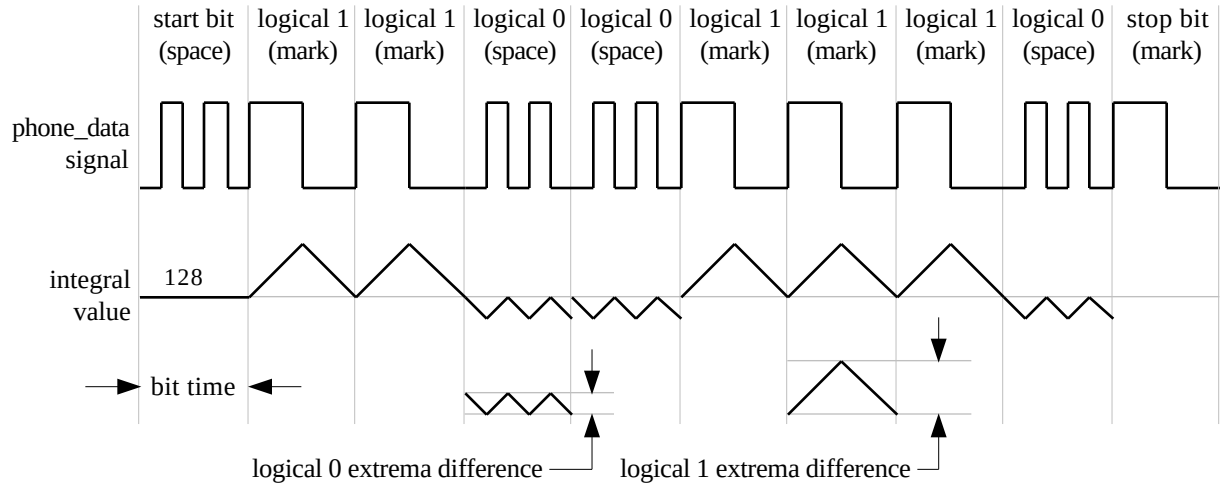
After the system initialization sequence, the program enters a loop which polls the function buttons and the `phone_data` port bit. If a function button closure is detected in this loop, the program branches to a routine which determines which button was pressed and responds with the appropriate function. If a signal at the `phone_data` port bit is detected, the software enters a pulse measuring routine which looks for the “ring signature” to determine if the phone is ringing. If the phone is not ringing, then the program clears the pulse duration counters and returns back to the beginning of the loop, thus starting the polling process over again.

After detecting a ring signature, the software enters a loop which looks for 100ms of dead time. The dead time indicates that the ring cycle has been completed and the caller ID information is to follow. When the 100ms of dead time is found, the software looks for 100 *consecutive* mark pulses from the phone line. Capturing 100 consecutive mark pulses from the phone line indicates to the software that the caller ID information is being transmitted. If this string of mark pulses is not found within the 1.3sec watchdog interval, the software returns to the `ring_check` label. This can occur if the phone goes “off hook” before the caller ID data is received or if the caller ID service is not available. Additionally, the absence of the 100 consecutive mark pulses may occur on ring signals after the first ring signal. This prevents the software from trying to acquire non-existent caller ID information on subsequent rings because the information is only transmitted after the first full ring cycle.

When the 100 consecutive mark pulses have been detected, the software begins to look for a start bit. The start bit is a space signal of 2200Hz. The software measures all pulse durations until it finds one of an acceptable space frequency. This is the first start bit of the data transmission. A “fatal CID error” is annunciated if a start bit is not detected within the watchdog interval. Again, this can occur if the phone goes off hook while the CID information is being transmitted.

After the software detects the first start bit, an independent bit clock is started. The internal 16 bit timer is used for this purpose. It is initialized with a value which will cause the timer flag to become asserted after the bit time has elapsed (833µs, in this case).

While the bit clock is running, the software integrates the signal found at the `phone_data` port bit. The value of the integral is initialized to 128 prior to executing the integration code. The integral will be incremented if `phone_data` is high and decremented if `phone_data` is low. Each iteration of the integration routine requires a constant amount of time, whether the `phone_data` port bit is high or low. The integration process is continued until the bit clock expires. The difference in integral extrema is used to make the distinction between a logical 0 and a logical 1. The integral value is reset to 128 at the beginning of each bit. Each bit is rolled onto a data byte until all eight bits of the byte have been received. In this way, the caller ID FSK demodulation can be accomplished in software. The following chart illustrates a typical FSK byte acquisition sequence.



To distinguish between zeros and ones, the software uses the difference in integral extrema (i.e. the maximum value minus the minimum value). This method of FSK demodulation provides fairly high noise margins in that a mis-synchronized bit clock will not contribute significantly to the outcome. In other words, the bit clock may be off by a few tens of microseconds while still allowing the software to reliably distinguish between 0 and 1. Additionally, fast spikes on `phone_data` will not contribute significantly to the integral value because the integral is essentially a lowpass filtered version of the input signal.

The stop bit is ignored to allow time for the software to administer the checksum value and prepare for the next integration series. Also, the caller ID specification states that up to ten stop bits may be inserted between data bytes. This specification makes it unreliable to assume that each data byte will be terminated by exactly one stop bit. For this reason, the software ignores the stop bit and loops back to looking for the next start bit.

After the caller ID transmission is complete, the software uses the checksum value to determine the validity of the data. If the checksums do not match, then an error annunciator (the epsilon character) is sent to the LCD display. Otherwise, the software begins to process the received caller ID data.

In processing the data, the software first converts the ASCII numeric representation of the month (i.e. 09 for September, 11 for November) to its three letter equivalent (i.e. Sep for September, Nov for November) and sends it to the fourth line of the LCD display. The date is processed and printed after the three letter month abbreviation. The time of day is then converted from ASCII military format (i.e. 11:08pm = 2308) to standard 12 hour format. The hours and minutes are separated by a colon (:) and printed to the fourth line of the display. The “am” or “pm” indicator is printed directly after the time.

The subroutines `validate_num` and `validate_char` are used throughout the display processing sequence to ensure that the software is using valid data. If the data is determined to be invalid, the software replaces the invalid character with an asterisk (*) or the invalid numeral with a question mark (?). These subroutines take advantage of the fact that the caller ID service can only transmit numerals and upper case letters - all other characters are invalid and generally indicate that the byte was somehow corrupted during acquisition. This is yet another feature which differentiates the CCID system from other caller ID systems. A typical commercially available caller ID system will not even attempt to work with corrupted data. If one of these units receives a checksum error, a “data error” message is displayed and the caller ID data is not decoded. In contrast, the CCID system will display all valid data and replace invalid data with fixed “wild card” characters. For example, suppose that the fictional company “Rocky Mountain Engineering” calls and some of the data gets corrupted during acquisition. The directory listing, which will be exactly 15 characters in length, would be something like “ROCKY MNT ENG ”. Also suppose that a few characters were corrupted. Some of the corrupted characters may be valid ASCII characters and some may not. A “conservative example” may look something like this on the CCID display: `R*CK* MMT *NG`. Although the data is quite corrupt, a human can “read-in” the meaning and have a pretty good idea who is calling. Some commercially

available systems would print “data error” and one would have no idea who is calling.

So, after the time and date are decoded and displayed, the software determines if the directory listing (the name and phone number of the calling party) is available. If the call is “blocked” by the calling party, the software branches to a “call blocked” routine which displays the `blocked_message` on the display. If the caller ID information is not available (i.e. “out of range”), the software will display the “out of area” on the second line of the display.

If the caller directory listing is available, the software will fetch it from RAM, validate all of the characters, format it, and send it to the LCD display. The LCD display now contains the month, date, and time on the fourth line, the caller’s phone number on the third line, and the caller’s name on the second line. The first line is reserved for indicating which record in the EEPROM database is currently displayed. See the enclosed photographs for typical formatted displays.

Now that the display contains all of the decoded and formatted information, the software proceeds to fetch the formatted data from the display controller’s RAM and store it in EEPROM. There are three lines of sixteen characters which contain pertinent caller ID information. All of these 48 characters are written to EEPROM to avoid the need to reformat them when they are retrieved. This method of storage, although somewhat inefficient, minimizes the amount of software overhead needed to administer the data.

The EEPROM is a 24C04 device which is capable of storing 512 eight bit bytes, split into two 256 byte pages. The EEPROM is interfaced to the host via the IIC bus. The software contains several IIC interface subroutines, all of which were developed, validated, and tested “off line” (i.e. out of the CCID base software). Reference Philips Semiconductor’s application note AN422 for detailed information about the IIC bus and the usage of the ‘752’s IIC capabilities.

After storing the formatted data in the EEPROM, the software enters the `send_cordless` section of code. This code segment puts the cordless phone’s base unit into “data transmit” mode by inhibiting its native data and replacing it with the CCID system data (see base unit schematic). The software begins the data transmission by sending five seconds worth of 20Hz pulses to the `xmit_data` port bit. This signal is used to “wake-up” the cordless handset and power-up the remote unit’s +5V power supply (see cordless handset hardware description). After the five seconds of 20Hz has been transmitted, the base unit software sends 33.3ms of 1.5KHz to synchronize the receiver to the pending incoming data. The 1.5KHz signal is followed by a “dummy” byte which is used to increase the noise immunity of the cordless data channel.

The cordless data is transmitted at 300BPS using FSK. The mark frequency (logical 1) is 300Hz and the space frequency (logical 0) is 900Hz. Although these frequencies were chosen somewhat arbitrarily, their values provide enough difference to allow the handset unit to easily distinguish between 0 and 1, even though it only runs at 6HMz. Each data byte is preceded by a start bit (space) and terminated with a stop bit (mark). Frequencies and data rate aside, this method is identical to the caller ID format, which makes it possible to reuse the base unit’s core demodulation software in the handset unit. The data bytes are sent by calling the `xmit_data_byte` subroutine for each data byte to be transmitted. The software finishes the transmission by sending a checksum word to the cordless handset.

The data transmission is achieved using the PWM output of the ‘752. This provides a low overhead method of generating the relatively low FSK frequencies without using multi-byte counters. The prescaler value is set for each frequency and the PWM does the rest. Additionally, the PWM output is in a totem pole configuration, which enables it to symmetrically source and sink current. This provides the additional benefit of maintaining a 50% duty cycle on the transmit data signal even though it is capacitively coupled to the base unit’s transmit pin.

The software returns to the beginning (`ring_check`) when the cordless transmission is complete.

Optional document

CCID Handset Unit Software Description

CCID Handset Software Description

Introduction & functional definition

The purpose of the CCID handset software is to receive the caller ID data from the base unit, process it, and display it on an LCD display. The applicable LCD display routines were copied, almost verbatim, from the base unit software. This is made possible by using displays which are controlled by the HD44780, which provides a standard interface between different display configurations. The FSK demodulation software used on the base unit is also used on the handset unit with a few modifications for data rate and FSK frequencies.

The major functional areas of the CCID handset unit software are outlined as follows:

- after power-up, latch-on the +5V power supply
- alert the user to the incoming data
- receive the incoming data from the cordless channel
- check the received data for validity (checksum)
- display the data on the LCD
- wait for 5 seconds (or until the phone stops ringing) before shutting down

As with the base unit software, the handset software has several error traps and error handling routines. The integrity of the received data is checked against a checksum and all ASCII characters are validated prior to being displayed. The handset software also alerts the user if no data was received or if the data is too corrupt to be worth displaying - each of which can happen if the cordless channel is too noisy for reliable data transmission.

Verbal flowchart

The following text describes the operation of the CCID handset software. The "verbal" flowchart is offered in lieu of a "traditional" flowchart because such a flowchart would be very large and difficult to interpret. Refer to the software listing for program labels and detailed comments about registers, bits, and timing.

When the system is powered-up, the processor begins execution at ROM address #000h. At this time, the handset's +5V power supply is latched-on by the first instruction at the `start` label. This ensures that the power supply will not "accidentally" turn off while the '750 and display need it. Once latched, the power supply can *only* be turned off by the '750.

After latching the +5V power supply, the software enables external interrupt INT0\. This function will be discussed later. Following the interrupt enable, the software initializes the two line LCD display using the `init_display` subroutine. This subroutine functions identically to that of the base unit. (the data port and control signals which drive the display are even connected to the same port as on the base unit!)

Once the display is initialized, the software flashes the handset `version_string` and a "waiting for data" message on the display. Meanwhile, the software waits for a maximum of about eight seconds while it looks for the 1.5KHz synchronization signal. If the synchronization signal is not found within the allotted eight seconds, then the program will branch to a "no data received" annunciation routine.

When the software finally detects the synchronization signal, it waits for a start bit in much the same manner as the base unit. The start bit is the first bit in the "dummy" byte which is sent by the base unit. This "dummy" byte is ignored (even though its value is zero) and does not contribute to the running checksum of the incoming data.

The cordless FSK data is acquired using the same integration scheme as the base unit uses on the caller ID data (see the base unit software description). The software algorithms for the FSK demodulation were copied directly from the

base unit and modified to account for the 6MHz clock of the ‘750 and different FSK frequencies and bit times. The integration concepts are identical.

It is known, a priori, that the base unit will transmit exactly 33 bytes of data. 32 of these bytes represent the middle two lines of the base unit’s LCD display and the 33rd byte is a checksum for the transmitted data. If the handset software does not receive more than 16 bytes of data, the “fatal error” message will be displayed. This can happen if the cordless channel suddenly becomes very noisy (i.e. static). If more than 16 bytes were received, the software will attempt to process and display the data, even though it may not be complete.

If the calculated checksum does not match the received checksum, the handset software will display the epsilon character in the second-to-rightmost position on the second line of the display. This informs the user that a checksum error occurred during the cordless transmission. It should be noted that the epsilon character from the base unit (if present) resides in the rightmost position of the second line. This allows the user to determine if an error occurred during the cordless transmission and/or during the original caller ID acquisition performed by the base unit.

The remainder of the data is validated by the `validate_char` subroutine before it is printed on the display. This subroutine is a modified version of the base unit software. Lowercase letters, parenthesis, dashes, asterisks, and the epsilon character are considered to be valid characters since the base unit can transmit them. An invalid character is replaced by a “box character” to inform the user that the character was corrupted during the cordless transmission.

When all 32 characters have been sent to the display, the software enters a wait loop which allows five seconds for the user to view the data. If, however, the phone is still ringing, the five second delay is reset. In other words, the user is allowed five seconds after the phone stops ringing to view the data. Most users will answer the phone (or elect not to) while the phone is still ringing, so the five second delay is adequate.

After the phone stops ringing and the five second delay is complete, the software enters the `power_down` code segment. At this point, the +5V power supply is turned-off and the oscillator is stopped (the ‘750 is put into “power down” mode). Stopping the oscillator serves a dual purpose: (1) it minimizes the load on the handset’s battery, and perhaps most importantly, (2) it prevents the processor from executing “random” code while its power supply decays (i.e. preventing latch-up). The handset system is now “off” and waiting for the next transmission.

As mentioned earlier, the external interrupt INT0\ is enabled at power-up. This hardware interrupt pin is connected to the “talk” button on the cordless handset. The interrupt is configured in the edge triggered mode so that edges on the talk button signal will activate the interrupt. The `talk_button` signal (see schematic) is normally low and oscillates when the talk button is pressed, thus creating edges on which the interrupt can be activated. The interrupt service routine for INT0\ immediately causes the program to jump to the `power_down` routine. Because this interrupt remains enabled for the duration of software execution, the user has the option to immediately answer the phone and cause the ‘750 to power itself down, thus allowing normal operation of the cordless phone. To summarize, the cordless handset system will immediately power down if the phone is answered any time during software execution.

Optional document

CCID Bill of Materials and Cost Estimate

CCID Bill of Materials and Cost Estimate

CCID Base Schematic v2.6:

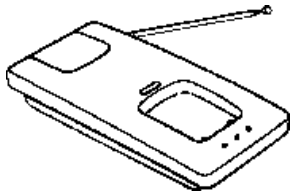
Quantity	Reference	Description	Unit cost	Subtotal
2	C1, C2	Capacitor, .1µF, 200V, 20%	$0.50	$1.00
1	C3	Capacitor, 22µF, 16V, tantalum	$0.10	$0.10
2	C4, C5	Capacitor, 33pF, 50V, 20%	$0.05	$0.10
1	C6	Capacitor, 2.2µF, 16V, tantalum	$0.10	$0.10
2	C7, C8	Capacitor, 1µF, 50V, 20%	$0.05	$0.10
1	C9	Capacitor, .01µF, 50V, 20%	$0.05	$0.05
1	CN1	Connector, dual row, right angle, modified	$0.10	$0.10
1	CN2	Connector, dual row, right angle, modified	$0.10	$0.10
6	CR1, CR2, CR4, CR4, CR5, CR6	Diode, general purpose, 1N4148	$0.01	$0.06
1	MOV1	Varistor, metal oxide, 130VRMS	$1.00	$1.00
2	Q1, Q2	FET, N channel, 1A, 60V, VN10KM	$0.20	$0.40
11	R1, R2, R3, R5, R8, R9, R10, R11, R14, R15, R22	Resistor, 10KΩ, 1/4W, 10%	$0.01	$0.11
1	R12	Resistor, 47KΩ, 1/4W, 10%	$0.01	$0.01
3	R13, R16, R21	Resistor, 1KΩ, 1/4W, 10%	$0.01	$0.03
1	R17	Resistor, 10Ω, 1/4W, 10%	$0.01	$0.01
1	R22	Potentiometer, 10KΩ, multiturn	$0.35	$0.35
2	R4, R18	Resistor, 1MΩ, 1/4W, 10%	$0.01	$0.02
4	R6, R7, R19, R20	Resistor, 100KΩ, 1/4W, 10%	$0.01	$0.04
4	SW1, SW2, SW3, SW4	Switch, pushbutton, momentary, N.O.	$0.25	$1.00
1	U1	Comparator, quad, O.C., LM339	$0.15	$0.15
1	U2	Microcontroller, 87C752, programmed	$3.00	$3.00
1	U3	EEPROM, 512 byte, IIC, 24C04	$1.50	$1.50
1	U4	Display, LCD, 4 line by 16 character	$5.00	$5.00
1	X1	Crystal, 12MHz	$1.00	$1.00
		Base unit total:		$15.33

CCID Handset Schematic v1.3:

Quantity	Reference	Description	Unit cost	Subtotal
2	C1, C2	Capacitor, .1µF, 50V, 20%	$0.05	$0.10
1	C3	Capacitor, 22µF, 16V, tantalum	$0.10	$0.10
1	C4	Capacitor, 100µF, 16V, electrolytic	$0.35	$0.35
1	C5	Capacitor, 2.2µF, 16V, tantalum	$0.10	$0.10
2	C6, C7	Capacitor, 33pF, 50V, 20%	$0.05	$0.10
1	C8	Capacitor, 1000pF, 50V, 20%	$0.05	$0.05
4	CR1, CR2, CR3, CR4	Diode, general purpose, 1N4148	$0.01	$0.04
1	L1	Inductor, 22µH	$0.50	$0.50
2	Q1, Q2	FET, N channel, 1A, 60V, VN10KM	$0.20	$0.40
1	R10	Resistor, 180KΩ, 1/4W, 10%	$0.01	$0.01
1	R11	Potentiometer, 10KΩ, multiturn	$0.35	$0.35
4	R2, R2, R5, R8	Resistor, 1MΩ, 1/4W, 10%	$0.01	$0.04
1	R3	Resistor, 1KΩ, 1/4W, 10%	$0.01	$0.01
1	R4	Resistor, 150KΩ, 1/4W, 10%	$0.01	$0.01
2	R6, R9	Resistor, 100KΩ, 1/4W, 10%	$0.01	$0.02
1	R7	Resistor, 2.2KΩ, 1/4W, 10%	$0.01	$0.01
1	U1	Comparator, quad, O.C., micropower, LP339	$0.50	$0.50
1	U2	DC-DC converter, MAX877	$2.50	$2.50
1	U2	Microcontroller, 87C750, programmed	$1.50	$1.50
1	U4	Display, LCD, 2 line by 16 character	$3.50	$3.50
1	X1	Crystal, 6MHz	$1.00	$1.00
		Handset unit total:		$11.19

Required document

Block diagram & Schematics



Cordless channel

AC wall transformer

Base unit's +5V supply

Phone line

Isolation

Function buttons

87C752

CCID v1.5

Cordless transmit hardware

EEPROM

4 x 16 LCD display

CCID Base Unit

Cordless receive hardware

4V phone battery

+5V supply

Handset buttons

87C750

CCID v1.3

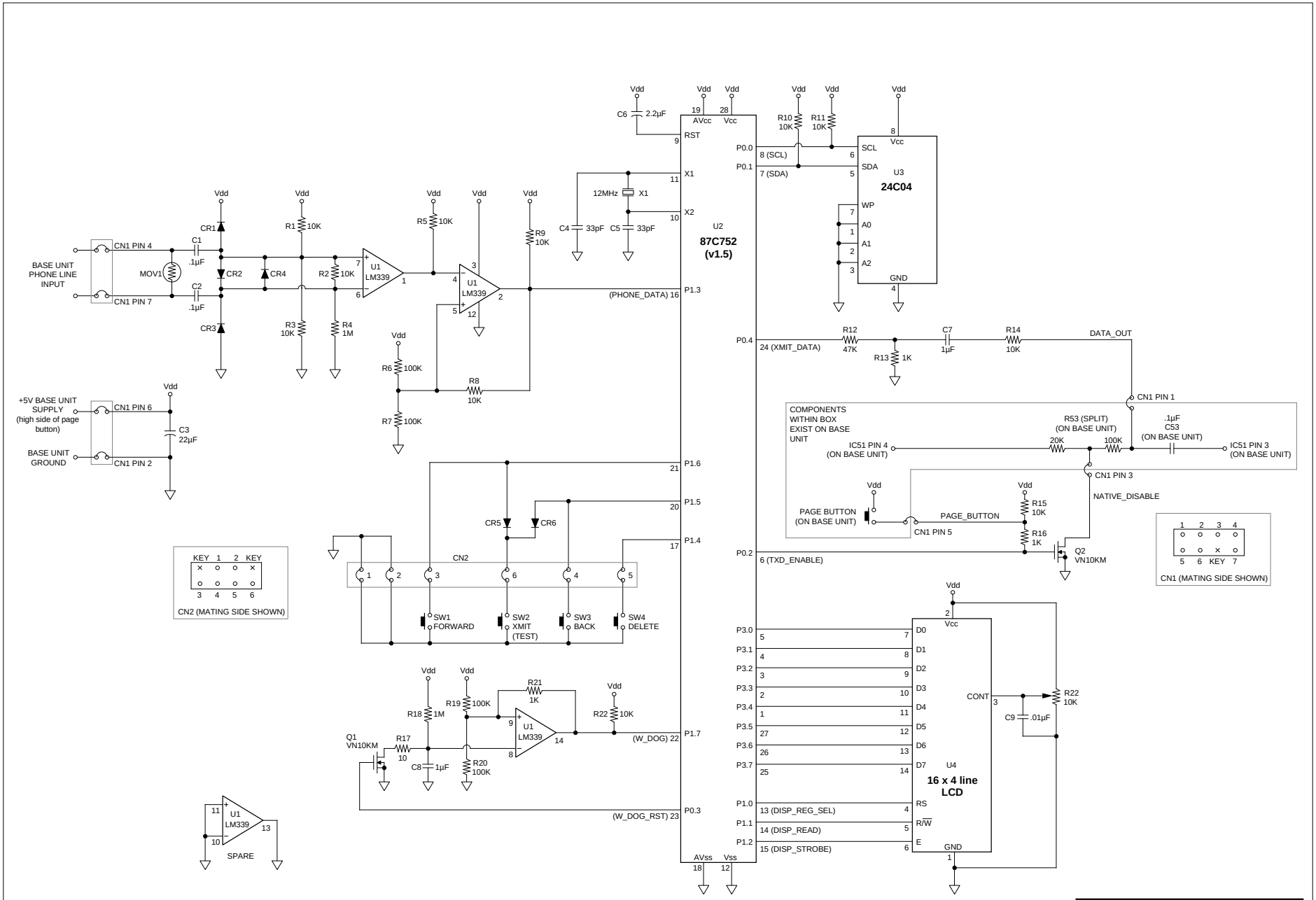
2 x 16 LCD display

CCID Handset Unit

CCID SYSTEM BLOCK DIAGRAM

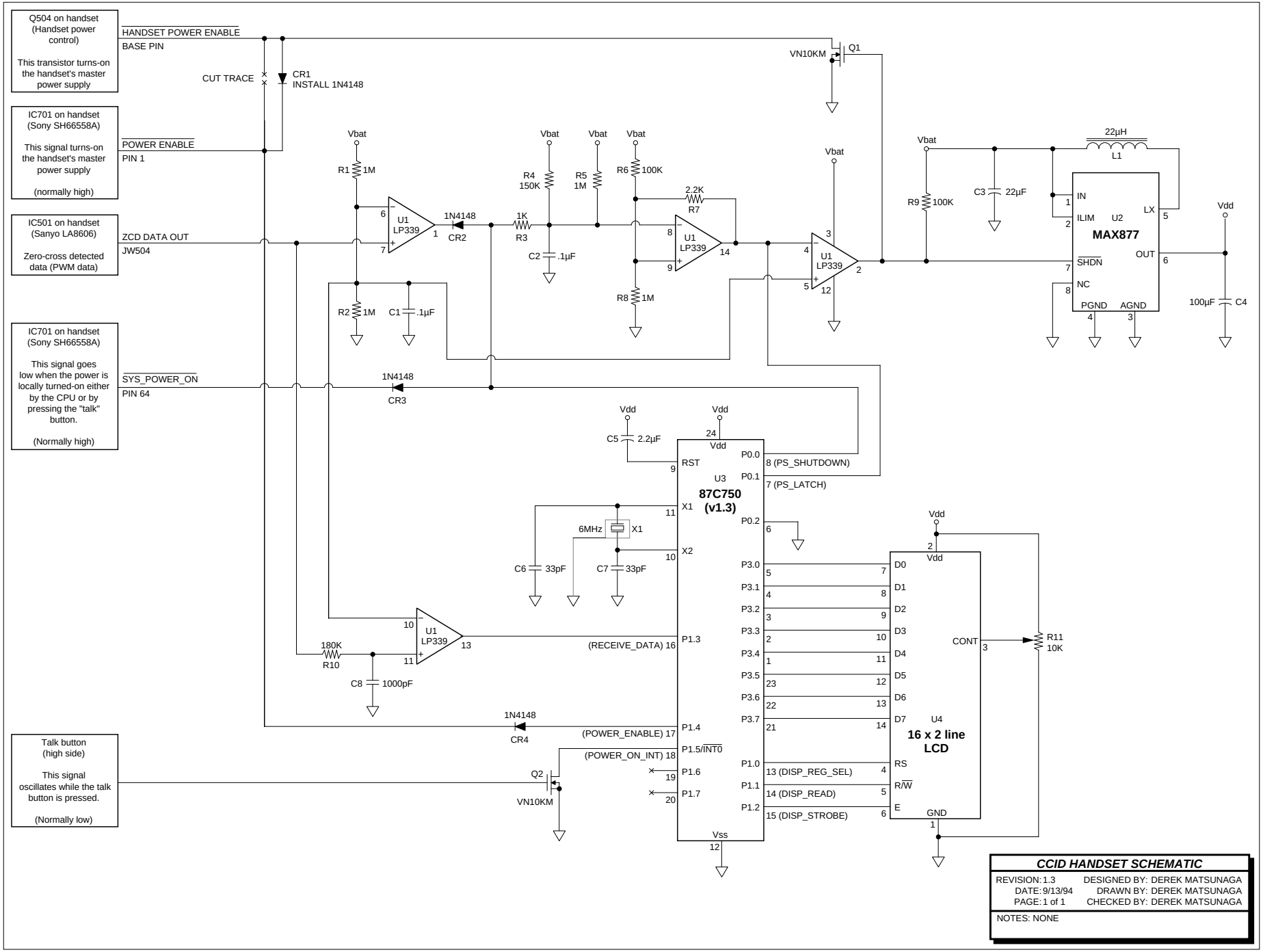
REVISION:	1.0	DESIGNED BY:	DEREK MATSUNAGA
DATE:	9/28/94	DRAWN BY:	DEREK MATSUNAGA
PAGE:	1 of 1	CHECKED BY:	DEREK MATSUNAGA

NOTES: NONE



CCID BASE SCHEMATIC	
REVISION: 2.6	DESIGNED BY: DEREK MATSUNAGA
DATE: 9/1/94	DRAWN BY: DEREK MATSUNAGA
PAGE: 1 of 1	CHECKED BY: DEREK MATSUNAGA

NOTES: ALL DIODES 1N4148.


Q504 on handset (Handset power control)
This transistor turns-on the handset's master power supply
HANDSET POWER ENABLE
BASE PIN
CUT TRACE
CR1
INSTALL 1N4148
VN10KM
Q1
IC701 on handset (Sony SH66558A)
This signal turns-on the handset's master power supply
(normally high)
POWER ENABLE
PIN 1
IC501 on handset (Sanyo LA8606)
Zero-cross detected data (PWM data)
ZCD DATA OUT
JW504
Vbat
R1 1M
R2 1M
C1 .1µF
U1 LP339
1N4148
CR2
R4 150K
R5 1M
R6 100K
R7 2.2K
R3 1K
C2 .1µF
R8 1M
R9 100K
C3 22µF
22µH
L1
MAX877
IN
ILIM
LX
SHDN
OUT
NC
PGND
AGND
Vdd
100µF C4
IC701 on handset (Sony SH66558A)
This signal goes low when the power is locally turned-on either by the CPU or by pressing the "talk" button.
(Normally high)
SYS_POWER_ON
PIN 64
1N4148
CR3
C5 2.2µF
U3
87C750 (v1.3)
RST
X1
X2
6MHz
C6 33pF
C7 33pF
P0.0
8 (PS_SHUTDOWN)
P0.1
7 (PS_LATCH)
P0.2
P1.3
(RECEIVE_DATA) 16
180K R10
C8 1000pF
1N4148
CR4
(POWER_ENABLE) 17
P1.4
P1.5/INT0
(POWER_ON_INT) 18
P1.6
P1.7
Talk button (high side)
This signal oscillates while the talk button is pressed.
(Normally low)
Q2
VN10KM
P3.0
P3.1
P3.2
P3.3
P3.4
P3.5
P3.6
P3.7
P1.0
13 (DISP_REG_SEL)
P1.1
14 (DISP_READ)
P1.2
15 (DISP_STROBE)
Vss
U4
16 x 2 line LCD
D0
D1
D2
D3
D4
D5
D6
D7
RS
R/W
E
GND
CONT
R11 10K
CCID HANDSET SCHEMATIC
REVISION: 1.3
DATE: 9/13/94
PAGE: 1 of 1
DESIGNED BY: DEREK MATSUNAGA
DRAWN BY: DEREK MATSUNAGA
CHECKED BY: DEREK MATSUNAGA
NOTES: NONE

Optional document

System Photographs

Photo #1: The completed CCID system prototype. (from left - handset unit, base unit)

Photo #2: The CCID modifications preserve the mechanical functionality of the phone. The unit can still be wall mounted.

Photo #3: A close-up photograph of the base unit's display shows a typical call record. Note the function buttons which flank the display.

Photo #4: A close-up photograph of the handset unit's display shows a typical call transmission.

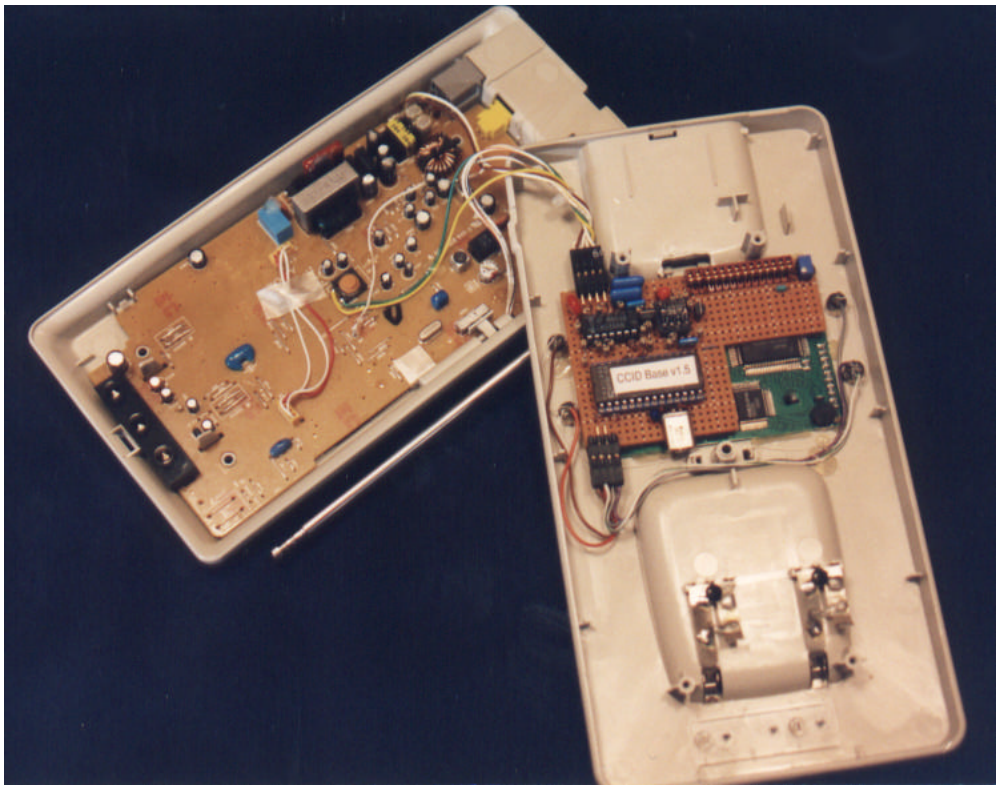


Photo #5: The CCID base unit circuitry is contained on a small perf board. Note the connections to the bottom half of the unit.

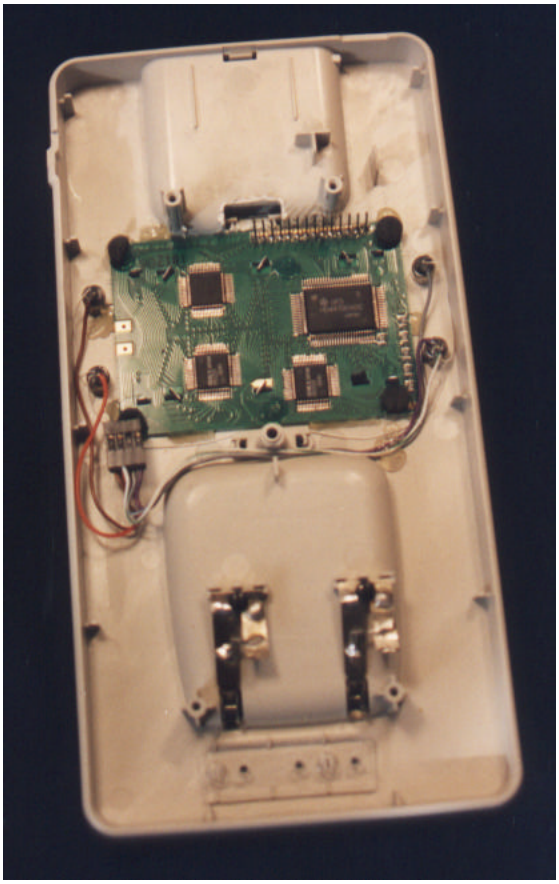


Photo #6: The under side of the top half of the base unit shows the rear of the 4 line by 16 character display and the four function buttons.

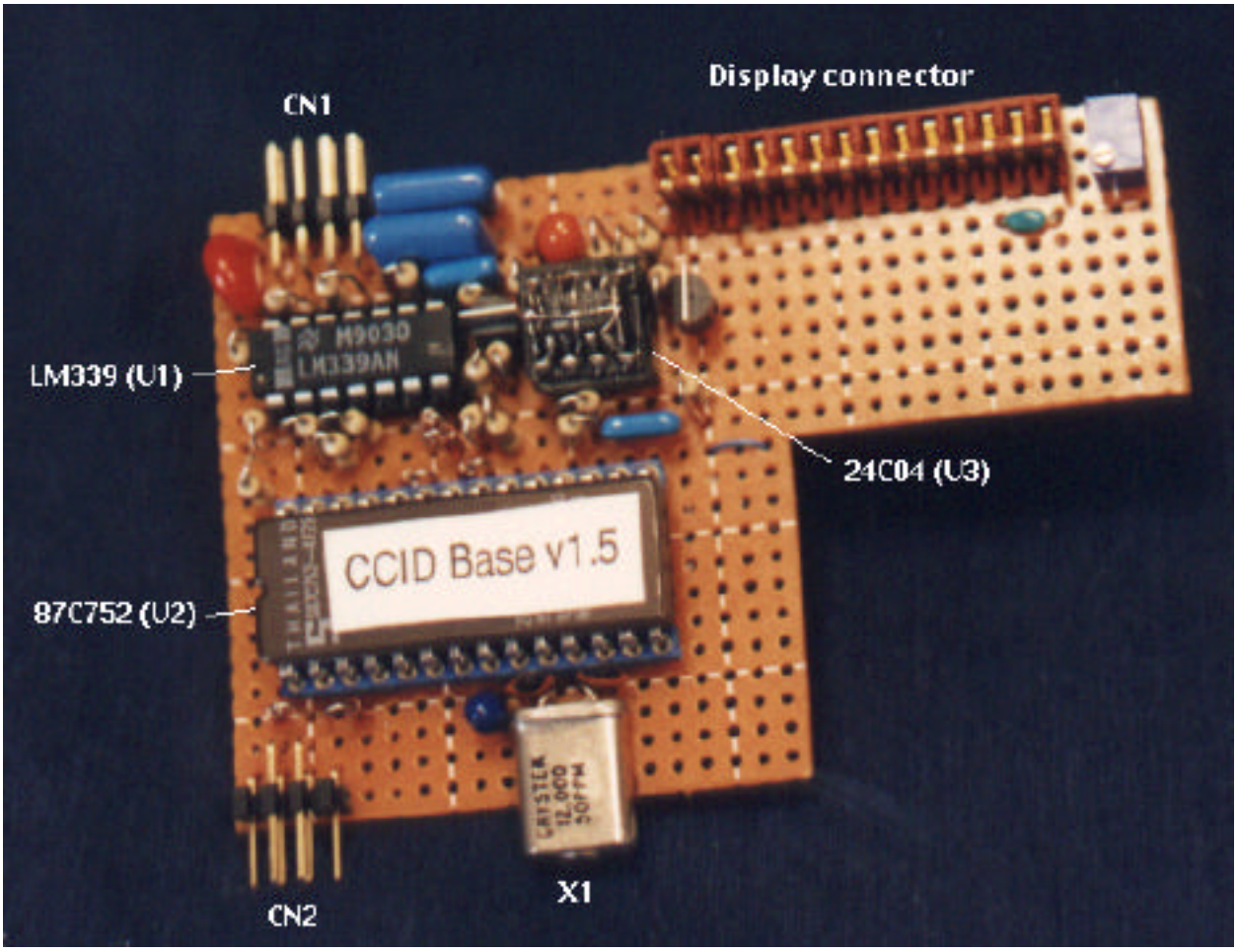


Photo #7: The perf board which contains the base unit circuitry.

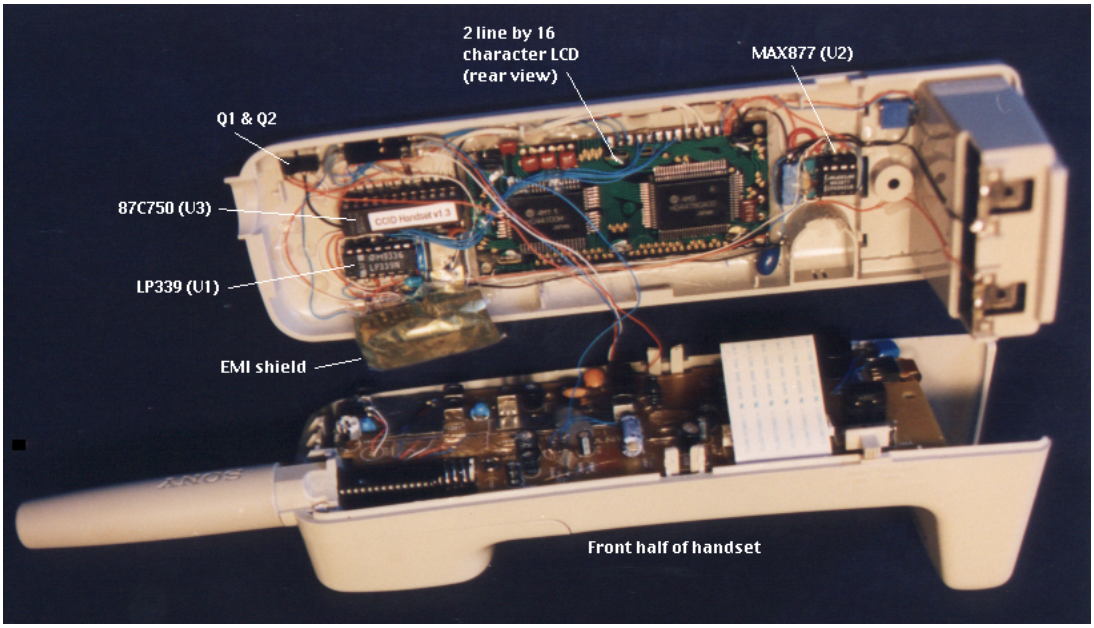


Photo #8: An inside view of the handset unit shows the extreme lack of space.
All CCID components are glued to the handset chassis.
There is no room for perf board or a through hole PCB.

Optional document

Software Validation and Testing Summary

CCID Software Validation and Testing Summary

Introduction

The CCID software plays a crucial role in the functionality of the system. The CCID system would not function without either the base unit software or the handset unit software. For this reason, it is important to develop reliable software for both processors.

Fortunately, both microcontrollers are from the 8XC75X family. They share many common attributes including I/O ports and instruction set. The software for both the base unit and the handset was written entirely in assembly language. This language was chosen because of the ROM and RAM limitations associated with the '750 and '752. It is not practical to push the entire stack into RAM every time a function is called, as some high level language compilers may do. Additionally, *absolute* control over the instruction execution time is necessary to perform the FSK demodulation. The Metalink assembler was downloaded from the Philips BBS to accomplish the software development tasks.

Every single line of assembly code for this project is commented with the number of machine cycles required for instruction execution and a brief, but informative, description of what the instruction is doing. This in itself serves as a form of "real time"software validation. As mentioned in the Project Summary, there were no "accidents" in developing the software - the use of each instruction was purposely engineered to perform a specific task. Both source code files are heavily commented with timing and execution information.

All of the software was developed and tested using the "burn and learn" method. That is, each software revision was assembled and programmed into the respective parts. Each part had to be UV erased prior to being reprogrammed. Emulators and debuggers were not used due to lack of availability (I don't own any). Additionally, an emulator would ground reference the phone line and produce an unmanageable 60Hz component on the phone_data pin. The relatively long device erasure cycles provide additional incentive to design the software correctly on the first iteration (there was no "hacking" development in this project)!

Testing summary

The following list highlights the testing which was performed on selected software modules.

- The ring detection (ring signature) software was independently tested by calling my own phone number from various sources such as pay phones, my office phone, and cellular phone.

- The Caller ID setup signal detection software was extensively tested in the same way.

- The FSK demodulation software was debugged and tested by activating the FSK demodulation software during caller ID transmission and observing the results on an eight bit LED display - stepping through RAM one location at a time.

- All display driver routines were tested out of the CCID system by writing special test software which would exercise the display functions, display fixed data, and read-back the fixed data into RAM. RAM was subsequently investigated using the 8 bit LEDs and stepping one location at a time. These tests were performed with 6MHz and 12MHz crystals.

- All IIC routines were tested out of the CCID system using special test software which would use the byte write, page write, byte read, and page read modes of the EEPROM. The contents of the EEPROM were observed and verified using a device programmer. The

device programmer was also used to seed the EEPROM with incorrect version_strings and erroneous address offset tables.

- All IIC error traps were tested by manually creating error conditions. The SDA and SCL pins were both disconnected. The entire EEPROM was removed from the circuit. The SDA and SCL lines were tied to ground.

- The data acquisition error traps were tested by injecting noise on the phone_data pin and disconnecting the phone line in the middle of a transmission.

- The display formatting routines were tested by seeding RAM with erroneous and illegal values (i.e. a month number of 25 and a time number of 100).

- The cordless data transmission software was debugged and tested using a hard wire test fixture between the base unit and the handset unit. Various error conditions were intentionally seeded and the results were observed. Among the error conditions tested are a noisy data channel and a non-existent data channel.

- The cordless data acquisition routines in the handset software were tested under conditions similar to those of the base unit's caller ID acquisition software.

- The handset's power management software was tested using special code which would cycle the power supply and respond to the "talk button" interrupt.

Software design methodology highlights

- The state of the carry bit is known (or known to be irrelevant) prior to all mathematical operations.

- Multiply defined registers are not used in any one section of code, including subroutines.

- All modules, code segments, and subroutines were tested "off line" prior to using them in this project.

- The number of machine cycles required by each instruction is known and accounted for.

- Adequate RAM is reserved for the stack to allow enough nested subroutine calls before the stack collides with data RAM.

- Through the use of timers and the hardware "watchdog", it is virtually impossible for the software to get "stuck" while waiting for an event.

- Built-in features such as IIC and PWM are used whenever possible

- Every assembly language line requires a comment which includes the number of required machine cycles.

- The software design philosphy approaches that of a medical device, although the CCID application is intended to be a cheap consumer product.

Required document

CCID Base Unit Software Listing

```
;Title:                Cordless Caller ID base unit software
;Author:               Derek Matsunaga
;Start date:           7/1/94

;Revision:             1.5
;Revision date:        9/25/94

;Target processor:     87C752
;Assembler:            Metalink v1.2i
;Hardware platform:    CCID base schematic v2.3

;Comment format:       A comment is required for each line of assembly code.  The "comment field" begins at the
;                      third tab stop from the left margin.  The first character in the comment field should be
;                      the number of machine cycles required to execute the instruction.  Following the number
;                      of machine cycles is another tab and the verbal comments which explain the function of
;                      the assembly instruction.  Comments are typically one sentence (or fragment) which begin
;                      with an uppercase letter and terminated with a period.  Comments which require more
;                      than one line may be continued at the same tab stop on the next line.

$MOD752                                                 ;Inform assembler of the target processor.

;=============================================================================================================
;Register definitions -------------------------------------------------------------------------------------

;Each of the eight internal registers (R0 through R7) are given relevant names to make the software more
;understandable.  Each register may be assigned more than one name so that its name is meaningful to the section
;of code in which it is used.  This is a "double edged sword" in that it makes the software more readable while
;it makes changes dangerous.  The name(s) of each register have been engineered such that they are not used
;for multiple purposes in any one section of code.  For this reason, one must exercise extreme caution when
;adding another name to a register definition or changing the register assignments altogether (i.e. swapping
;R0 and R3).

original_value        equ    r0                         ;Used to store the original ASCII value in the
                                                        ;validate_char subroutine.
trans                 equ    r0                         ;Used to keep track of port bit transitions.
temp                  equ    r0                         ;Used as a general purpose temporary register.
ring_count            equ    r0                         ;Used to count ring pulses from the phone line.

ram_ptr               equ    r1                         ;Used to point to a specific location in internal RAM.

max_value             equ    r2                         ;Used to keep track the FSK integral maxima.
ls_digit              equ    r2                         ;Used to store the value of a least significant decimal
                                                        ;digit.
low_eeprom_addr       equ    r2                         ;Used to store the low byte of the EEPROM address.
low_count_l           equ    r2                         ;Used to store the low byte of a 16 bit counter.

min_value             equ    r3                         ;Used to keep track the FSK integral minima.
ms_digit              equ    r3                         ;Used to store the value of a most significant decimal
                                                        ;digit.
high_eeprom_addr      equ    r3                         ;Used to store the high byte of the EEPROM address.
low_count_h           equ    r3                         ;Used to store the high byte of a 16 bit counter.

bit_count             equ    r4                         ;Used to count the number of received bits.
iic_error_code        equ    r4                         ;Used to store IIC error codes.

integral              equ    r5                         ;Used to store the FSK integral value.
mark_count            equ    r5                         ;Used to count mark frequency pulses.
am_pm_byte            equ    r5                         ;Used to store ASCII "a" or ASCII "m" to append to the
                                                        ;time.
record_number         equ    r5                         ;Used to store the EEPROM record number.

data_byte             equ    r6                         ;Used to store the data byte currently being received.
temp_aux              equ    r6                         ;Used as an auxiliary general purpose temporary register.

byte_count            equ    r7                         ;Used to store the number of bytes received.
```

```
;Additional equates and bit definitions ----------------------------------------------------------------------

checksum              equ   dph                     ;Used to store a checksum value.
line_count            equ   dpl                     ;Used to store the number of line transmitted to the
                                                    ;cordless handset.

disp_port             equ   p3                      ;The display data port (see schematic).
disp_reg_sel          bit   p1.0                    ;The display register select bit (see schematic).
disp_read             bit   p1.1                    ;The display read/write (see schematic).
disp_strobe           bit   p1.2                    ;The display data latch bit (see schematic).

data_port             equ   p1                      ;The port at which all inputs may be read (see schematic).

delete_key            bit   p1.4                    ;The delete button port bit (see schematic).
back_key              bit   p1.5                    ;The back button port bit (see schematic).
forward_key           bit   p1.6                    ;The forward button port bit (see schematic).
phone_data            bit   p1.3                    ;The bit at which the squared-up phone line signal is read.
w_dog_rst             bit   p0.3                    ;The reset watchdog timer bit.
w_dog                 bit   p1.7                    ;The watchdog timer bit.
xmit_data             bit   p0.4                    ;The transmit data bit (to the cordless transmitter).
txd_enable            bit   p0.2                    ;The bit that presses and holds the page button on the
                                                    ;base unit.


;Constant definitions ----------------------------------------------------------------------------------------

data_ram              equ   015h                    ;The beginning of data RAM.  This value was chosen to allow
                                                    ;up to three nested subroutine calls before the stack
                                                    ;collides with data RAM.

key_mask              equ   01110000b               ;ANDed with the data_port to remove everything but the
                                                    ;function buttons.
phone_mask            equ   00001000b               ;ANDed with the data_port to remove everything but the
                                                    ;phone_data bit.
eeprom_dev_addr       equ   10100000b               ;The device address of the EEPROM.
iic_bus_request       equ   01110000b               ;A constant used to request mastership of the IIC bus.
i2con_clear           equ   00011100b               ;A constant used to clear the i2con register.
iic_receive           equ   10111100b               ;Used to put the IIC master into receive mode.
iic_release           equ   10111101b               ;Used to release control of the IIC bus.
iic_stop              equ   00100001b               ;Used to send an IIC stop condition.
iic_clr_drdy          equ   00100000b               ;Used to clear IIC data ready.
iic_start             equ   00100010b               ;Used to issue an IIC start condition.
disp_record_addr      equ   020h                    ;An off-screen display location residing in the display's
                                                    ;DD RAM used to store the number of the currently displayed
                                                    ;record.
timer_start           equ   00010000b               ;Used with TCON to start the internal timer.
timer_stop            equ   00000000b               ;Used with TCON to stop the internal timer.
;=================================================================================================================


;Power-on (hardware) reset routine ---------------------------------------------------------------------------

;When the CPU power-up, the program counter is loaded with the reset vector and execution begins at address
;#000h.  The beginning of the "main" program actually start here.  The display and EEPROM are initialized
;and the most recent call record is displayed.  This code is located here to make better use of program ROM
;space.  In other words, these few lines of code fill the gap between the reset location and the timer I
;interrupt location.  After displaying the most recent call record, the program initializes the ring counters
;and jumps to the ring_loop, skipping-over the timer I ISR.

                      org   000h                    ;   Reset location.
start:                mov   tcon,#timer_stop        ;2  Clear the timer control location.
                      clr   txd_enable              ;1  Allow the phone base unit to operate normally.
                      mov   ie,#000h                ;2  Disable all interrupts.
                      clr   xmit_data               ;1  Set the data output low.
                      acall init_display            ;2  Initialize and clear the LCD display.
```

```
                    acall  init_eeprom                ;2  Initialize the EEPROM (if necessary) and get the
                                                      ;   number of stored records into the accumulator.
                    acall  disp_record                ;2  Display the latest record (whose number is in the
                                                      ;   accumulator from calling init_eeprom).
ring_check:         mov    ring_count,#014h           ;1  Load the ring_count register with the 20 cycle
                                                      ;   countdown value.
init_low_time:      mov    low_count_l,#000h          ;1  Initialize the low byte of the inter-ring cycle low
                                                      ;   counter.
                    mov    low_count_h,#010h          ;1  Initialize the high byte of the inter-ring cycle low
                                                      ;   counter.
                    sjmp   ring_loop                  ;2  Skip over the time I ISR and go to the main program.
;-----------------------------------------------------------------------------------------------------------------


;Interrupt service routine: Timer I -----------------------------------------------------------------------------

;The timer I interrupt service routine is used to detect IIC errors.  If the IIC signals get "stuck" for more
;than about 1ms, timer I will expire and the program will be vectored to address #01bh.  This ISR loads the
;iic_error_code register with ASCII zero (#030h) and sends it to the IIC error annunciation routine.

                    org    01bh                       ;   Timer I interrupt location.
timer_i_int:        mov    iic_error_code,#030h       ;1  Load the error code register with the ASCII value of
                                                      ;   zero.
                    ajmp   iic_error                  ;2  Annunciate IIC error code 0: Timer I expiration.
;-----------------------------------------------------------------------------------------------------------------


;Main program ---------------------------------------------------------------------------------------------------
;The ring_check routine checks the phone line for a ring.  A "characteristic" ring consists of about two to three
;seconds of 20ms pulses which are separated by about 30ms.  This corresponds to a ring frequency of about 20Hz.
;The software looks for 20 consecutive 15ms highs at the phone_data port bit.  After detecting the first 15ms
;high, the software allows about 50ms for the next 15ms pulse to start.  If another 15ms pulse is not found
;within the 50ms, the counters are reset and the software continues to monitor the function buttons and the
;status of phone_data.

;Due to the relatively long pulsewidths and deadtimes, the use of 16 bit counters is required.  The data pointer,
;DPTR, is used to count the number of loop iterations during which phone_data is high.  The data pointer serves
;as a low overhead sixteen bit counter because it can be directly loaded and incremented using a single
;instruction.  The low time counter consists of two eight bit registers.  The lower eight bits are stored in
;low_count_l and the upper eight bits are stored in low_count_h.

;After detecting 20 consecutive ring pulses, the software allows about three seconds for the ring signal to
;finish.  In order to detect the completion of a legitimate ring, a deadtime of at least 100ms must be present
;at phone_data during the three second ring termination loop.  If the 100ms of deadtime is not detected, then
;the software resets the ring counters and continues to monitor phone_data and the function buttons.

ring_loop:          jb     phone_data,inc_ring         ;2  If phone_data is high, increment the high counter.
                    mov    dptr,#0000h                ;2  Reset the data pointer.
key_check:          jnb    delete_key,delete_keypress  ;2  If the "delete" key is pressed, proceed to execute it.
                    jnb    forward_key,forward_keypress ;2 If the "forward" key is pressed, proceed to execute it.
                    jnb    back_key,back_keypress      ;2  If the "back" key is pressed, proceed to execute it.
                    djnz   low_count_l,ring_loop       ;2  Decrement the lower byte of the low counter and repeat
                                                       ;   until is reaches zero (rolls over).
                    djnz   low_count_h,ring_loop       ;2  If the lower byte of the low counter rolls over,
                                                       ;   decrement the high byte and repeat until the high
                                                       ;   byte reaches zero.
                    sjmp   ring_check                  ;2  Reinitialize all counters if a 15ms pulse was not
                                                       ;   detected during the 50ms interval.

inc_ring:           inc    dptr                        ;2  Increment the data pointer if phone_data was high.
                    mov    a,dph                       ;1  Load the upper byte of the data pointer into the
                                                       ;   accumulator for a subtraction.
                    clr    c                           ;1  Explicitly clear the carry bit.
                    subb   a,#004h                     ;1  Subtract the value that corresponds to about 15ms.
                    jz     dec_ring_count              ;2  If the result is zero, then a 15ms pulse was detected
                                                       ;   so decrement the number of required pulses.
```

```
                    sjmp   init_low_time              ;2  Keep checking phone_data.

dec_ring_count:     jb     phone_data,$               ;2  Wait until phone_data goes low.
                    djnz   ring_count,init_low_time   ;2  Keep looking for 15ms pulses until the required number
                                                      ;   has been detected.
                    mov    dptr,#0000h                ;2  Reset the data pointer.  It will now be used to count
                                                      ;   the three seconds allowed for the ring to finish.
reset_deadtime:     mov    low_count_l,#000h          ;1  Reset the low byte of the low time counter.
                    mov    low_count_h,#009h          ;1  Reset the high byte of the low time counter.
deadtime_check:     inc    dptr                       ;2  Increment the data pointer each time through the loop.
                    mov    a,dph                      ;1  Load the high byte of the data pointer into the
                                                      ;   accumulator.
                    inc    a                          ;1  Increment the accumulator.
                    jz     ring_check                 ;2  If the accumulator (the high byte of the data pointer)
                                                      ;   rolls over to zero, then its initial value was #0ffh,
                                                      ;   indicating the three seconds has expired.
                    mov    b,#009h                    ;2  Load the B register with a delay loop countdown value.
                                                      ;   The delay will consist of 4*B machine cycles and allows
                                                      ;   the software to count a 3 second interval using a
                                                      ;   16 bit counter.
delay_loop:         jb     phone_data,reset_deadtime  ;2  Reset the deadtime counter if phone_data becomes
                                                      ;   asserted (i.e. canceling the deadtime).
                    djnz   b,delay_loop               ;2  Repeat until B reaches zero.
                    djnz   low_count_l,deadtime_check ;2  Keep looking for deadtime until the low byte of the
                                                      ;   low count register rolls to zero (256 iterations).
                    djnz   low_count_h,deadtime_check ;2  After 256 times through the loop, decrement the high
                                                      ;   byte of the deadtime counter until it reaches zero.
                    sjmp   deadtime_found             ;2  After the high byte reaches zero, proceed to look
                                                      ;   for the caller ID data.

;The following section of code is used to implement the button functions.  If either the "forward" or "back" key
;is pressed individually, the software will jump to a common routine (keypress) to implement their functions.
;If the "forward" and "back" keys are pressed simultaneously through the wired OR configuration of the "xmit"
;key, then the software uses a long jump to execute the send_cordless routine which will transmit the currently
;displayed record to the cordless handset.  If the "delete" key is pressed, the software jumps to the
;delete_keypress section of code where the currently displayed record is deleted from the EEPROM.

;After performing the forward, back, or delete operations, the keys are debounced for 55.3ms before the software
;returns to the ring_check section of code.  The "xmit" key is debounced by virtue of the time required to
;transmit the cordless data (several seconds).

forward_keypress:   jnb    back_key,long_xmit_jump    ;2  If the back_key is pressed at the same time as the
                                                      ;   forward_key, proceed to transmit the current record
                                                      ;   over the cordless channel.
                    sjmp   keypress                   ;2  Go to the general keypress routine if only the
                                                      ;   forward_key is pressed.
back_keypress:      jnb    forward_key,long_xmit_jump ;2  If the forward_key is pressed at the same time as the
                                                      ;   back_key, proceed to transmit the current record
                                                      ;   over the cordless channel.
                    sjmp   keypress                   ;2  Go to the general keypress routine if only the
                                                      ;   back_key is pressed.
long_xmit_jump:     ajmp   send_cordless              ;2  Send the currently displayed record to the handset (a
                                                      ;   long jump is needed because the send_cordless routine
                                                      ;   is out of the relative addressing range).

keypress:           acall  init_eeprom                ;2  Call the EEPROM initialization routine to get the
                                                      ;   number of records into the accumulator.
                    mov    b,a                        ;1  Temporarily store the number of records in the
                                                      ;   B register.
                    mov    a,#disp_record_addr        ;1  Load the accumulator with the off-screen DD RAM.
                    acall  set_disp_address           ;2  Set the cursor to the off-screen position.
                    acall  read_disp_data             ;2  Get the currently displayed record number.
                    jz     key_debounce               ;2  If there are no records, debounce the key with no
                                                      ;   further action.
                    jnb    forward_key,forward        ;2  If the forward key was pressed, go to the forward
                                                      ;   code segment.
```

```
                        jnb     back_key,back               ;2  If the back key was pressed, go to the back code
                                                            ;   segment.
                        sjmp    ring_check                  ;2  Go back to ring_check if no keys are pressed.
back:                   cjne    a,#001h,decrement_record    ;2  If the current record number is not 1, then it is
                                                            ;   all right to decrement the displayed record.
                        sjmp    key_debounce                ;2  Debounce the key with no further action if the current
                                                            ;   record is the first.
forward:                cjne    a,b,increment_record        ;2  If the currently displayed record is not equal to the
                                                            ;   number of records, then it is OK to display the next
                                                            ;   record.
                        sjmp    key_debounce                ;2  Debounce the key with no further action.
increment_record:       inc     a                           ;1  Increment the record number.
                        sjmp    update_display              ;2  Display the new record on the screen.
decrement_record:       dec     a                           ;1  Decrement the record number.
update_display:         acall   disp_record                 ;2  Display the new record.


;The key_debounce routine is used to debounce the function keys after their respective functions have been
;performed.  The actual debounce time is 55.296ms and is derived as follows:

;                       12 machine cycles in check_key * #1200h iterations of check_key = 55,296us

;It should be noted that the xmit key is not debounce by key_debounce.  The xmit key is debounced by virtue
;of the amount of time required to transmit the data (several seconds).

key_debounce:           mov     dptr,#0000h                 ;2  Initialize the data pointer to zero.
check_key:              mov     a,data_port                 ;1  Load the key status into the accumulator.
                        anl     a,#key_mask                 ;1  Remove the irrelevant bits.
                        xrl     a,#key_mask                 ;1  Exclusive OR the button bits with the key mask.  If
                                                            ;   any keys are pressed, the result will be non-zero.
                        jnz     key_debounce                ;2  Reset the debounce time counter if any one of the
                                                            ;   keys is still pressed.
                        inc     dptr                        ;2  Increment the data pointer.
                        mov     a,dph                       ;1  Load the accumulator with the high byte of the data
                                                            ;   pointer.
                        cjne    a,#012h,check_key           ;2  Continue to check the keys until the time expires.
                        ajmp    ring_check                  ;2  Go back to looking for a ring.

;The delete_keypress routine is executed when the delete_key is pressed.  The routine first calls init_eeprom
;to get the number of stored records into the accumulator.  If there are no records, then the keys are
;debounced with no further action.  If the number of records in the EEPROM is non-zero, the software loads
;the number of the currently displayed record from the off-screen DD RAM location and passes it to the
;delete_record subroutine.  After returning from the delete_record subroutine, the program determines if the
;deleted record was the last record.  If so, the program proceeds to display the last record on the screen.  If
;not, then the program displays the record number of the previously deleted record (i.e. the record database
;was moved-up from the deleted record).  In such a case, record 7 would become record 6, record 8 would become
;record 7 and so on.  The display is then updated and the keys are debounced prior to returning to ring_check.

delete_keypress:        acall   init_eeprom                 ;2  Call the EEPROM initialization routine to get the
                                                            ;   number of records into the accumulator.
                        jz      key_debounce                ;2  If there are no records in the EEPROM, debounce the
                                                            ;   keys and go back to ring_check.
                        mov     a,#disp_record_addr         ;1  Load the accumulator with the address of the record
                                                            ;   holder in DD RAM.
                        acall   set_disp_address            ;2  Move the cursor to this location
                        acall   read_disp_data              ;2  Get the record number from DD RAM into the accumulator.
                        mov     temp_aux,a                  ;1  Store the currently displayed record number in the
                                                            ;   auxiliary temporary register.
                        acall   delete_record               ;2  Delete the record.
                        acall   init_eeprom                 ;2  Call the EEPROM initialization routine to get the
                                                            ;   number of records into the accumulator.
                        mov     b,temp_aux                  ;2  Temporarily store the record number in the B register
                                                            ;   for a compare operation.
                        cjne    a,b,test_record             ;2  See if the deleted record number is now greater than
                                                            ;   the total number of records.
                        sjmp    update_screen               ;2  If the deleted record number is now equal to the
```

```
                                                ;   total number of records, then display the last
                                                ;   record on the screen.
test_record:        jc      update_screen       ;2  If the carry bit is set, then the deleted record
                                                ;   number exceeds the total number of records, so
                                                ;   display the last record, which is in the accumulator.
                    mov     a,temp_aux          ;1  If the carry bit is clear, then the deleted record
                                                ;   number is less than the total number of records, so
                                                ;   load the accumulator with the deleted record number
                                                ;   to display the record that took its place.
update_screen:      acall   disp_record         ;2  Display the record contained in the accumulator.
                    sjmp    key_debounce        ;2  Debounce the keys and go back to ring_check.

;When the ring has completed and 100ms of deadtime has been detected, the phone line is checked for about 83ms
;of the mark frequency.  The channel seizure signal is skipped-over and the program looks for 100 continuous
;mark signals (the carrier frequency).  The phone line input signal will be asserted for 416us (1/2 mark cycle)
;for every mark cycle.  To account for noise and small changes in the bit times, the software will consider any
;pulsewidth between 312us and 520us to be of mark frequency (416us +/- 25%).  After receiving 100 of these
;pulses in a row, the software will begin to look for a space, which signifies the beginning of the data.  The
;bit time corresponding to 1200 baud is 0.833ms (833 machine cycles with a 12MHz oscillator).  The external
;timer (the "watchdog") is polled during this operation so that the software doesn't get "stuck" looking
;for the 100 consecutive mark signals.  This is necessary in case the phone goes off hook prior to the
;transmission of the CID information (i.e. an early answer or an answering machine picks-up) or if the CID
;service is not available altogether.

deadtime_found:     setb    w_dog_rst           ;1  Discharge the watchdog capacitor.
                    jnb     w_dog,$             ;2  Wait until the watchdog bit is asserted.
                    clr     w_dog_rst           ;1  Allow the watchdog capacitor to charge.

start_count:        mov     mark_count,#064h    ;1  Initialize the mark_count to #100d for countdown.

pulse_wait:         jb      phone_data,init_mark_max    ;2  If phone_data becomes asserted, start measuring
                                                ;   the pulse duration.
                    jb      w_dog,pulse_wait    ;2  Loop until the watchdog expires.
long_ring_jump:     ajmp    ring_check          ;2  Go back to ring_check if the carrier frequency
                                                ;   cannot be found within the watchdog interval.

init_mark_max:      mov     a,#068h             ;1  Initialize the accumulator with the maximum assertion
                                                ;   time (520us).
port_high:          dec     a                   ;1  Decrement the accumulator.
                    jz      out_of_bounds       ;2  Get-out of the loop if the accumulator reaches zero.
                    jb      phone_data,port_high    ;2  Repeat while phone_data is high.

;When the program reaches the timeout label, the accumulator has #068h minus the number of port_high
;iterations that were detected or zero if the 520us limit was exceeded.  The accumulator value corresponding
;to the minimum pulsewidth of 312us is calculated as follows:
;
;                           312us = 312 machine cycles
;
;                           312 cycles / 5 cycles in port_high = 62.4 port_high iterations (call it 62)
;
;                           62 decrements from #068h = 42 (#02ah)
;
;So, #02ah is the accumulator value corresponding to the minimum pulse width and the accumulator
;value corresponding to a pulsewidth greater than 520us is zero.  If the accumulator value is
;between #001h and #02ah, then the pulse is considered to be that of a valid mark frequency.

timeout:            clr     c                   ;1  Explicitly clear the carry bit.
                    subb    a,#02ah             ;1  This will set the carry bit if the accumulator
                                                ;   has a value between zero and #02ah.
                    jnc     out_of_bounds       ;2  The pulse was too short if the c bit is clear.

                    djnz    mark_count,pulse_wait   ;2  Decrement the mark_count and repeat the pulse test.
                    sjmp    init_start          ;2  When mark_count reaches zero, get ready to look for
                                                ;   a start bit.
out_of_bounds:      jnb     w_dog,long_ring_jump    ;2  Go back to ring_check if the watchdog expires.  An
                                                ;   absolute jump is required because ring_check is
```

```
                                                ;    out of the relative addressing range.
                        jb      phone_data,out_of_bounds    ;2  Loop until the port goes low in case of timeout.
                        sjmp    start_count                 ;2  Restart looking for the 100 consecutive mark pulses.

;Now that the 1200hz carrier has been detected (the 100 consecutive mark pulses), the program can start
;looking for a start bit.  According to the Caller ID protocol, each data byte is preceded by a start
;bit (logical 0, 2200Hz).  The following section of code monitors the data port (phone_data) for a 2200Hz
;pulse.  Ideally, the duration of the pulse assertion will be 0.227ms (i.e .5/2200Hz).  However, to
;increase noise margins, the software will apply a tolerance to this duration.  So, a pulse as short as
;0.105ms and as long as 0.280ms will be allowed.  This tolerance is arbitrarily based on experimentally
;determined values.

;The software will accomplish the 2200Hz pulse detection by looking for a transition at the phone_data
;port.  First, the state of the data_port is loaded and everything but the phone_data bit is masked-off.
;The phone_data bit is stored as an eight bit word so that the 8051 exclusive OR instruction can be
;utilized.  After storing the state of phone_data, the software proceeds to monitor phone_data for a
;transition.  A transition is detected by exclusive-ORing the previous state of phone_data with its
;present state.  A non-zero result indicates that a transition has occurred.

;It should be noted that transition detection is necessary because the FSK data may start on either a
;rising edge or a falling edge.  This makes it impractical to look for one or the other because either
;can occur, depending on the unknown history of the transmission.

;Immediately after a transition has occurred, the internal timer is started to independently keep track
;of when the transition occurred.  Meanwhile, a software counter is used to determine the duration of the
;pulse.  If the duration of the pulse is within the established bounds (105us to 260us), the software
;assumes that it was the first pulse of a start bit and proceeds to acquire the remaining data bits
;in the byte.  If not, the counters are reset and the software proceeds to measure the duration of
;the next pulse.  The software is allowed one watchdog interval to acquire the start bit.

;After detecting the first legitimate pulse of the start bit, the value of the internal counter is used
;to set-up the bit counter in the data bit acquisition code.  In other words, the internal timer is
;used to "mark" the beginning of the start bit so that the following bits will be correctly spaced in
;time.  This method is used for each individual data byte received so that timing errors will not
;"stack-up" throughout the duration of the CID transmission.  Additionally, the CID specification indicates
;that up to 10 mark bits may be placed at the end of any data byte, so it is necessary to explicitly look
;for start bits rather that assuming that a byte will be sent every 833us after the first start bit is
;detected.

init_start:             mov     ram_ptr,#data_ram           ;1  Reset the RAM pointer to the beginning of RAM.
                        mov     checksum,#000h              ;2  Clear the checksum location.
                        mov     byte_count,#000h            ;1  Reset the byte counter.

                        setb    w_dog_rst                   ;1  Discharge the watchdog capacitor.
                        jnb     w_dog,$                     ;2  Wait until the watchdog bit is asserted.
                        clr     w_dog_rst                   ;1  Allow the watchdog cap to charge.

;The start_check code segment loads the current state of the data_port, masks-off everything but the
;phone_data bit, and stores the result in the trans register, which is used to store the previous state
;of the phone_data bit.

;The external watchdog timer is monitored in this section of the program.  If it expires while looking for
;a start bit, then the program assumes that no caller ID information is available and jumps to an annunciator
;routine.  This will indicate that a call has been received but there is something wrong with the CID system.

;After the first data byte has been received, a 15ms counter is started prior to receiving the next byte.  If
;this counter expires before a new start bit is received, then the software assumes that the CID transmission
;is complete and goes to the checksum evaluation routine.  This timer provides the first of two means by which
;the software can detect the end of the CID transmission.  In other words, a span of 15ms without a start bit
;means that the data transmission is finished.  This coincides with the CID specification which specifies a
;maximum of 10 mark signals (about 8.3ms) between data bytes.  The timer value of 15ms was arbitrarily
;chosen to provide noise margin.

;As usual, the initial timer values are set such that the timer overflow flag (TF) will be set after 15ms
;elapses.  The initial timer value is #ffffh-15,000 machine cycles = #c567h.
```

```
start_check:          mov    a,data_port               ;1  Copy the status of the input port to the accumulator.
                      anl    a,#phone_mask             ;1  Mask-off everything but the phone_data bit.
                      mov    trans,a                   ;1  Put the state of phone_data into the trans register.

                      mov    tcon,#timer_stop          ;2  Stop the timer and clear TF.
                      mov    th,#0c5h                  ;2  Initialize the high timer byte with the 20ms count
                                                       ;   value.
                      mov    tl,#067h                  ;2  Initialize the low timer byte with the 20ms count
                                                       ;   value.
                      mov    a,byte_count              ;1  Copy the byte count into the accumulator.
                      jz     transition_test           ;2  If the byte_count is zero, then do not start the 20ms
                                                       ;   timer.
                      mov    tcon,#timer_start         ;2  Start the timer if more than one data byte has been
                                                       ;   received.

;The transition_test loop loads the current state of the data_port, removes the irrelevant bits, and
;exclusive ORs the phone_data bit with its previous state.  If the result is non-zero, a transition has
;occurred.  If not, the program continues to loop until a transition is detected or the watchdog expires or
;until the 15ms timer expires (if more than one data byte has been received).

transition_test:      mov    a,data_port               ;1  Copy the status of the input port to the accumulator.
                      anl    a,#phone_mask             ;1  Mask-off everything but the phone_data.
                      xrl    a,trans                   ;1  Exclusive OR the current state of phone_data with
                                                       ;   its previous state.
                      jnz    count_setup               ;2  The result will be non-zero if a transition has
                                                       ;   occurred, so start timing the pulse width.
                      jb     tf,jump_check_data        ;2  If the 15ms has elapsed without detecting a start bit,
                                                       ;   then the data transmission must be complete.
                      jb     w_dog,transition_test     ;2  Continue to look for a transition until the
                                                       ;   watchdog expires.
                      ajmp   fatal_cid_error           ;2  If the watchdog expires while looking for a start bit,
                                                       ;   then something is wrong with the CID system.  In this
                                                       ;   case, alert the user that a call has been received
                                                       ;   but there is no CID available.

jump_check_data:      ajmp   check_data                ;2  An absolute jump is required to get from this program
                                                       ;   area to the check_data segment because it is out of
                                                       ;   the relative addressing bounds.

;The count_setup code segment is used to place a marker in time after a transition is detected.  Following
;a successful start bit pulse detection, this marker is used to determine the actual start time of the
;beginning of the start bit.

count_setup:          mov    a,data_port               ;1  Copy the status of the input port to the accumulator.
                      anl    a,#phone_mask             ;1  Mask-off everything but the phone_data.
                      mov    trans,a                   ;1  Put the state of phone_data into the trans register.
                      mov    tcon,#timer_stop          ;2  Stop the timer and clear TF.

;The timer is seeded with the expected bit time of the start bit minus a few tens of microseconds to account
;for time that has elapsed since the transition was detected.  This value is presently set to 700us but is
;subject to change.  This is one of the key "tuning" parameters which may be adjusted to reduce the frequency
;of checksum errors.

                      mov    th,#0fdh                  ;2  Initialize the high timer byte with the expected bit
                                                       ;   time (minus a few tens of microseconds to account for
                                                       ;   machine cycles that have elapsed since the transition
                                                       ;   was detected).
                      mov    tl,#043h                  ;2  Initialize the low timer byte with the expected bit
                                                       ;   time (minus a few tens of microseconds to account for
                                                       ;   machine cycles that have elapsed since the transition
                                                       ;   was detected).
                      mov    tcon,#timer_start         ;2  Start the timer.
                      mov    b,#025h                   ;2  Initialize B with the maximum pulsewidth time.

;The count_length code segment determines the duration of the pulse.  If it is too short or too
;long, the program returns to looking for another start bit.  However, if the watchdog expires while looking
```

```
;for a start bit, the program will assume that there is a problem with the CID system.  In this case, the
;user will be informed that a call has been received but no CID is available.

;A normal pass through the count_length loop requires 7 machine cycles.  The B register has been initialized
;with #025h (#37d) which corresponds to the maximum pulse width of 260us (7 cycles * 37 iterations = 259).
;If the B register is decremented fewer than #025h - #016h = #00fh (#15d) times, then the pulse was too short.
;Note that 15 iterations of 7 machine cycles corresponds to the minimum pulse width of 105us.

count_length:         mov    a,data_port                ;1  Copy the status of the input port to the accumulator.
                      anl    a,#phone_mask              ;1  Mask-off everything but the phone_data.
                      xrl    a,trans                    ;1  Exclusive OR the current state of phone_data with
                                                        ;   its previous state.
                      jnz    exit                       ;2  If the result is non-zero, a transition has occurred.
                      djnz   b,count_length             ;2  If B reaches zero, the pulse was too long.  Otherwise,
                                                        ;   continue to measure its duration.
                      jb     w_dog,transition_test      ;2  Restart the transition test if the pulse was too
                                                        ;   long and the watchdog has not expired.
                      ajmp   fatal_cid_error            ;2  If the watchdog expires while looking for a start bit,
                                                        ;   then something is wrong with the CID system.  In this
                                                        ;   case, alert the user that a call has been received
                                                        ;   but there is no CID available.

exit:                 mov    a,b                        ;1  Copy the pulse width count into the accumulator to
                                                        ;   prepare for a direct subtraction.
                      clr    c                          ;1  Explicitly clear the carry bit.
                      subb   a,#016h                    ;1  Subtract the minimum number of decrement iterations
                                                        ;   from the actual number of decrement iterations.
                      jc     start_bit_delay            ;2  The pulse was of an acceptable duration if the carry
                                                        ;   bit is set, so proceed to the next section of code.
                      jb     w_dog,start_check          ;2  Restart the transition test if the pulse was too
                                                        ;   short and the watchdog has not expired.
                      ajmp   fatal_cid_error            ;2  If the watchdog expires while looking for a start bit,
                                                        ;   then something is wrong with the CID system.  In this
                                                        ;   case, alert the user that a call has been received
                                                        ;   but there is no CID available.

;The CID information is transmitted at 1200bps, which means that each bit will last for .833ms or
;833 machine cycles with a 12MHz oscillator.  During each bit time, the software will integrate the
;phone_data signal to determine if it represents a logical zero (2200Hz) or a logical one (1200Hz).

;The initial value of the integral is set to 128.  A high phone_data signal will cause the integral
;to be incremented while a low phone_data signal will cause the integral to be decremented.  Although
;the integral value will always be close to 128 at the end of the bit time, the maximum excursions
;from 128 can be used to distinguish between a zero (2200Hz) and a one (1200Hz).  The lower frequency
;signal will have a greater difference in extrema of the integral than that of the higher frequency.
;Integrating the phone_data signal also provides noise margin in the bit timing.  Since only the maximum
;and minimum integral values are significant, the bit clock can "leak" into adjacent bits while still
;differentiating between 1200Hz and 2200Hz.

;The start_bit_delay loop allows the bit timer to expire after detecting the first pulse of a start bit.
;The count_setup code segment started the internal timer with an initial value which will center the
;2200Hz start bit in an 833us window.

start_bit_delay:      jnb    tf,start_bit_delay         ;2  Wait for the timer to expire before looking for
                                                        ;   the first data bit.
                      mov    bit_count,#008h            ;1  Initialize the bit counter with 8 bits (countdown).
                      mov    data_byte,#000h            ;1  Clear the data_byte register.

;The internal timer is used as the bit clock.  It is seeded with a value which will assert the timer flag after
;814us has elapsed.  The 814us is slightly less than the 833us bit time, which allows for a few microseconds of
;post processing on the data bit within the 833us bit time.  The timer value is #ffffh-833us=#fcd1h.

get_bit:              mov    tcon,#timer_stop           ;2  Stop the timer.
                      mov    th,#0fch                   ;2  Initialize the high timer byte with the bit time.
                      mov    tl,#0d1h                   ;2  Initialize the low timer byte with the bit time.
                      mov    tcon,#timer_start          ;2  Start the timer
```

```
                    mov   integral,#080h          ;1  Set the initial integral value to 128.
                    mov   max_value,#080h         ;1  Set the initial maximum value to 128.
                    mov   min_value,#080h         ;1  Set the initial minimum value to 128.

bit_start:          jb    tf,bit_timeout          ;2  Go to bit_timeout if the timer has expired.
                    jb    phone_data,inc_sum      ;2  Increment the integral if phone_data is high.

dec_sum:            dec   integral                ;1  Since phone_data is low, decrement the integral.
                    mov   b,integral              ;2  Copy the integral value into the B register.
                    mov   a,min_value             ;1  Copy the latest minimum value into A for comparison.
                    cjne  a,b,new_min             ;2  If the integral and the old minimum value are not
                                                  ;   equal, see if the value of integral constitutes a
                                                  ;   new minimum.
                    sjmp  burn_cycles             ;2  Burn cycles to make everything even.

new_min:            jc    burn_cycles             ;2  If the carry bit is set, then the integral value
                                                  ;   is not a new minimum. (min_value<integral)
                    mov   min_value,b             ;2  Make min_value equal to the integral value.
                    sjmp  bit_start               ;2  Continue to integrate phone_data.

inc_sum:            inc   integral                ;1  Since phone_data is high, increment the integral.
                    mov   b,integral              ;2  Copy the integral value into the B register.
                    mov   a,max_value             ;1  Copy the latest maximum value into A for comparison.
                    cjne  a,b,new_max             ;2  If the integral and the old maximum value are not
                                                  ;   equal, see if the value of integral constitutes a
                                                  ;   new maximum.
                    sjmp  burn_cycles             ;2  Burn cycles to make everything even.

new_max:            jnc   burn_cycles             ;2  If the carry bit is clear, then max_value is less
                                                  ;   than the integral, hence no new maximum exists.
                                                  ;   (integral<max_value)
                    mov   max_value,b             ;2  Make max_value equal to the integral value.
                    sjmp  bit_start               ;2  Continue to integrate phone_data.

;The purpose of the burn_cycles code segment is to "time pad" the integration sequence so that all
;operations, whether they are new_max, new_min, or neither, will take the same amount of time.  This
;ensures that the slope (gain) of the integration remains constant in all cases.  From the bit_start
;label, all iterations require 16 machine cycles (16us with a 12MHz oscillator).

burn_cycles:        nop                           ;1  Use one machine cycle.
                    nop                           ;1  Use one machine cycle.
                    sjmp  bit_start               ;2  Continue to integrate phone_data.

;After the bit timer has expired, it is necessary to determine whether the most recently acquired bit was
;a logical 0 (2200Hz) or a logical one (1200Hz).  This is done by taking the difference between the
;extrema of the integral.  In other words, the difference between max_value and min_value is used
;to make the distinction between zero and one.

;Ideally, a 1200Hz signal will produce an extrema difference of 26.  This value is derived as follows:
;
;                   1200Hz = 833us per cycle so half of a cycle will last 833us/2 = 417us
;
;                   Since each iteration of the integral routine requires 16us, the ideal difference
;                   between maximum and minimum will be 417us/16us = 26.
;
;The extrema difference for a 2200Hz (logical zero) signal is calculated in a similar fashion:
;
;                   2200Hz = 455us per cycle so half of a cycle will last 455us/2 = 227us
;
;                   Since each iteration of the integral routine requires 16us, the ideal difference
;                   between maximum and minimum will be 227us/16us = 14.
;
;The extrema difference for a constant signal is calculated in a similar fashion:
;
;                   0Hz = 833us per cycle
;
```

```
;                       Since each iteration of the integral routine requires 16us, the ideal difference
;                       between maximum and minimum will be 833us/16us = 52.

;Fortunately, the difference between a logical zero and a logical one is quite significant.  The threshold
;will be set slightly above the center of the two ideal extrema differences.  This will allow logical zeros
;(2200Hz) to "leak" into logical ones (1200Hz) if the bit timer is not properly synchronized without
;causing errors.  This value is arbitrarily chosen to be 24.  So, an extrema difference less than 24 will
;be considered a zero and an extrema difference of 24 or more will be considered a logical one.  This will
;allow for fairly high noise margins in that spurious spikes on the phone_data signal or a mis-synchronized
;bit clock will not contribute significantly to the outcome.  In other words, fast spikes on phone_data and
;variations in bit synchronization will be integrated-out (lowpass filtered).  Furthermore, an extrema
;of 32 or more will indicate that the transmission has stopped (i.e. a constant signal for more than
;32*16=512us).  This is the second of two ways in which the software can determine that the transmission
;is complete.

bit_timeout:            mov     tcon,#timer_stop            ;2  Stop the timer.
                        mov     a,max_value                 ;1  Copy the maximum value into A for a subtraction.
                        clr     c                           ;1  Explicitly clear the carry bit.
                        subb    a,min_value                 ;1  Subtract the minimum value from the maximum value.
                        mov     max_value,a                 ;1  Temporarily store the extrema difference in the
                                                            ;   max_value register.
                        clr     c                           ;1  Explicitly clear the carry bit.
                        subb    a,#01fh                     ;1  Subtract decimal 31 to see if the difference in the
                                                            ;   extrema is 32 or more.  If so, then the transmission
                                                            ;   has ceased.
                        jnc     check_data                  ;2  If the carry bit is clear, then the transmission has
                                                            ;   ceased, so stop getting bits.
                        mov     a,max_value                 ;1  Load the accumulator with the max_value register,
                                                            ;   which now contains the extrema difference.
                        clr     c                           ;1  Explicitly clear the carry bit.
                        subb    a,#015h                     ;1  Subtract the zero/one threshold.  If C is set, then
                                                            ;   the bit was a zero.
                        cpl     c                           ;1  Complement the carry bit to make it representative
                                                            ;   of the received bit.
                        mov     a,data_byte                 ;1  Load the data_byte into the accumulator.
                        rrc     a                           ;1  Roll C onto the data_byte.
                        mov     data_byte,a                 ;1  Put the data_byte back with the new bit rolled onto
                                                            ;   it.
                        jnb     w_dog,fatal_cid_error       ;2  If the watchdog expires in this section of code, then
                                                            ;   there is a major problem with some portion of the
                                                            ;   CID system.  If this is the case, alert the user that
                                                            ;   a call has been received but the CID information is
                                                            ;   extremely corrupted or not available.
                        djnz    bit_count,get_bit           ;2  Get another bit until eight bits have been received.

;At this point, all eight bits of the current data byte have been received.  The software stores the most
;recently received data byte in RAM and adds it to an ongoing checksum value.  The ram_ptr is moved to the
;next location and the byte_count is incremented before the software starts looking for the next start bit.
;Note that the stop bit is not processed - the software simply returns to looking for the next start bit.  This
;is necessary because the CID specification indicates that it may be possible to have up to 10 stop bits between
;data bytes.

                        mov     @ram_ptr,a                  ;1  Copy the most recently received byte into the current
                                                            ;   memory location.
add_checksum:           add     a,checksum                  ;1  Add the most recently received byte to the checksum.
                        mov     checksum,a                  ;1  Put the checksum back.
                        inc     ram_ptr                     ;1  Increment the ram_ptr.
                        inc     byte_count                  ;1  Increment the byte_count register.
                        ajmp    start_check                 ;2  Go back to start_check to look for the next start bit.

;If the program reaches the fatal_cid_error label, then something major has gone wrong in the CID system.  The
;primary cause of such a failure is answering the phone (going off hook) while CID is being transmitted.  This
;trap provides a graceful failure mode for such cases.  This section of code first clears the LCD display and
;then prints the 16 character error message on the second line of the display.

fatal_cid_error:        mov     a,#00000001b                ;1  Load the accumulator with the clear display command.
```

```
                        acall  send_command                ;2  Clear the LCD display.
                        mov    dptr,#fatal_message         ;1  Load the data pointer with the address of the
                                                           ;   "fatal error" message.
                        acall  send_message                ;2  Send the "fatal error" message to the display.
                        ajmp   save_screen                 ;2  Save the error message in EEPROM.  The program will
                                                           ;   go back to ring_check from the save_screen routine.

;The check_data routine is used to compare the received checksum with the actual checksum.  The checksum is
;determined by the lower eight bits of the sum of all received data bytes.  The checksum register contains
;this number plus the actual checksum as sent by CID.  Because the checksum sent by CID is the twos complement
;of the sum of all received data bytes, the calculated checksum may simply be added to the checksum sent by CID.
;This has already been done by the add_checksum routine.  So, if checksum is zero, then the data is good.
;Otherwise, the data is considered to be corrupted and the user must be alerted.  The LCD display is cleared
;at the check_data label to prepare the display for the new data.

check_data:             mov    a,#00000001b                ;1  Load the accumulator with the clear display command.
                        acall  send_command                ;2  Clear the LCD display.
                        mov    a,checksum                  ;1  Load the accumulator with the checksum value.
                        jz     get_date                    ;2  If the checksum is zero, then the data is probably
                                                           ;   valid, so get the current date from RAM.

;The data_bad routine is used to annunciate a checksum error.  This is done by placing an epsilon symbol in the
;rightmost position of line 3 of the LCD display.  Since line 3 is always used to display the originator's phone
;number and the processed phone number data is only 14 characters in length, the 16th character position will
;always be available for the epsilon character.  So, if the data is bad, the software will move the display's
;cursor to this position and print an epsilon (#0e3h in the HD44780 character set).  Also, if a checksum error
;has occurred, "all bets are off" with regards to data integrity.  For this reason, the subsequent date and
;time display routines have error traps which check for valid time and date characters.  The validate_num
;subroutine is used to determine if a valid ASCII numeral exists in the time and date locations in RAM.  If
;not, the current time and date character is replaced with a question mark.

data_bad:               mov    a,#01fh                     ;1  Load the accumulator with the address of the right
                                                           ;   most character of line 3.
                        acall  set_disp_address            ;2  Set the cursor to this position.
                        mov    a,#0e3h                     ;1  Load the accumulator with the value of epsilon.
                        acall  send_disp_data              ;2  Send the epsilon character to the display to indicate
                                                           ;   that a checksum error has occurred.

;The purpose of the get_date section of code is to get the time and date from the CID information.  The get_date
;code segment will retrieve the date from the received data, strip leading zeros, and write it to the bottom
;left of the LCD display.  Then, the time will be retrieved from data RAM, converted from military to standard
;time, appended with "am" or "pm", and sent to the lower right corner of the LCD display.  All date and time
;information is preprocessed by the validate_num subroutine to help ensure that the data is valid.  It should
;be noted that the validate_num subroutine cannot fully ensure that the data is valid because it is possible
;to have a valid incorrect numeral (i.e. a 1 instead of a 3, etc.).

get_date:               mov    a,#050h                     ;1  Load the accumulator with the address of the first
                                                           ;   character of the LCD display's last line.
                        acall  set_disp_address            ;2  Set the display RAM pointer to this location.  The
                                                           ;   next character write to the display will occur at
                                                           ;   this location.

                        mov    ram_ptr,#data_ram+4         ;1  Load the ram pointer with the address of the beginning
                                                           ;   of the time and date information (see memory map).
                        mov    b,#000h                     ;2  Initialize the B register with the month lookup table
                                                           ;   offset.
                        mov    a,@ram_ptr                  ;1  Load the accumulator with the first byte of the date.
                        acall  validate_num                ;2  Ensure that the byte is a valid numeral.
                        jz     month_error                 ;2  If the accumulator is zero, then the first digit of the
                                                           ;   month number is invalid, so don't try to use it.
                        clr    c                           ;1  Explicitly clear the carry bit.
                        subb   a,#030h                     ;1  Subtract the ASCII value of zero.  The accumulator now
                                                           ;   contains the absolute value of the first digit of the
                                                           ;   month number.
                        jz     month_ones                  ;2  If the result is zero, then the month number is less
                                                           ;   than 10 (i.e. earlier than October).  This indicates
```

```
                                                ;   that an unnecessary leading zero is present in the
                                                ;   month number.
                        mov     b,#00ah         ;2  Load decimal ten into the B register to indicate that
                                                ;   the month number is greater than 9.
month_ones:             inc     ram_ptr         ;1  Move the RAM pointer to the next RAM location (the
                                                ;   second month byte).
                        mov     a,@ram_ptr      ;1  Load the month's ones digit into the accumulator.
                        acall   validate_num    ;2  Ensure that the byte is a valid numeral.
                        jz      month_error     ;2  If the accumulator is zero, then the last digit of the
                                                ;   month number is invalid, so don't try to use it.
                        clr     c               ;1  Explicitly clear the carry bit.
                        subb    a,#030h         ;1  Subtract the ASCII value of zero from the month's
                                                ;   ones digit to leave the accumulator holding the
                                                ;   absolute value of the month's ones digit.
                        add     a,b             ;1  Add the B register to the accumulator.  The B register
                                                ;   either holds zero or ten, depending on the value
                                                ;   of the month's tens digit.  The accumulator now holds
                                                ;   the absolute value of the month (i.e. November =
                                                ;   #00bh).
                        mov     b,a             ;1  Copy the accumulator into B.
                        clr     c               ;1  Explicitly clear the carry bit.
                        subb    a,#00ch         ;1  Subtract 12 to determine if the month is valid.
                        jnc     month_error     ;2  The carry bit will be clear if the month is greater
                                                ;   than 12.
                        mov     a,#003h         ;1  Load the accumulator with the length of the month
                                                ;   abbreviation to prepare for a multiplication.
                        mul     ab              ;4  Multiply the B register by the accumulator (3) to
                                                ;   obtain the lookup table offset for the month
                                                ;   abbreviation.
                        sjmp    month_lookup    ;2  Decode the three letter month abbreviation

;It should be noted that the result of the multiplication can never exceed 8 bits in length because the maximum
;month number is 12 and the B register always contains 3.  So, the maximum result will be 48, which easily
;fits into the lower eight bits of the multiplication result.  For this reason, the accumulator will contain
;exactly the lookup table offset for the current month and the B register will always contain exactly zero.

month_error:            mov     a,#000h         ;1  Load the accumulator with the lookup table offset
                                                ;   of the month error indicator.

;The month_lookup code segment is used to send the three character month abbreviation to the display.  The
;three character month abbreviations are stored in ROM at the months label.  A month of zero indicates an
;error.

month_lookup:           mov     dptr,#months    ;2  Load the data pointer bytes with the starting address
                                                ;   of the month abbreviation lookup table.
                        mov     temp,#003h      ;1  Load the temp register with the loop countdown value
                                                ;   (3 characters in the month abbreviation).
                        mov     b,a             ;1  Copy the accumulator value to the B register because
                                                ;   the accumulator will be destroyed by the lookup table
                                                ;   operation.
month_loop:             movc    a,@a+dptr       ;2  Load the accumulator with the first ASCII byte of the
                                                ;   month abbreviation from the lookup table.
                        acall   send_disp_data  ;2  Send the ASCII byte to the display.
                        inc     b               ;1  Increment the stored offset value to the next ASCII
                                                ;   character in the month abbreviation.
                        mov     a,b             ;1  Put the offset value back into the accumulator.
                        djnz    temp,month_loop ;2  Loop until all three month abbreviation characters
                                                ;   have been sent.

;The three letter month abbreviation (or the three question marks in case of error) and the date are separated
;by a space to improve display readability.

                        mov     a,#020h         ;1  Load the accumulator with the ASCII value of a space.
                        acall   send_disp_data  ;2  Send the space to the display.

;The date code segment loads the ASCII values of the date, as sent by CID, and prepares them for display.
```

```
;Leading zeros are not displayed and each of the two possible date digits are validated using the validate_num
;subroutine.  Erroneous date digits are replaced with question marks.

date:                   mov     ram_ptr,#data_ram+6         ;1  Move the data RAM pointer to the first digit of the
                                                            ;   date.
                        mov     a,@ram_ptr                  ;1  Load the accumulator with the first ASCII byte of
                                                            ;   the date.
                        acall   validate_num                ;2  Ensure that the byte is a valid numeral.
                        jnz     date_msd                    ;2  If the accumulator is not zero, then the byte is a
                                                            ;   valid ASCII numeral.
                        mov     a,#'?'                      ;1  Load the accumulator with the ASCII value of a
                                                            ;   question mark.
                        acall   send_disp_data              ;2  Send the question mark to the display.
                        sjmp    date_lsd                    ;2  Try to decode the last date digit.
date_msd:               clr     c                           ;1  Explicitly clear the carry bit.
                        subb    a,#030h                     ;1  Subtract the ASCII value of a zero.
                        jz      date_lsd                    ;2  If the most significant digit of the date is a zero,
                                                            ;   do not display it.
                        add     a,#030h                     ;1  Add back the ASCII value of a zero.
                        acall   send_disp_data              ;2  Send the digit out to the display.
date_lsd:               inc     ram_ptr                     ;1  Move the data RAM pointer to the least significant
                                                            ;   digit of the date.
                        mov     a,@ram_ptr                  ;1  Load the accumulator with the second ASCII byte of
                                                            ;   the date.
                        acall   validate_num                ;2  Ensure that the byte is a valid numeral.
                        jnz     display_date                ;2  If the accumulator is not zero, then the byte is a
                                                            ;   valid ASCII numeral, so print it.
                        mov     a,#'?'                      ;1  Load the accumulator with the ASCII value of a
                                                            ;   question mark.
display_date:           acall   send_disp_data              ;2  Send it out to the display.

;The get time code segment fetches the CID time from data RAM, processes it, and sends it to the display.
;Each time byte is checked for ASCII validity by the validate_num subroutine.  Since the time information
;is sent by CID in military format, it must be converted to a "standard" 12 hour format.  This requires
;an unusual amount of program code because the time is sent in ASCII and must be converted to "standard" format
;and appended with "am" or "pm", depending on the time of day or night.  Additionally, leading zeros are not
;displayed and the hours and minutes are separated by a colon character (:).  <-- this is NOT a smiley face!

get_time:               mov     a,#059h                     ;1  Load the accumulator with the display address of
                                                            ;   the time display.
                        acall   set_disp_address            ;2  Set the display's internal address pointer to position
                                                            ;   #059h.

                        inc     ram_ptr                     ;1  Increment the data RAM pointer to the most significant
                                                            ;   digit of the hours.
                        mov     a,@ram_ptr                  ;1  Load the accumulator with the MSD of the hours.

                        acall   validate_num                ;2  Ensure that the byte is a valid numeral.
                        jz      hour_error                  ;2  If the accumulator is zero, then the byte is not a
                                                            ;   valid ASCII numeral, so go to the error segment.
                        clr     c                           ;1  Explicitly clear the carry bit.
                        subb    a,#030h                     ;1  Subtract the ASCII value of a zero.  The accumulator
                                                            ;   now holds the absolute value of the MSD of the hours.
                        mov     b,#00ah                     ;2  Load the B register with decimal ten to convert the
                                                            ;   time to an absolute value.
                        mul     ab                          ;4  Multiply the absolute value of the hours MSD by ten.
                        mov     temp,a                      ;1  Store the result in the temporary register.
                        inc     ram_ptr                     ;1  Increment the data RAM pointer to the least significant
                                                            ;   digit of the hours.
                        mov     a,@ram_ptr                  ;1  Load the accumulator with the LSD of the hours.
                        acall   validate_num                ;2  Ensure that the byte is a valid numeral.
                        jz      hour_error                  ;2  If the accumulator is zero, then the byte is not a
                                                            ;   valid ASCII numeral, so go to the error segment.
                        clr     c                           ;1  Explicitly clear the carry bit.
                        subb    a,#030h                     ;1  Subtract the ASCII value of a zero.  The accumulator
                                                            ;   now holds the absolute value of the LSD of the hours.
```

```
                    add     a,temp                  ;1  Add the MSD of the hours to the LSD of the hours to
                                                    ;   obtain the absolute value of the hours (in military
                                                    ;   format).
                    jz      midnight                ;2  If the absolute value of the hours is zero, then it
                                                    ;   is between midnight and 1:00am.
                    mov     temp,a                  ;1  Copy the absolute hour value into the temporary
                                                    ;   register.
                    clr     c                       ;1  Explicitly clear the carry bit.
                    subb    a,#00ch                 ;1  Subtract 12 from the hours.
                    jz      noon                    ;2  If the result is zero, then it is 1200 hours and
                                                    ;   therefore it is noon.
                    jc      am                      ;2  If the carry bit is set, then the time is before noon.
pm:                 acall   hex_to_ascii            ;2  Convert the absolute value to two ASCII bytes.
                    mov     am_pm_byte,#'p'         ;1  Load the am_pm_byte with the ASCII value for "p".
                    sjmp    print_time              ;2  Print the time.
am:                 mov     a,temp                  ;1  Copy the original absolute hour value into A.
                    acall   hex_to_ascii            ;2  Convert the absolute value to two ASCII bytes.
                    mov     am_pm_byte,#'a'         ;1  Load the am_pm_byte with the ASCII value for "a".
                    sjmp    print_time              ;2  Print the time.
noon:               mov     ms_digit,#'1'           ;1  Load the hours ls_digit with the ASCII value of "1".
                    mov     ls_digit,#'2'           ;1  Load the hours ls_digit with the ASCII value of "2".
                    mov     am_pm_byte,#'p'         ;1  Load the am_pm_byte with the ASCII value for "p".
                    sjmp    print_time              ;2  Print the time.
midnight:           mov     ms_digit,#'1'           ;1  Load the hours ls_digit with the ASCII value of "1".
                    mov     ls_digit,#'2'           ;1  Load the hours ls_digit with the ASCII value of "2".
                    mov     am_pm_byte,#'a'         ;1  Load the am_pm_byte with the ASCII value for "a".
                    sjmp    print_time              ;2  Print the time.

hour_error:         mov     am_pm_byte,#'?'         ;1  Load the am_pm_byte with the ASCII value for "?".
                    mov     ms_digit,#'?'           ;1  Load the hours ls_digit with the ASCII value of "?".
                    mov     ls_digit,#'?'           ;1  Load the hours ls_digit with the ASCII value of "?".
                    sjmp    print_msd               ;2  Print the time error.

print_time:         mov     a,ms_digit              ;1  Load the accumulator with the most significant hours
                                                    ;   digit.
                    clr     c                       ;1  Explicitly clear the carry bit.
                    subb    a,#030h                 ;1  Subtract the ASCII value of a zero.
                    jnz     print_msd               ;2  If the result is not-zero, then there is not a leading
                                                    ;   zero.
                    mov     ms_digit,#020h          ;1  Load the ms_digit with the ASCII value of a space.
print_msd:          mov     a,ms_digit              ;1  Load the ASCII value of the most significant hours
                                                    ;   digit.
                    acall   send_disp_data          ;2  Send the most significant time digit to the display.
                    mov     a,ls_digit              ;1  Load the accumulator with the least significant hours
                                                    ;   digit.
                    acall   send_disp_data          ;2  Send the least significant hours digit to the display.
                    mov     a,#':'                  ;1  Load the accumulator with the ASCII value of a colon.
                    acall   send_disp_data          ;2  Send the colon out to the display.

                    mov     ram_ptr,#data_ram+10    ;1  Increment the RAM pointer to the most significant
                                                    ;   minutes digit.
                    mov     a,@ram_ptr              ;1  Load the accumulator with the most significant minutes
                                                    ;   digit.
                    acall   validate_num            ;2  Ensure that the byte is a valid numeral.
                    jnz     disp_ten_minute         ;2  If the accumulator is not zero, then the byte is a
                                                    ;   valid ASCII numeral, so print it on the display.
                    mov     a,#'?'                  ;1  Load the accumulator with the ASCII value of a
                                                    ;   question mark.
disp_ten_minute:    acall   send_disp_data          ;2  Send it out to the display.
                    inc     ram_ptr                 ;1  Increment the RAM pointer to the least significant
                                                    ;   minutes digit.
                    mov     a,@ram_ptr              ;1  Load the accumulator with the least significant minutes
                                                    ;   digit.
                    acall   validate_num            ;2  Ensure that the byte is a valid numeral.
                    jnz     disp_minute             ;2  If the accumulator is not zero, then the byte is a
                                                    ;   valid ASCII numeral, so print it on the display.
```

```
                        mov     a,#'?'                  ;1  Load the accumulator with the ASCII value of a
                                                        ;   question mark.
disp_minute:            acall   send_disp_data          ;2  Send it out to the display.
                        mov     a,am_pm_byte            ;1  Load the accumulator with the am_pm_byte.
                        acall   send_disp_data          ;2  Send the 'a' or the 'p' out to the display.
                        mov     a,#'m'                  ;1  Load the accumulator with the ASCII value of 'm'.
                        acall   send_disp_data          ;2  Send the 'm' out to the display.

;The check_blocked section of code is used to determine if the caller ID information was blocked by the calling
;party or if the CID information is not available (out of area).  The program first determines if there were
;fewer than 20 bytes received from the CID transmission.  If so, then the CID information is not available.
;Because the phone number and name of the calling party are not present when the call is blocked or is out
;of range, the maximum message length will not exceed 20 bytes.

;The CID protocol uses ASCII 'O' to indicate an out of range call and ASCII 'P' to indicate a private (blocked)
;call.  The ASCII value of 'O' is #04fh and the ASCII value of 'P' is #050h.  This relationship provides a
;convenient way to differentiate between the two even under high noise conditions.  Note that the lower nibble
;of ASCII 'O' is all ones and the lower nibble of 'P' is all zeros.  Additionally, the LSB of the upper nibble
;is clear for ASCII 'O' and set for ASCII 'P'.  In other words, there are a total of five bits in the status
;byte which should differ between ASCII 'O' and ASCII 'P'.  This fact will be used to drastically enhance the
;noise immunity of the differentiation.

;After complementing bit 4 of the status indicator byte, the software will differentiate between 'O' and 'P'
;by looking for more than two asserted bits in the lower five bits of the status indicator.  This process
;is illustrated below:

;                       ASCII 'O' = #04fh = #0100 1111b
;                       ASCII 'P' = #050h = #0101 0000b

;The result of complementing bit 4 of the ASCII bytes is shown below:

;                       ASCII 'O' = #04fh = #0101 1111b
;                       ASCII 'P' = #050h = #0100 0000b

;Note that the lower five bits are either all asserted or all zero.  Looking for three asserted bits in the
;lower five bits of the ASCII values (with bit 4 complemented) allows two of the five bits to be corrupted
;without affecting the outcome of the determination.  Additionally, the upper three bits are not significant
;so the net result is a noise margin of up to five bits in a single eight bit word!

;To summarize, the software will first determine if fewer than 20 bytes were received.  If so, the software
;will assume that the call was either blocked or is out of the area and proceed to check the status byte.
;After complementing bit four of the status byte, the software will look for at least three asserted bits
;in the lower five bits of the status byte.  If three asserted bits are found, the software will send the
;"out of area" message to the display.  If three asserted bits are not found, the software will send the
;"call blocked" message to the display.

check_blocked:          mov     a,byte_count            ;1  Load the total number of received bytes into the
                                                        ;   accumulator.
                        clr     c                       ;1  Explicitly clear the carry bit.
                        subb    a,#014h                 ;1  Subtract 20 from the number of received bytes.
                        jnc     get_phone_number        ;2  The carry bit will be clear if more than 20 bytes
                                                        ;   have been received, so proceed to get the phone
                                                        ;   number.
                        mov     temp,#000h              ;1  Clear the temporary register which will be used
                                                        ;   to count asserted bits.
                        mov     b,#005h                 ;2  Initialize the B register with the eight bit count
                                                        ;   down value.
                        mov     ram_ptr,#data_ram+14    ;1  Load the RAM pointer with the address of the CID
                                                        ;   absent status byte.
                        mov     a,@ram_ptr              ;1  Load the accumulator with the CID absent status
                                                        ;   byte.
                        add     a,#00010000b            ;1  Complement bit five of the status byte as discussed
                                                        ;   earlier.
                        anl     a,#00011111b            ;1  Mask-off the upper three bits.

bit_count_loop:         clr     c                       ;1  Explicitly clear the carry bit.
```

```
                        rrc     a                       ;1  Rotate the accumulator (right) into the carry bit.
                        jc      inc_bit_count           ;2  If the carry bit is set, increment the asserted bit
                                                        ;   counter
                        djnz    b,bit_count_loop        ;2  Continue until all five bits have been rotated.
                        sjmp    determine_block         ;2  Get out of the loop if five bits have been rotated.
inc_bit_count:          inc     temp                    ;1  Increment the asserted bit count register.
                        djnz    b,bit_count_loop        ;2  Continue until all five bits have been rotated.

determine_block:        mov     a,temp                  ;1  Move the asserted bit count into the accumulator.
                        clr     c                       ;1  Explicitly clear the carry bit.
                        subb    a,#003h                 ;1  Subtract the threshold from the accumulator.
                        jc      call_blocked            ;2  If the carry bit is set, then the status byte is
                                                        ;   ASCII 'P', meaning the call was blocked.
out_of_area:            mov     dptr,#area_message      ;2  Load the data pointer with the address of the
                                                        ;   "out of area" message.
                        acall   send_message            ;2  Send the "out of area" message to the display.
                        ajmp    save_screen             ;2  Skip-over the phone number and name software and
                                                        ;   write the "out of area" message to EEPROM and return
                                                        ;   to ring_check.

call_blocked:           mov     dptr,#blocked_message   ;2  Load the data pointer with the address of the
                                                        ;   "call blocked" message.
                        acall   send_message            ;2  Send the "call blocked" message to the display.
                        ajmp    save_screen             ;2  Skip-over the phone number and name software and
                                                        ;   write the "call blocked" message to EEPROM.

;The get_phone_number section of code is used to fetch the phone number information from internal RAM, format
;it, and send it out to the third line of the LCD display.  If the program reaches this label, the software
;has already determined that the call was not blocked and was not "out of area", hence, the phone number is
;assumed to be available.

;Get_phone_number starts by positioning the LCD cursor on the leftmost character of the third line.  Then,
;a left parenthesis is immediately displayed.  The area code is then retrieved from RAM and displayed.  The
;area code is terminated by displaying a right parenthesis and a space.  The first three digits of the phone
;number (the prefix) are then retrieved from RAM and displayed.  Preceded by a dash character (-), the
;remaining four digits of the phone number (the suffix) are displayed.  This phone number format requires
;14 characters on the third line.

get_phone_number:       mov     ram_ptr,#data_ram+14    ;1  Load the ram pointer with the offset of the first
                                                        ;   data byte of the phone number.
                        mov     a,#010h                 ;1  Load the accumulator with the address of the first
                                                        ;   character of the third line of the LCD display.
                        acall   set_disp_address        ;2  Set the display address.
                        mov     a,#'('                  ;1  Load the accumulator with the ASCII value of a left
                                                        ;   parenthesis.
                        acall   send_disp_data          ;2  Send the left parenthesis out to the display.
                        mov     b,#003h                 ;2  Load the B register with the area code length count
                                                        ;   down value.
area_code:              mov     a,@ram_ptr              ;1  Load the accumulator with the ASCII value of the
                                                        ;   area code digit.
                        acall   validate_char           ;2  Call the character validation subroutine to ensure
                                                        ;   that the character is printable.  This will return
                                                        ;   an asterisk if the character is invalid in case of
                                                        ;   a bit error.
                        acall   send_disp_data          ;2  Send the character out to the display.
                        inc     ram_ptr                 ;1  Increment the RAM pointer to the next location.
                        djnz    b,area_code             ;2  Get the next digit of the area code until all three
                                                        ;   digits have been displayed.
                        mov     a,#')'                  ;1  Load the accumulator with the ASCII value of a right
                                                        ;   parenthesis.
                        acall   send_disp_data          ;2  Send the right parenthesis out to the display.
                        mov     a,#020h                 ;1  Load the accumulator with the ASCII value of a space.
                        acall   send_disp_data          ;2  Send the space out to the display.

                        mov     b,#003h                 ;2  Load the B register with the prefix countdown value.
prefix:                 mov     a,@ram_ptr              ;1  Load the accumulator with the current prefix digit.
```

```
                        acall  validate_char              ;2  Call the character validation subroutine to ensure
                                                          ;   that the character is printable.  This will return
                                                          ;   an asterisk if the character is invalid in case of
                                                          ;   a checksum error.
                        acall  send_disp_data             ;2  Send the character out to the display.
                        inc    ram_ptr                    ;1  Increment the RAM pointer to the next location.
                        djnz   b,prefix                   ;2  Get the next digit of the prefix until all three
                                                          ;   digits have been displayed.
                        mov    a,#'-'                     ;1  Load the accumulator with the ASCII value of a dash.
                        acall  send_disp_data             ;2  Send the dash character out to the display.

                        mov    b,#004h                    ;2  Load the B register with the suffix countdown value.
suffix:                 mov    a,@ram_ptr                 ;1  Load the accumulator with the current suffix digit.
                        acall  validate_char              ;2  Call the character validation subroutine to ensure
                                                          ;   that the character is printable.  This will return
                                                          ;   an asterisk if the character is invalid in case of
                                                          ;   a bit error.
                        acall  send_disp_data             ;2  Send the character out to the display.
                        inc    ram_ptr                    ;1  Increment the RAM pointer to the next location.
                        djnz   b,suffix                   ;2  Get the next digit of the suffix until all four
                                                          ;   digits have been displayed.

;The get_name section of code retrieves the directory listing information from internal RAM and sends it to the
;second line of the LCD display.  Experimental characterization of the CID name information indicates that the
;name information is ALWAYS 15 characters in length, regardless of the actual length of the originator's name.
;Characters beyond the actual length of the originator's name are spaces.  This makes it quite convenient to
;display the name on a single line of the LCD display because the software can always display 15 consecutive
;characters AND the additional characters (spaces) will automatically erase unused positions on the display.

                        mov    a,#040h                    ;1  Load the accumulator with the address of the first
                                                          ;   character of the second line of the LCD display.
                        acall  set_disp_address           ;2  Position the cursor to the second line.
                        mov    ram_ptr,#data_ram+26       ;1  Load the RAM pointer with the name information offset.
                        mov    b,#00fh                    ;2  Load the B register with the 15 character countdown
                                                          ;   value
get_name:               mov    a,@ram_ptr                 ;2  Load the accumulator with the current name byte.
                        acall  validate_char              ;2  Call the character validation subroutine to ensure
                                                          ;   that the character is printable.  This will return
                                                          ;   an asterisk if the character is invalid in case of
                                                          ;   a bit error.
                        acall  send_disp_data             ;2  Send the character out to the display.
                        inc    ram_ptr                    ;1  Increment the RAM pointer to the next location.
                        djnz   b,get_name                 ;2  Get the next character in the name until all 15
                                                          ;   characters have been displayed.

;The purpose of the save_screen section of code is to write a "display image" to the EEPROM.  All pertinent
;caller ID information is displayed on the lower three lines of the LCD display.  ALL of this information,
;including blank spaces, is retrieved from the LCD display and written to the EEPROM.  Although this method
;of storage is not very efficient in terms of information per byte, it is used to avoid having to reprocess
;the information into a displayable format when it is retrieved.  Also, it simplifies the storage process by
;avoiding any compression algorithms.  The driving criteria in deciding the EEPROM storage method is to
;minimize the amount of code needed to deal with the stored data.

;Three lines of information each containing 16 characters implies that 48 bytes of EEPROM are needed for each
;call record.  Additionally, a few bytes of overhead are required to keep track of the records.  So, with a
;512 byte EEPROM, the storage is limited to the ten most recent calls.

;A total of 480 bytes are required to store a maximum of ten 48 byte records in EEPROM.  This leaves 32 bytes
;available for storage overhead.  The first sixteen EEPROM bytes (addresses 0 through 15) are used to store the
;software version string which is used to determine if the EEPROM has been initialized.  So, the EEPROM
;housekeeping data can begin at EEPROM address 16.

;Addresses 16 through 25 contain the absolute offset of each record (1 through 10) in EEPROM.  This is best
;illustrated in the following table:

;                       EEPROM address                    EEPROM data (hex)
```

```
;                       16                      0f
;                       17                      1f
;                       18                      2f
;                       19                      3f
;                       20                      4f
;                       21                      5f
;                       22                      6f
;                       23                      7f
;                       24                      8f
;                       25                      9f

;If the software needed to access record #5, it would look at EEPROM location 21 to get the offset multiplier
;for the address of record #5.  The multiplier value is stored in the lower nibble of the EEPROM data.  A value
;of #f in this nibble indicates that the record is available (not being used).  The software would then multiply
;this number by 48 to determine the absolute EEPROM address which contains record #5.  This method of storage
;minimizes the amount of software overhead required to manipulate the database.  For example, suppose record
;number 4 was deleted.  In this case, all records greater than 4 would have to be moved down within the database.
;Using the address offset table as shown above, the software only needs to "shuffle" a few bytes in the table.
;Similarly, if the number of stored calls "rolls over" beyond 10, the software only needs to change the offsets
;in the table, rather than going through the entire database and actually rearranging the records.  An offset
;value of #0nfh (where n is the address offset to be shifted into the lower nibble when the record is in use)
;indicates that the record is available.

;The EEPROM address of the stored data begins at address 32.

save_screen:        acall  init_eeprom              ;2  The init_eeprom subroutine is called to get the
                                                    ;   record information from the beginning of the
                                                    ;   EEPROM.  The number of records is returned in the
                                                    ;   accumulator.
                    cjne   a,#00ah,save_record      ;2  If there are not ten records in the EEPROM, proceed
                                                    ;   to write the new record to the EEPROM.
ten_records:        mov    a,#001h                  ;1  Load the accumulator with the number of the first
                                                    ;   record in EEPROM so that it can be deleted.
                    acall  delete_record            ;2  Delete the first record to make room for the new
                                                    ;   record.
save_record:        acall  init_eeprom              ;2  The init_eeprom subroutine is called to get the
                                                    ;   record information from the beginning of the
                                                    ;   EEPROM.  The number of records is returned in the
                                                    ;   accumulator and the EEPROM address offset table
                                                    ;   is in RAM starting at the beginning of data RAM.
                    add    a,#data_ram              ;1  Add the address of the beginning of RAM to the
                                                    ;   address offset pointer.
                    mov    ram_ptr,a                ;1  Put the RAM address offset into the RAM pointer.
                    mov    a,@ram_ptr               ;1  Load the EEPROM address offset into the accumulator.
                    anl    a,#0f0h                  ;1  Knock-off the low nibble.
                    mov    b,a                      ;1  Store the high nibble in the B register.
                    swap   a                        ;1  Swap the high and the low nibbles of the accumulator.
                    orl    a,b                      ;1  Combine the two nibbles so that they are both equal.
                    mov    @ram_ptr,a               ;1  Put the updated address offset back into the address
                                                    ;   offset table.
                    mov    record_number,a          ;1  Store the EEPROM address offset in the record_number
                                                    ;   register.
                    acall  wr_offset_table          ;2  Write the updated EEPROM address offset table back
                                                    ;   to the EEPROM using "page write"

                    mov    a,#010h                  ;1  Load the accumulator with the address of the first
                                                    ;   character of the third line of the LCD display.
                    acall  set_disp_address         ;2  Move the cursor to the third line.
                    mov    b,#010h                  ;2  Load the B register with the 16 character countdown
                                                    ;   value.
                    mov    ram_ptr,#data_ram        ;1  Load the RAM pointer with the beginning of RAM.
get_third_line:     acall  read_disp_data           ;2  Get the character from the display.
                    mov    @ram_ptr,a               ;1  Place the character in the RAM buffer.
                    inc    ram_ptr                  ;1  Increment the RAM pointer.
                    djnz   b,get_third_line         ;2  Repeat until all 16 characters have been read from the
                                                    ;   third line of LCD display and put into RAM.
```

```
                    mov    a,record_number          ;1  Load the accumulator with the EEPROM offset address
                                                    ;   multiplier.
                    anl    a,#00fh                  ;1  Knock-off the high nibble.
                    mov    record_number,a          ;1  Put it back into record_number.
                    mov    b,#030h                  ;2  Load the B register with the 48 byte multiplier value.
                    mul    ab                       ;4  Multiply the desired record address offset by the
                                                    ;   48 byte record length.  The accumulator now contains
                                                    ;   the low order byte of the absolute EEPROM address and
                                                    ;   the B register now contains the high order byte of the
                                                    ;   absolute EEPROM address.
                    add    a,#020h                  ;1  Add the EEPROM address of the first record.
                    jnc    load_address             ;2  If the carry flag is not set, then the low EEPROM
                                                    ;   address did not overflow so the high EEPROM address
                                                    ;   does not need to be incremented.
                    inc    b                        ;1  If the low EEPROM address overflowed, increment the
                                                    ;   high EEPROM address.
load_address:       mov    low_eeprom_addr,a        ;1  Put the accumulator into the low EEPROM address.
                    mov    high_eeprom_addr,b       ;1  Put the B register into the high EEPROM address.
                    mov    ram_ptr,#data_ram        ;1  Position the RAM pointer to the beginning of RAM.
                    acall  wr_eeprom_page           ;2  Write the third line of the display to the EEPROM
                                                    ;   using "page write" mode.

                    mov    a,#040h                  ;1  Load the accumulator with the address of the first
                                                    ;   character of the second line of the LCD display.
                    acall  set_disp_address         ;2  Position the cursor on the second line.
                    mov    b,#020h                  ;2  Load the B register with the 32 character countdown
                                                    ;   value.
                    mov    ram_ptr,#data_ram        ;1  Reset the RAM pointer back to the beginning.
get_second_line:    acall  read_disp_data           ;2  Get the character from the display.
                    mov    @ram_ptr,a               ;1  Place the character in the RAM buffer.
                    inc    ram_ptr                  ;1  Increment the RAM pointer.
                    djnz   b,get_second_line        ;2  Repeat until all 32 characters have been read from the
                                                    ;   LCD display and put into RAM.
                    mov    a,low_eeprom_addr        ;1  Load the low EEPROM address into the accumulator.
                    add    a,#010h                  ;1  Add 16 to the low EEPROM address to advance it to the
                                                    ;   next 16 byte page.
                    jnc    write_line_2             ;2  If the carry bit is clear, then the low EEPROM address
                                                    ;   did not overflow so the high EEPROM address does not
                                                    ;   need to be incremented.
                    inc    high_eeprom_addr         ;1  Increment the high EEPROM address if the low EEPROM
                                                    ;   address overflowed.
write_line_2:       mov    low_eeprom_addr,a        ;1  Store the new low EEPROM address in its register.
                    mov    ram_ptr,#data_ram        ;1  Position the RAM pointer to the second line of the
                                                    ;   display image.
                    acall  wr_eeprom_page           ;2  Write the line to the EEPROM.
                    mov    a,low_eeprom_addr        ;1  Load the low EEPROM address into the accumulator.
                    add    a,#010h                  ;1  Add 16 to the low EEPROM address to advance it to the
                                                    ;   next 16 byte page.
                    jnc    write_line_4             ;2  If the carry bit is clear, then the low EEPROM address
                                                    ;   did not overflow so the high EEPROM address does not
                                                    ;   need to be incremented.
                    inc    high_eeprom_addr         ;1  Increment the high EEPROM address if the low EEPROM
                                                    ;   address overflowed.
write_line_4:       mov    low_eeprom_addr,a        ;1  Store the new low EEPROM address in its register.
                    mov    ram_ptr,#data_ram+16     ;1  Position the RAM pointer to the fourth line of the
                                                    ;   display image.
                    acall  wr_eeprom_page           ;2  Write the line to the EEPROM.
                    acall  init_eeprom              ;2  Call the EEPROM initialization routine to get the
                                                    ;   number of records into the accumulator.
                    acall  disp_record              ;2  Display the record number header and redisplay the
                                                    ;   record.

;The xmit_data code segment is used to transmit the caller ID data to the cordless handset.  Since the display
;on the cordless handset can only display two lines of 16 characters, the amount of data that is transmitted
;does not include everything displayed on the base unit's four line by 16 character display.  Only the calling
```

```
;party's name and phone number are transmitted.  This information has already been formatted for 16 characters
;per line and exists in the base station's LCD DD RAM.  In other words, the second and third lines from the
;base station are transmitted directly to the handset.  The xmit_data code segment begins by loading the second
;and third lines of the base unit's LCD display into a 32 buffer in internal RAM.

;Included in the 32 data bytes is the base unit's epsilon character if a checksum error occurred during CID
;data acquisition.  This will enable the handset user to determine the integrity of the originally acquired
;data.  A separate cordless checksum indicator is used to annunciate errors caused during cordless transmission.

;The base unit is then put into a "data transmission mode" by de-asserting txd_enable.  This effectively presses
;(and holds) the page button on the base unit.  This ensures that all of the necessary base unit transmit
;circuitry is powered-up and ready to transmit data.  Additionally, any data that the base unit attempts to send
;over the wireless channel is inhibited.  This enables the xmit_data port of the CPU to transmit its own data in
;place of the normal cordless "page" data.

;After placing the base unit into the data transmission mode, a 25Hz square wave is sent over the wireless
;channel via xmit_data.  This is necessary to guaranty that the remote processor "wakes-up" because the cordless
;handset polls the wireless channel only once every 3.5 seconds.  Sending 5 seconds of signal will ensure that
;the signal is detected by the cordless handset (providing it is turned-on and within range).

;After the cordless handset receives one of the 20ms (25Hz) pulses, the remote power supply is turned-on and the
;remote processor powers-up and readies itself to receive the caller ID data.  This process requires far less
;than the allotted five seconds.  The five second signal is more than adequate to ensure that the remote +5V
;power supply stabilizes and the processor is ready to receive data.

;The five seconds of 25Hz square wave is immediately followed by 33.3ms of 1500Hz signal.  This signal is used
;to help synchronize the handset unit to the data and therefore increase noise margins.  The frequency of this
;signal is near the upper limit of the cordless bandwidth and was chosen so that high frequency noise (static
;on the cordless channel) would play a less significant role in the data acquisition.  The 33.3ms of 1.5KHz
;is immediately followed by a "dummy" byte (#000h) which is sent to aid in receiver synchronization.  This
;"dummy" byte has no data significance and is eventually discarded by the receiver.

;The PWM output of the CPU is used for data transmission for the following two reasons.  First, the PWM is
;"self running" in that it requires very little software overhead to generate a known frequency.  Secondly,
;when in PWM mode, the output is actively pulled high, which enables the output signal to symmetrically
;capacitively couple through C7 (see schematic).

;The data is transmitted using FSK at 300bps.  The frequency of the space signal (logical zero) is 900Hz and the
;frequency of the mark signal (logical one) is 300Hz.  These frequencies were chosen to enable the remote
;processor (which runs at only 6MHz) plenty of margin in discriminating between zeros and ones.  Additionally,
;the 900Hz is well within the bandwidth limit of the cordless channel.

;There are two lines of 16 characters which need to be transmitted to the cordless handset.  Additionally, a
;checksum byte is sent to ensure data integrity.  So, a total of 33 data bytes need to be sent.

;Prior to sending the data, the second and third lines of the LCD display are loaded into data RAM.  This is
;necessary because the display may take too long to retrieve its data in a real-time transmission mode.

send_cordless:      mov   pwcm,#080h              ;2  Load the PWM duty cycle SFR with a 50% value.
                    mov   checksum,#000h          ;2  Clear the checksum word.
                    mov   ram_ptr,#data_ram       ;1  Position the RAM pointer to the beginning of RAM.
load_disp_data:     mov   line_count,#002h        ;2  Load the line_count location with a 2 line countdown
                                                  ;   value.
                    mov   a,#040h                 ;1  Load the accumulator with the address of the second
                                                  ;   line of the LCD display.
get_line:           acall set_disp_address        ;2  Position the cursor on the second line.
                    mov   b,#010h                 ;2  Load the B register with the 16 character countdown
                                                  ;   value.
get_disp_char:      acall read_disp_data          ;2  Get the ASCII value of the current character on the
                                                  ;   base unit's LCD display.
                    mov   @ram_ptr,a              ;1  Put the data byte into the current RAM location.
                    mov   temp,a                  ;1  Temporarily store the display byte.
                    mov   a,checksum              ;1  Load the checksum into the accumulator.
                    add   a,temp                  ;1  Add the current character to the checksum.
                    mov   checksum,a              ;1  Put the new checksum back into the checksum location.
                    inc   ram_ptr                 ;1  Increment the RAM pointer to the next location.
```

```
                    djnz   b,get_disp_char             ;2  Get the next display byte until all 16 have been
                                                      ;   loaded into RAM.
                    mov    a,#010h                    ;1  Load the accumulator with the address of the third
                                                      ;   line on the LCD display.
                    djnz   line_count,get_line        ;2  Get the third line of the LCD display and put it into
                                                      ;   RAM.
                    mov    @ram_ptr,checksum          ;2  Put the checksum into the 33rd RAM location.

transmit_data:      setb   txd_enable                 ;1  Disable the base unit's data and enable the CPU
                                                      ;   to transmit data over the cordless channel.

                    mov    byte_count,#0fah           ;1  Load the byte_count register with a 250 pulse
                                                      ;   countdown value.
send_20ms_pulse:    mov    tcon,#timer_stop           ;2  Stop the timer and clear TF.
                    mov    th,#0b1h                   ;2  Initialize the high timer byte to expires after 20ms.
                    mov    tl,#0dfh                   ;2  Initialize the low timer byte to expire after 20ms.
                    mov    tcon,#timer_start          ;2  Start the timer.
                    cpl    xmit_data                  ;1  Complement the xmit_data bit.
                    jnb    tf,$                       ;2  Loop until the timer expires.
                    djnz   byte_count,send_20ms_pulse ;2  Send another pulse until 250 20ms pulses have been
                                                      ;   sent.

                    mov    tcon,#timer_stop           ;2  Stop the timer and clear TF.
                    mov    tl,#0cah                   ;2  Initialize the low timer byte to expire after 33.3ms.
                    mov    th,#07dh                   ;2  Initialize the high timer byte to expires after 33.3ms.
                    mov    pwmp,#00fh                 ;2  Load the PWM prescaler with the 1.5KHz division value.
                    mov    pwena,#001h                ;2  Start the PWM output.
                    mov    tcon,#timer_start          ;2  Start the timer.
                    jnb    tf,$                       ;2  Loop until the 33.3ms timer expires.

send_dummy_byte:    mov    a,#000h                    ;1  Load the accumulator with the "dummy" byte.
                    acall  xmit_data_byte             ;2  Send the "dummy" byte to get things synchronized on
                                                      ;   the handset.

                    mov    ram_ptr,#data_ram          ;2  Position the RAM pointer to the beginning of RAM.
                    mov    b,#020h                    ;2  Load the B register with the 32 byte countdown value.
send_data:          mov    a,@ram_ptr                 ;1  Load the current data byte into the accumulator.
                    acall  xmit_data_byte             ;2  Send the current display character over the cordless
                                                      ;   channel.
                    inc    ram_ptr                    ;1  Increment the RAM pointer to the next location.
                    djnz   b,send_data                ;2  Loop until all 32 data bytes have been sent.
                    mov    pwena,#000h                ;2  Stop the PWM.
                    clr    txd_enable                 ;1  Release control of the "page" button.
                    ajmp   ring_check                 ;2  Go back to ring_check to restart the program and
                                                      ;   poll the function buttons.

;This is the end of the main program.
;-------------------------------------------------------------------------------------------------------------


;*************************** Cordless data byte transmission subroutines ************************************

;                   xmit_data_byte: used to send a full data byte over the cordless channel.
;                   xmit_data_bit: called by xmit_data_byte to send individual bits over the cordless channel.

;*************************************************************************************************************


;Subroutine: xmit_data_byte ----------------------------------------------------------------------------------

;The purpose of the xmit_data_byte subroutine is to transmit a full eight bit data byte over the cordless
;channel.  This subroutine also takes care of sending a leading start bit (900Hz) and a trailing stop bit
;(300Hz).

;The data byte to be transmitted must be in the accumulator and the txd_enable bit must be set upon entry.
;The subroutine sends a start bit (space) over the cordless channel, followed by the eight data bits in the
```

```
;accumulator, and terminates the byte with a stop bit (mark),

;Registers used:      Accumulator, bit_count
;Registers affected:  none
;SFRs used:           none
;Bits used:           c
;Calls:               xmit_data_bit

xmit_data_byte:       mov   bit_count,#008h          ;1  Load the bit_count register with the eight bit
                                                     ;   countdown value.
send_start_bit:       clr   c                        ;1  Clear the carry bit to send a logical 0 (start bit).
                      acall xmit_data_bit            ;2  Transmit the start bit.
send_byte:            rrc   a                        ;1  Rotate the next LSB into the carry bit.
                      acall xmit_data_bit            ;2  Transmit the current data bit.
                      djnz  bit_count,send_byte      ;2  Continue to transmit data bits until the entire byte
                                                     ;   has been sent.
send_stop_bit:        setb  c                        ;1  Set the carry bit to send a logical 1 (stop bit).
                      acall xmit_data_bit            ;2  Transmit the stop bit.
                      ret                            ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------

;Subroutine: xmit_data_bit ----------------------------------------------------------------------------------

;The purpose of the xmit_data_bit subroutine is to transmit a single data bit over the cordless channel.  The
;frequency of a logical zero (space) is 900Hz and the frequency of a logical one (mark) is 300Hz.  The data
;rate is 300BPS, which implies a bit time of 3.33ms.

;Upon entry, the carry bit must reflect the bit to be sent (i.e. set for a logical 1 and clear for a logical
;zero).  This subroutine uses the PWM output to achieve the FSK frequencies.  The PWM prescaler is loaded
;with the appropriate divider value while the timer is used to clock the bit time.  The PWM output must
;be enabled prior to calling this subroutine.  Furthermore, txd_enable must be set by the calling code.

;Registers used:      none
;Registers affected:  none
;SFRs used:           tl, th, tcon, pwmp
;Bits used:           c
;Calls:               none


xmit_data_bit:        mov   tcon,#timer_stop         ;2  Stop the timer and clear TF.
                      mov   tl,#0fah                 ;2  Initialize the low timer byte to expire after 3.33ms.
                      mov   th,#0f2h                 ;2  Initialize the high timer byte to expires after 3.33ms.
                      jc    send_one                 ;2  Send a logical one if the carry bit is set.
send_zero:            mov   pwmp,#019h               ;2  Load the PWM prescaler with the 900Hz division value
                                                     ;   for a logical 0.
                      sjmp  xmit_bit                 ;2  Send the data bit.
send_one:             mov   pwmp,#04dh               ;2  Load the PWM prescaler with the 300Hz division value
                                                     ;   for a logical 1.
xmit_bit:             mov   tcon,#timer_start        ;2  Start the timer.
                      jnb   tf,$                     ;2  Loop until the timer expires (the bit time).
                      ret                            ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------


;Fixed messages ---------------------------------------------------------------------------------------------
;The following ASCII text is used for standard messages which may be displayed on the LCD display.  The data is
;fixed in ROM.

version_string:       db    ' CCID Base v1.5 '       ;Must be exactly sixteen characters in length.
fatal_message:        db    'Fatal data error'       ;Must be exactly sixteen characters in length.
area_message:         db    '  Out of area   '       ;Must be exactly sixteen characters in length.
blocked_message:      db    '  Call blocked  '       ;Must be exactly sixteen characters in length.
no_records_msg:       db    ' No call record '       ;Must be exactly sixteen characters in length.
iic_error_msg:        db    'IIC error code  '       ;Must be exactly eight characters in length.
record_message:       db    'Record '                ;Must be exactly seven characters in length.
;-------------------------------------------------------------------------------------------------------------
```

```
;Lookup table: months ---------------------------------------------------------------------------------------

;The months lookup table provides the three letter abbreviations for the months.  When accessing the lookup
;table, the software must reference the months label and provide an offset which is equal to the month number
;to obtain the proper text.  The first three characters in the lookup table are used to annunciate a month
;error and have an offset of zero.

months:               db     '???JanFebMarAprMayJunJulAugSepOctNovDec'
;-----------------------------------------------------------------------------------------------------------



;******************************** Display & formatting subroutines ********************************************

;                          init_display: used to initialize and clear the display after power-up.
;                          send_command: used to send a command byte to the display.
;                          send_disp_data: used to send a data byte to the display.
;                          set_disp_address: used to set the position of the cursor on the display.
;                          read_disp_data: used to read data from the display.
;                          disp_busy: used to check the busy status of the display.
;                          validate_char: used to validate received ASCII characters.
;                          validate_num: used to check the validity of ASCII numerals.
;                          hex_to_ascii: used to convert a hexadecimal number to its decimal ASCII components.
;                          send_message: used to send a fixed 16 byte message to line 2 of the display.

;***********************************************************************************************************


;Subroutine: init_display ----------------------------------------------------------------------------------

;According to the HD44780 datasheet, the display and controller hardware requires 10ms after power-up to
;guaranty a successful internal power-on reset.  The software will wait for 15ms to provide a few extra
;milliseconds of margin.  Since 15ms requires 15,000 (#3a98h) machine cycles with a 12MHz oscillator,
;the internal 16 bit timer will be used to achieve the delay.  The timer bytes are initialized with
;#ffffh-#3a98h = #c567h so that the timer overflow flag will be set when 15,000 machine cycles have elapsed.
;After the initial power-up delay, an HD44780 "function set" command is sent to the display to configure
;it for an 8 bit data interface.  The HD44780 requires a maximum of 120us to complete this command.  The
;display's busy flag cannot be checked during this operation, so the send_command subroutine is not used.
;Instead, the "function set" command is "hard sent".  All subsequent operations can use the busy flag.

;Registers used:      Accumulator, disp_port
;Registers affected:  a, disp_port
;SFRs used:           tl, th, tcon, tf
;Bits used:           disp_reg_sel, disp_read, disp_strobe
;Calls:               send_command


init_display:         mov    disp_port,#0ffh            ;2  Set all display data bits high so they cannot sink
                                                        ;   current.
                      clr    disp_strobe                ;1  Set the display strobe low.
                      mov    tl,#067h                   ;2  Load the low timer byte with the start value.
                      mov    th,#0c5h                   ;2  Load the high timer byte with the start value.
                      mov    tcon,#timer_start          ;2  Start the timer.
init_loop:            jnb    tf,init_loop               ;2  Loop until the timer overflow flag is set.
                      mov    tcon,#timer_stop           ;2  Stop the timer and clear the overflow flag.
                      clr    disp_reg_sel               ;1  Clear the display register select bit to indicate
                                                        ;   that a command is going to be sent.
                      clr    disp_read                  ;1  Clear the display read bit to indicate that data
                                                        ;   is going to be written.
                      mov    disp_port,#00111000b       ;2  Load the display port with the display function
                                                        ;   set data. (8 bit interface, 2 lines, 5x7 font)
                      setb   disp_strobe                ;1  Set the display enable bit.
                      nop                               ;1  Wait for one instruction cycle for timing margin.
                      nop                               ;1  Wait for one instruction cycle for timing margin.
```

```
                    clr   disp_strobe               ;1  Clear the display enable bit to provide a falling
                                                    ;   edge.  This clocks the data into the display.
                    mov   a,#028h                   ;1  Load the accumulator with a 120us delay value.
init_delay:         dec   a                         ;1  Decrement the accumulator.
                    jnz   init_delay                ;2  Loop until the accumulator reaches zero.

;After the initialization delays, the display is turned-on and cleared and the cursor (display RAM pointer)
;is placed in the home position.

                    mov   a,#00001100b              ;1  Load the accumulator with the display on/off
                                                    ;   command. (display on, cursor off, no blink)
                    acall send_command              ;2  Send the command out to the display.
                    mov   a,#00000110b              ;1  Load the accumulator with the entry mode byte.
                                                    ;   (increment cursor, no scroll)
                    acall send_command              ;2  Send the command out to the display.
                    mov   a,#00000001b              ;1  Load the accumulator with the clear display
                                                    ;   and home cursor command.
                    acall send_command              ;2  Send the command out to the display.
                    ret                             ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------

;Subroutine: send_command -----------------------------------------------------------------------------------

;The purpose of the send_command subroutine is to send a command word to the LCD display.  The command
;word must be in the accumulator prior to calling this subroutine.  This subroutine operates in the
;"closed loop" mode by ensuring that the HD44780 is not busy prior to sending the command.  Also, the
;"closed loop" mode enables the software to send commands to the display as rapidly as possible as opposed
;to using fixed worst-case delays.  The accumulator is not affected by this subroutine.

;Registers used:      Accumulator, disp_port
;Registers affected:  disp_port
;SFRs used:           none
;Bits used:           disp_reg_sel, disp_read, disp_strobe
;Calls:               disp_busy

send_command:       acall disp_busy                 ;2  Make sure the display is not busy.
                    clr   disp_reg_sel              ;1  Clear the display register select bit to indicate
                                                    ;   that a command is going to be sent.
                    clr   disp_read                 ;1  Clear the display read bit to indicate that data
                                                    ;   is going to be written.
                    mov   disp_port,a               ;1  Copy the accumulator contents to the display
                                                    ;   data port.
                    nop                             ;1  Wait for one machine cycle to provide timing margin.
                    setb  disp_strobe               ;1  Set the display enable bit.
                    nop                             ;1  Wait for one machine cycle to provide timing margin.
                    clr   disp_strobe               ;1  Clear the display enable bit to provide a falling
                                                    ;   edge.  This clocks the data into the display.
                    ret                             ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------

;Subroutine: send_disp_data ---------------------------------------------------------------------------------

;The purpose of the send_disp_data subroutine is to send a character to the LCD display.  The ASCII value
;of the character must be in the accumulator prior to calling this subroutine.  The character is displayed
;at the current DD RAM location internal to the HD44780.  This subroutine operates in the "closed loop" mode by
;ensuring that the HD44780 is not busy prior to sending the data.  Also, the "closed loop" mode enables the
;software to send commands to the display as rapidly as possible as opposed to using fixed worst-case delays.
;The accumulator is not affected by this subroutine.

;Registers used:      Accumulator, disp_port
;Registers affected:  disp_port
;SFRs used:           none
;Bits used:           disp_reg_sel, disp_read, disp_strobe
;Calls:               disp_busy

send_disp_data:     acall disp_busy                 ;2  Make sure the display is not busy.
```

```
                    setb   disp_reg_sel               ;1  Set the display register select bit to indicate
                                                      ;   that a data word is going to be sent.
                    clr    disp_read                  ;1  Clear the display read bit to indicate that data
                                                      ;   is going to be written.
                    mov    disp_port,a                ;1  Copy the accumulator contents to the display
                                                      ;   data port.
                    nop                               ;1  Wait for one machine cycle to provide timing margin.
                    setb   disp_strobe                ;1  Set the display enable bit.
                    nop                               ;1  Wait for one machine cycle to provide timing margin.
                    clr    disp_strobe                ;1  Clear the display enable bit to provide a falling
                                                      ;   edge.  This clocks the data into the display.
                    ret                               ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------

;Subroutine: set_disp_address ---------------------------------------------------------------------------

;The purpose of the set_disp_address subroutine is to move the HD44780's internal DD RAM pointer to a specific
;location.  The desired location must be in the accumulator prior to calling this subroutine.  This subroutine
;operates in the "closed loop" mode by ensuring that the display is not busy prior to setting the address.
;Also, the "closed loop" mode enables the software to send commands to the display as rapidly as possible as
;opposed to using fixed worst-case delays.

;Registers used:      Accumulator
;Registers affected:  Accumulator
;SFRs used:           none
;Bits used:           none
;Calls:               send_command, disp_busy

set_disp_address:    acall  disp_busy                  ;2  Make sure the display is not busy.
                     orl    a,#10000000b               ;1  Set the MSB of the accumulator to indicate to the
                                                       ;   display that the address pointer is going to be
                                                       ;   modified.  (the display address is contained in the
                                                       ;   lower seven bits of the accumulator)
                     acall  send_command               ;2  Send the new address pointer to the display.  This is
                                                       ;   a nested subroutine call.
                     ret                               ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------

;Subroutine: read_disp_data -----------------------------------------------------------------------------

;The purpose of the read_disp_data subroutine is to read a character from the LCD display.  The ASCII value
;of the character is read from the current DD RAM location internal to the HD44780 and is returned in the
;accumulator.  This subroutine operates in the "closed loop" mode by waiting for the HD44780 to finish an
;operation before getting the data.

;Registers used:      Accumulator, disp_port
;Registers affected:  Accumulator, disp_port
;SFRs used:           none
;Bits used:           disp_reg_sel, disp_read, disp_strobe
;Calls:               disp_busy

read_disp_data:      acall  disp_busy                  ;2  Make sure the display is not busy.
                     mov    disp_port,#0ffh            ;1  Set all disp_port bits high to configure them as
                                                       ;   inputs.
                     setb   disp_reg_sel               ;1  Set the display register select bit to indicate
                                                       ;   that a data word is going to be read.
                     setb   disp_read                  ;1  Set the display read bit to indicate that data
                                                       ;   is going to be read.
                     nop                               ;1  Wait for one machine cycle to provide timing margin.
                     setb   disp_strobe                ;1  Set the display enable bit.
                     nop                               ;1  Wait for one machine cycle to provide timing margin.
                     mov    a,disp_port                ;1  Copy the display data into the accumulator.
                     clr    disp_strobe                ;1  Clear the display enable bit.
                     ret                               ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------
```

```
;Subroutine: disp_busy ----------------------------------------------------------------------------------

;The purpose of the disp_busy subroutine is to ensure that the HD44780 is not busy.  If it is busy, the
;subroutine will wait until it is not.  This subroutine is called by any display communication routines
;prior to sending or receiving data.

;Registers used:      disp_port
;Registers affected:  disp_port
;SFRs used:           none
;Bits used:           disp_reg_sel, disp_read, disp_strobe
;Calls:               none

disp_busy:            mov   disp_port,#0ffh          ;1  Set the display port bit high to configure them
                                                     ;   as inputs and prevent them from sinking current.
                      clr   disp_reg_sel             ;1  Clear the display register select bit to indicate
                                                     ;   that the busy flag is going to be read.
                      setb  disp_read                ;1  Set the display read bit to indicate that a read
                                                     ;   operation is going to take place.
                      setb  disp_strobe              ;1  Set the display strobe.
                      nop                            ;1  Wait one machine cycle to provide timing margin.
busy:                 jb    disp_port.7,busy         ;2  Loop the busy flag is clear.
                      clr   disp_strobe              ;1  Clear the display strobe.
                      ret                            ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------

;Subroutine: validate_char ------------------------------------------------------------------------------

;The purpose of the validate_char subroutine is to determine whether or not a character value is a valid
;ASCII character generated by caller ID.  Since caller ID is only capable of sending uppercase letters A
;through Z, numerals zero through 9, and spaces, any other character value will be considered invalid.  In
;other words, a value other than the aforementioned was most likely corrupted during the data acquisition
;sequence.  It should be noted, however, that this method of determining character validity is by no means
;bullet proof.  It is possible to have one or more corrupted bits in an ASCII value and still have a valid
;ASCII character.  This subroutine is called as an error minimization function.

;If the character is determined to be invalid, it will be replaced with an asterisk (*) and returned via
;the accumulator.  The calling code segment will the send the asterisk to the LCD display which will inform
;the user that the character has been corrupted.

;Upon entering this subroutine, the accumulator must have the ASCII value to be checked.  When the subroutine
;is finished, the accumulator will contain the original value if it is a valid caller ID character or an
;asterisk if not.

;Registers used:      Accumulator, original_value
;Registers affected:  Accumulator (if data is invalid), original_value
;SFRs used:           none
;Bits used:           c
;Calls:               none

validate_char:        mov   original_value,a         ;1  Copy the original character value into a temporary
                                                     ;   holding register.
                      clr   c                        ;1  Explicitly clear the carry bit.
                      subb  a,#020h                  ;1  Subtract the ASCII value of a space from the
                                                     ;   accumulator.
                      jc    invalid_char             ;2  If the carry bit is set, the value was less than
                                                     ;   #020h and hence, is invalid.
                      jz    valid_char               ;2  If the accumulator is zero, then the character
                                                     ;   is a space and is valid.
                      subb  a,#010h                  ;1  Subtract the difference between ASCII zero and ASCII
                                                     ;   space.
                      jc    invalid_char             ;2  If the original value was less than ASCII zero
                                                     ;   (#030h) and not a space, then it was invalid.
                      subb  a,#00ah                  ;1  Subtract one more than the difference between ASCII
                                                     ;   nine and ASCII zero.
                      jc    valid_char               ;2  If the carry bit is set, then the character is a
                                                     ;   numeral.
```

```
                        subb   a,#007h                  ;1  Subtract the difference between an ASCII colon (:)
                                                        ;   and ASCII "A".
                        jc     invalid_char             ;2  If the carry bit is set, then the character is between
                                                        ;   ASCII colon and ASCII ampersand and is therefore
                                                        ;   invalid.
                        subb   a,#01ah                  ;1  Subtract the difference between the next ASCII value
                                                        ;   greater than "Z" and "A".
                        jnc    invalid_char             ;2  If the carry bit is clear, then the value was
                                                        ;   greater than "Z" and hence invalid.
valid_char:             mov    a,original_value         ;1  Since the character was determined to be valid,
                                                        ;   move the original character into the accumulator.
                        ret                             ;2  Return to the calling address with the valid character
                                                        ;   in the accumulator.
invalid_char:           mov    a,#02ah                  ;1  Since the character was determined to be invalid,
                                                        ;   place the ASCII value for an asterisk (*) in the
                                                        ;   accumulator.
                        ret                             ;2  Return to the calling address with an asterisk in the
                                                        ;   accumulator.
;-------------------------------------------------------------------------------------------------------------

;Subroutine: validate_num -----------------------------------------------------------------------------------

;The purpose of the validate_num subroutine is to determine whether or not a character value is a valid
;ASCII numeral generated by caller ID.  This subroutine is used to ensure that the date and time code
;segments process only valid ASCII numerals (i.e. 0 through 9).  If the character is determined to be invalid,
;then a zero is returned in the accumulator.  If the character is valid, then its ASCII value is returned in the
;accumulator.  It should be noted, however, that this method of determining numeral validity is by no means
;bullet proof.  It is possible to have one or more corrupted bits in an ASCII numeral and still have a valid
;ASCII numeral.  This subroutine is called as an error minimization function.

;Upon entering this subroutine, the accumulator must have the ASCII value to be checked.  When the subroutine
;is finished, the accumulator will contain the original value if it is a valid caller ID numeral or zero
;if not.

;Registers used:      Accumulator
;Registers affected:  Accumulator (if data is invalid)
;SFRs used:           none
;Bits used:           c
;Calls:               none

validate_num:           clr    c                        ;1  Explicitly clear the carry bit.
                        subb   a,#030h                  ;1  Subtract the ASCII value of a zero from the
                                                        ;   accumulator.
                        jc     invalid_num              ;2  If the carry bit is set, the value was less than
                                                        ;   #030h and hence, is invalid.
                        add    a,#0f6h                  ;1  Add decimal 246 to set the carry bit if the ASCII
                                                        ;   value of the character is greater than that of ASCII
                                                        ;   nine.
                        jnc    valid_num                ;2  The carry flag will be clear if the character is
                                                        ;   between zero and nine, inclusive.
invalid_num:            mov    a,#000h                  ;1  Load the accumulator with zero because the character
                                                        ;   is not a valid ASCII numeral.
                        ret                             ;2  Return to the calling address with zero in the
                                                        ;   accumulator.
valid_num:              subb   a,#0c6h                  ;1  Subtract the difference between decimal 246 and the
                                                        ;   ASCII value of zero to obtain the original ASCII
                                                        ;   value of the valid numeral.
                        ret                             ;2  Return to the calling address with the valid ASCII
                                                        ;   numeral in the accumulator.
;-------------------------------------------------------------------------------------------------------------

;Subroutine: hex_to_ascii -----------------------------------------------------------------------------------

;The purpose of the hex_to_ascii subroutine is to convert a hexadecimal value into two ASCII bytes.  The
;hexadecimal value is contained in the accumulator upon entry.  Upon exit, ls_digit contains the ASCII
;value of the least significant digit and ms_digit contains the most significant (tens) digit.  This routine
```

```
;uses the Decimal Adjust instruction to simplify the conversion.  For example, if the accumulator contains
;#032h (#050d) on entry, ms_digit will contain #035h (ASCII "5") and ls_digit will contain #030h (ASCII 0)
;upon exit.  Similarly, if the accumulator contains #00fh (#015d) on entry, ms_digit will contain #031h
;(ASCII "1") and ls_digit will contain #035h (ASCII "5") upon exit.

;Registers used:      Accumulator, ls_digit, ms_digit, b
;Registers affected:  Accumulator, ls_digit, ms_digit, b
;SFRs used:           none
;Bits used:           c, ac
;Calls:               none

hex_to_ascii:        mov   b,a                      ;1  Copy the accumulator into the B register.
                     mov   a,#000h                  ;1  Clear the accumulator.

count_loop:          clr   c                        ;1  Explicitly clear the carry bit.
                     clr   ac                       ;1  Explicitly clear the AC bit.
                     inc   a                        ;1  Increment the accumulator.
                     da    a                        ;1  Decimal adjust the accumulator.
                     djnz  b,count_loop             ;2  Repeat the process until the original hexadecimal
                                                    ;   value is counted down.

;The accumulator now contains a decimal adjusted value representing the lower two decimal digits in BCD format.
;If the original hexadecimal value was greater than #063h (#099d), then the carry flag will be set.  This
;subroutine only converts the tens and ones digits into BCD format.  Higher order digits are ignored.

                     mov   ls_digit,a               ;1  Copy the accumulator into ls_digit.
                     swap  a                        ;1  Swap the accumulator nibbles to place the most
                                                    ;   significant BCD digit in the low nibble.
                     mov   ms_digit,a               ;1  Copy the accumulator into ms_digit.
                     mov   a,ls_digit               ;1  Copy the BCD value of ls_digit into the accumulator
                                                    ;   to prepare for addition,
                     anl   a,#00fh                  ;1  Knock-off the high nibble of ls_digit.
                     add   a,#030h                  ;1  Add the ASCII value of zero to ls_digit.
                     mov   ls_digit,a               ;1  Put the ASCII value of the least significant digit
                                                    ;   into ls_digit.
                     mov   a,ms_digit               ;1  Copy the BCD value of ms_digit into the accumulator
                                                    ;   to prepare for addition.
                     anl   a,#00fh                  ;1  Knock-off the high nibble of ms_digit.
                     add   a,#030h                  ;1  Add the ASCII value of zero to ms_digit.
                     mov   ms_digit,a               ;1  Put the ASCII value of the most significant digit
                                                    ;   into ms_digit.
                     ret                            ;2  Return to the calling address with the ones digit
                                                    ;   in ls_digit and the tens digit in ms_digit, both
                                                    ;   in ASCII format.
;-------------------------------------------------------------------------------------------------------------

;Subroutine: send_message ------------------------------------------------------------------------------------

;The purpose of the send_message subroutine is to send a 16 character message to the second line of the LCD
;display.  On entry, the data pointer (DPTR) must contain the address of the message to be sent.  This
;subroutine assumes that exactly 16 characters are to be sent.

;Registers used:      Accumulator, b, temp
;Registers affected:  Accumulator, b, temp
;SFRs used:           none
;Bits used:           none
;Calls:               send_disp_data, set_disp_address

send_message:        mov   b,#010h                  ;2  Load the B register with the 16 character countdown
                                                    ;   value.
                     mov   temp,#000h               ;1  Initialize the temporary register with the offset
                                                    ;   value.
                     mov   a,#040h                  ;1  Load the accumulator with the address of the second
                                                    ;   line of the LCD display.
                     acall set_disp_address         ;2  Set the display address pointer.
```

```
message_loop:         mov    a,temp                    ;1  Load the accumulator with the offset value.
                      movc   a,@a+dptr                 ;2  Load the next message byte to be sent into the
                                                       ;   accumulator.
                      acall  send_disp_data            ;2  Send it out to the display.
                      inc    temp                      ;1  Increment the data pointer offset value.
                      djnz   b,message_loop            ;2  Repeat until all 16 bytes have been sent.
                      ret                              ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------


;*********************************** EEPROM & IIC subroutines ************************************************

;                          init_eeprom: used to determine if the EEPROM has been used before.
;                          wr_eeprom_byte: used to write a single byte to the EEPROM.
;                          wr_offset_table: used to write the EEPROM address offset table to the EEPROM.
;                          wr_eeprom_page: used to write 16 bytes to the EEPROM in fast "page mode".
;                          delete_record: used to delete a single record from the EEPROM.
;                          disp_record: used to retrieve and display a designated record.
;                          send_iic_byte: used to send a data byte over the IIC bus.
;                          send_eeprom_addr: used to send the EEPROM device address over the IIC bus.
;                          send_iic_stop: used to generate an IIC stop condition.
;                          rd_eeprom_byte: used to read a single byte from the EEPROM.
;                          iic_mater_req: used to request mastership of the IIC bus.
;                          get_offset_table: used to retrieve the EEPROM address offset table and place it in RAM.
;                          rd_eeprom_page: used to read 16 bytes from the EEPROM.
;                          iic_error: used to annunciate IIC errors.

;*************************************************************************************************************


;Subroutine: init_eeprom ----------------------------------------------------------------------------------

;The purpose of the init_eeprom subroutine is to determine if the EEPROM has been used in this application
;before.  If not, the EEPROM is initialized by writing the version string to the first 16 bytes and writing
;#0nfh, where n is the EEPROM address offset that will be used to write data to the record, to addresses 16
;through 32.  The high nibble of each address offset (n) is initialized with the record number and the low
;nibble is initialized to #f to indicate that the record is available.  In this case, zero is returned in the
;accumulator to indicate that there are no records stored in EEPROM.

;If the EEPROM has been used in this application before (i.e. the first 16 bytes match version_string),
;the number of valid data records (records whose address offsets are NOT #0nfh) is returned in the accumulator.
;This number also represents the last stored record so the caller of the subroutine can display it.  The
;EEPROM record offset table is left in the lower 16 bytes of RAM on exit.

;Registers used:      accumulator, high_eeprom_address, low_eeprom_address, ram_ptr, b, dptr, temp,
;                     record_number
;Registers affected:  accumulator, high_eeprom_address, low_eeprom_address, ram_ptr, b, dptr, temp,
;                     record_number
;SFRs used:           none
;Bits used:           c
;Calls:               rd_eeprom_page, get_offset_table, wr_eeprom_page, wr_offset_table

init_eeprom:          mov    high_eeprom_addr,#000h    ;1  Load the high EEPROM address in preparation to
                                                       ;   read the first 16 EEPROM bytes.
                      mov    low_eeprom_addr,#000h     ;1  Load the low EEPROM address in preparation to
                                                       ;   read the first 16 EEPROM bytes.
                      mov    ram_ptr,#data_ram         ;1  Load the RAM pointer with the beginning address
                                                       ;   of data RAM to store the EEPROM data.
get_ver_string:       acall  rd_eeprom_page            ;2  Read the first 16 bytes of EEPROM into RAM starting
                                                       ;   at data_ram.
                      mov    ram_ptr,#data_ram         ;1  Reset the RAM pointer to the beginning of data RAM.
                      mov    b,#010h                   ;2  Load the B register with the 16 byte countdown value.
                      mov    dptr,#version_string      ;2  Load the 16 bit data pointer with the address of
                                                       ;   the version string.
                      mov    temp,#000h                ;1  Clear the temporary register which is used to store
                                                       ;   the offset for the MOVC instruction.
```

```
;At this point the lower 16 bytes of data ram contain an image of the first 16 bytes in the EEPROM.
;The software subtracts the current byte of the version_string from the corresponding byte in the EEPROM image.
;If the result is not zero, then the EEPROM has an incorrect (or non-existent) version string and needs to
;be initialized under this version of software.  If the result is zero, then the two bytes are equal and the
;software proceeds to check the next byte until all 16 version_string bytes have been tested.

check_version:      mov   a,temp                    ;1  Load the accumulator with the offset value.
                    movc  a,@a+dptr                 ;2  Load the next version string byte into the accumulator.
                    clr   c                         ;1  Explicitly clear the carry bit.
                    subb  a,@ram_ptr                ;1  Subtract the value that was loaded into RAM from the
                                                    ;   EEPROM.
                    jnz   new_eeprom                ;2  If the result of the subtraction is not zero, then
                                                    ;   the current byte of the version string does not
                                                    ;   match the corresponding byte retrieved from the
                                                    ;   EEPROM, thus indicating that the EEPROM has not been
                                                    ;   used under this software version before.
                    inc   temp                      ;1  Increment the data pointer offset value.
                    inc   ram_ptr                   ;1  Increment the RAM pointer.
                    djnz  b,check_version           ;2  Repeat until all 16 bytes have been checked.

;If the first 16 bytes in the EEPROM match version_string, then the EEPROM has been initialized under the
;current version of software.  In this case, the old_eeprom section of code loads the accumulator with the
;number of records in the EEPROM.  Since the lower 16 bytes of RAM contain an image of the first 16 bytes of
;EEPROM, the entire EEPROM address offset table is not in RAM.  The following section of code loads the
;address offset table from the EEPROM and places it in the lower 16 bytes of RAM.  However, only the
;first ten bytes are significant.

old_eeprom:         acall get_offset_table          ;2  Read the address offset table from EEPROM into RAM.
                    mov   ram_ptr,#data_ram         ;1  Position the RAM pointer to the beginning of RAM.
                    mov   b,#00ah                   ;2  Load the B register with the 10 record countdown
                                                    ;   value.
                    mov   record_number,#000h       ;1  Initialize the record counter to zero.
count_records:      mov   a,@ram_ptr                ;1  Load the accumulator with the address offset of the
                                                    ;   record pointed to by ram_ptr.
                    orl   a,#0f0h                   ;1  Set the high nibble of the accumulator to #f to test
                                                    ;   for record availability.
                    inc   a                         ;1  Increment the accumulator to make it roll to zero if
                                                    ;   it is #0ffh.
                    jz    continue_count            ;2  If the accumulator rolls to zero, then the record is
                                                    ;   blank, therefore do no increment the record counter.
                    inc   record_number             ;1  Increment the record counter because a valid record
                                                    ;   was found.
continue_count:     inc   ram_ptr                   ;1  Increment the RAM pointer to the next location.
                    djnz  b,count_records           ;2  Continue to count records until the address offset
                                                    ;   table has been tested.
                    mov   a,record_number           ;1  Copy the record count into the accumulator.
                    ret                             ;2  Return to the calling address with the number of
                                                    ;   records in accumulator.

;If it has been determined that the EEPROM is new or has not been used under the current software revision,
;then it must be initialized.  The version_string is written to the first 16 bytes of the EEPROM using the
;wr_eeprom_page subroutine.  The EEPROM address pointer is then moved to location 16 and #0nfh is written
;ten times into the address offset table in the page write mode.  The accumulator is returned containing
;zero to indicate that there are no call records stored in the EEPROM.

new_eeprom:         mov   ram_ptr,#data_ram         ;1  Reset the RAM pointer to the beginning of data RAM.
                    mov   b,#010h                   ;2  Load the B register with the 16 byte countdown value.
                    mov   dptr,#version_string      ;2  Load the 16 bit data pointer with the address of
                                                    ;   the version string.
                    mov   temp,#000h                ;1  Clear the temporary register which is used to store
                                                    ;   the offset for the MOVC instruction.

prep_version:       mov   a,temp                    ;1  Load the accumulator with the offset value.
                    movc  a,@a+dptr                 ;2  Load the next version string byte into the accumulator.
                    mov   @ram_ptr,a                ;1  Put the current byte of the version string into the
```

```
                                                ;   current RAM location.
                    inc     ram_ptr             ;1  Increment the RAM pointer.
                    inc     temp                ;1  Increment the version_string offset.
                    djnz    b,prep_version      ;2  Repeat until all 16 version string bytes have been
                                                ;   transferred to RAM.
                    mov     ram_ptr,#data_ram   ;1  Load the ram pointer with the beginning of RAM.
                    mov     high_eeprom_addr,#000h  ;1  Load the high EEPROM address for page write.
                    mov     low_eeprom_addr,#000h   ;1  Load the low EEPROM address for page write.
                    acall   wr_eeprom_page      ;2  Write the first 16 bytes of RAM to EEPROM.

                    mov     b,#00ah             ;2  Load the B register with the 10 record countdown value.
                    mov     ram_ptr,#data_ram   ;1  Position the RAM pointer to the beginning of data ram.
                    mov     temp,#000h          ;1  Initialize the temporary register with zero.  The
                                                ;   lower nibble of this register will be used to determine
                                                ;   "n" for the upper nibble of the EEPROM address offset.
write_blanks:       mov     a,temp              ;1  Load the accumulator with the current address offset
                                                ;   high nibble.
                    swap    a                   ;1  Put the address offset into the high nibble.
                    orl     a,#00fh             ;1  Set the low nibble of the address offset to #f to
                                                ;   indicate that the record is available.
                    mov     @ram_ptr,a          ;1  Put #0nfh at the current RAM location to indicate an
                                                ;   unused record.
                    inc     temp                ;1  Increment the temporary register which holds the
                                                ;   high nibble of the EEPROM offset address in its low
                                                ;   nibble.
                    inc     ram_ptr             ;1  Increment the RAM pointer.
                    djnz    b,write_blanks      ;2  Continue to fill the first ten bytes of RAM with #0ffh.
                    acall   wr_offset_table     ;2  Write the initialized EEPROM address offset table to
                                                ;   the EEPROM.
                    mov     a,#000h             ;1  Load the accumulator with zero.
                    ret                         ;2  Return to the calling address with the number of
                                                ;   records in the accumulator.
;-------------------------------------------------------------------------------------------------------------


;Subroutine: wr_eeprom_byte ---------------------------------------------------------------------------

;The purpose of the wr_eeprom_byte subroutine write a single byte to a specific location in the EEPROM.  The
;byte to be written must be contained in the accumulator, the high address byte must be in the high_eeprom_addr
;register, and the low address byte must be in the low_eeprom_addr register.  Although the high address byte
;for the 24C04 EEPROM only has one significant bit, a full byte (high_eeprom_addr) is reserved for this byte.
;This will allow for possible EEPROM expansion.  The LSB of high_eeprom_addr is used to specify which 256
;byte "bank" will be addressed in the 24C04.  See the 24C04 datasheet for further addressing information.

;This subroutine begins by requesting mastership of the IIC bus.  If mastership is not obtained, then an IIC
;error code is loaded and passed to an error annunciation routine.  No attempts to restore the IIC bus are
;made following this error.  Timer I and its interrupt are used to detect "stuck" IIC lines.

;An IIC start condition is automatically generated after bus mastership has been obtained (see IIC SFR listings
;for further information).  The subroutine proceeds to combine the high_eeprom_addr with the eeprom_dev_addr to
;form the slave address of the EEPROM.  The LSB of this slave address is cleared to indicate that a write is
;about to take place.  After compiling the slave address byte, it is sent to the EEPROM via the send_iic_byte
;subroutine.

;The next step is to send the EEPROM "sub-address" which indicates which location in the 256 byte "bank" is
;being addressed.  This is done simply by sending the low_eeprom_addr byte directly to the EEPROM via the
;send_iic_byte subroutine.

;After all of the address information has been sent to the EEPROM, the data byte may be sent.  Again, this is
;done by simply calling the send_iic_byte subroutine.  Following the successful IIC transmission of the device
;address, the byte address, and the data byte, this subroutine generates an IIC stop condition which releases
;the bus, stops timer I, and disables the timer I interrupt.

;Registers used:      Accumulator, low_eeprom_addr, high_eeprom_addr, b
;Registers affected:  Accumulator, b
;SFRs used:           none
```

```
;Bits used:           none
;Calls:              iic_master_req, send_eeprom_addr, send_iic_byte, send_iic_stop

wr_eeprom_byte:      acall  iic_master_req              ;2  Request IIC bus mastership.

make_address:        mov    b,a                         ;1  Temporarily store the byte to be written in the B
                                                        ;   register to free-up the accumulator.
                     mov    a,high_eeprom_addr          ;1  Load the accumulator with the high byte of the
                                                        ;   EEPROM address.
                     rl     a                           ;1  Rotate the accumulator left to shift the MSB of the
                                                        ;   high address byte into bit 1.
                     orl    a,#eeprom_dev_addr          ;1  Combine the shifted high address and the device
                                                        ;   address.
                     anl    a,#11111110b                ;1  Ensure that the MSB is clear to signify an EEPROM
                                                        ;   write.
                     acall  send_eeprom_addr            ;2  Send the EEPROM device address over the IIC bus.
send_sub_addr:       mov    a,low_eeprom_addr           ;1  Load the accumulator with the EEPROM sub-address.
                     acall  send_iic_byte               ;2  Send the EEPROM sub-address over the IIC bus.
send_data_byte:      mov    a,b                         ;1  Copy the data byte back into the accumulator
                     acall  send_iic_byte               ;2  Send the data byte to the EEPROM via the IIC bus.
                     acall  send_iic_stop               ;2  Send an IIC stop condition to release the bus.
                     ret                                ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------


;Subroutine: wr_offset_table ----------------------------------------------------------------------------

;The purpose of the wr_offset_table subroutine is to write the EEPROM address offset table to the EEPROM.  An
;image of the EEPROM offset table must be in RAM beginning at data_ram prior to calling this subroutine.  This
;subroutine must be located directly before the wr_eeprom_page subroutine.  This subroutine simply sets the high
;EEPROM address to zero, the low EEPROM address to 16, and the RAM pointer to the beginning of data RAM.  The
;subroutine then flows (versus using a call instruction) into the wr_eeprom_page subroutine to perform a page
;write of the EEPROM offset table.

;Registers used:      high_eeprom_addr, low_eeprom_addr, ram_ptr
;Registers affected:  high_eeprom_addr, low_eeprom_addr, ram_ptr
;SFRs used:           none
;Bits used:           none
;Constants used:      data_ram
;Calls:               flows into wr_eeprom_page


wr_offset_table:     mov    low_eeprom_addr,#010h       ;1  Load the low EEPROM address with the beginning of the
                                                        ;   address offset table.
                     mov    high_eeprom_addr,#000h      ;1  Load the high EEPROM address with the beginning of the
                                                        ;   address offset table.
                     mov    ram_ptr,#data_ram           ;1  Move the RAM pointer to the beginning of data RAM.

;After setting-up the EEPROM address and the RAM pointer, the wr_offset_table subroutine flows into the
;wr_eeprom_page subroutine.  The return from subroutine instruction is located at the end of the wr_eeprom_page
;subroutine.
;-------------------------------------------------------------------------------------------------------------


;Subroutine: wr_eeprom_page -----------------------------------------------------------------------------

;The purpose of the wr_eeprom_page subroutine write a 16 byte page of data to the EEPROM.  The low and high
;bytes of the starting address of the page must be in low_eeprom_addr and high_eeprom_addr, respectively.
;The ram_ptr register must contain the RAM address of the beginning of the data to be written.  This subroutine
;assumes that a full 16 bytes is available to be written.

;This subroutine takes advantage of the "page write" mode of the 24C04.  In this mode, 16 continuous bytes of
;data are sent to the EEPROM.  These bytes are temporarily stored in the EEPROM's internal buffer so that they
;may all be written at once, thus saving 15 slow EEPROM write cycles.

;The EEPROM is first addressed in the usual fashion by sending the devices address followed by the word address.
```

```
;However, instead of sending an IIC stop condition after the first data byte, the subroutine sends 16 data bytes
;and then generates an IIC stop condition.  At that time, the EEPROM initiates its internal write cycle.

;Registers used:      Accumulator, low_eeprom_addr, high_eeprom_addr, b, ram_ptr
;Registers affected:  Accumulator, b
;SFRs used:           i2cfg,i2con
;Bits used:           eti
;Constants used:      eeprom_dev_addr
;Calls:               iic_master_req, send_eeprom_addr, send_iic_byte, send_iic_stop

wr_eeprom_page:       acall  iic_master_req              ;2  Request IIC bus mastership.

make_page_addr:       mov    a,high_eeprom_addr          ;1  Load the accumulator with the high byte of the
                                                         ;   EEPROM address.
                      rl     a                           ;1  Rotate the accumulator left to shift the MSB of the
                                                         ;   high address byte into bit 1.
                      orl    a,#eeprom_dev_addr          ;1  Combine the shifted high address and the device
                                                         ;   address.
                      anl    a,#11111110b                ;1  Ensure that the MSB is clear to signify an EEPROM
                                                         ;   write.
                      acall  send_eeprom_addr            ;2  Send the EEPROM device address over the IIC bus.
send_page_addr:       mov    a,low_eeprom_addr           ;1  Load the accumulator with the EEPROM sub-address.
                      acall  send_iic_byte               ;2  Send the EEPROM sub-address over the IIC bus.
                      mov    b,#010h                     ;2  Load the B register with the 16 byte countdown value.

send_page_bytes:      mov    a,@ram_ptr                  ;1  Load the accumulator with the current data byte.
                      acall  send_iic_byte               ;2  Send the data byte to the EEPROM via the IIC bus.
                      inc    ram_ptr                     ;1  Increment the RAM pointer.
                      djnz   b,send_page_bytes           ;2  Repeat the process until all 16 bytes have been sent.
                      acall  send_iic_stop               ;2  Send an IIC stop condition to release the bus.
                      ret                                ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------------


;Subroutine: delete_record ----------------------------------------------------------------------------------

;The purpose of the delete_record subroutine is to delete a record from the EEPROM and adjust the EEPROM address
;offset table accordingly.  The record number to be deleted must be in the accumulator when calling this
;subroutine.

;This subroutine loads the EEPROM address offset table from the EEPROM and places it in the lower 16 bytes of
;RAM, although only the first ten bytes are significant.  The subroutine starts at the record number to be
;deleted and moves the address offset values from all subsequent records down one position.  Then, the offset
;address of the deleted record is placed into the upper nibble of the tenth record.  In this way, the location
;of the deleted record in EEPROM can be overwritten later, as needed (much like a circular buffer).

;Registers used:      accumulator, record_number, ram_ptr, temp
;Registers affected:  accumulator, record_number, ram_ptr, temp
;SFRs used:           none
;Bits used:           c
;Calls:               get_offset_table, wr_offset_table

delete_record:        mov    record_number,a             ;1  Store the record number to be deleted in the record
                                                         ;   number register.
                      acall  get_offset_table            ;2  Read the EEPROM address offset table into RAM.
                      mov    a,#00ah                     ;1  Load the accumulator with the maximum record number.
                      clr    c                           ;1  Explicitly clear the carry bit.
                      subb   a,record_number             ;1  Subtract the record to be deleted.  The accumulator
                                                         ;   now contains the number of "move down" iterations
                                                         ;   which need to be performed.
                      jz     tenth_record                ;2  If the result is zero, then the tenth record needs
                                                         ;   to be deleted.
                      mov    b,a                         ;1  Load the B register with the number of "move down"
                                                         ;   iterations.
                      mov    a,#data_ram                 ;1  Load the accumulator with the beginning of RAM.
                      add    a,record_number             ;1  Add the record number to the beginning of RAM.
```

```
                        dec     a                       ;1  Subtract 1 from the RAM address so that the accumulator
                                                        ;   contains the RAM address of the record to be deleted.
                        mov     ram_ptr,a               ;1  Load the RAM address of the record into the RAM
                                                        ;   pointer.
                        mov     a,@ram_ptr              ;1  Load the accumulator with the EEPROM address offset
                                                        ;   of the record to be deleted.
                        mov     temp,a                  ;1  Temporarily store the EEPROM address offset of the
                                                        ;   record to be deleted.
move_down:              inc     ram_ptr                 ;1  Increment the RAM pointer to point to the next record.
                        mov     a,@ram_ptr              ;1  Load the accumulator with the address offset of the
                                                        ;   next record.
                        dec     ram_ptr                 ;1  Decrement the RAM pointer to point to the previous
                                                        ;   record.
                        mov     @ram_ptr,a              ;1  Place the EEPROM address offset of the next record
                                                        ;   into this record.
                        inc     ram_ptr                 ;1  Increment the RAM pointer to point to the next record.
                        djnz    b,move_down             ;2  Repeat until all records above the deleted one have
                                                        ;   been moved down.

                        mov     a,temp                  ;1  Load the accumulator with the EEPROM address offset
                                                        ;   of the deleted record.
                        orl     a,#00fh                 ;1  Set the lower nibble to #f to indicate that the
                                                        ;   record is available.
                        mov     ram_ptr,#data_ram+9     ;1  Load the RAM pointer with the address of the tenth
                                                        ;   record EEPROM address offset.
                        mov     @ram_ptr,a              ;1  Put the EEPROM address offset of the deleted record
                                                        ;   into that of the tenth record.
                        sjmp    update_table            ;2  Update the EEPROM address offset table.

tenth_record:           mov     ram_ptr,#data_ram+9     ;1  Load the RAM pointer with the address of the tenth
                                                        ;   record EEPROM address offset.
                        mov     a,@ram_ptr              ;1  Load the accumulator with the EEPROM address offset
                                                        ;   of the tenth record.
                        orl     a,#00fh                 ;1  Set the lower nibble to #f to indicate that it is
                                                        ;   available.
                        mov     @ram_ptr,a              ;1  Put it back into the tenth record of the table.
update_table:           acall   wr_offset_table         ;2  Write the update EEPROM address offset table back
                                                        ;   to the EEPROM using "page write"
                        ret                             ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------------


;Subroutine: disp_record ------------------------------------------------------------------------------------------

;The purpose of the disp_record subroutine is to display a stored data record on the LCD display.  The record
;number must be in the accumulator on entry.  If the record number is zero, then the subroutine will clear the
;display and send the "No call record" message to the second line of the display via the send_message subroutine.
;The software version_string is then sent to first line of the LCD display.

;If the record number is non-zero, the subroutine will load the EEPROM address offset table into RAM and
;retrieve the address offset of the record.  Then, the address offset is multiplied by 48 to obtain the
;absolute EEPROM address of the designated record.  The EEPROM address pointer is then moved to this address
;and each byte in the record is sent to the display until all 48 bytes have been sent.

;After retrieving and displaying the lower three lines of the display, the software determines the number of
;stored records and sends "Record X of Y" to the first line of the display.  "X" is the currently displayed
;record and "Y" is the total number of stored records.  The record number is stored in the off-screen DD RAM
;location.

;Registers used:        record_number, accumulator, temp_aux, ram_ptr, low_eeprom_addr, high_eeprom_addr,
;                       temp, b, ls_digit, ms_digit
;Registers affected:    record_number, accumulator, temp_aux, ram_ptr, low_eeprom_addr, high_eeprom_addr,
;                       temp, b, ls_digit, ms_digit
;SFRs used:             none
;Bits used:             c
;Calls:                 store_rec_no, get_offset_table, set_disp_address, rd_eeprom_byte, send_disp_data,
```

```
;                   init_eeprom, hex_to_ascii, send_message, send_command


disp_record:        mov   record_number,a           ;1  Store the record desired record number in the
                                                    ;   record_number register.
                    mov   temp_aux,a                ;1  Temporarily store the record number in the auxiliary
                                                    ;   temporary register.
                    jnz   process_record            ;2  If the record number is not zero, then process the
                                                    ;   record.
                    ajmp  no_call_record            ;2  If the record number is zero, then display the
                                                    ;   "no call record" message.

process_record:     acall store_rec_no              ;2  Store the record number in DD RAM.
                    acall get_offset_table          ;2  Read the address offset table into RAM.
                    mov   a,record_number           ;1  Load the desired record number into the accumulator.
                    add   a,#data_ram               ;1  Add the address of the beginning of RAM to the
                                                    ;   desired record number.
                    dec   a                         ;1  Decrement the accumulator so that it points to the
                                                    ;   address offset of the desired record.
                    mov   ram_ptr,a                 ;1  Position the RAM pointer to the desired record in
                                                    ;   the address offset table.
                    mov   a,@ram_ptr                ;1  Load the accumulator with the address offset of the
                                                    ;   desired record.
                    anl   a,#00fh                   ;1  Knock-off the high nibble.
                    mov   b,#030h                   ;2  Load the B register with the 48 byte record length.
                    mul   ab                        ;4  Multiply the desired record address offset by the
                                                    ;   48 byte record length.  The accumulator now contains
                                                    ;   the low order byte of the absolute EEPROM address and
                                                    ;   the B register now contains the high order byte of the
                                                    ;   absolute EEPROM address.
                    add   a,#020h                   ;1  Add the address of the beginning of EEPROM data.
                    jnc   load_low_addr             ;2  The high EEPROM address does not need to be incremented
                                                    ;   if the accumulator did not overflow.
                    inc   b                         ;1  Increment the high EEPROM address if the accumulator
                                                    ;   overflowed.
load_low_addr:      mov   low_eeprom_addr,a         ;1  Put the accumulator into the low EEPROM address.
                    mov   high_eeprom_addr,b        ;1  Put the B register into the high EEPROM address.
                    mov   a,#010h                   ;1  Load the accumulator with the address of the first
                                                    ;   character of the third line of the LCD display.
                    acall set_disp_address          ;2  Position the cursor on the third line.
                    mov   temp,#010h                ;2  Load the temporary register with the 16 character
                                                    ;   countdown value.
recall_line_3:      acall rd_eeprom_byte            ;2  Get the byte at the current EEPROM address.  The byte
                                                    ;   is returned in the accumulator.
                    acall send_disp_data            ;2  Send the current character to the display.
                    mov   a,low_eeprom_addr         ;2  Load the low EEPROM address into the accumulator.
                    inc   a                         ;1  Increment the low EEPROM address.
                    mov   low_eeprom_addr,a         ;2  Put the incremented low address back into its register.
                    jnz   continue_line_3           ;2  If the low EEPROM address rolled over to zero, then
                                                    ;   increment the high EEPROM address.
                    inc   high_eeprom_addr          ;1  Increment the high EEPROM address if necessary.
continue_line_3:    djnz  temp,recall_line_3        ;2  Continue getting record bytes until the second line
                                                    ;   of the display is filled.
                    mov   a,#040h                   ;1  Load the accumulator with the address of the first
                                                    ;   character of the second line of the LCD display.
                    acall set_disp_address          ;2  Position the cursor on the second line.
                    mov   temp,#020h                ;2  Load the temporary register with the 32 character
                                                    ;   countdown value.
recall_line_2:      acall rd_eeprom_byte            ;2  Get the byte at the current EEPROM address.  The byte
                                                    ;   is returned in the accumulator.
                    acall send_disp_data            ;2  Send the current character to the display.
                    mov   a,low_eeprom_addr         ;2  Load the low EEPROM address into the accumulator.
                    inc   a                         ;1  Increment the low EEPROM address.
                    mov   low_eeprom_addr,a         ;2  Put the incremented low address back into its register.
                    jnz   continue_line_2           ;2  If the low EEPROM address rolled over to zero, then
                                                    ;   increment the high EEPROM address.
```

```
                    inc     high_eeprom_addr        ;1  Increment the high EEPROM address if necessary.
continue_line_2:    djnz    temp,recall_line_2      ;2  Continue getting record bytes until the second line
                                                    ;   of the display is filled.

                    mov     a,#000h                 ;1  Load the accumulator with the address of the first
                                                    ;   character of line 1 of the display in preparation
                                                    ;   to clear it.
                    acall   set_disp_address        ;2  Position the cursor on the first line.
                    mov     b,#010h                 ;2  Load the B register with a 16 character countdown
                                                    ;   value.
clear_line_one:     mov     a,#020h                 ;1  Load the accumulator with the ASCII value of a space.
                    acall   send_disp_data          ;2  Send the space to the display.
                    djnz    b,clear_line_one        ;2  Continue to fill line 1 with spaces until it is clear.

                    mov     a,#000h                 ;1  Load the accumulator with the address of the first
                                                    ;   character of the first line of the LCD display.
                    acall   set_disp_address        ;2  Set the cursor to the new display address.
                    acall   init_eeprom             ;2  Put the number of records into the accumulator.
                    cjne    a,#00ah,less_than_ten   ;2  If there are fewer than 10 records, then put a leading
                                                    ;   space in the header.
                    sjmp    setup_rec_msg           ;2  Display the record number header with no leading space
                                                    ;   if there are 10 records.
less_than_ten:      mov     a,#020h                 ;1  Load the accumulator with the ASCII value of a space.
                    acall   send_disp_data          ;2  Send the space to the display.
setup_rec_msg:      mov     dptr,#record_message    ;2  Load the data pointer with the address of the record
                                                    ;   message header.
                    mov     temp,#000h              ;1  Reset the temp register for use as an address offset
                                                    ;   pointer.
                    mov     b,#007h                 ;2  Load the B register with the 7 byte countdown value.
disp_rec_msg:       mov     a,temp                  ;1  Load the accumulator with the message character address
                                                    ;   offset.
                    movc    a,@a+dptr               ;2  Load the next message byte to be sent into the
                                                    ;   accumulator.
                    acall   send_disp_data          ;2  Send it out to the display.
                    inc     temp                    ;1  Increment the data pointer offset value.
                    djnz    b,disp_rec_msg          ;2  Repeat until all 16 bytes have been sent.

                    mov     a,temp_aux              ;1  Load the accumulator with the record number from the
                                                    ;   auxiliary temporary register.
                    acall   hex_to_ascii            ;2  Convert the record number to ASCII.
                    mov     a,ms_digit              ;1  Load the most significant digit into the accumulator.
                    clr     c                       ;1  Explicitly clear the carry flag.
                    subb    a,#030h                 ;1  Subtract the ASCII value of zero.
                    jz      send_ls_digit           ;2  If the result is zero, then there is a leading zero
                                                    ;   that should not be displayed.
                    mov     a,ms_digit              ;1  Load the most significant digit into the accumulator.
                    acall   send_disp_data          ;2  Send the MSD to the display.
send_ls_digit:      mov     a,ls_digit              ;1  Load the ones digit into the accumulator.
                    acall   send_disp_data          ;2  Send the record number to the display.
                    mov     a,#020h                 ;1  Load the accumulator with the ASCII value of a space.
                    acall   send_disp_data          ;2  Send the space to the display.
                    mov     a,#'o'                  ;1  Load the accumulator with the ASCII value of "o".
                    acall   send_disp_data          ;2  Send the "o" to the display.
                    mov     a,#'f'                  ;1  Load the accumulator with the ASCII value of "f".
                    acall   send_disp_data          ;2  Send the "f" to the display.
                    mov     a,#020h                 ;1  Load the accumulator with the ASCII value of a space.
                    acall   send_disp_data          ;2  Send the space to the display.

                    acall   init_eeprom             ;2  Call the init_eeprom subroutine to get the number of
                                                    ;   stored records into the accumulator.
                    acall   hex_to_ascii            ;2  Convert the number of stored records to ASCII values.
                    mov     a,ms_digit              ;1  Load the ASCII value of the most significant digit
                                                    ;   into the accumulator.
                    clr     c                       ;1  Explicitly clear the carry bit.
                    subb    a,#030h                 ;1  Subtract the ASCII value of zero.
                    jz      ones_record             ;2  If the result is zero, then there is a leading zero
```

```
                                                ;   in the number of records.  Don't display it.
                    mov   a,ms_digit            ;1  Load the ASCII value of the most significant digit
                                                ;   into the accumulator.
                    acall send_disp_data        ;2  Send the most significant digit to the display.
ones_record:        mov   a,ls_digit            ;1  Load the ASCII value of the least significant digit
                                                ;   into the accumulator.
                    acall send_disp_data        ;2  Send the digit to the display.
                    ret                         ;2  Return to the calling address.


no_call_record:     mov   a,#001h               ;1  Load the accumulator with the clear display and
                                                ;   home cursor command.
                    acall send_command          ;2  Clear the display.
                    mov   dptr,#no_records_msg  ;1  Load the data pointer with the address of the
                                                ;   "no call record" message.
                    acall send_message          ;2  Send the "no call record" message to the display.
                    mov   a,#000h               ;1  Load the accumulator with the address of the first
                                                ;   line.
                    acall set_disp_address      ;2  Move the cursor to the new position.
                    mov   dptr,#version_string  ;2  Load the data pointer with the address of the version
                                                ;   string.
                    mov   b,#010h               ;2  Load the B register with the 16 byte countdown value.
                    mov   temp,#000h            ;1  Clear the temporary register which is used to hold
                                                ;   the address offset of the message character.
disp_ver_string:    mov   a,temp                ;1  Load the accumulator with the message character address
                                                ;   offset.
                    movc  a,@a+dptr             ;2  Load the next message byte to be sent into the
                                                ;   accumulator.
                    acall send_disp_data        ;2  Send it out to the display.
                    inc   temp                  ;1  Increment the data pointer offset value.
                    djnz  b,disp_ver_string     ;2  Repeat until all 16 bytes have been sent.

                    mov   record_number,#000h   ;1  Put zero into the record number.
store_rec_no:       mov   a,#disp_record_addr   ;1  Load the accumulator with the address of unused DD RAM.
                    acall set_disp_address      ;2  Move the cursor off the screen.
                    mov   a,record_number       ;1  Load the accumulator with the record number.
                    acall send_disp_data        ;2  Store the record number in the DD RAM.
                    ret                         ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------


;Subroutine: send_iic_byte ------------------------------------------------------------------------------

;The purpose of the send_iic_byte subroutine is to send a single eight bit byte over the IIC bus.  Upon entry,
;the accumulator must contain the byte to be sent.  Mastership of the IIC bus is assumed.  IIC start and stop
;conditions are not generated by this subroutine.  After the eight bits have been sent, the subroutine looks
;for an IIC acknowledge from the EEPROM.  If an acknowledge is not detected, then an IIC error code is generated.
;Timer I should be enabled to detect "stuck" IIC signals before calling this subroutine.

;Registers used:      Accumulator, bit_count, iic_error_code (if applicable)
;Registers affected:  Accumulator, bit_count
;SFRs used:           i2cfg,i2con, i2dat
;Bits used:           atn, drdy, rdat
;Constants used:      i2con_clear, iic_receive
;Calls:               iic_error (if applicable)

send_iic_byte:      mov   bit_count,#008h       ;1  Load the bit_count register with the 8 bit countdown
                                                ;   value.
send_bit:           mov   i2dat,a               ;1  Load the MSB of I2DAT with the current bit.
                    mov   i2con,#i2con_clear    ;2  Clear IIC ARL, STR, and STP.
                    rl    a                     ;1  Rotate the accumulator left to place the next bit
                                                ;   in the MSB position.
                    jnb   atn,$                 ;2  Loop until the next bus condition is detected.
                    jb    drdy,send_bit_cont    ;2  If the bus condition is DRDY, then continue to send
                                                ;   bits.
                    mov   iic_error_code,#'2'   ;1  Load the accumulator with the ASCII value for "2".
```

```
                    ajmp   iic_error                ;2  Annunciate IIC error code 2: IIC bit xmit error.
send_bit_cont:      djnz   bit_count,send_bit       ;2  Continue to send bits until all eight have been sent.

                    mov    i2con,#iic_receive       ;2  Put the IIC hardware into receive mode.
                    jnb    atn,$                    ;2  Loop until the next bus condition is detected.
                    jnb    rdat,send_done           ;2  Exit the subroutine if the acknowledge bit was
                                                    ;   received.
                    mov    iic_error_code,#'3'      ;1  Load the accumulator with the ASCII value for "3".
                    ajmp   iic_error                ;2  Annunciate IIC error code 3: No EEPROM acknowledge.
send_done:          ret                             ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------


;Subroutine: send_eeprom_addr ---------------------------------------------------------------------------

;The purpose of the send_eeprom_addr subroutine is to provide an IIC write routine which is dedicated
;specifically to sending the IIC address to the EEPROM.  This is necessary because of the relatively long
;write times required by EEPROMs.  While the EEPROM is writing data, it does not respond to any IIC commands.
;This subroutine continually sends the IIC address of the EEPROM until it responds or the external hardware
;timer (the watchdog) expires.  If an acknowledge bit is received after transmitting the EEPROM address, then
;the EEPROM is not busy and is able to take the next byte of data.  If the hardware watchdog expires during
;this subroutine, then an IIC error is reported.  Mastership of the IIC bus is assumed on entering this
;subroutine.  The address of the EEPROM must be in the accumulator on entry.  This subroutine implements
;"acknowledge polling" as discussed in the 24c04 datasheet.

;The timer I interrupt is disabled at the beginning of the subroutine because the external hardware timer (the
;watchdog) is used to detect "stuck" conditions.  This is simpler than using an ISR which differentiates between
;an EEPROM write cycle and a stuck bus.  The interrupt is re-enabled before exiting the subroutine.

;Registers used:      Accumulator, bit_count, iic_error_code (if applicable)
;Registers affected:  Accumulator, bit_count
;SFRs used:           i2cfg, i2con, i2dat
;Bits used:           w_dog_rst, w_dog, eti, tirun, atn, drdy, rdat, master
;Constants used:      i2con_clear, iic_receive, iic_release,
;Calls:               iic_error (if applicable)

send_eeprom_addr:   clr    eti                      ;1  Disable the timer I interrupt because the hardware
                                                    ;   watchdog will be used.
                    clr    tirun                    ;1  Stop timer I.
                    setb   w_dog_rst                ;1  Discharge the watchdog capacitor.
                    jnb    w_dog,$                  ;2  Wait until the watchdog bit is asserted.
                    clr    w_dog_rst                ;1  Allow the watchdog cap to charge.

reset_bit_count:    mov    bit_count,#008h          ;1  Load the bit_count register with the 8 bit countdown
                                                    ;   value.
send_addr:          mov    i2dat,a                  ;1  Load the MSB of I2DAT with the current bit.
                    mov    i2con,#i2con_clear       ;1  Clear IIC ARL, STR, and STP.
                    rl     a                        ;1  Rotate the accumulator left to place the next bit
                                                    ;   in the MSB position.
                    jnb    atn,$                    ;2  Loop until the next bus condition is detected.
                    jb     drdy,send_addr_cont      ;2  If the bus condition is DRDY, then continue to send
                                                    ;   bits.
                    mov    iic_error_code,#'2'      ;1  Load the error code with the ASCII value for "2".
                    ajmp   iic_error                ;2  Annunciate IIC error code 2: IIC bit xmit error.
send_addr_cont:     djnz   bit_count,send_addr      ;2  Continue to send bits until all eight have been sent.

                    mov    i2con,#iic_receive       ;2  Put the IIC hardware into receive mode.
                    jnb    atn,$                    ;2  Loop until the next bus condition is detected.
                    jnb    rdat,send_addr_done      ;2  Exit the subroutine if the acknowledge bit was
                                                    ;   received.  This means that the EEPROM is available.
                    mov    i2con,#iic_release       ;2  Release the IIC bus and generate a stop condition.
                    jnb    atn,$                    ;2  Loop until the next IIC event.
                    mov    i2cfg,#iic_bus_request   ;2  Request IIC mastership and generate a start condition.
                    jnb    atn,$                    ;2  Loop until the IIC bus is available.
                    jb     master,try_again         ;2  If the microcontroller gets IIC bus mastership,
                                                    ;   try to send the EEPROM address again.
```

```
                    mov   iic_error_code,#'1'          ;1  Load the error code with the ASCII value for "1".
                    ajmp  iic_error                    ;2  Annunciate IIC error code 1: IIC bus not granted.
try_again:          jb    w_dog,reset_bit_count        ;2  Try again until the watchdog expires.
                    mov   iic_error_code,#'4'          ;1  Load the error code with the ASCII value for "4".
                    ajmp  iic_error                    ;2  Annunciate IIC error code 4: Can't find EEPROM.
send_addr_done:     setb  tirun                        ;1  Clear and start timer I.
                    setb  eti                          ;1  Enable the timer I interrupt.
                    ret                                ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------------


;Subroutine: send_iic_stop ----------------------------------------------------------------------------------------

;The purpose of the send_iic_stop subroutine is to generate an IIC stop condition and release mastership of the
;IIC bus.  This is typically done after a full transmission has been completed.  Nothing is expected to be
;passed to this subroutine.  This subroutine does not modify any registers.  Timer I is stopped and its interrupt
;is disabled.

;Registers used:      none
;Registers affected:  none
;SFRs used:           i2con
;Bits used:           atn, eti, tirun, mastrq
;Constants used:      iic_stop, iic_clr_drdy
;Calls:               none

send_iic_stop:      clr   mastrq                       ;1  Release IIC bus mastership.
                    mov   i2con,#iic_stop              ;2  Generate a stop condition.
                    jnb   atn,$                        ;2  Loop until the next IIC event.
                    mov   i2con,#iic_clr_drdy          ;2  Clear DRDY.
                    jnb   atn,$                        ;2  Loop until the next IIC event.
                    clr   eti                          ;1  Disable the timer I interrupt.
                    clr   tirun                        ;1  Clear and stop timer I.
                    ret                                ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------------


;Subroutine: rd_eeprom_byte ---------------------------------------------------------------------------------------

;The purpose of the rd_eeprom_byte subroutine read a single byte from a specific location in the EEPROM.  The
;high address byte must be in the high_eeprom_addr register and the low address byte must be in the
;low_eeprom_addr register.  Although the high address byte for the 24C04 EEPROM only has one significant bit,
;a full byte (high_eeprom_addr) is reserved for this byte.  This will allow for possible EEPROM expansion.  The
;LSB of high_eeprom_addr is used to specify which 256 byte "bank" will be addressed in the 24C04.  See the 24C04
;datasheet for further addressing information.

;This subroutine begins by requesting mastership of the IIC bus.  If mastership is not obtained, then an IIC
;error code is loaded and passed to an error annunciation routine.  No attempts to restore the IIC bus are
;made following this error.  Timer I and its interrupt are used to detect "stuck" IIC lines and must be
;enabled prior to calling this subroutine.

;An IIC start condition is automatically generated after bus mastership has been obtained (see IIC SFR listings
;for further information).  The subroutine proceeds to combine the high_eeprom_addr with the eeprom_dev_addr to
;form the slave address of the EEPROM.  The LSB of this slave address is cleared to indicate that a write is
;about to take place.  This is done to provide the "dummy write" to the EEPROM which sets its address pointer.

;The next step is to send the EEPROM "sub-address" which indicates which location in the 256 byte "bank" is
;being addressed.  This is done simply by sending the low_eeprom_addr byte directly to the EEPROM via the
;send_iic_byte subroutine.

;After sending the EEPROM slave address (in the write mode) and the byte sub-address, the software waits for
;an acknowledge and then reissues a start condition and sends the EEPROM's slave address again, this time with
;the LSB of the address set to 1 to indicate a read function.  The EEPROM will then transmit the requested byte
;to the master.  The master then responds with a "negative acknowledge" and issues an IIC stop condition to
;indicate that the transfer is complete.  The received byte is returned in the accumulator.

;Registers used:      Accumulator, low_eeprom_addr, high_eeprom_addr, b, bit_count,
```

```
;                       iic_error_code (if applicable)
;Registers affected:    Accumulator, b, bit_count
;SFRs used:             i2cfg,i2con, i2dat
;Bits used:             atn, drdy
;Constants used:        eeprom_dev_addr, iic_bus_request, iic_start, iic_clr_drdy
;Calls:                 iic_master_req, send_eeprom_addr, send_iic_byte, send_iic_stop,
;                       iic_error (if applicable)

rd_eeprom_byte:         acall  iic_master_req               ;2  Request IIC bus mastership.

make_rd_address:        mov    a,high_eeprom_addr           ;1  Load the accumulator with the high byte of the
                                                            ;   EEPROM address.
                        rl     a                            ;1  Rotate the accumulator left to shift the MSB of the
                                                            ;   high address byte into bit 1.
                        orl    a,#eeprom_dev_addr           ;1  Combine the shifted high address and the device
                                                            ;   address.
                        anl    a,#11111110b                 ;1  Ensure that the MSB is clear to signify an EEPROM
                                                            ;   "dummy" write.
                        mov    b,a                          ;1  Copy the slave address into the B register for later
                                                            ;   use.
                        acall  send_eeprom_addr             ;2  Send the EEPROM device address over the IIC bus.
send_rd_sub_addr:       mov    a,low_eeprom_addr            ;1  Load the accumulator with the EEPROM sub-address.
                        acall  send_iic_byte                ;2  Send the EEPROM sub-address over the IIC bus and wait
                                                            ;   for an acknowledge from the EEPROM.
                        mov    i2con,#iic_start             ;2  Send repeated starts.
                        jnb    atn,$                        ;2  Loop until the next IIC event.
                        mov    i2con,#iic_clr_drdy          ;2  Clear DRDY.
                        jnb    atn,$                        ;2  Loop until the start is sent.
                        mov    a,b                          ;1  Get the slave address which was previously stored.
                        orl    a,#00000001b                 ;1  Set the LSB to indicate an EEPROM read.
                        acall  send_iic_byte                ;2  Send the EEPROM slave address over the IIC bus and
                                                            ;   wait for an acknowledge from the EEPROM.
                        mov    bit_count,#009h              ;1  Load the bit_count register with the eight bit count
                                                            ;   down value plus a buffer clear.
                        mov    a,#000h                      ;1  Clear the accumulator.
get_eeprom_bit:         orl    a,i2dat                      ;1  Put the current IIC bit into the MSB of the
                                                            ;   accumulator.
                        rl     a                            ;1  Rotate the next bit into the MSB position.
                        jnb    atn,$                        ;2  Loop until the next IIC event.
                        jb     drdy,get_continue            ;2  Continue to get bits if DRDY is set.
                        mov    iic_error_code,#'5'          ;1  Load the IIC error register with IIC error code 5:
                                                            ;   "IIC bit receive error".
                        ajmp   iic_error                    ;2  Display the error.
get_continue:           djnz   bit_count,get_eeprom_bit     ;2  Repeat until all 8 bits have been received.
                        mov    i2dat,#10000000b             ;2  Send a "negative acknowledge" to the EEPROM.
                        acall  send_iic_stop                ;2  Send an IIC stop condition to release the bus.
                        ret                                 ;2  Return to the calling address with the received byte
                                                            ;   in the accumulator.
;-------------------------------------------------------------------------------------------------------------------


;Subroutine: iic_master_req ---------------------------------------------------------------------------------------

;The purpose of the iic_master_req subroutine is to request mastership of the IIC bus.  If mastership is granted,
;then an IIC start condition is automatically generated.  If the IIC bus is not available, then IIC error code 1
;is sent to the IIC error annunciation routine and the program is halted.

;Registers used:        iic_error_code (if applicable)
;Registers affected:    none
;SFRs used:             i2cfg,i2con
;Bits used:             atn, drdy, tirun, eti, master
;Constants used:        iic_bus_request, iic_receive
;Calls:                 iic_error (if applicable)

iic_master_req:         mov    i2con,#iic_receive           ;2  Clear all IIC flags.
                        clr    tirun                        ;2  Stop and clear timer I.
```

```
                        setb   tirun                        ;2  Start timer I.
                        setb   eti                          ;1  Enable the timer I interrupt.
                        mov    i2cfg,#iic_bus_request       ;2  Load the IIC configuration SFR with the bus request
                                                            ;   byte (sets master, clears and starts timer I,
                                                            ;   clears CT1 and CT0 for 12MHz operation)
                                                            ;   An IIC start condition is automatically generated
                                                            ;   if the bus is granted.
                        jnb    atn,$                        ;2  Loop until the IIC bus is available or a timer I
                                                            ;   interrupt occurs.
                        jb     master,master_exit           ;2  If the microcontroller gets IIC bus mastership,
                                                            ;   proceed to make the EEPROM device address.
                        mov    iic_error_code,#'1'          ;1  Load the error code register with the ASCII value of
                                                            ;   "1".
                        ajmp   iic_error                    ;2  Annunciate IIC error code 1: IIC bus not granted.
master_exit:            ret                                 ;2  Return to the calling address, IIC bus granted.
;-------------------------------------------------------------------------------------------------------------------


;Subroutine: get_offset_table ---------------------------------------------------------------------------------

;The purpose of the get_offset_table subroutine is to load the EEPROM offset table from the EEPROM and put it
;into RAM beginning at data_ram.  This subroutine must be located directly before the rd_eeprom_page subroutine.
;This subroutine simply sets the high EEPROM address to zero, the low EEPROM address to 16, and the RAM pointer
;to the beginning of data RAM.  The subroutine then flows (versus using a call instruction) into the
;rd_eeprom_page subroutine to load the offset table into RAM.

;Registers used:        high_eeprom_addr, low_eeprom_addr, ram_ptr
;Registers affected:    high_eeprom_addr, low_eeprom_addr, ram_ptr
;SFRs used:             none
;Bits used:             none
;Constants used:        data_ram
;Calls:                 flows into rd_eeprom_page

get_offset_table:       mov    low_eeprom_addr,#010h        ;1  Load the low EEPROM address with the beginning of the
                                                            ;   address offset table.
                        mov    high_eeprom_addr,#000h       ;1  Load the high EEPROM address with the beginning of the
                                                            ;   address offset table.
                        mov    ram_ptr,#data_ram            ;1  Move the RAM pointer to the beginning of data RAM.

;After setting-up the EEPROM address and the RAM pointer, the get_offset_table subroutine flows into the
;rd_eeprom_page subroutine.  The return from subroutine instruction is located at the end of the rd_eeprom_page
;subroutine.
;-------------------------------------------------------------------------------------------------------------------


;Subroutine: rd_eeprom_page -----------------------------------------------------------------------------------

;The purpose of the rd_eeprom_page subroutine is to read 16 consecutive bytes from the EEPROM and store them in
;an internal RAM buffer.  Upon entry, the starting EEPROM address must be contained in high_eeprom_addr and
;low_eeprom_addr.  The complete EEPROM addresses must be valid (i.e. between 0 and 512-16=496).  If the beginning
;EEPROM address is greater than 496, then the EEPROM read sequence will wrap around to the beginning of the
;high EEPROM bank.  The ram_ptr register must contain the beginning location of the RAM buffer.

;This subroutine does not use the sequential read mode of the 24C04.  Instead, the rd_eeprom_byte subroutine
;is called 16 consecutive times.  This reduces the amount of EEPROM support code needed and should be more than
;fast enough for this application.  The byte_count register is used for the 16 byte countdown value.

;Registers used:        high_eeprom_addr, low_eeprom_addr, ram_ptr, accumulator, byte_count
;Registers affected:    ram_ptr, accumulator, byte_count
;SFRs used:             none
;Bits used:             c
;Constants used:        none
;Calls:                 rd_eeprom_byte

rd_eeprom_page:         mov    byte_count,#010h             ;1  Load the byte_count register with the 16 byte
                                                            ;   countdown value.
```

```
rd_page_loop:         acall  rd_eeprom_byte              ;2  Call the EEPROM byte read routine.
                      mov    @ram_ptr,a                 ;2  Store the byte that was read in the current RAM
                                                        ;   location.
                      inc    ram_ptr                    ;1  Increment the RAM pointer.
                      mov    a,low_eeprom_addr          ;1  Load the accumulator with the low EEPROM address byte.
                      clr    c                          ;1  Explicitly clear the carry bit.
                      add    a,#001h                    ;1  Add one to the low EEPROM address.
                      mov    low_eeprom_addr,a          ;1  Put the incremented low EEPROM address back.
                      jnc    dec_count                  ;2  If the carry bit is clear, there was no low_eeprom_
                                                        ;   addr overflow.
                      mov    high_eeprom_addr,#001h     ;1  Set the MSB of the EEPROM address, if necessary.
dec_count:            djnz   byte_count,rd_page_loop    ;2  Repeat the process until 16 bytes have been read.
                      ret                               ;1  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------

;Subroutine: iic_error ----------------------------------------------------------------------------------------

;The purpose of the iic_error subroutine is to annunciate an IIC error by displaying it on the LCD display.  No
;attempts are made to repair the IIC bus.  The errors are codified according to the following table:

;                           Error #                   Meaning
;                           -------                   -------
;                           0                         Timer I expiration
;                           1                         IIC bus not granted
;                           2                         IIC bit xmit error
;                           3                         No EEPROM acknowledge
;                           4                         Can't find EEPROM
;                           5                         IIC bit receive error

;The ASCII value of the error code must be in the iic_error_code register prior to calling this subroutine.
;The subroutine displays the error condition on the message line of the LCD display and loops until the system
;is reset (i.e. power is cycled).  This is a fatal error and must be corrected before the system can be used
;again.  For troubleshooting purposes, the state of the IIC bus is not changed by this subroutine.

;Registers used:      Accumulator, B, iic_error_code, DPTR
;Registers affected:  none
;SFRs used:           i2con
;Bits used:           none
;Constants used:      iic_error_msg
;Calls:               send_message, set_disp_address, send_disp_data

iic_error:            mov    dptr,#iic_error_msg        ;1  Load the data pointer with the address of the
                                                        ;   "IIC error code" message.
                      acall  send_message               ;2  Send the "IIC error code" message to the display.

                      mov    a,#04fh                    ;1  Load the accumulator with the address of the right
                                                        ;   most character of the message line.
                      acall  set_disp_address           ;2  Position the cursor at that character.
                      mov    a,iic_error_code           ;1  Load the accumulator with the ASCII value of the
                                                        ;   error code.
                      acall  send_disp_data             ;2  Send the error code to the display.
infinite_loop:        sjmp   infinite_loop              ;2  Loop until the system is reset.
;-------------------------------------------------------------------------------------------------------------

end
```

Required document

CCID Handset Software Listing

```
;Title:                Cordless Caller ID handset software
;Author:               Derek Matsunaga
;Start date:           9/1/94

;Revision:             1.3
;Revision date:        9/13/94

;Target processor:     87C750
;Assembler:            Metalink v1.2i
;Hardware platform:    CCID handset schematic v1.3

;Comment format:       A comment is required for each line of assembly code.  The "comment field" begins at the
;                      third tab stop from the left margin.  The first character in the comment field should be
;                      the number of machine cycles required to execute the instruction.  Following the number
;                      of machine cycles is another tab and the verbal comments which explain the function of
;                      the assembly instruction.  Comments are typically one sentence (or fragment) which begin
;                      with an uppercase letter and terminated with a period.  Comments which require more
;                      than one line may be continued at the same tab stop on the next line.

$MOD752                                                 ;Inform assembler of the target processor.

;=============================================================================================================
;Register definitions -------------------------------------------------------------------------------------

;Each of the eight internal registers (R0 through R7) are given relevant names to make the software more
;understandable.  Each register may be assigned more than one name so that its name is meaningful to the section
;of code in which it is used.  This is a "double edged sword" in that it makes the software more readable while
;it makes changes dangerous.  The name(s) of each register have been engineered such that they are not used
;for multiple purposes in any one section of code.  For this reason, one must exercise extreme caution when
;adding another name to a register definition or changing the register assignments altogether (i.e. swapping
;R0 and R3).

original_value        equ    r0                         ;Used to store the original ASCII value in the
                                                        ;validate_char subroutine.
trans                 equ    r0                         ;Used to keep track of port bit transitions.
temp                  equ    r0                         ;Used as a general purpose temporary register.

ram_ptr               equ    r1                         ;Used to point to a specific location in internal RAM.

max_value             equ    r2                         ;Used to keep track the FSK integral maxima.
interrupt_count       equ    r2                         ;Used to count timer interrupts.

min_value             equ    r3                         ;Used to keep track the FSK integral minima.
pulse_count_low       equ    r3                         ;Used to store the low byte of a long pulse measuring
                                                        ;count.

bit_count             equ    r4                         ;Used to count the number of received bits.
pulse_count_high      equ    r4                         ;Used to store the high byte of a long pulse measuring
                                                        ;count.

integral              equ    r5                         ;Used to store the FSK integral value.
max_pw                equ    r5                         ;Used to store a maximum pulse width value.
mark_count            equ    r5                         ;Used to count mark frequency pulses.

data_byte             equ    r6                         ;Used to store the data byte currently being received.
pulse_count           equ    r6                         ;Used to count pulses.

byte_count            equ    r7                         ;Used to store the number of bytes received.


;Additional equates and bit definitions -------------------------------------------------------------------

checksum              equ    dph                        ;Used to store a checksum value.

disp_port             equ    p3                         ;The display data port (see schematic).
disp_reg_sel          bit    p1.0                       ;The display register select bit (see schematic).
```

```
disp_read             bit    p1.1                    ;The display read/write (see schematic).
disp_strobe           bit    p1.2                    ;The display data latch bit (see schematic).

data_port             equ    p1                      ;The port at which all inputs may be read (see schematic).

receive_data          bit    p1.3                    ;The port bit where received data is read (see schematic).
ps_shutdown           bit    p0.0                    ;The +5V power supply shutdown bit (open drain!).
ps_latch              bit    p0.1                    ;The +5V power supply latch-on bit (open drain!).
power_enable          bit    p1.4                    ;The signal which indicates that the handset has been
                                                     ;turned-on by its own CPU - used to detect handset ringing.


;Constant definitions ---------------------------------------------------------------------------------

data_ram              equ    015h                    ;The beginning of data RAM.  This value was chosen to allow
                                                     ;up to three nested subroutine calls before the stack
                                                     ;collides with data RAM.

receive_mask          equ    00001000b               ;ANDed with data_port to remove everything but the
                                                     ;receive_data bit.
timer_start           equ    00010100b               ;Used with TCON to start the internal timer and edge detect
                                                     ;on INT0.
timer_stop            equ    00000100b               ;Used with TCON to stop the internal timer and edge detect
                                                     ;on INT0.
;================================================================================================================


;Power-on (hardware) reset routine --------------------------------------------------------------------

;When the CPU power-up, the program counter is loaded with the reset vector and execution begins at address
;#000h.  The next command sends the program counter to the "start" label.

                      org    000h                    ;    Reset location.
                      sjmp   start                   ;2   Jump to the start label of the main program.
;----------------------------------------------------------------------------------------------------------


;Interrupt service routine: INT0\ ---------------------------------------------------------------------

;The hardware interrupt INT0\ service routine is used to immediately power-down the handset CPU, display, and
;power supply.  This interrupt is activated if the "TALK" button on the cordless handset is pressed.

                      org    003h                    ;    INT0\ interrupt location.
external_int_0:       clr    ea                      ;1   Disable all interrupts.
                      ajmp   power_down              ;2   Jump to the power-down routine.
;----------------------------------------------------------------------------------------------------------


;Interrupt service routine: Counter/timer -------------------------------------------------------------

;The timer interrupt service routine is used to achieve a multi-second delay.  The interrupt_count register
;is decremented each time the ISR is activated and the "no data" message is displayed if the interrupt_count
;reaches zero.  This ISR is used specifically for waiting 5 seconds for the caller ID data to be transmitted
;by the base unit.  During this 5 second wait period, the LCD display is toggled on and off every 300ms to
;indicate that the system is working properly.  (See the HD44780 datasheet for more information about turning
;the display on and off)  If the 5 seconds expires, the display is turned on and the "no data" message is
;displayed.

                      org    00bh                    ;    Internal timer interrupt location.
timer_interrupt:      mov    tcon,#00001000b         ;2   Stop the timer and clear TF.
                      djnz   interrupt_count,exit_int ;2  Decrement interrupt_count and get-out if it is not
                                                     ;    zero.
                      mov    a,#00001100b            ;1   Load the accumulator with the display on command.
                      acall  send_command            ;2   Turn the display on.
                      ajmp   no_data                 ;2   If 5 seconds has elapsed, annunciate the "no data"
                                                     ;    error.
```

```
exit_int:             mov   a,interrupt_count          ;1  Load the interrupt count into the accumulator.
                      anl   a,#00000011b               ;1  Knock-off all but the two LSBs.
                      xrl   a,#00000011b               ;1  Exclusive OR the accumulator with #003h to see if the
                                                       ;   interrupt_count is divisible by 3.
                      jnz   reset_int_timer            ;2  If the result is non-zero, then reset the timer and
                                                       ;   and exit the ISR.
                      cpl   b.2                        ;1  Toggle the display on/off bit.
                      mov   a,b                        ;1  Load the accumulator with the display control bit.
                      acall send_command               ;2  Turn the display on or off, depending on the status
                                                       ;   of the display on/off bit.
reset_int_timer:      mov   tl,#0afh                   ;2  Initialize the low timer byte to expire after 100ms.
                      mov   th,#03ch                   ;2  Initialize the high timer byte to expire after 100ms.
                      mov   tcon,#timer_start          ;2  Start the timer.
                      reti                             ;2  Return from the interrupt.
;-------------------------------------------------------------------------------------------------------------------


;Main program -----------------------------------------------------------------------------------------------------

;Following power-on reset, the program immediately branches to the "start" label.  At this time, the +5V power
;supply is latched on such that it can only be turned-off by software.  The LCD display is initialized, the
;timer is stopped, and the power on interrupt is enabled.  This interrupt immediately shuts the system down when
;the "talk" button on the cordless handset is pressed.

start:                clr   ps_latch                   ;1  Latch the +5V power supply on so that it can only be
                                                       ;   turned-off by software.
                      setb  ps_shutdown                ;1  Release control of the +5V power supply shutdown
                                                       ;   signal.
                      mov   tcon,#timer_stop           ;2  Stop the timer and clear TF.
                      acall init_display               ;2  Initialize and clear the LCD display.
                      mov   ie,#10000001b              ;2  Enable external interrupt 0, the handset power on
                                                       ;   detector (goes low when the "talk" button is pressed).

;After the LCD display has been initialized, the software displays the software version_string on the first
;line of the LCD display and the "waiting for data" message on the second line.  This indicates that the
;CPU and display are powered-up and alive since it may take up to four seconds for the base unit to transmit
;the data.

                      mov   dptr,#version_string       ;2  Point the data pointer to the version_string address.
                      mov   a,#000h                    ;1  Load the accumulator with the address of the first
                                                       ;   character of the first line on the LCD display.
                      acall send_message               ;2  Send the version_string to line 1.
                      mov   dptr,#wait_message         ;2  Point the data pointer to the wait_message address.
                      mov   a,#040h                    ;1  Load the accumulator with the address of the first
                                                       ;   character of the second line on the LCD display.
                      acall send_message               ;2  Send the wait_message to line 2.

;Prior to sending the actual caller ID data, the base unit transmits 50 consecutive 1.5KHz cycles.  This
;33.3ms of 1.5KHz signal is used to help synchronize the handset unit to the incoming data.  By waiting for this
;period of 1.5KHz cycles, the handset unit will not be "fooled" by spurious pulses which may look like start
;bits while the 25Hz "wake-up" signal (or a handset ring signal) is being sent.

;The handset software will look for 5 (out of 50 possible) consecutive 1.5KHz high pulses at receive_data.  The
;low time of ;the 1.5KHz signal is not relevant.  Each high portion of a 1.5KHz signal should be 333us in length.
;An arbitrary tolerance of +/-20% will be applied to increase noise margin.  So, any signal whose high duration
;is between 266us and 400us will be considered to be part of the 1.5KHz signal.  The method of pulse detection
;is very similar to that which is used on the base unit.  Slight modifications have been made to account for
;processor speed and signal frequency.

;A total of about eight seconds is allowed to obtain the 1.5KHz pulses after the initial 20ms "wake-up" pulses are
;received.  This is necessary because the base unit transmits four seconds of the "wake-up" signal before it
;sends the actual caller ID information, and the handset CPU may be awakened by a ring before the base unit
;has a chance to acquire and decode the CID data.  The timer interrupt will be used to achieve this long period.
;Initializing the timer with #3cafh will cause a timer interrupt to occur every 100ms, so a total of 80
;interrupts will occur in eight seconds.  The ISR will decrement interrupt_count until it reaches zero.  If
;interrupt_count reaches zero, then the eight seconds have expired and the "no data" message is displayed.  If
```

```
;this is the case, it it is assumed that the handset was accidentally awakened or that the cordless data channel
;is too noisy to transmit data.

;If the 20 consecutive 1.5KHz pulses are detected within the allotted eight seconds, then the timer interrupt is
;disabled and the software proceeds to wait for a start bit (space signal).

                        mov     b,#00001000b                ;2  Load the B register with the display on/off  command.
                        mov     tcon,#timer_stop            ;2  Stop the timer and clear TF.
                        mov     tl,#0afh                    ;2  Initialize the low timer byte to expire after 100ms.
                        mov     th,#03ch                    ;2  Initialize the high timer byte to expire after 100ms.
                        mov     interrupt_count,#050h       ;1  Load the interrupt counter with the 80 interrupt
                                                            ;   countdown value.
                        setb    et0                         ;1  Enable the timer interrupt.
                        mov     tcon,#timer_start           ;2  Start the timer.

start_count:            mov     mark_count,#005h            ;1  Initialize the mark_count to #005d for countdown.

pulse_wait:             jnb     receive_data,$              ;2  Loop until receive_data goes high.

init_mark_max:          mov     a,#028h                     ;1  Initialize the accumulator with the maximum assertion
                                                            ;   time (400us).
port_high:              dec     a                           ;1  Decrement the accumulator.
                        jz      out_of_bounds               ;2  Get-out of the loop if the accumulator reaches zero.
                        jb      receive_data,port_high      ;2  Repeat while receive_data is high.

;When the program reaches the timeout label, the accumulator has #028h minus the number of port_high
;iterations that were detected or zero if the 1000us limit was exceeded.  The accumulator value corresponding
;to the minimum pulsewidth of 266us is calculated as follows:
;
;                           266us = 133 machine cycles
;
;                           133 cycles / 5 cycles in port_high = 27 port_high iterations
;
;                           27 decrements from #028h = 13 (#00dh)
;
;So, #00dh is the accumulator value corresponding to the minimum pulse width and the accumulator
;value corresponding to a pulsewidth greater than 400us is zero.  If the accumulator value is
;between #001h and #00dh, then the pulse is considered to be that of a valid 1.5KHz frequency.

timeout:                clr     c                           ;1  Explicitly clear the carry bit.
                        subb    a,#00eh                     ;1  This will set the carry bit if the accumulator
                                                            ;   has a value between zero and #00dh.
                        jnc     out_of_bounds               ;2  The pulse was too short if the c bit is clear.
                        djnz    mark_count,pulse_wait       ;2  Decrement the mark_count and repeat the pulse test.
                        sjmp    data_start                  ;2  When mark_count reaches zero, get ready to look for
                                                            ;   a start bit.
out_of_bounds:          jb      receive_data,$              ;2  Loop until the port goes low in case of timeout.
                        sjmp    start_count                 ;2  Restart looking for the 15 consecutive 600Hz pulses.

;Now that the 5 consecutive 1.5KHz pulses have been detected, the cordless handset software looks for a start
;bit.  The start bit consists of a space signal (900Hz) which is sent by the base unit.  A square wave with a
;900Hz frequency will have a high time of (1/900Hz)/2=556us.  To account for small changes in the frequency
;which may be caused by the wireless channel, a tolerance of 15% is applied to the 556us assertion time of the
;start bit.  So, a pulse whose duration is between 473us and 639us will be considered to be part of a start
;bit.  The start bit detection software is largely similar to that of the base unit with a few modifications
;for clock speed and assertion times.

data_start:             clr     et0                         ;1  Disable the timer interrupt.
                        mov     a,#00001100b                ;1  Load the accumulator with the display on command.
                        acall   send_command                ;2  Turn the display on.
                        mov     ram_ptr,#data_ram           ;1  Reset the RAM pointer to the beginning of RAM.
                        mov     byte_count,#022h            ;1  Reset the byte counter for a 33 byte countdown since
                                                            ;   it is known that the base unit will transmit 32 data
                                                            ;   bytes, a "dummy" byte, and a checksum word.
start_bit_check:        mov     a,data_port                 ;1  Copy the status of the input port to the accumulator.
                        anl     a,#receive_mask             ;1  Mask-off everything but the receive_data.
```

```
                    mov   trans,a                   ;1  Put the state of receive_data into the trans register.

look_for_trans:     mov   a,data_port               ;1  Copy the status of the input port to the accumulator.
                    anl   a,#receive_mask           ;1  Mask-off everything but the receive_data.
                    xrl   a,trans                   ;1  Exclusive OR the current state of receive_data with
                                                    ;   its previous state.
                    jz    look_for_trans            ;2  The result is zero, no transition has occurred, so
                                                    ;   continue to look for a transition.
transition_found:   mov   a,data_port               ;1  Copy the status of the input port to the accumulator.
                    anl   a,#receive_mask           ;1  Mask-off everything but the receive_data.
                    mov   trans,a                   ;1  Put the state of receive_data into the trans register.
                    mov   max_pw,#02dh              ;1  Load the max_pw register with the maximum pulse width.

measure_duration:   mov   a,data_port               ;1  Copy the status of the input port to the accumulator.
                    anl   a,#receive_mask           ;1  Mask-off everything but the receive_data.
                    xrl   a,trans                   ;1  Exclusive OR the current state of receive_data with
                                                    ;   its previous state.
                    jnz   determine_width           ;2  The result is non-zero, a transition has occurred, so
                                                    ;   determine its duration.
                    djnz  max_pw,measure_duration   ;2  If the maximum pulse width had not been exceeded,
                                                    ;   continue to measure its duration.

;Each pass through the measure_duration loop consists of 7 instruction cycles.  So, a maximum pulse width of
;639us will require about 45 (#02d) passes through the loop.  As with the maximum pulse width, the minimum pulse
;width of 473us will require 34 iterations of the measure_duration code segment.  34 (or more) decrements from
;the original max_pw value of 45 results in a max_pw value of 11 (or less).

;If the previous pulse was too long to be considered a start bit, the software will measure the remaining
;pulse duration.  If the remaining duration exceeds 15ms, then it is assumed that the transmission has
;ceased.  This prevents the software from getting "stuck" while looking for a start bit in the event of
;a mid-transmission failure.  15ms requires 7500 machine cycles with a 6MHz oscillator.

;The actual duration of the pulse_too_long routine (if a transition does not occur within the window) is about:
;                   7 machine cycles per low bit * #0d7h low bit decrements = 1505 machine cycles.
;                   1505 machine cycles per high bit decrement * 5 high bit decrements = 7525 machine cycles.
;                   7525 machine cycles = 15.050ms with a 6MHz oscillator.

pulse_too_long:     mov   pulse_count_high,#005h    ;1  Initialize the high byte of the 15ms timer.
                    mov   pulse_count_low,#0d7h     ;1  Initialize the low byte of the 15ms timer.
long_loop:          mov   a,data_port               ;1  Copy the status of the input port to the accumulator.
                    anl   a,#receive_mask           ;1  Mask-off everything but the receive_data.
                    xrl   a,trans                   ;1  Exclusive OR the current state of receive_data with
                                                    ;   its previous state.
                    jnz   transition_found          ;2  If the result is non-zero, then a transition has
                                                    ;   occurred within the 15ms window, so determine if
                                                    ;   it is a start bit.
                    djnz  pulse_count_low,long_loop ;2  Decrement the low byte of the 15ms measurement value
                                                    ;   and repeat the local transition test if it is not zero.
                    djnz  pulse_count_high,long_loop;2  Decrement the high byte of the 15ms measurement value
                                                    ;   and repeat the local transition test if it is not zero.
                    ajmp  check_data                ;2  Check the data integrity if the transmission has
                                                    ;   ceased.
determine_width:    mov   a,max_pw                  ;1  Copy the pulse width count into the accumulator to
                                                    ;   prepare for a direct subtraction.
                    clr   c                         ;1  Explicitly clear the carry bit.
                    subb  a,#00bh                   ;1  Subtract the minimum number of decrement iterations
                                                    ;   from the actual number of decrement iterations.
                    jc    init_start                ;2  The pulse was of an acceptable duration if the carry
                                                    ;   bit is set, so proceed to the next section of code.
                    sjmp  start_bit_check           ;2  If the pulse was too short, look for the next
                                                    ;   transition.

;Information from the base unit is transmitted at 300bps, which means that each bit will last for 3.33ms or
;1667 machine cycles with a 6MHz oscillator.  During each bit time, the software will integrate the
;receive_data signal to determine if it represents a logical zero (900Hz) or a logical one (300Hz).  This
;FSK integration scheme is very similar to that which the base unit performs on the CID data.
```

```
;The initial value of the integral is set to 128.  A high receive_data signal will cause the integral
;to be incremented while a low receive_data signal will cause the integral to be decremented.  Although
;the integral value will always be close to 128 at the end of the bit time, the maximum excursions
;from 128 can be used to distinguish between a zero (900Hz) and a one (300Hz).  The lower frequency
;signal will have a greater difference in extrema of the integral than that of the higher frequency.
;Integrating the receive_data signal also provides noise margin in the bit timing.  Since only the maximum
;and minimum integral values are significant, the bit clock can "leak" into adjacent bits while still
;differentiating between 900Hz and 300Hz.

;The start_bit_delay loop calculates the duration of the start bit pulse as determined by the previous
;section of code.  Then, a delay loop is executed to allow the start bit time to expire before the software
;starts to acquire the actual data bits.

;The max_pw register contains the number of decrement iterations which were performed on the start bit pulse.
;Since each decrement iteration requires 7 machine cycles, the duration of the start bit pulse will be 7 times
;the number of decrement iterations performed, in machine cycles.  With a bit time of 3.33ms, the amount of time
;before the actual data begins is 3.33ms minus the time spent measuring the first pulse.

init_start:           mov    a,#02dh                  ;1  Load the accumulator with the original max_pw value
                                                      ;   from the start bit detection code.
                      clr    c                        ;1  Explicitly clear the carry bit.
                      subb   a,max_pw                 ;1  Subtract the last value of max_pw.  The accumulator
                                                      ;   now contains the actual number of decrement iterations
                                                      ;   performed on the first start bit pulse.
                      inc    a                        ;1  Increment the accumulator to account for the machine
                                                      ;   cycles that have elapsed since the start bit was
                                                      ;   detected.
                      inc    a                        ;1  Increment the accumulator to account for the machine
                                                      ;   cycles that have elapsed since the start bit was
                                                      ;   detected.

;At this point, the accumulator contains the number of machine cycles (divided by 7) spent detecting the start
;bit thus far.  The bit time of 3.333ms equates to about 1667 machine cycles with a 6MHz oscillator.  So, the
;software needs to wait for 1667-7*(accumulator) cycles before the first data bit arrives.  Or, if a delay loop
;can be engineered to require 7 machine cycles per iteration, then that loop would need to be executed
;(1667/7=238 - accumulator) times.  In other words, a 7 machine cycle delay loop would have the effect of
;dividing the delay equation by seven, thus eliminating a multiplication and enabling the use of an eight bit
;counter to accomplish the delay.

                      mov    b,a                      ;1  Store the number of decrement iterations in the B
                                                      ;   register to prepare for subtraction.
                      mov    a,#0eeh                  ;1  Load the accumulator with the 238 iteration start
                                                      ;   value.
                      clr    c                        ;1  Explicitly clear the carry bit.
                      subb   a,b                      ;1  Subtract the number of decrement iterations from the
                                                      ;   238 iteration start value.  The accumulator now
                                                      ;   contains the number of 7 cycle delay iterations needed
                                                      ;   before the data bits arrive.
                      mov    temp,a                   ;1  Load the temporary register with the number of 7 cycle
                                                      ;   delay iterations for countdown.
start_bit_delay:      mul    ab                       ;4  Used as a single byte instruction to produce four NOPs.
                      nop                             ;1  Do nothing for one cycle.
                      djnz   temp,start_bit_delay     ;2  Continue to loop until the start bit time has
                                                      ;   completely elapsed.  The start_bit_delay routine
                                                      ;   requires 7 machine cycles per iteration.

;Now that the start bit time has elapsed, the actual data bits can be acquired.  The software for the data byte
;acquisition is almost identical to that of the base unit with a few modifications for clock frequency, bit time,
;and FSK frequencies.

;To make the timer expire after the 3.333ms bit time, it must be initialized with #f97ch to allow 1667 machine
;cycles to elapse before expiration.

                      mov    bit_count,#008h          ;1  Initialize the bit counter with 8 bits (countdown).
                      mov    data_byte,#000h          ;1  Clear the data_byte register.
```

```
get_bit:            mov   tcon,#timer_stop          ;2  Stop the timer.
                    mov   th,#0f9h                  ;2  Initialize the high timer byte.
                    mov   tl,#07ch                  ;2  Initialize the low timer byte.
                    mov   tcon,#timer_start         ;2  Start the timer
                    mov   integral,#080h            ;1  Set the initial integral value to 128.
                    mov   max_value,#080h           ;1  Set the initial maximum value to 128.
                    mov   min_value,#080h           ;1  Set the initial minimum value to 128.

bit_start:          jb    tf,bit_timeout            ;2  Go to bit_timeout if the timer has expired.
                    jb    receive_data,inc_sum      ;2  Increment the integral if receive_data is high.

dec_sum:            dec   integral                  ;1  Since receive_data is low, decrement the integral.
                    mov   b,integral                ;2  Copy the integral value into the B register.
                    mov   a,min_value               ;1  Copy the latest minimum value into A for comparison.
                    cjne  a,b,new_min               ;2  If the integral and the old minimum value are not
                                                    ;   equal, see if the value of integral constitutes a
                                                    ;   new minimum.
                    sjmp  burn_cycles               ;2  Burn cycles to make everything even.

new_min:            jc    burn_cycles               ;2  If the carry bit is set, then the integral value
                                                    ;   is not a new minimum. (min_value<integral)
                    mov   min_value,b               ;2  Make min_value equal to the integral value.
                    sjmp  bit_start                 ;2  Continue to integrate receive_data.

inc_sum:            inc   integral                  ;1  Since receive_data is high, increment the integral.
                    mov   b,integral                ;2  Copy the integral value into the B register.
                    mov   a,max_value               ;1  Copy the latest maximum value into A for comparison.
                    cjne  a,b,new_max               ;2  If the integral and the old maximum value are not
                                                    ;   equal, see if the value of integral constitutes a
                                                    ;   new maximum.
                    sjmp  burn_cycles               ;2  Burn cycles to make everything even.

new_max:            jnc   burn_cycles               ;2  If the carry bit is clear, then max_value is less
                                                    ;   than the integral, hence no new maximum exists.
                                                    ;   (integral<max_value)
                    mov   max_value,b               ;2  Make max_value equal to the integral value.
                    sjmp  bit_start                 ;2  Continue to integrate receive_data.

;The purpose of the burn_cycles code segment is to "time pad" the integration sequence so that all
;operations, whether they are new_max, new_min, or neither, will take the same amount of time.  This
;ensures that the slope (gain) of the integration remains constant in all cases.  From the bit_start
;label, all iterations require 16 machine cycles (32us with a 6MHz oscillator).

burn_cycles:        nop                             ;1  Use one machine cycle.
                    nop                             ;1  Use one machine cycle.
                    sjmp  bit_start                 ;2  Continue to integrate receive_data.

;After the bit timer has expired, it is necessary to determine whether the most recently acquired bit was
;a logical 0 (900Hz) or a logical one (300Hz).  This is done by taking the difference between the
;extrema of the integral.  In other words, the difference between max_value and min_value is used
;to make the distinction between zero and one.

;Ideally, a 300Hz signal will produce an extrema difference of 52.  This value is derived as follows:
;
;                   300Hz = 3333us per cycle so half of a cycle will last 3333us/2 = 1667us
;
;                   Since each iteration of the integral routine requires 32us, the ideal difference
;                   between maximum and minimum will be 1667us/32us = 52.
;
;The extrema difference for a 900Hz (logical zero) signal is calculated in a similar fashion:
;
;                   900Hz = 1111us per cycle so half of a cycle will last 1111us/2 = 556us
;
;                   Since each iteration of the integral routine requires 32us, the ideal difference
;                   between maximum and minimum will be 556us/32us = 17.
```

```
;
;The extrema difference for a constant signal is calculated in a similar fashion:
;
;                       0Hz = 3333us per cycle
;
;                       Since each iteration of the integral routine requires 32us, the ideal difference
;                       between maximum and minimum will be 3333us/32us = 104.

;Fortunately, the difference between a logical zero and a logical one is quite significant.  The threshold
;is arbitrarily chosen to be 35.  So, an extrema difference less than 35 will be considered a zero and
;an extrema difference of 35 or more will be considered a logical one.  This will allow for fairly high noise
;margins in that spurious spikes on the receive_data signal or a mis-synchronized bit clock will not contribute
;significantly to the outcome.  In other words, fast spikes on receive_data and variations in bit synchronization
;will be integrated-out (lowpass filtered).  Furthermore, an extrema of 90 or more will indicate that the
;transmission has stopped (i.e. a constant signal for more than 90*32=2880us).

bit_timeout:            mov     tcon,#timer_stop            ;2  Stop the timer.
                        mov     a,max_value                 ;1  Copy the maximum value into A for a subtraction.
                        clr     c                           ;1  Explicitly clear the carry bit.
                        subb    a,min_value                 ;1  Subtract the minimum value from the maximum value.
                        mov     max_value,a                 ;1  Temporarily store the extrema difference in the
                                                            ;   max_value register.
                        clr     c                           ;1  Explicitly clear the carry bit.
                        subb    a,#05ah                     ;1  Subtract decimal 90 to see if the difference in the
                                                            ;   extrema is 90 or more.  If so, then the transmission
                                                            ;   has ceased.
                        jnc     check_data                  ;2  If the carry bit is clear, then the transmission has
                                                            ;   ceased, so stop getting bits.
                        mov     a,max_value                 ;1  Load the accumulator with the max_value register,
                                                            ;   which contains the extrema difference.
                        clr     c                           ;1  Explicitly clear the carry bit.
                        subb    a,#025h                     ;1  Subtract the zero/one threshold.  If C is set, then
                                                            ;   the bit was a zero.
                        cpl     c                           ;1  Complement the carry bit to make it representative
                                                            ;   of the received bit.
                        mov     a,data_byte                 ;1  Load the data_byte into the accumulator.
                        rrc     a                           ;1  Roll C onto the data_byte.
                        mov     data_byte,a                 ;1  Put the data_byte back with the new bit rolled onto
                                                            ;   it.
                        djnz    bit_count,get_bit           ;2  Get another bit until eight bits have been received.

store_byte:             mov     @ram_ptr,a                  ;1  Copy the most recently received byte into the current
                                                            ;   memory location.
                        inc     ram_ptr                     ;1  Increment the ram_ptr.
                        djnz    byte_count,long_start_jump  ;2  Decrement the byte counter and get the next byte until
                                                            ;   it reaches zero.
                        sjmp    check_data                  ;2  Go back to start_check to look for the next start bit.

long_start_jump:        ajmp    start_bit_check             ;2  Go back to looking for the next start bit.


;The check_data routine determines the integrity of the received data.  First, check_data determines if more than
;16 bytes were received.  If not, then the program annunciates a "fatal data error" because something gone
;extremely wrong during the transmission or reception of the data.  The software will not attempt to display
;fewer than 16 bytes because it is assumed that the data is corrupt beyond recognition.

;However, more than 16 bytes were received, the software will tally-up a checksum on the received data and
;compare it to the checksum that was supposedly sent by the base unit.  If the two checksums do not match, the
;epsilon character is placed in RAM so that it will appear at the second-to-last character on the second line
;of the LCD display.  This way, the user can identify a cordless transmission error AND a caller ID checksum
;error on the cordless display. (i.e. the rightmost character on the second line will contain the epsilon
;character if a caller ID checksum error occurred at the base unit while the second-to-rightmost character
;on the second line will contain the epsilon character if a checksum error occurred during the cordless
;transmission)  On any given transmission, any combination of checksum errors (or lack thereof) can occur.

check_data:             mov     a,byte_count                ;1  Load the accumulator with the final value of
```

```
                                                ;   byte_count.
                        clr     c               ;1  Explicitly clear the carry bit.
                        subb    a,#012h         ;1  Subtract 18 to see if more than 16 characters have
                                                ;   been received (to include the "dummy" character).
                        jnc     fatal_error     ;2  If the carry bit is clear, then fewer than 16 data
                                                ;   bytes have been received.  This means that there
                                                ;   is not enough data to be worth displaying, so
                                                ;   annunciate the fatal error.
                        mov     checksum,#000h  ;1  Clear the checksum location.
                        mov     ram_ptr,#data_ram+1 ;1  Position the RAM pointer to the beginning of data RAM
                                                ;   after the "dummy" byte.
                        mov     b,#020h         ;2  Load the B register with a 32 byte countdown value.
add_checksum:           mov     a,@ram_ptr      ;1  Load the accumulator with the current data byte.
                        add     a,checksum      ;1  Add checksum to the accumulator.
                        mov     checksum,a      ;1  Put the new checksum back into its location.
                        inc     ram_ptr         ;1  Move the RAM pointer to the next location.
                        djnz    b,add_checksum  ;2  Repeat until all 32 bytes of data have been totaled.
                        mov     b,@ram_ptr      ;2  Load the B register with the checksum value that was
                                                ;   sent by the base unit.
                        clr     c               ;1  Explicitly clear the carry bit.
                        subb    a,b             ;1  Subtract the calculated checksum from that which was
                                                ;   sent by the base unit.
                        jz      data_good       ;2  If the result is zero, then the checksums match and the
                                                ;   data has probably not been corrupted.

;The data_bad section of code overwrites the RAM location that holds the second-to-last character of the second
;line with the epsilon character to indicate that a checksum error occurred during the cordless transmission.

data_bad:               mov     ram_ptr,#data_ram+31    ;1  Position the RAM pointer to where the second to last
                                                ;   character of the second line should be in RAM.
                        mov     a,#0e3h         ;1  Load the accumulator with the value of epsilon.
                        mov     @ram_ptr,a      ;1  Place the epsilon character in RAM to be sent to the
                                                ;   display.

;The data_good section of code simply moves the first 32 bytes of received data from RAM to the display.  Note
;that the first 32 bytes of received data begin at data_ram+1 because the "dummy" byte occupies the first
;location in data RAM.

data_good:              mov     b,#010h         ;2  Load the B register with a 16 character countdown
                                                ;   value.
                        mov     ram_ptr,#data_ram+1 ;1  Position the RAM pointer to the beginning of data RAM,
                                                ;   which is where the first line of text resides after
                                                ;   the "dummy" byte.
                        mov     a,#000h         ;1  Load the accumulator with the address of the first
                                                ;   line on the LCD display.
                        acall   set_disp_address    ;2  Move the display cursor to the first line.
send_line_1:            mov     a,@ram_ptr      ;1  Load the current character into the accumulator.
                        acall   validate_char   ;2  Ensure the character is valid.
                        acall   send_disp_data  ;2  Send the character to the LCD display.
                        inc     ram_ptr         ;1  Increment the RAM pointer to the next location.
                        djnz    b,send_line_1   ;2  Repeat the sequence until all 16 characters of the
                                                ;   first line have been displayed.
                        mov     b,#010h         ;2  Load the B register with a 16 character countdown
                                                ;   value.
                        mov     a,#040h         ;1  Load the accumulator with the address of the second
                                                ;   line on the LCD display.
                        acall   set_disp_address    ;2  Move the display cursor to the second line.
send_line_2:            mov     a,@ram_ptr      ;1  Load the current character into the accumulator.
                        acall   validate_char   ;2  Ensure the character is valid.
                        acall   send_disp_data  ;2  Send the character to the LCD display.
                        inc     ram_ptr         ;1  Increment the RAM pointer to the next location.
                        djnz    b,send_line_2   ;2  Repeat the sequence until all 16 characters of the
                                                ;   second line have been displayed.

;The show_message section of code waits for five seconds while it polls the power_enable bit.  If the
;power_enable bit goes low during this time for more than 500ms, the message will be displayed for the
```

```
;remaining time during which power_enable is low.  This indicates that the phone is ringing.  When the phone
;stops ringing, the power_enable bit will go high and the CCID CPU will wait for another five seconds before
;shutting itself down.

;If the power_enable bit does not go low for more than 500ms during the allotted five seconds, then the phone
;has stopped ringing and the software will wait for another five seconds before shutting itself down.

;Five seconds with a 6MHz oscillator requires 2,500,000 machine cycles which equals about 38 (#026h) full
;timer expirations.

show_message:         mov   b,#026h                 ;2  Initialize the B register with the 38 expiration
                                                    ;   countdown value.
                      mov   dptr,#0000h             ;2  Reset the data pointer which will be used to measure
                                                    ;   the low duration of power_enable.

start_timer:          mov   tcon,#timer_stop        ;2  Stop the timer and clear TF.
                      mov   tl,#000h                ;2  Initialize the low timer byte for a full timer count.
                      mov   th,#000h                ;2  Initialize the high timer byte for a full timer count.
                      mov   tcon,#timer_start       ;2  Start the timer.

ring_check:           jb    tf,timer_exp_count      ;2  If the timer expires, decrement the number of remaining
                                                    ;   expirations and reset the timer.
                      jb    power_enable,reset_ring ;2  If the power_enable bit is high, keep resetting the
                                                    ;   ring counter.

;The duration of the low portion of power_enable needs to be measured to determine if the phone is still ringing.
;If the low signal at power_enable lasts for more than 500ms, then the phone is ringing.  If this is the case,
;the software will wait until the power_enable bit goes high before executing the final five second delay
;prior to shutdown.

;If the low signal at power_enable does not last for more than 500ms, then the software will continue to count
;down the five seconds at ring_check prior to executing the final five second delay.

;If power_enable is low, a complete iteration of the ring_check loop will require 13 machine cycles.  This
;means that the ring_check loop will be executed 19,231 (#4b1f) times for a 500ms low at power_enable.  Since
;the power_enable_low section of code only checks the high byte of DPTR, the actual low time will be valid
;in .4992 seconds (i.e. DPTR = #4b00h).

power_enable_low:     inc   dptr                    ;2  Increment the data pointer since power_enable is low.
                      mov   a,dph                   ;1  Load the high byte of the data pointer into the
                                                    ;   accumulator.
                      clr   c                       ;1  Explicitly clear the carry bit.
                      subb  a,#04bh                 ;1  Subtract the number of expected increments during
                                                    ;   a 500ms low at power_enable.
                      jz    phone_ringing           ;2  If the result is zero, then the power_enable signal
                                                    ;   has been low for 500ms.
                      sjmp  ring_check              ;2  Go back to checking power_enable and the timer flag.

reset_ring:           mov   dptr,#0000h             ;2  Reset the data pointer which will be used to measure
                                                    ;   the low duration of power_enable.
                      sjmp  ring_check              ;2  Go back to checking power_enable and the timer flag.

timer_exp_count:      djnz  b,start_timer           ;2  Continue counting time until the timer has expired
                                                    ;   38 times.
                      sjmp  wait_5_seconds          ;2  Execute the final five seconds delay before power
                                                    ;   down.

;If a 500ms low was detected at power_enable, then the phone is still ringing.  It is desirable to display
;the received data for the duration of the phone ring so the user will have a chance to see the data before
;answering the phone.  So, the phone_ringing section of code simply loops until the phone stops ringing.  After
;the power_enable bit goes high (the phone stops ringing), the program jumps to a five second delay prior to
;shutting itself down.

phone_ringing:        jnb   power_enable,$          ;2  Loop until power_enable goes high, which will indicate
                                                    ;   that the phone is no longer ringing.
                      sjmp  wait_5_seconds          ;2  Execute the final five second delay before power down.
```

```
;The purpose of the fatal_error section of code is to annunciate a fatal data error.  This can occur if more than
;zero and fewer than 16 characters were received over the cordless channel.  This section of code displays the
;the software version string on the first line of the LCD display and the "fatal error" message on the second
;line.  The program then jumps to the five second delay before shutting itself down.

fatal_error:          clr   et0                       ;1  Disable the timer interrupt.
                      mov   dptr,#version_string      ;2  Load the data pointer with the address of the
                                                      ;   version_string.
                      mov   a,#000h                   ;1  Load the accumulator with the address of the first
                                                      ;   line of the display.
                      acall send_message              ;2  Send the version_string to the display.
                      mov   dptr,#fatal_error_msg     ;2  Load the data pointer with the address of the "fatal
                                                      ;   error" message.
                      mov   a,#040h                   ;1  Load the accumulator with the address of the second
                                                      ;   line of the display.
                      acall send_message              ;2  Send the "no data" message to the display.
                      sjmp  wait_5_seconds            ;2  Allow five seconds for the user to read the display
                                                      ;   before shutting down.

;The no_data section of code annunciates to the user that there is no data being transmitted by the base unit.
;This can occur if the handset CPU and display were accidentally awakened by an erroneous 20ms pulse, which is
;required to enable the +5V converter.  The "no data" error may also indicate that the data channel is too
;noisy to transmit data.  This section of code places the "no data" message on the second line of the LCD
;display and the software version_string on the first.

;After the "no data" message is sent to the display, the CPU will wait for about 5 seconds before it turns off
;the +5V supply to enable the user to view the message.

no_data:              clr   et0                       ;1  Disable the timer interrupt.
                      mov   dptr,#version_string      ;2  Load the data pointer with the address of the
                                                      ;   version_string.
                      mov   a,#000h                   ;1  Load the accumulator with the address of the first
                                                      ;   line of the display.
                      acall send_message              ;2  Send the version_string to the display.
                      mov   dptr,#no_data_string      ;2  Load the data pointer with the address of the "no
                                                      ;   data" message.
                      mov   a,#040h                   ;1  Load the accumulator with the address of the second
                                                      ;   line of the display.
                      acall send_message              ;2  Send the "no data" message to the display.

;The wait_five_seconds section of code creates a five second delay.  Five seconds with a 6MHz oscillator requires
;2,500,000 machine cycles which equals about 38 (#026h) full timer expirations.

wait_5_seconds:       mov   ie,#10000001b             ;2  Enable external interrupt 0, the handset power on
                                                      ;   detector (goes low when the "talk" button is pressed).
                      mov   b,#026h                   ;2  Initialize the B register with the 38 expiration
                                                      ;   countdown value.
reset_timer:          mov   tcon,#timer_stop          ;2  Stop the timer and clear TF.
                      mov   tl,#000h                  ;2  Initialize the low timer byte for a full timer count.
                      mov   th,#000h                  ;2  Initialize the high timer byte for a full timer count.
                      mov   tcon,#timer_start         ;2  Start the timer.

wait_loop:            jnb   tf,wait_loop              ;2  Loop until the timer expires.
                      djnz  b,reset_timer             ;2  Continue counting time until the timer has expired
                                                      ;   38 times.

;The power_down section of code releases the power supply latch and turns-off the +5V power supply.  The CPU
;is put into "power down" mode immediately following the +5V power supply shutdown.  This stops the oscillator
;and prevents the CPU from executing unknown code while the power supply discharges.

power_down:           setb  ps_latch                  ;1  Release the power supply latch.
                      clr   ps_shutdown               ;1  Pull the open-drain output low to disable the +5V
                                                      ;   power supply.
                      mov   pcon,#00000010b           ;2  Put the CPU into power down mode while the +5V supply
                                                      ;   bleeds down.  This stops the oscillator and the only
```

```
                                                    ;    way to start-up again is with a hardware reset.

;This is the end of the main program.
;-------------------------------------------------------------------------------------------------------------


;Fixed messages -----------------------------------------------------------------------------------------------
;The following ASCII text is used for standard messages which may be displayed on the LCD display.  The data is
;fixed in ROM.

version_string:       db     'CCID Handset 1.3'          ;Must be exactly sixteen characters in length.
wait_message:         db     'Waiting for data'          ;Must be exactly sixteen characters in length.
fatal_error_msg:      db     'Fatal data error'          ;Must be exactly sixteen characters in length.
no_data_string:       db     'No data received'          ;Must be exactly sixteen characters in length.
;-------------------------------------------------------------------------------------------------------------


;****************************** Display & formatting subroutines ******************************************

;Note: All of these subroutines were copied from the base unit.  Some of them have been slightly modified for use
;in the handset software.  Modifications are noted in the header comments of each subroutine.

;                     init_display: used to initialize and clear the display after power-up
;                     send_command: used to send a command byte to the display
;                     send_disp_data: used to send a data byte to the display
;                     set_disp_address: used to set the position of the cursor on the display
;                     disp_busy: used to check the busy status of the display
;                     send_message: used to send a 16 byte message to the first line of the LCD display
;                     validate_char: used to validate received ASCII characters

;*************************************************************************************************************

;Subroutine: init_display -------------------------------------------------------------------------------------
;This subroutine has been copied from the base unit software.  Slight modifications have been made to account
;for the handset CPU's lower oscillator speed.

;According to the HD44780 datasheet, the display and controller hardware requires 10ms after power-up to
;guaranty a successful internal power-on reset.  The software will wait for 15ms to provide a few extra
;milliseconds of margin.  Since 15ms requires 7500 (#1d4ch) machine cycles with a 6MHz oscillator,
;the internal 16 bit timer will be used to achieve the delay.  The timer bytes are initialized with
;#ffffh-#1d4ch = #e2b3h so that the timer overflow flag will be set when 7500 machine cycles have elapsed.
;After the initial power-up delay, an HD44780 "function set" command is sent to the display to configure
;it for an 8 bit data interface.  The HD44780 requires a maximum of 120us to complete this command.  The
;display's busy flag cannot be checked during this operation, so the send_command subroutine is not used.
;Instead, the "function set" command is "hard sent".  All subsequent operations can use the busy flag.

;Registers used:      Accumulator, disp_port
;Registers affected:  a, disp_port
;SFRs used:           tl, th, tcon, tf
;Calls:               send_command


init_display:         mov    disp_port,#0ffh          ;2  Set all display data bits high so they cannot sink
                                                      ;   current.
                      clr    disp_strobe              ;1  Set the display strobe low.
                      mov    tl,#0b3h                 ;2  Load the low timer byte with the start value.
                      mov    th,#0e2h                 ;2  Load the high timer byte with the start value.
                      mov    tcon,#timer_start        ;2  Start the timer.
init_loop:            jnb    tf,init_loop             ;2  Loop until the timer overflow flag is set.
                      mov    tcon,#000h               ;2  Stop the timer and clear the overflow flag.
                      clr    disp_reg_sel             ;1  Clear the display register select bit to indicate
                                                      ;   that a command is going to be sent.
                      clr    disp_read                ;1  Clear the display read bit to indicate that data
                                                      ;   is going to be written.
                      mov    disp_port,#00111000b     ;2  Load the display port with the display function
                                                      ;   set data. (8 bit interface, 2 lines, 5x7 font)
```

```
                    setb   disp_strobe              ;1  Set the display enable bit.
                    nop                             ;1  Wait for one instruction cycle for timing margin.
                    clr    disp_strobe              ;1  Clear the display enable bit to provide a falling
                                                    ;   edge.  This clocks the data into the display.
                    mov    a,#080h                  ;1  Load the accumulator with a countdown value to
                                                    ;   provide at least 120us of delay.
init_delay:         dec    a                        ;1  Decrement the accumulator.
                    jnz    init_delay               ;2  Continue to decrement the accumulator until it reaches
                                                    ;   zero.

;After the initialization delays, the display is turned-on and cleared and the cursor (display RAM pointer)
;is placed in the home position.

                    mov    a,#00001100b             ;1  Load the accumulator with the display on/off
                                                    ;   command. (display on, cursor off, no blink)
                    acall  send_command             ;2  Send the command out to the display.
                    mov    a,#00000110b             ;1  Load the accumulator with the entry mode byte.
                                                    ;   (increment cursor, no scroll)
                    acall  send_command             ;2  Send the command out to the display.
                    mov    a,#00000001b             ;1  Load the accumulator with the clear display
                                                    ;   and home cursor command.
                    acall  send_command             ;2  Send the command out to the display.
                    ret                             ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------


;Subroutine: send_command ------------------------------------------------------------------------------------
;This subroutine has been copied verbatim from the base unit software with no modifications.

;The purpose of the send_command subroutine is to send a command word to the LCD display.  The command
;word must be in the accumulator prior to calling this subroutine.  This subroutine operates in the
;"closed loop" mode by ensuring that the HD44780 is not busy prior to sending the command.  Also, the
;"closed loop" mode enables the software to send commands to the display as rapidly as possible as opposed
;to using fixed worst-case delays.  The accumulator is not affected by this subroutine.

;Registers used:      Accumulator, disp_port
;Registers affected:  none
;SFRs used:           none
;Calls:               disp_busy

send_command:       acall  disp_busy                ;2  Make sure the display is not busy.
                    clr    disp_reg_sel             ;1  Clear the display register select bit to indicate
                                                    ;   that a command is going to be sent.
                    clr    disp_read                ;1  Clear the display read bit to indicate that data
                                                    ;   is going to be written.
                    mov    disp_port,a              ;1  Copy the accumulator contents to the display
                                                    ;   data port.
                    nop                             ;1  Wait for one machine cycle to provide timing margin.
                    setb   disp_strobe              ;1  Set the display enable bit.
                    nop                             ;1  Wait for one machine cycle to provide timing margin.
                    clr    disp_strobe              ;1  Clear the display enable bit to provide a falling
                                                    ;   edge.  This clocks the data into the display.
                    ret                             ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------


;Subroutine: send_disp_data ----------------------------------------------------------------------------------
;This subroutine has been copied verbatim from the base unit software with no modifications.

;The purpose of the send_disp_data subroutine is to send a character to the LCD display.  The ASCII value
;of the character must be in the accumulator prior to calling this subroutine.  The character is displayed
;at the current DD RAM location internal to the HD44780.  This subroutine operates in the "closed loop" mode by
;ensuring that the HD44780 is not busy prior to sending the data.  Also, the "closed loop" mode enables the
;software to send commands to the display as rapidly as possible as opposed to using fixed worst-case delays.
;The accumulator is not affected by this subroutine.
```

```
;Registers used:            Accumulator, disp_port
;Registers affected:  none
;SFRs used:                 none
;Calls:                     disp_busy

send_disp_data:      acall  disp_busy                ;2  Make sure the display is not busy.
                     setb   disp_reg_sel             ;1  Set the display register select bit to indicate
                                                     ;   that a data word is going to be sent.
                     clr    disp_read                ;1  Clear the display read bit to indicate that data
                                                     ;   is going to be written.
                     mov    disp_port,a              ;1  Copy the accumulator contents to the display
                                                     ;   data port.
                     nop                             ;1  Wait for one machine cycle to provide timing margin.
                     setb   disp_strobe              ;1  Set the display enable bit.
                     nop                             ;1  Wait for one machine cycle to provide timing margin.
                     clr    disp_strobe              ;1  Clear the display enable bit to provide a falling
                                                     ;   edge.  This clocks the data into the display.
                     ret                             ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------


;Subroutine: set_disp_address ---------------------------------------------------------------------------
;This subroutine has been copied verbatim from the base unit software with no modifications.

;The purpose of the set_disp_address subroutine is to move the HD44870's internal DD RAM pointer to a specific
;location.  The desired location must be in the accumulator prior to calling this subroutine.  This subroutine
;operates in the "closed loop" mode by ensuring that the display is not busy prior to setting the address.
;Also, the "closed loop" mode enables the software to send commands to the display as rapidly as possible as
;opposed to using fixed worst-case delays.

;Registers used:      Accumulator
;Registers affected:  Accumulator
;SFRs used:           none
;Calls:               send_command

set_disp_address:    acall  disp_busy                ;2  Make sure the display is not busy.
                     orl    a,#10000000b             ;1  Set the MSB of the accumulator to indicate to the
                                                     ;   display that the address pointer is going to be
                                                     ;   modified.  (the display address is contained in the
                                                     ;   lower seven bits of the accumulator)
                     acall  send_command             ;2  Send the new address pointer to the display.  This is
                                                     ;   a nested subroutine call.
                     ret                             ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------


;Subroutine: disp_busy ----------------------------------------------------------------------------------
;This subroutine has been copied verbatim from the base unit software with no modifications.

;The purpose of the disp_busy subroutine is to ensure that the HD44780 is not busy.  If it is busy, the
;subroutine will wait until it is not.  This subroutine is called by any display communication routines
;prior to sending or receiving data.

;Registers used:      none
;Registers affected:  none
;SFRs used:           none
;Calls:               none

disp_busy:           mov    disp_port,#0ffh          ;2  Set the display port bit high to configure them
                                                     ;   as inputs and prevent them from sinking current.
                     clr    disp_reg_sel             ;1  Clear the display register select bit to indicate
                                                     ;   that the busy flag is going to be read.
                     setb   disp_read                ;1  Set the display read bit to indicate that a read
                                                     ;   operation is going to take place.
                     setb   disp_strobe              ;1  Set the display strobe.
                     nop                             ;1  Wait one machine cycle to provide timing margin.
```

```
busy:               jb     disp_port.7,busy          ;2  Loop the busy flag is clear.
                    clr    disp_strobe               ;1  Clear the display strobe.
                    ret                              ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------


;Subroutine: send_message ------------------------------------------------------------------------------------
;This subroutine has been copied from the base unit software.  The subroutine has been modified to allow the
;message to be displayed at any location on the screen.  The address of the starting location on the LCD must
;be in the accumulator prior to calling this subroutine.

;The purpose of the send_message subroutine is to send a 16 character message to the second line of the LCD
;display.  On entry, the data pointer (DPTR) must contain the address of the message to be sent.  This
;subroutine assumes that exactly 16 characters are to be sent.

;Registers used:      Accumulator, dptr, b, temp
;Registers affected:  Accumulator, dptr, b, temp
;SFRs used:           none
;Calls:               send_disp_data, set_disp_address

send_message:       acall  set_disp_address          ;2  Set the display address pointer.
                    mov    b,#010h                   ;2  Load the B register with the 16 character countdown
                                                     ;   value.
                    mov    temp,#000h                ;1  Initialize the temporary register with the offset
                                                     ;   value.

message_loop:       mov    a,temp                    ;1  Load the accumulator with the offset value.
                    movc   a,@a+dptr                 ;2  Load the next message byte to be sent into the
                                                     ;   accumulator.
                    acall  send_disp_data            ;2  Send it out to the display.
                    inc    temp                      ;1  Increment the data pointer offset value.
                    djnz   b,message_loop            ;2  Repeat until all 16 bytes have been sent.
                    ret                              ;2  Return to the calling address.
;-------------------------------------------------------------------------------------------------------------


;Subroutine: validate_char -----------------------------------------------------------------------------------
;This subroutine has been copied from the base unit software.  The subroutine has been modified to allow dashes,
;parenthesis, asterisks, the epsilon character, and lower case letters a through z.  A box character is now
;used to indicate an invalid character.

;The purpose of the validate_char subroutine is to determine whether or not a character value is a valid
;ASCII character sent by the base unit.  Since the base unit is only capable of sending uppercase letters A
;through Z, numerals zero through 9, spaces, dashes, parenthesis, asterisks, and the epsilon character, any
;other character value will be considered invalid.  In other words, a value other than the aforementioned was
;most likely corrupted during the data acquisition sequence.  It should be noted, however, that this method of
;determining character validity is by no means bullet proof.  It is possible to have one or more corrupted bits
;in an ASCII value and still have a valid ASCII character.  This subroutine is called as an error minimization
; and annunciation function.

;If the character is determined to be invalid, it will be replaced with a box character (HD44780 ASCII #0dbh)
;and returned via the accumulator.  The calling code segment will the send the box character to the LCD display
;which will inform the user that the character has been corrupted.  This way, base unit character errors will be
;indicated by an asterisk and characters which were corrupted during the cordless transmission will be indicted
;by a box character.

;Upon entering this subroutine, the accumulator must have the ASCII value to be checked.  When the subroutine
;is finished, the accumulator will contain the original value if it is a valid character or an asterisk if not.

;Registers used:      Accumulator, original_value
;Registers affected:  Accumulator (if data is invalid)
;SFRs used:           none
;Calls:               none

validate_char:      mov    original_value,a          ;1  Copy the original character value into a temporary
                                                     ;   holding register.
```

```
                        clr     c                       ;1  Explicitly clear the carry bit.
                        subb    a,#020h                 ;1  Subtract the ASCII value of a space from the
                                                        ;   accumulator.
                        jc      invalid_char            ;2  If the carry bit is set, the value was less than
                                                        ;   #020h and hence, is invalid.
                        jz      valid_char              ;2  If the accumulator is zero, then the character
                                                        ;   is a space and is valid.
                        subb    a,#008h                 ;1  Subtract the difference between ASCII zero and ASCII
                                                        ;   left parenthesis "(".
                        jc      invalid_char            ;2  If the original value was less than ASCII "("
                                                        ;   (#028h) and not a space, then it was invalid.
                        jz      valid_char              ;2  If the result is zero, then the character is a left
                                                        ;   parenthesis and therefore is valid.
                        dec     a                       ;1  Decrement the accumulator to see if the character is
                                                        ;   a right parenthesis ")".
                        jz      valid_char              ;2  If the result is zero, then the character is a right
                                                        ;   parenthesis and therefore is valid.
                        dec     a                       ;1  Decrement the accumulator to see if the character is
                                                        ;   an asterisk "*".
                        jz      valid_char              ;2  If the result is zero, then the character is an
                                                        ;   asterisk and therefore is valid.
                        mov     a,original_value        ;1  Load the original character value into the accumulator.
                        clr     c                       ;1  Explicitly clear the carry bit.
                        subb    a,#02dh                 ;1  Subtract the ASCII value of a dash "-".
                        jz      valid_char              ;2  If the result is zero, then the character is a dash and
                                                        ;   is therefore valid.

;At this point, all valid characters with ASCII values less than that of zero have been tested.

                        mov     a,original_value        ;1  Load the original character value into the accumulator.
                        clr     c                       ;1  Explicitly clear the carry bit.
                        subb    a,#030h                 ;1  Subtract the ASCII value of zero.
                        jc      invalid_char            ;2  If the carry bit is set, then the value was less than
                                                        ;   that of zero and is therefore invalid.
                        subb    a,#00ah                 ;1  Subtract one more than the ASCII value of nine to
                                                        ;   set the carry bit if it is a numeral.
                        jc      valid_char              ;2  If the carry bit is set, then the character is a
                                                        ;   numeral and is therefore valid.

;At this point, all valid numerals been tested.

                        mov     a,original_value        ;1  Load the original character value into the accumulator.
                        clr     c                       ;1  Explicitly clear the carry bit.
                        subb    a,#041h                 ;1  Subtract the ASCII value of uppercase A.
                        jc      invalid_char            ;2  If the carry bit is set, then the value was less than
                                                        ;   that of an uppercase A and is therefore invalid.

;At this point, all ASCII values less than uppercase A have been tested.

                        subb    a,#01ah                 ;1  Subtract one more than the difference between ASCII Z
                                                        ;   and ASCII A to set the carry bit if the value is a
                                                        ;   valid uppercase ASCII character.
                        jc      valid_char              ;2  The character is a valid uppercase character if the
                                                        ;   carry bit is set.

;At this point, all uppercase letters have been tested.

                        mov     a,original_value        ;1  Load the original character value into the accumulator.
                        clr     c                       ;1  Explicitly clear the carry bit.
                        subb    a,#061h                 ;1  Subtract the ASCII value of lowercase a.
                        jc      invalid_char            ;2  If the carry bit is set, then the value was less than
                                                        ;   that of a lowercase a and is therefore invalid.

;At this point, all ASCII values less than lowercase a have been tested.

                        subb    a,#01ah                 ;1  Subtract one more than the difference between ASCII z
```

```
                                                ;   and ASCII a to set the carry bit if the value is a
                                                ;   valid lowercase ASCII character.
                    jc      valid_char          ;2  The character is a valid lowercase character if the
                                                ;   carry bit is set.

;At this point, all lowercase letters have been tested, so test for the epsilon character.

                    mov     a,original_value    ;1  Load the original character value into the accumulator.
                    clr     c                   ;1  Explicitly clear the carry bit.
                    subb    a,#0e3h             ;1  Subtract the ASCII value of epsilon.
                    jnz     invalid_char        ;2  If the result is not zero, then the character is not
                                                ;   an epsilon and is therefore invalid.
valid_char:         mov     a,original_value    ;1  Since the character was determined to be valid,
                                                ;   move the original character into the accumulator.
                    ret                         ;2  Return to the calling address with the valid character
                                                ;   in the accumulator.
invalid_char:       mov     a,#0dbh             ;1  Since the character was determined to be invalid,
                                                ;   place the ASCII value for box character in the
                                                ;   accumulator.
                    ret                         ;2  Return to the calling address with a box character in
                                                ;   the accumulator.
;-------------------------------------------------------------------------------------------------------------
end
```