INTRODUCTION

Many people carry one or more magnetically encoded cards with them for accessing a range of services. Perhaps the most common example is the credit card or bank ATM card, but increasingly they are being used for access control, employee time logging, customer loyalty schemes, club membership and other applications. This application note describes data formats found on these cards, and details an algorithm to decode that data with suitable hardware.

Often a card reader will be incorporated into a point-of-sale (POS) terminal or built into a security device. In a larger system, the reader control electronics may be integrated with other devices, however it is often useful to use a microcontroller to decode the data from the card remotely and send the data via a serial link to a host processor. The PIC12C509 is a good example of a suitable microcontroller, as it requires few external components, and is small enough to be incorporated into the card reader assembly itself. This allows a self-contained card reader with serial data output to be built.

This document details the typical data found on a bank card, but there are also many custom-encoded cards in circulation for other purposes. However, even these cards usually conform to the ISO Track 2 standard which will be described later.

DATA ENCODING

Data is encoded on the magnetic stripe on a card in a similar way to original computer tape systems, mainly because at the time they were introduced, tape technology was widely available. While the details of the card data formats given here are brief, a full description of how the data is physically encoded on the magnetic stripe can be found in International Standards Organization document ISO7811/2-1985. In the US, this is also known as ANSI x 4.16-1983, and in the UK, as BS7106:Part 2:1989. Full specifications for all aspects of "identification cards", including the physical size of the card and the embossed information on the front, can be found in ISO7811 parts 1 to 6.

The magnetic stripe on bank cards and credit cards typically has three tracks of information. Track 1 is alphanumeric and usually records the card holder's name in addition to their card number. Track 2 (the center track) is numeric only and encodes the card holder's card number and expiration date. Track 3 is typically not used, or reserved for the card organization's own use, and has many different data encoding standards.

To read each track requires a magnetic reading head at the appropriate distance from the edge of the card. The position of each track is shown in Figure 1.

FIGURE 1: POSITION OF ISO TRACKS 1, 2 AND 3

This application note deals specifically with data encoded on Track 2. This data is numeric only and so is compact and easy to read, and there are many card reading modules with a single head in the Track 2 position available. In recent years, there has been a trend for organizations to read data from Track 1, thus allowing POS terminals to display the cardholder's name on the receipt.

This application note deals specifically with data encoded on Track 2. This data is numeric only and so is compact and easy to read, and there are many card reading modules with a single head in the Track 2 position available. In recent years, there has been a trend for organizations to read data from Track 1, thus allowing POS terminals to display the cardholder's name on the receipt.
Most card readers have three wires for data output, plus of course, one each for power and ground. They are typically powered from a 5V DC supply with TTL compatible output signals. Inside the reader assembly is a magnetic reader head, like a cassette tape head. A small circuit converts the analog signal from the head into clock and data signals, and a signal to indicate a card is present. For this application note, a Panasonic card reader (part no. PCR100-ND from Digi-Key) is used. Typically the signals are all active low, which means a high voltage (+5V) represents logic '0' and a low voltage (0V) represents logic '1'. Table 1 shows details of the connection to the reader module, with wire colors for the Panasonic interface cable (Digi-Key PCR101-ND).

**TABLE 1: CARD READER MODULE CONNECTIONS**

<table>
<thead>
<tr>
<th>Wire Color</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brown</td>
<td>CLD</td>
<td>Card presence indicator. When low, a card is in the reader.</td>
</tr>
<tr>
<td>Red</td>
<td>RCL</td>
<td>Clock signal. When low, the data bit on the RDT pin is valid.</td>
</tr>
<tr>
<td>Orange</td>
<td>RDT</td>
<td>Data signal. Data bits are read from the card sequentially and output on this pin. When low, the data bit is a '1' and when high, it is a '0'. The data is only valid when the RCE pin is low.</td>
</tr>
<tr>
<td>Yellow</td>
<td>+5V supply</td>
<td>Connect to power supply.</td>
</tr>
<tr>
<td>Green</td>
<td>0V</td>
<td>Connect to ground.</td>
</tr>
<tr>
<td>Blue</td>
<td>Frame Ground</td>
<td>Connect to ground if necessary.</td>
</tr>
</tbody>
</table>

For a reader with more than one read head there will be more than one clock and data line, and the software to read the card becomes more complicated.

Figure 2 shows a representation of the signals generated by the card reader as a card is passed through. First, **CLD** goes low to indicate a card is in the reader, then a series of pulses on **RCL** indicate when the data on the **RDT** pin is valid. The sequence shown is for the first character on Track 2, which is the start sentinel 1011b. This is encoded LSb first and followed by a parity bit.

Since the card is being passed through the reader by a human, the timing of the **RCL** pulse will be irregular, but the speed of the card as it passes the read head is so slow with respect to the operation of the microcontroller's sampling loop, that this is not really a problem.

**FIGURE 2: CARD READER SIGNALS**

![Card Reader Signals Diagram]
CARD DATA FORMAT

Data is encoded LSb first on the three tracks as follows:

**Track 1 - IATA**

The data standard for Track 1 was developed by the International Air Transportation Association (IATA) and contains alphanumeric information for airline ticketing or other database accesses. On a credit card, this track typically contains the cardholder's name as embossed on the front of the card. The specification allows up to 79 characters. Each character is 7 bits long, comprising a 6 bit character code and a parity bit. The data is encoded at 210 bpi (bits per inch).

<table>
<thead>
<tr>
<th>SS</th>
<th>FC</th>
<th>PAN</th>
<th>FS</th>
<th>CC</th>
<th>Name</th>
<th>FS</th>
<th>Additional Data</th>
<th>ES</th>
<th>LRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS.............................................................................................................................</td>
<td>........................................................................................................................................</td>
<td>........................................................................................................................................</td>
<td>........................................................................................................................................</td>
<td>........................................................................................................................................</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC.............................................................................................................................</td>
<td>Format Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAN.....................................................................................</td>
<td>Primary Account Number (19 digits max.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FS.............................................................................................................................</td>
<td>Field Separator</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC ........................................................................................................</td>
<td>Country Code (3 characters max.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name........................................................................................................</td>
<td>Name (26 characters max.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.............................................................................................................................</td>
<td>........................................................................................................................................</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRC .................................................................................</td>
<td>Longitudinal Redundancy Check Character</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Additional Data:
- Expiration Date (4 characters)
- Interchange Designator (1 Character)
- Service Code (2 characters)
- Custom Data

**Track 2 - ABA**

The data standard for Track 2 was developed by the American Bankers Association (ABA), and contains numeric information only. On a credit card, this track typically contains the cardholder's credit card number as embossed on the front of the card. The specification allows up to 40 digits. Each digit is 5 bits long, comprising a 4-bit BCD digit and a parity bit. The data is encoded at 75 bpi.

<table>
<thead>
<tr>
<th>SS</th>
<th>PAN</th>
<th>FS</th>
<th>Additional Data</th>
<th>ES</th>
<th>LRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS.............................................................................................................................</td>
<td>........................................................................................................................................</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PAN.....................................................................................</td>
<td>Primary Account Number (19 digits max.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FS.............................................................................................................................</td>
<td>Field Separator</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES.............................................................................................................................</td>
<td>........................................................................................................................................</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRC .................................................................................</td>
<td>Longitudinal Redundancy Check Character</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Additional Data:
- Country Code (3 characters)
- Expiration Date (4 characters)
- Interchange Designator (3 Character)
- Service Code (3 characters)
- Custom Data
Track 3 - THRIFT

The data standard for Track 3 was developed by the Thrift Industry, and contains numeric only information which may be re-recorded or updated when the card is used. There are many different uses and specifications for Track 3, so no details are shown here. The Track 3 specification allows up to 107 digits. Each digit is 5 bits long, a 4-bit BCD digit and a parity bit. The data is encoded at 210 bpi.

While the Primary Account Number (PAN) can be up to 19 digits, a MasterCard PAN is variable up to 16 digits, and VISA is 13 or 16 digits, including a modulo-10 check digit.

Each of the three specifications includes three special characters: a start sentinel, an end sentinel and an LRC (Longitudinal Redundancy Check) character. This means that the actual number of characters that can be stored is three less than the maximum specified. The sentinel codes are special character codes that are used to tell the microprocessor that it is reading the data where the start and end of the data is. Any unused space before or after the data on the card is filled with zeroes. The LRC character provides one of the error detection mechanisms described below.

ERROR DETECTION

There are two error detection methods incorporated into the data encoding standard. The first is parity checking. For alphanumeric data, there are 7 bits per character. The lower 6 bits are the character itself, and the MSb is a parity bit. Each character is encoded with odd parity, meaning that the total number of '1's in the character will be odd. Similarly for numeric data, there are 5 bits per character, 4 are the character itself, and the MSb is the parity bit. This is shown in Table 2 and Table 3. To check the parity, count the number of '1's in each character as it is read from the card. If the count is even, then there was a parity error when reading that character.

The LRC is a character code which is calculated when the card is written and checked when the card is read. It contains a bit pattern that makes the total number of '1's encoded in the corresponding bit location of all the characters (including the start and end sentinel and the LRC itself) in the data on that track even. To check the LRC, XOR all of the character codes, ignoring the parity bits, as they are read from the card, including the start and end sentinels and the LRC itself. The result (excluding the parity bit) should be zero.

The reason for having two error detection methods is to make the error detection more robust. For example, if a character is read with two bits wrong, the parity will appear to be okay, but the LRC check will fail.
## CHARACTER SET

### TABLE 2: TRACK 1 AT 7 BITS PER CHARACTER (PARITY BIT NOT SHOWN)

<table>
<thead>
<tr>
<th>D3:D0</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SPC</td>
<td>0</td>
<td></td>
<td>P</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>A</td>
<td></td>
<td>Q</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>B</td>
<td></td>
<td>R</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>C</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>0100</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
</tr>
<tr>
<td>0101</td>
<td>% (start sentinel)</td>
<td>5</td>
<td>E</td>
<td>U</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>F</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>G</td>
<td></td>
<td>W</td>
</tr>
<tr>
<td>1000</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>X</td>
</tr>
<tr>
<td>1001</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
</tr>
<tr>
<td>1010</td>
<td>J</td>
<td></td>
<td></td>
<td>Z</td>
</tr>
<tr>
<td>1011</td>
<td>K</td>
<td></td>
<td></td>
<td>[</td>
</tr>
<tr>
<td>1100</td>
<td>L</td>
<td></td>
<td></td>
<td>/</td>
</tr>
<tr>
<td>1101</td>
<td>-</td>
<td></td>
<td></td>
<td>]</td>
</tr>
<tr>
<td>1110</td>
<td>.</td>
<td></td>
<td>N</td>
<td>^(separator)</td>
</tr>
<tr>
<td>1111</td>
<td>/</td>
<td></td>
<td>? (end sentinel)</td>
<td>O</td>
</tr>
</tbody>
</table>

Characters not shown are not supported in the alphanumeric character set, although they may appear on the card in the LRC position. The three shaded characters may differ for national character sets.

### TABLE 3: TRACK 2 AND 3 AT 5 BITS PER CHARACTER (PARITY BIT SHOWN)

<table>
<thead>
<tr>
<th>P</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Data Decoding

Knowing what the card reader signals mean, and how the characters are encoded, makes it a simple matter to devise an algorithm to decode the data from a card when it is swiped through the reader. The card509.asm file for the PIC12C509 is very compact and may be easily adapted for other PICmicro® devices.

The program is designed to read data from a Track 2 magnetic card reader, because equipment for reading this track is widely available, and Track 2 has a manageably small number of bits encoded. This is important since the data is buffered before sending it out via the serial port. As each character is read, its parity bit is checked, then stored in a memory buffer. After reading the end sentinel, the LRC is read and checked, and all of the data characters in the buffer are sent out serially from an I/O pin. If there are any parity errors, an error flag is set, and if the LRC check is bad, then another error flag is set. The state of these two flags is indicated by two characters sent after the card data. On power-up and after every card read, the PICmicro device sends ‘Ready’ from the serial port.

The card data is stored in a memory buffer, which is a block of data memory not used by any variable in the program. Since each character is a 4-bit BCD digit, each byte can hold two characters, so 20 bytes are reserved, enough to hold 40 characters. In fact, space is only needed for 37 characters as there is no need to store the start and end sentinels or the LRC character.

On power-up, all of the program memory from 0x07 to 0x0F is cleared. This is not strictly necessary, but some registers are not specifically initialized on reset and may contain random data on power up.

The main loop starts by clearing the memory buffer and initializing the memory pointers and other variables. When starting, the bad_LRC flag is set and it is only cleared if a bad LRC is not found after reading all the data from the card. Also, the 4 bits that hold the LRC check in the parity register are initialized to the same bit pattern as the start sentinel. This is because the start sentinel is never stored, but must be included in the LRC calculation.

Next, the program waits to see if a card is present and loops indefinitely while the CLD signal is high. Once it is low, the program drops through to another loop to wait for the RCL line to go low. When the RCL line is low, the data line is valid, so the RDT line can be tested. Remember the RDT line is active low, so if it is high, the data on the card is a ‘0’ and the carry flag is cleared. If RDT is low, the data on the card is a ‘1’, therefore the carry flag is set and the parity bit in the parityLRC register is toggled. Toggling the parity bit is like having a one bit counter, which is all that is needed to count ‘1’s and see if there is an odd number.

A byte of memory is reserved as an input buffer, char_buf, and a single bit as a flag, found_start. Each time a bit is read from the reader, it is placed in the carry flag as described above, and the input buffer is cleared. Once it is low, the data line is valid, so the RDT flag is cleared at the beginning of the main loop, and while it is clear, the top 5 bits are checked for the start sentinel bit pattern every time a new bit is rotated in from the reader. As soon as the start sentinel is seen, the found_start flag is set.

When the carry flag is rotated into the input buffer with RRF char_buf, the LSb of char_buf rotates out into the carry. Until the start sentinel is seen, the low three bits of char_buf are continually cleared, so a zero rotates out and is of no concern. Once the start sentinel is seen, the bits that were read need to be grouped into 5-bit characters. This is done by setting bit 4 of char_buf when ready to read a new character. When 5 bits have been rotated in from the reader, the bit that was set will be rotated out into the carry flag. This bit is known as a sentinel bit (not to be confused with the start and end sentinels on the card).

The carry flag is checked to see if the sentinel bit has rolled out, and if it has, then the top 5 bits of char_buf contain a character from the card. The program checks the parity (by looking at the parity bit in the parityLRC register), then XORs the character with the LRC to update it. If the character is not the end sentinel or LRC, the parity bit is discarded and the 4-bit character is stored in the memory buffer. If it was the end sentinel, a flag (found_end) is set to show that the next character will be the LRC and that it’s possible to finish.

When the last character (the LRC) has been read, the program jumps to the dump_buffer routine, or if the buffer has been filled, sets the buf_end flag, which causes a jump there.

Characters are stored and fetched from the memory buffer by the get_put_char routine. The variable buf_ptr effectively points to a particular nibble in the PIC12C509 register banks. The read_buf flag indicates whether to store or fetch from the buffer, and the character is moved between char_buf and the buffer accordingly. The buffer locations are not in a contiguous address space and some care must be taken to deal with register banks correctly.

The dump_buffer routine loops through the memory buffer address space, takes each character (each nibble), converts it to an ASCII code and then calls the send_char routine to send the character out serially. If the PICmicro device is connected to a serial port on a PC running a terminal program, the data from the card will appear in the terminal window. When all characters have been sent, a ‘P’ is sent if there was a parity error, and an ‘L’, if there was an LRC error. If there were no errors, a period ‘.’ is sent, then the program loops back, clears the buffer, and waits for another card.

A simple serial output routine, send_char, sends the character code held in the W register serially from an output pin. It is timed to run at 1200 baud, no parity, 1
stop bit with a 4 MHz oscillator, and the PC serial port should be configured appropriately to receive it. The send_char routine could easily be replaced by a routine that displays the character on an LCD module, for example. Higher baud rates could be achieved using an external crystal, but the internal oscillator has been seen to be stable enough to run at 1200 baud with no errors.

**CIRCUIT LAYOUT**

As can be seen from Figure 3, there is very little to do other than wire up the PICmicro device directly to the reader. The 5V supply can be taken from a bench Power Supply Unit, or a 9V battery and a voltage regulator. The connections to the reader and to the PC serial port should match the I/O pin declarations in the code. For this application the PICmicro device should be programmed for internal oscillator mode, internal MCLR and watchdog disabled. The connections to the PC via the serial port should use a level shift IC, such as the MAX232A from Maxim. (This device is not shown in Figure 3.)

A simple program is included, CARDLOG.EXE, which monitors a PC serial port for data from the reader circuit shown. It records each unique card number as it is seen, and logs the date and time it was used, together with a notification when the number was seen for the first time. Once the card numbers are in a list in a PC program, they can be easily manipulated for the applications mentioned earlier in this document. However, it would be possible to extend the CARD509.ASM program to store card numbers in an external EEPROM, for example, or to verify card numbers read against those stored in an EEPROM to construct a stand-alone access control system or card data logger.

**CONCLUSION**

Although smart cards are gaining greater acceptance, magnetic cards have been around for some time, and it seems they will remain in use for a few years to come. This application note demonstrates the simplicity of reading magnetic card data using a low-cost embedded microcontroller, and interfacing to a larger, more complex system for many diverse applications.

**FIGURE 3: CIRCUIT SIMPLIFIED BLOCK DIAGRAM**
FIGURE 4: SIMPLIFIED FLOW DIAGRAM

Start

Initialize PORTS
Clear Variables

Main Loop

Wait for Card Present
Card = 0?

Clear Memory Buffer
Initialize Variables

Send "Ready" From Serial Port

Wait for Data Valid
CLK = 0?

Read Data Line

Data Bit is '1'
Set Carry and Toggle Parity Counter

Data Bit is '0'
Clear Carry

Rotate c into Character Buffer

Check for Sentinel Bit
C = 1?

Has Start Sentinel Been Seen Yet?

Y

Is SS in Character Buffer?

Y

Set found_start

Prepare For Next Character

N

N

N

N

N

Y

Update LRC

Has End Sentinel Been Seen Yet?

Y

Discard Parity Bit and Store Character?

Set found_end

Is End Sentinel in Character Buffer?

Y

Y

Is Buffer Full?

Set bad_parity Flag

N

N

N

N

N

Y

Is LRC Ok?

Y

Clear bad_LRC Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?

Y

Set bad_parity Flag

Update LRC

Is LRC Ok?

Y

Clear bad_parity Flag

Is Buffer Full?

Y

Dump Buffer

Main Loop

Is Parity Ok?
**APPENDIX A: SOURCE CODE**

MPASM 01.50 Released       CARD509.ASM  9-29-1999  14:09:04       PAGE  1

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>LINE</th>
<th>SOURCE TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>; Card Reader</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td>; Written by    A M Errington</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td>; Device        PIC 12C509 (8 pins)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0005</td>
<td>; Clock speed   1Mhz Tcy=1us</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0006</td>
<td>; Resonator     4MHz internal RC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0007</td>
<td>; Reset circuit Internal MCLR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td>; Watchdog Disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0009</td>
<td>;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>; Uses Panasonic track 2 card reader or any typical magnetic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>; card reader mechanism. Sends data out serially at 1200bd.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0012</td>
<td>;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0013</td>
<td>; This software is Copyright 1998 Andrew M Errington</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0014</td>
<td>;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0015</td>
<td>; This file is best viewed with hard tabs set to 4 character</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0016</td>
<td>; spacing.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0018</td>
<td>TITLE &quot;Card Reader&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0019</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0020</td>
<td>LIST C=120, b=4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0021</td>
<td>LIST P=12C509</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0022</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0023</td>
<td>ERRORLEVEL -305 ; Suppress &quot;Using default</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0024</td>
<td>; destination of 1 (file).&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0025</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0026</td>
<td>; *******************************************************</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0027</td>
<td>; General Equates</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0028</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0029</td>
<td>; PIC12C509 standard registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0030</td>
<td>EQU 0x00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0031</td>
<td>EQU 0x01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0032</td>
<td>EQU 0x02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0033</td>
<td>EQU 0x03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0034</td>
<td>EQU 0x04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035</td>
<td>EQU 0x05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0036</td>
<td>EQU 0x06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0037</td>
<td>; lower 5 bits only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0038</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0039</td>
<td>; I/O port bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0040</td>
<td>EQU 0x00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0041</td>
<td>EQU 0x01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0042</td>
<td>EQU 0x02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0043</td>
<td>EQU 0x03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0044</td>
<td>EQU 0x04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0045</td>
<td>EQU 0x05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0046</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0047</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0048</td>
<td>; Status register bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0049</td>
<td>EQU 0x00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0050</td>
<td>EQU 0x01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0051</td>
<td>EQU 0x02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0052</td>
<td>EQU 0x03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0053</td>
<td>EQU 0x04</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Card Reader

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>LINE SOURCE TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000005</td>
<td>00054</td>
<td>PA0 EQU 0x05 ; Program page select</td>
</tr>
<tr>
<td>00000007</td>
<td>00055</td>
<td>GPWUF EQU 0x07 ; GPIO reset bit</td>
</tr>
<tr>
<td>00000005</td>
<td>00056</td>
<td>RP0 EQU 0x05 ; Register page select</td>
</tr>
<tr>
<td>00000000</td>
<td>00057</td>
<td>; Other control bits</td>
</tr>
<tr>
<td>00000000</td>
<td>00058</td>
<td>LSB EQU 0x00</td>
</tr>
<tr>
<td>00000007</td>
<td>00059</td>
<td>MSB EQU 0x07</td>
</tr>
<tr>
<td>00000007</td>
<td>00060</td>
<td>; Other useful constants</td>
</tr>
<tr>
<td>00000000</td>
<td>00061</td>
<td>start_code EQU b’01011000’ ; Start Sentinel bit pattern, shifted up into the top 5 bits</td>
</tr>
<tr>
<td>000000F8</td>
<td>00062</td>
<td>end_code EQU b’11111000’ ; End Sentinel bit pattern, shifted up into the top 5 bits</td>
</tr>
<tr>
<td>00000058</td>
<td>00063</td>
<td>buf_ptr EQU 0x07 ; card data buffer pointer (nibbles)</td>
</tr>
<tr>
<td>00000008</td>
<td>00064</td>
<td>num_chr EQU 0x08 ; Number of characters read from card</td>
</tr>
<tr>
<td>00000009</td>
<td>00065</td>
<td>count EQU 0x09 ; General 8 bit counter</td>
</tr>
<tr>
<td>00000000A</td>
<td>00066</td>
<td>flag EQU 0x0A ; Control flags</td>
</tr>
<tr>
<td>00000000B</td>
<td>00067</td>
<td>char_buf EQU 0x0B ; Character buffer, input and serial output</td>
</tr>
<tr>
<td>00000000C</td>
<td>00068</td>
<td>parityLRC EQU 0x0C ; Parity/LRC workspace</td>
</tr>
<tr>
<td>00000000D</td>
<td>00069</td>
<td>temp EQU 0x0D ; Temporary workspace</td>
</tr>
<tr>
<td>00000010</td>
<td>00070</td>
<td>lo_mem EQU 0x10 ; Memory buffer start address:</td>
</tr>
<tr>
<td>00000010</td>
<td>00071</td>
<td>count EQU 0x09 ; Memory buffer start address:</td>
</tr>
</tbody>
</table>

#DEFINE IFSET BTFSC
#DEFINE IFCLR BTFSS
#DEFINE IFZ SKPNZ
#DEFINE IFNZ SKPZ
#DEFINE IFC SKPNC
#DEFINE IFNC SKPC
#DEFINE SKPSET BTFSS
#DEFINE SKPCLR BTFSC

---

Card Reader constants

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>LINE SOURCE TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000007</td>
<td>00096</td>
<td>buf_ptr EQU 0x07 ; card data buffer pointer (nibbles)</td>
</tr>
<tr>
<td>00000008</td>
<td>00097</td>
<td>num_chr EQU 0x08 ; Number of characters read from card</td>
</tr>
<tr>
<td>00000009</td>
<td>00098</td>
<td>count EQU 0x09 ; General 8 bit counter</td>
</tr>
<tr>
<td>00000000A</td>
<td>00099</td>
<td>flag EQU 0x0A ; Control flags</td>
</tr>
<tr>
<td>00000000B</td>
<td>00100</td>
<td>char_buf EQU 0x0B ; Character buffer, input and serial output</td>
</tr>
<tr>
<td>00000000C</td>
<td>00101</td>
<td>parityLRC EQU 0x0C ; Parity/LRC workspace</td>
</tr>
<tr>
<td>00000000D</td>
<td>00102</td>
<td>temp EQU 0x0D ; Temporary workspace</td>
</tr>
<tr>
<td>00000010</td>
<td>00103</td>
<td>; EQU 0x0E ; unused</td>
</tr>
<tr>
<td>00000010</td>
<td>00104</td>
<td>; EQU 0x0F ; unused</td>
</tr>
<tr>
<td>00000010</td>
<td>00105</td>
<td>lo_mem EQU 0x10 ; Memory buffer start address:</td>
</tr>
<tr>
<td>00000010</td>
<td>00106</td>
<td>; EQU 0x11 ; Track 2 of the magnetic card contains</td>
</tr>
</tbody>
</table>
Card Reader

LOC OBJECT CODE LINE SOURCE TEXT
VALUE

00107 ; EQU 0x12 ; at most 40 4-bit characters, including
00108 ; EQU 0x13 ; the start sentinel, end sentinel and
00109 ; EQU 0x14 ; LRC, so 20 bytes are reserved to store
00110 ; EQU 0x15 ; all of them. In fact only 37 nibbles
00111 ; EQU 0x16 ; are used as the start and end sentinels
00112 ; EQU 0x17 ; and the LRC are never stored in this
00113 ; EQU 0x18 ; application
00114 ; EQU 0x19
00115 ; EQU 0x1A
00116 ; EQU 0x1B
00117 ; EQU 0x1C
00118 ; EQU 0x1D
00119 ; EQU 0x1E
00120 ; EQU 0x1F
00121
00122 ; 0x20 to 0x2F are mapped to 0x00 to 0x1F, so the buffer
00123 ; continues from 0x30 onwards
00124
00125 ; EQU 0x30
00126 ; EQU 0x31
00127 ; EQU 0x32
00128 hi_mem EQU 0x33 ; Memory buffer end
00129 ; EQU 0x34 ; unused
00130 ; EQU 0x35 ; unused
00131 ; EQU 0x36 ; unused
00132 ; EQU 0x37 ; unused
00133 ; EQU 0x38 ; unused
00134 ; EQU 0x39 ; unused
00135 ; EQU 0x3A ; unused
00136 ; EQU 0x3B ; unused
00137 ; EQU 0x3C ; unused
00138 ; EQU 0x3D ; unused
00139 ; EQU 0x3E ; unused
00140 ; EQU 0x3F ; unused
00141
00142
00143 ; Derived constants
00144
00145 ; buf_sz is the actual number of nibbles available in the
00146 ; buffer. If the buffer continues into Bank 1 care must be
00147 ; taken to correct for the discontinuity in address space.
00148 ; Here the assembler signals an error if portions of the buffer
00149 ; are in the wrong banks.
00150
00151 if lo_mem > 0x1F
00152
00153 ERROR "Buffer start address (lo_mem) must be in Bank 0"
00154
00155 endif
00156
00157 if hi_mem > 0x1F && hi_mem < 0x30
00158
00159 ERROR "Buffer end address (hi_mem) must be in upper half of Bank 1"
Card Reader

; Flag register bit meanings
; bit   7  6  5  4  3  2  1  0 -> found start sentinel
;       |  |  |  |  |  |  +-----> found end sentinel
;       |  |  |  |  |  +--------> bad parity
;       |  |  |  |  +-----------> bad LRC
;       |  |  |  +--------------> reached end of buffer
;       |  |  |  +-----------------> R/^W flag for buffer operations
;       |  |  +--------------------> unused
;       |  +-----------------------> unused

00000000          00185 found_start EQU 0
00000001          00186 found_end   EQU 1
00000002          00187 bad_parity  EQU 2
00000003          00188 bad_LRC     EQU 3
00000004          00189 buf_end     EQU 4
00000005          00190 read_buf    EQU 5

00191
00192
00193 ; ParityLRC register bit meanings
00194
00195 ; bit   7  6  5  4  3  2  1  0 -> unused
00196 ;       |  |  |  |  |  |  +-----> unused
00197 ;       |  |  |  |  |  +--------> unused
00198 ;       |  |  |  |  +-----------> LRC bit 0
00199 ;       |  |  |  +-----------------> LRC bit 1
00200 ;       |  |  |  +-----------------> LRC bit 2
00201 ;       |  |  |  +-----------------> LRC bit 3
00202 ;       |  +-----------------------> parity bit
00203
00204 ; Note: Later code relies on these bits remaining in this position
00205
00206
00207 ; I/O pin declarations
00208
00209 ; The card reader connects to +5V and 0V, and has three signal
00210 ; lines connected to the following I/O pins:
00211
00000000          00212 ser_out     EQU GP0     ; serial TxD pin to host
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>LINE</th>
<th>SOURCE TEXT</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>00213</td>
<td>card</td>
<td>EQU GP1</td>
<td>; ^CLD signal (low when card present)</td>
</tr>
<tr>
<td>00000002</td>
<td>00214</td>
<td>clock</td>
<td>EQU GP2</td>
<td>; ^RCL signal (low when data valid)</td>
</tr>
<tr>
<td>00000003</td>
<td>00215</td>
<td>signal</td>
<td>EQU GP3</td>
<td>; ^RDT signal from magstripe</td>
</tr>
<tr>
<td></td>
<td>00216</td>
<td></td>
<td></td>
<td>GP4 and GP5 are still available for I/O or for a crystal if</td>
</tr>
<tr>
<td></td>
<td>00217</td>
<td></td>
<td></td>
<td>required.</td>
</tr>
<tr>
<td></td>
<td>00218</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00219</td>
<td></td>
<td></td>
<td>; Compilation options</td>
</tr>
<tr>
<td></td>
<td>00220</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>00221</td>
<td></td>
<td></td>
<td>; The invert_tx option changes the sense of the ser_out line</td>
</tr>
<tr>
<td></td>
<td>00222</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>00225</td>
<td>invert_tx</td>
<td>EQU 0</td>
<td>; 0 = Idle (logical ’1’) is 0V</td>
</tr>
<tr>
<td>00226</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 00227    |            |           |             | ; ************************************************************
| 00228    |            |           |             | ; Program code starts here |
| 00229    |            |           |             | ; ************************************************************
| 00230    |            |           |             | |
| 00231    |            |           |             | ; The PIC12C509 reset vector jumps to top of memory, then the |
| 00232    |            |           |             | ; program counter rolls to 0x00 after loading the RC osc. |
| 00233    |            |           |             | ; calibration value into W |
| 00234    |            |           |             | |
| 00235    |            |           |             | |
| 0000    | 00236      | ORG      | 0xD0         | |
| 0000 0025 | 00237     | MOVWF   | OSCCAL       | |
| 00238    |            |           |             | |
| 0001 0A31 | 00239     | GOTO    | start        | |
| 00240    |            |           |             | |
| 00241    |            |           |             | ; ************************************************************
| 00242    |            |           |             | ; Subroutines |
| 00243    |            |           |             | ; |
| 00244    |            |           |             | ; |
| 00245    |            |           |             | ; |
| 00246    |            |           |             | ; |
| 00247    |            |           |             | ; ************************************************************
| 0002    | 00248      | send_char |             | |
| 00249    |            |           |             | |
| 00250    |            |           |             | ; Call send_char with an ASCII character code in W. This is a |
| 00251    |            |           |             | ; simple serial output routine which sends the character out |
| 00252    |            |           |             | ; serially on an output pin at 1200 baud, 8 data bits, no parity, |
| 00253    |            |           |             | ; 1 stop bit. Assume the PIC oscillator is running at 4MHz. |
| 00254    |            |           |             | ; |
| 00255    |            |           |             | ; The baud rate of 1200 baud was chosen as it will work with the |
| 00256    |            |           |             | ; 12C509 internal RC oscillator generating the timing. Higher |
| 00257    |            |           |             | ; baud rates require tighter timing tolerance, and will therefore |
| 00258    |            |           |             | ; require a crystal. |
| 00259    |            |           |             | ; |
| 00260    |            |           |             | ; Normally serial communication (RS232) requires a negative |
| 00261    |            |           |             | ; voltage between -5V to -15V to represent a ’1’, and a positive |
| 00262    |            |           |             | ; voltage between +5V and +15V to represent a ’0’. Most PC |
| 00263    |            |           |             | ; serial ports will switch at +/-3V, and in fact will often work |
| 00264    |            |           |             | ; with 0V and 5V, so it is possible to use a PIC I/O pin, set |
| 00265    |            |           |             | ; high for a logic ’0’ and low for a logic ’1’. A 1k resistor |
LOC | OBJECT CODE | LINE | SOURCE TEXT
--- | ----------- | --- | ---
0026 | 02B | MOVWF char_buf | ; Store the character code (in W) to character buffer
0026 | 0287 | | ; placed in series with the Tx line will limit the current, and
0026 | 0266 | ; This is probably acceptable for experimental purposes. For
0026 | 0267 | ; robustness, however, it may be desirable to include level shift
0026 | 0268 | IC, such as the MAX232A from Maxim. The invert_tx compilation
0026 | 0269 | option can be used to alter the sense of transmitted bits if
0026 | 0270 | necessary.
0026 | 0271 | ;
0026 | 0272 | ; At 1200 baud each bit cell is just over 833us in length. At
0026 | 0273 | power up the serial output line is set to its idle state (logic
0026 | 0274 | '1'). At the start of transmission it is taken to logic '0'
0026 | 0275 | for one bit time to generate the start bit. Next, the 8 bits
0026 | 0276 | of character data are shifted out, LSB first, by rolling them
0026 | 0277 | down into the carry. The program sets or clears the serial
0026 | 0278 | line pin according to whether the carry represents a logic '0'
0026 | 0279 | or '1'. Finally the line is held at logic '1' for at least one
0026 | 0280 | bit time for the stop bit. The line then rests at this state
0026 | 0281 | (idle) until it is time to send the next byte
0026 | 0282 | ;
0026 | 0283 | ; Bit cell timing is done by counting clock cycles: 1 instruction
0026 | 0284 | is 1us, jumps and skips are 2us.
0026 | 0285 | ;
0026 | 0286 | 0002 002B | 00287 | MOVWF char_buf | ; Store the character code (in W)
0026 | 00288 | | ; to character buffer
0026 | 00289 | 0003 0C0A | 00290 | MOVLW .10 | ; Set the number of bits (including
0026 | 00291 | | ; start and stop bits) in count
0026 | 00292 | 0004 0029 | 00293 | MOVWF count | ;
0026 | 00294 | | ; Clear carry because the start bit
0026 | 00295 | 0005 0403 | 00296 | CLRC | ; is a '0'
0026 | 00297 | 0006 006 | 00298 | bit_loop | ;
0026 | 00299 | 0006 0703 | 00299 | IFNC | ; serial pin logic '0'
0026 | 00300 | | ;
0026 | 00301 | 0007 0506 | 00302 | endif | ;
0026 | 00303 | 0008 0603 | 00304 | IFC | ; serial pin logic '1'
0026 | 00305 | | ;
0026 | 00306 | 0009 0406 | 00307 | else | ;
0026 | 00308 | | ;
0026 | 00309 | 00A0 090F | 00310 | CALL bit_delay | ; Make up the bit time to 833us
0026 | 00311 | | ;
0026 | 00312 | 00B0 032B | 00313 | RRF char_buf | ; Roll LSB of char_buf into carry,
0026 | 00314 | | ; and the '1' from the bit_delay
0026 | 00315 | | ; routine into the MSB
0026 | 00316 | 00C0 02E9 | 00317 | DECFSZ count | ; Loop until all bits have been
0026 | 00318 | | ; shifted out.
LOC OBJECT CODE   LINE SOURCE TEXT
  VALUE

00319
000E 0800 00320 RETLW 0
00321 00322
000F 00323 bit_delay
00324
00325 ; The bit length should be 833us in total (=833 Tcy). This
00326 ; routine and the bit loop routine take 21 Tcy, leaving 812 Tcy
00327 ; to waste. The delay loop is 4 cycles long, so loop for 203
00328 ; times. This is done by loading a counter with 255 - 203 = 52
00329 ; and incrementing it every time around the loop. When the
00330 ; counter reaches 255 it overflows and sets the carry flag. As
00331 ; a side effect this routine returns to the bit loop just before
00332 ; the RRF instruction with carry set, which will roll a '1' into
00333 ; char_buf, which is then used as the stop bit when it rolls out
00334 ; again after being shifted 8 times.
00335
000F 0C34 00336 MOVWL .52 ; Initialise temp
0100 002D 00337 MOVWF temp
00338
0011 0C01 00339 MOVWL .1 ; Put 1 in W for incrementing temp
00340
0012 0A13 00341 GOTO $+1 ; Waste 2 cycles
0013 0000 00342 NOP ; Waste 1 cycle
00343
0014 00344
0014 01ED 00345 ADDWF temp ; Increment temp 1
0015 0703 00346 IFNC ; Did it overflow? 1
0016 0A14 00347 GOTO delay_loop ; No: go round again 2
00348
00349 ; ---
00350 ; time = 4 Tcy
00351
0017 0800 00352 RETLW 0 ; Yes: return
00353 00354
00355 ; *******************************************************
0018
0018 00356 get_put_char
00357
00358 ; This subroutine deals with buffer operations, either storing a
00359 ; character from char_buf to the buffer or fetching it from the
00360 ; buffer. The routine uses buf_ptr (the logical buffer address)
00361 ; to calculate the physical address for the character.
00362 ;
00363 ; The 4 bit character will be stored at the current "memory
00364 ; location" in the buffer. The buffer is a large chunk of RAM
00365 ; from 0x10 to 0x1F and 0x30 to 0x33. Two "memory locations"
00366 ; are contained in each byte, one in each nibble.
00367 ;
00368 ; The variable buf_ptr points to the next free logical memory
00369 ; location, and the constants lo_mem and hi_mem record the
00370 ; physical start and end locations of the RAM block. The
00371 ; constant buf_sz holds the number of nibbles (or "memory
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>LINE</th>
<th>SOURCE TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td>00372</td>
<td>;</td>
<td>locations*) that can be filled.</td>
</tr>
<tr>
<td></td>
<td>00373</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00374</td>
<td>;</td>
<td>Note there is a discontinuity in the address space, which must</td>
</tr>
<tr>
<td></td>
<td>00375</td>
<td>;</td>
<td>be dealt with when mapping the logical memory location to the</td>
</tr>
<tr>
<td></td>
<td>00376</td>
<td>;</td>
<td>physical memory address</td>
</tr>
<tr>
<td></td>
<td>00377</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00378</td>
<td>;</td>
<td>To calculate the position in memory to store the current</td>
</tr>
<tr>
<td></td>
<td>00379</td>
<td>;</td>
<td>character, divide buf_ptr by 2 and add lo_mem to give the</td>
</tr>
<tr>
<td></td>
<td>00380</td>
<td>;</td>
<td>address in RAM. Next, check if this exceeds the 0x1F address</td>
</tr>
<tr>
<td></td>
<td>00381</td>
<td>;</td>
<td>range by checking bit 5 of the resultant address, and if</td>
</tr>
<tr>
<td></td>
<td>00382</td>
<td>;</td>
<td>necessary force the address into the 0x3n address space by</td>
</tr>
<tr>
<td></td>
<td>00383</td>
<td>;</td>
<td>setting bit 4. Check whether buf_ptr is odd or even by</td>
</tr>
<tr>
<td></td>
<td>00384</td>
<td>;</td>
<td>examining its LSB to see whether to store the character in the</td>
</tr>
<tr>
<td></td>
<td>00385</td>
<td>;</td>
<td>upper or lower nibble.</td>
</tr>
<tr>
<td></td>
<td>00386</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00387</td>
<td>;</td>
<td>When buf_ptr is zero it is pointing at the first &quot;memory location&quot;, which</td>
</tr>
<tr>
<td></td>
<td>00388</td>
<td>;</td>
<td>is the low nibble of the first byte.</td>
</tr>
<tr>
<td></td>
<td>00389</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00390</td>
<td>;</td>
<td>All RAM in the memory buffer is cleared at the beginning of</td>
</tr>
<tr>
<td></td>
<td>00391</td>
<td>;</td>
<td>the main loop, so it is not necessary to clear each &quot;memory location&quot; before</td>
</tr>
<tr>
<td></td>
<td>00392</td>
<td>;</td>
<td>storing anything there.</td>
</tr>
<tr>
<td></td>
<td>00393</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>00394</td>
<td>;</td>
<td>Note that for a 'put' operation the character arrives here in</td>
</tr>
<tr>
<td></td>
<td>00395</td>
<td>;</td>
<td>the upper nibble of char_buf, and for a 'get' operation the</td>
</tr>
<tr>
<td></td>
<td>00396</td>
<td>;</td>
<td>character is returned in the lower nibble.</td>
</tr>
<tr>
<td></td>
<td>00397</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>0018</td>
<td>0307</td>
<td>RRF</td>
<td>buf_ptr,W ; load W with buf_ptr/2. Carry flag is rolled in.</td>
</tr>
<tr>
<td>00398</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00399</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0019</td>
<td>0024</td>
<td>MOVWF</td>
<td>FSR ; and use the FSR to point to it</td>
</tr>
<tr>
<td>00401</td>
<td></td>
<td></td>
<td>Upper bits of FSR are forced to 1 so junk in Carry flag doesn't matter.</td>
</tr>
<tr>
<td>00402</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00403</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00404</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001A</td>
<td>OC10</td>
<td>MOVLW</td>
<td>lo_mem ; add the buffer start address</td>
</tr>
<tr>
<td>00405</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001B</td>
<td>01E4</td>
<td>ADDWF</td>
<td>FSR ; to get the physical address to</td>
</tr>
<tr>
<td>00406</td>
<td></td>
<td></td>
<td>store the character</td>
</tr>
<tr>
<td>00407</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00408</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00409</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001C</td>
<td>06A4</td>
<td>IFSET</td>
<td>FSR,RP0 ; Check for overflow into the</td>
</tr>
<tr>
<td>00410</td>
<td></td>
<td></td>
<td>second register page and set bit 4 to move into 0x3n address space</td>
</tr>
<tr>
<td>00411</td>
<td></td>
<td></td>
<td>space if necessary</td>
</tr>
<tr>
<td>00412</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00413</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00414</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001E</td>
<td>06AA</td>
<td>IFSET</td>
<td>flag,read_buf ; check whether this is a read</td>
</tr>
<tr>
<td>00415</td>
<td></td>
<td></td>
<td>or write operation</td>
</tr>
<tr>
<td>001F</td>
<td>0A26</td>
<td>GOTO</td>
<td>get_char ; check whether this is a read</td>
</tr>
<tr>
<td>00416</td>
<td></td>
<td></td>
<td>or write operation</td>
</tr>
<tr>
<td>00417</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0020</td>
<td></td>
<td>put_char</td>
<td></td>
</tr>
<tr>
<td>00418</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0020</td>
<td>020B</td>
<td>MOVF</td>
<td>char_buf,W ; Move the character (in high nibble) to W</td>
</tr>
<tr>
<td>00419</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00420</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00421</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0021</td>
<td>0707</td>
<td>ICLR</td>
<td>buf_ptr,LSB ; except if LSB of buf_ptr is '0'</td>
</tr>
<tr>
<td>00422</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0022</td>
<td>038B</td>
<td>SWAPF</td>
<td>char_buf,W ; then the destination is an even nibble, so swap the character</td>
</tr>
<tr>
<td>00423</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00424</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MPASM 01.50 Released         CARD509.ASM   9-29-1999  14:09:04         PAGE 9
Card Reader

LOC  OBJECT CODE     LINE SOURCE TEXT
VALUE

00425                                 ; to the low nibble
00426
0023 0120 00427                  IORWF INDF            ; since the buffer was cleared
00428                                 ; OR the character into place
00429
0024 0C29 00430                  MOVLW buf_sz + 1      ; set limit for 'put' operation
00431                                 ; to size of buffer
00432
0025 0A2C 00433                  GOTO get_put_done
00434
0026 00435                  get_char
0026 0200 00436                  MOVF INDF,W          ; Fetch data from buffer to W
00437
0027 0607 00438                  IFSET buf_ptr,LSB     ; if LSB of buf_ptr is set the
0028 0380 00439                  SWAPF INDF,W          ; desired character is an odd
00440                                 ; nibble, so swap the nibbles
00441
0029 0E0F 00442                  ANDLW 0x0F            ; mask off upper nibble
00443
002A 002B 00444                  MOVLW char_buf        ; move it to the character buffer
00445
002B 0208 00446                  MOVF num_chr,W       ; set limit for 'get' operation
00447                                 ; to number of characters read
00448                                 ; from card
00449
002C 00450                  get_put_done
002C 02A7 00451                  INCF buf_ptr         ; increment memory pointer.
00452
002D 0187 00453                  XORWF buf_ptr,W       ; check if this was the last
00454                                 ; nibble in the buffer by
00455                                 ; comparing against W (either
00456                                 ; buf_sz or num_chr)
00457
002E 0643 00458                  IFZ                            ; if it was,
002F 058A 00459                  BSF flag,buf_end    ; then set a flag
00460
0030 0800 00461                  RETLW 0
00462
00463                                 ; End of subroutines
00464
00465
00466                                ; ***************************************************************
00467                                ; Main Program starts here                                   
00468                                ; ***************************************************************
00469
0031 00470                  start
0031 0063 00471                  CLRF STATUS
00472
0032 0CC0 00473                  MOVLW B’11000000’    ; Disable GPIO pull-ups and wake
00474                                 ; up feature
00475
0033 0002 00476                  OPTION
00477

© 1999 Microchip Technology Inc.
LOC  OBJECT CODE    LINE SOURCE TEXT
     VALUE

0034 0C0E           00478         MOVLW   B'00001110'     ; Set GPIO <1:3> as inputs...
0035 0006           00479         TRIS    GPIO            ; Note: GP3 is always input
0036 0066           00480         CLRF    GPIO            ; GPIO outputs all 0
00481                                 ; except for invert_tx condition
00482     if invert_tx
00483          BSF    GPIO,ser_out    ; except for invert_tx condition
00484     endif
00485     endif
00486     endif
00487                                 ; Clear RAM from 0x07 to 0x0F
00488                                 ; for neatness
00489                                 ;
0037 0C07           00490         MOVLW   0x07            ; Load start address (0x07) into
0038 0024           00491         MOVWF   FSR             ; the FSR
00492                                 ;
003A 02A4           00493 clrloop
003A 0C10           00494         MOVLW   lo_mem          ; Fetch buffer start address
0040 0060           00495         MOVWF   FSR             ;
0040 02A4           00496 clr_buf_loop
003F 0C10           00497 clr_buf_loop
0041 0060           00498 clr_buf_loop
0042 02A4           00499 clr_buf_loop
0043 06A4           00500     IFSET   FSR,RP0          ; If FSR points to register page 1
0044 0584           00501     BSF    FSR,4           ; set bit 4 to move into 0x3n
0044 0523           00502     ; address space
0045 0204           00503     MOVF   FSR,W          ; Check for buffer end address.
0046 0FF4           00504     XORLW   (hi_mem + 1) | 0xC0
0047 0743           00505     IFNZ                         ; If not end then loop around
0048 0A41           00506     GOTO   clr_buf_loop
Card Reader

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>LINE</th>
<th>SOURCE</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0049 0067</td>
<td>00531</td>
<td>CLRF</td>
<td>buf_ptr</td>
<td>; Initialise buffer pointer to 0</td>
</tr>
<tr>
<td>00532</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>004A 0C28</td>
<td>00533</td>
<td>MOVLW</td>
<td>buf_sz</td>
<td>; Initialise the number of</td>
</tr>
<tr>
<td>00534</td>
<td>0028</td>
<td>MOVWF</td>
<td>num_chr</td>
<td>; characters read to the maximum</td>
</tr>
<tr>
<td>00535</td>
<td></td>
<td></td>
<td>; in case of overflow later</td>
<td></td>
</tr>
<tr>
<td>00536</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>004C 0C58</td>
<td>00537</td>
<td>MOVLW</td>
<td>start_code</td>
<td>; Initialise the LRC to the start</td>
</tr>
<tr>
<td>00538</td>
<td></td>
<td>MOVWF</td>
<td>parityLRC</td>
<td>; sentinel code.</td>
</tr>
<tr>
<td>00539</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>004E 006A</td>
<td>00540</td>
<td>CLRF</td>
<td>flag</td>
<td>; Initialise control flags to</td>
</tr>
<tr>
<td>00541</td>
<td>BSF</td>
<td>flag,bad_LRC</td>
<td>; zero then set the bad_LRC</td>
<td></td>
</tr>
<tr>
<td>00542</td>
<td></td>
<td></td>
<td>; flag. Assume the LRC is bad</td>
<td></td>
</tr>
<tr>
<td>00543</td>
<td></td>
<td></td>
<td>; until the check at the end.</td>
<td></td>
</tr>
<tr>
<td>00544</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0050 0C52</td>
<td>00545</td>
<td>MOVLW</td>
<td>'R'</td>
<td>; Send &quot;Ready&quot; from serial port</td>
</tr>
<tr>
<td>00546</td>
<td>CALL</td>
<td>send_char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00547</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0052 0C65</td>
<td>00548</td>
<td>MOVLW</td>
<td>'e'</td>
<td></td>
</tr>
<tr>
<td>00549</td>
<td>CALL</td>
<td>send_char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00550</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0054 0C61</td>
<td>00551</td>
<td>MOVLW</td>
<td>'a'</td>
<td></td>
</tr>
<tr>
<td>00552</td>
<td>CALL</td>
<td>send_char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00553</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0056 0C64</td>
<td>00554</td>
<td>MOVLW</td>
<td>'d'</td>
<td></td>
</tr>
<tr>
<td>00555</td>
<td>CALL</td>
<td>send_char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00556</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0058 0C79</td>
<td>00557</td>
<td>MOVLW</td>
<td>'y'</td>
<td></td>
</tr>
<tr>
<td>00558</td>
<td>CALL</td>
<td>send_char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00559</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>005A 0C0D</td>
<td>00560</td>
<td>MOVLW</td>
<td>.13</td>
<td>; Send CR LF from serial port</td>
</tr>
<tr>
<td>00561</td>
<td>CALL</td>
<td>send_char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00562</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>005B 0902</td>
<td>00563</td>
<td>IFSET</td>
<td>GPIO,card</td>
<td>; Check ^CARD line</td>
</tr>
<tr>
<td>005C 0C0A</td>
<td>00564</td>
<td>MOVVLW</td>
<td>.10</td>
<td></td>
</tr>
<tr>
<td>005D 0902</td>
<td>00565</td>
<td>CALL</td>
<td>send_char</td>
<td></td>
</tr>
<tr>
<td>00566</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>005E 006B</td>
<td>00567</td>
<td>CLRF</td>
<td>char_buf</td>
<td>; Clear character input buffer</td>
</tr>
<tr>
<td>00568</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>005F</td>
<td>00569</td>
<td>wait_card</td>
<td></td>
<td></td>
</tr>
<tr>
<td>005F 0626</td>
<td>00570</td>
<td>IFSET</td>
<td>GPIO,clock</td>
<td>; Check ^CLK line</td>
</tr>
<tr>
<td>0060 0A5F</td>
<td>00571</td>
<td>GOTO</td>
<td>wait_card</td>
<td>; if it's high then keep waiting</td>
</tr>
<tr>
<td>00572</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00573</td>
<td>: ^CARD is low, so a card has started passing through the reader</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00574</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0061</td>
<td>00575</td>
<td>wt_clk_lo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0061 0646</td>
<td>00576</td>
<td>IFSET</td>
<td>GPIO,clock</td>
<td>; Check ^CLK line</td>
</tr>
<tr>
<td>0062 0A61</td>
<td>00578</td>
<td>GOTO</td>
<td>wt_clk_lo</td>
<td>; If it's high then keep waiting</td>
</tr>
<tr>
<td>00579</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00580</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00581</td>
<td>: ^CLK is low, so valid data is on the ^DATA pin. If ^DATA is</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00582</td>
<td>low the data bit on the card is a '1', so set carry and toggle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00583</td>
<td>; the parity bit counter. If ^DATA is high the data bit on the</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LOC  OBJECT CODE     LINE SOURCE TEXT
VALUE

00584  ; card is a ‘0’, so clear the carry. Roll the carry flag into
00585  ; the character buffer.
00586  ;
00587  ; All this data processing must be done as quickly as possible,
00588  ; but fortunately the card is being swiped by a human, so from
00589  ; the micro’s point of view it is all happening very slowly.
00590
00591  0063    chk_data
0066 0666          00592  IFSET   GPIO,signal     ; Check ‘DATA
0064 0A69          00593  GOTO   data_0          ; If it’s high, data bit is ’0’
00594
0065 065      data_1
0065 0503          00595  BSF     STATUS,C        ; Otherwise it’s low so data bit
00596          00597  ; is ’1’, so set carry flag
00598
0066 0C80          00599  MOVWL   0x80            ; and toggle parity bit in
0067 01AC          00600  XORWF   parityLRC       ; parityLRC register
00601
0068 0703          00602  BTFSS   STATUS,C        ; Use that fact that carry is
00603          00604  ; set to skip the next line.
00605
0069 0605          00606  BCF     STATUS,C        ; bit is ’0’, so clear carry
00607
006A 0608    store_bit
006A 032B          00609  RRF     char_buf        ; shift data bit in carry flag
00610          00611  ; into the input buffer, and
00612          00613  ; shift LSB out into carry flag.
00614
0061           00615  ; If the start sentinel code has not yet been seen the LSB will
00616  ; have been ’0’, so carry will be ’0’. If the start sentinel
00617  ; code has been seen then there will have been a sentinel bit
00618  ; set in char_buf which falls out after shifting 5 bits (one
00619  ; character) in.
00620
006B 0603          00619  IFC                     ; So, check the carry flag
00621 006C 0A7C      00620  GOTO   got_char
00622
00623  ; Otherwise, here a bit has just been read. Check if the start
00624  ; sentinel code has ever been seen. If it has then a sentinel
00625  ; bit will drop out after each character, which is dealt with by
00626  ; the code above. If not then it is necessary to check for the
00627  ; start sentinel code after reading each bit.
00628
006D 060A          00628  IFSET   flag,found_start; Has the start code been seen?
006E 0A79          00629  GOTO   wt_clk_hi        ; Yes, so wait for ^CLK to go
00631          00632  ; high again and get next bit.
00633          00634  ; No, so check for the start
00635 0CF8          00635  B’11111000’        ; The start code is five bits
00636
MPASM 01.50 Released  CARD509.ASM  9-29-1999  14:09:04  PAGE 13
Card Reader

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>LINE</th>
<th>SOURCE TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VALUE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00637</td>
<td></td>
<td></td>
<td>; long, so mask off the low 3</td>
</tr>
<tr>
<td>00638</td>
<td>016B</td>
<td>00639</td>
<td>ANDWF char_buf ; bits in the buffer (which are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>; probably ’0’ anyway)</td>
</tr>
<tr>
<td>00640</td>
<td>0C58</td>
<td>00641</td>
<td>MOVWF start_code ; and compare start_code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00642</td>
<td>XORWF char_buf,W ; to the buffer</td>
</tr>
<tr>
<td>00643</td>
<td>0743</td>
<td>00644</td>
<td>IFNZ ; Is it the start code?</td>
</tr>
<tr>
<td>00645</td>
<td>0A79</td>
<td>00646</td>
<td>GOTO wt_clk_hi ; No, so wait for ^CLK to go high</td>
</tr>
<tr>
<td>00647</td>
<td>050A</td>
<td>00648</td>
<td>BSF flag,found_start; Yes, so set a flag</td>
</tr>
<tr>
<td>00649</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00650</td>
<td>next_char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00651</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00652</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00653</td>
<td>04EC</td>
<td>00654</td>
<td>BCF parityLRC,MSB ; clear the parity flag,</td>
</tr>
<tr>
<td>00655</td>
<td>006B</td>
<td>00656</td>
<td>CLRF char_buf ; clear the input buffer,</td>
</tr>
<tr>
<td>00657</td>
<td>058B</td>
<td>00658</td>
<td>BSF char_buf,4 ; and set a sentinel bit</td>
</tr>
<tr>
<td>00659</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00660</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00661</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00662</td>
<td>wt_clk_hi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00663</td>
<td>0746</td>
<td>00664</td>
<td>IFCLR GPIO,clock ; Check ^CLK line</td>
</tr>
<tr>
<td>00665</td>
<td>0A79</td>
<td>00666</td>
<td>GOTO wt_clk_hi ; Keep waiting whilst it’s low</td>
</tr>
<tr>
<td>00667</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00668</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00669</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00670</td>
<td>got_char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00671</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00672</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00673</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00674</td>
<td>char_buf 7 6 5 4 3 2 1 0</td>
<td>; First, check the parity of the character just read. The</td>
<td></td>
</tr>
<tr>
<td>00675</td>
<td>P D3 D2 D1 D0 0 0 0</td>
<td>; characters on the card are encoded with odd parity, and before</td>
<td></td>
</tr>
<tr>
<td>00676</td>
<td></td>
<td></td>
<td>; each character is read the parity bit in parityLRC is cleared.</td>
</tr>
<tr>
<td>00677</td>
<td></td>
<td></td>
<td>; This bit is toggled every time a ’1’ is read for the current</td>
</tr>
<tr>
<td>00678</td>
<td></td>
<td></td>
<td>; character, which means that if the character was read correctly</td>
</tr>
<tr>
<td>00679</td>
<td></td>
<td></td>
<td>; this bit will be ’1’.</td>
</tr>
<tr>
<td>00680</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00681</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00682</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00683</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00684</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00685</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00686</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00687</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00688</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00689</td>
<td>07EC</td>
<td>00690</td>
<td>IFCLR parityLRC,MSB ; If parity bit is ’0’</td>
</tr>
</tbody>
</table>
Card Reader

LOC    OBJECT CODE     LINE SOURCE TEXT
VALUE

007D  054A           00690       BSF    flag,bad_parity ; set the parity error flag
            00691
            00692 ; Now XOR char_buf with the parityLRC register to update the LRC
            00693 ; state. The LRC portion of the parityLRC register was
            00694 ; initialised with the code for the start sentinel. Every time
            00695 ; a character is read it is XORed with the parityLRC register.
            00696 ; Any bits in the char_buf register which are set will toggle
            00697 ; bits in the LRC portion of parityLRC. If there have been no
            00698 ; bit errors after reading all the characters and the LRC from
            00700 ; the card, then the LRC bits in the parityLRC register will all
            00701 ; be zero.
            00702
007E  020B           00703       MOVF    char_buf,W      ; Copy char_buf to W
            00704       XORWF   parityLRC       ; XOR with the parityLRC register
            00705                                 ; to update the LRC calculation
            00706
            00707                                 ; If the end sentinel has not
0080  072A           00708       IFCLR   flag,found_end  ; yet been seen then this is
            00709       GOTO    not_LRC         ; not the LRC, so store it
            00710
0082  020C           00711       MOVF    parityLRC,W     ; Otherwise it was the LRC, so
3  0E78           00712       ANDLW   b'01111000'     ; get the LRC check from the
            00713                                 ; parityLRC register, and mask
            00714                                 ; off the parity flag
            00715
0084  0643           00716       IFZ                     ; If it is zero then the LRC was
            00717       BCF    flag,bad_LRC ; okay so clear the bad_LRC flag
            00718
0086  0207           00719       MOVF    buf_ptr,W       ; Copy the value of buffer pointer
            00720       MOVWF   num_chr         ; to num_chr
3  0028           00721
            00722       GOTO    dump_buffer     ; and dump it out
            00723
            00724
0089  0A95           00725       GOTO    dump_buffer     ; and dump it out
3  0A95           00726
3  0CF8           00727       MOVLW   end_code        ; Is this the end sentinel?
            00728
008A  018B           00729       MOVF    char_buf,W      ; If so, the next character is
            00730       BSF    flag,found_end  ; the LRC, so set a flag
            00731
008B  0643           00732       IFZ                     ; and don’t bother storing it
3  052A           00733       GOTO    next_char
            00734
008D  0643           00735       RLF     char_buf        ; discard parity by shifting it
            00736                                 ; out, leaving the 4 bit
            00737                                 ; character in the upper nibble
            00738
008F  036B           00739       MOVLW   0xF0            ; mask off the lower nibble
            00740       ANDWF   char_buf
            00741
0090  0CF0           00742       CALL    get_put_char    ; and store the character

LOC  OBJECT  CODE   LINE  SOURCE   TEXT
       VALUE

00743

0093  078A  00744  IFCLR   flag,buf_end  ; Is the buffer full?
00745  GOTO   next_char       ; no, so get the next character
00746
00747                                    ; Otherwise, fall through...
00748
00749 ; Jump to dump_buffer when the buffer is full. This routine
00750 ; loops through each "location" in the memory buffer, and sends
00751 ; the character at that location out serially on an output pin.
00752
0095  0067  00753  dump_buffer
0095                00754  CLRF    buf_ptr         ; Load buffer pointer with 0
00755
0096  05AA  00756  BSF     flag,read_buf   ; Set the flag to read mode
0097  048A  00757  BCF     flag,buf_end    ; Clear the buf_end flag
00758
0098  0918  00759  loop_buffer
0098                00760  CALL    get_put_char    ; Get character from buffer
00761
0099  0C30  00762  MOVLW   .48             ; convert to ASCII by adding 48
009A  01CB  00763  ADDWF   char_buf, W     ; and put the result in W
00764
009B  0902  00765  CALL    send_char       ; and send the character
00766
009C  078A  00767  IFCLR   flag,buf_end    ; have we emptied the buffer?
009D  0A98  00768  GOTO   loop_buffer ; No, so loop around
00769
009E  0C2E  00770  ; After sending the contents of the buffer, two more characters
009F  064A  00771  ; are sent to indicate any errors. If there was bad parity on
00A0  0C50  00772  ; any character a "P" is sent, and if there was a bad LRC an "L"
00A1  0902  00773  ; is sent. If either condition was okay we send a period "."
00A2  0A3F  00774
009E  0C2E  00775  MOVLW   '.'             ; Load ASCII "." into W
00776
009F  064A  00777  IFSET   flag,bad_parity ; If parity was ever bad
00A0  0C50  00778  MOVLW   'P'             ; load ASCII "P" into W instead
00779
00A1  0902  00780  CALL    send_char       ; then send the character
00781
00A2  0C2E  00782  MOVLW   '.'             ; Load ASCII "." into W again
00783
00A3  066A  00784  IFSET   flag,bad_LRC  ; If LRC was bad
00A4  0C4C  00785  MOVLW   'L'             ; load ASCII 'L' into W instead
00786
00A5  0902  00787  CALL    send_char       ; and send the character
00788
00A6  0C0D  00789  MOVLW   .13             ; Send CR LF from serial port
00A7  0902  00790  CALL    send_char
00791
00A8  0C0A  00792  MOVLW   .10             ;
00A9  0902  00793  CALL    send_char
00794
00AA  0A3F  00795  GOTO   main_loop       ; Back to the beginning and wait
Card Reader

LOC OBJECT CODE     LINE SOURCE TEXT
VALUE

00796 ; for another card.
00797
00798 ; The end is nigh...
00799
00800 END

SYMBOL TABLE
LABEL                             VALUE
C                                 00000000
DC                                00000001
FSR                               00000004
GP0                                00000000
GP1                                00000001
GP2                                00000002
GP3                                00000003
GP4                                00000004
GP5                                00000005
GPIO                               00000006
GPWUF                              00000007
IFC                                00000007
IFCLR                              BTFSS
IFNC                               SKPC
IFNZ                               SKPNZ
IFSET                              BTFSC
IFZ                                SKPC
INDF                               00000000
LSB                                00000000
MSB                                00000007
OSCCAL                             00000005
PA0                                00000005
PC                                 00000002
PD                                 00000003
RP0                                00000005
SKPCLR                             BTFSC
SKPSET                             BTFSC
STATUS                             00000003
TMR0                               00000001
TO                                 00000004
Z                                  00000002
__12C509                            00000001
bad_LRC                             00000003
bad_parity                          00000002
bit_delay                           0000000F
bit_loop                            00000006
buf_end                             00000004
buf_ptr                             00000007
buf_sz                              00000028
card                                00000001
char_buf                            0000000B
chk_data                             00000063
clock                               00000002
clr_buf_loop                         00000041
clrloop                              00000039
count                               00000009
data_0                               00000069
data_1                               00000065
delay_loop                           00000014
dump_buffer                          00000095
end_code                             000000F8
flag                                 0000000A
found_end                            00000001
<table>
<thead>
<tr>
<th>SYMBOL TABLE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>found_start</td>
<td>00000000</td>
</tr>
<tr>
<td>get_char</td>
<td>00000026</td>
</tr>
<tr>
<td>get_put_char</td>
<td>00000018</td>
</tr>
<tr>
<td>get_put_done</td>
<td>0000002C</td>
</tr>
<tr>
<td>got_char</td>
<td>0000007C</td>
</tr>
<tr>
<td>hi_mem</td>
<td>00000033</td>
</tr>
<tr>
<td>invert_tx</td>
<td>00000000</td>
</tr>
<tr>
<td>lo_mem</td>
<td>00000010</td>
</tr>
<tr>
<td>loop_buffer</td>
<td>00000098</td>
</tr>
<tr>
<td>main_loop</td>
<td>0000003F</td>
</tr>
<tr>
<td>next_char</td>
<td>00000076</td>
</tr>
<tr>
<td>not_LRC</td>
<td>00000089</td>
</tr>
<tr>
<td>num_chr</td>
<td>00000008</td>
</tr>
<tr>
<td>parityLRC</td>
<td>0000000C</td>
</tr>
<tr>
<td>put_char</td>
<td>00000020</td>
</tr>
<tr>
<td>read_buf</td>
<td>00000005</td>
</tr>
<tr>
<td>send_char</td>
<td>00000002</td>
</tr>
<tr>
<td>ser_out</td>
<td>00000000</td>
</tr>
<tr>
<td>signal</td>
<td>00000003</td>
</tr>
<tr>
<td>start</td>
<td>00000031</td>
</tr>
<tr>
<td>start_code</td>
<td>00000058</td>
</tr>
<tr>
<td>store_bit</td>
<td>0000006A</td>
</tr>
<tr>
<td>temp</td>
<td>0000000D</td>
</tr>
<tr>
<td>wait_card</td>
<td>0000005F</td>
</tr>
<tr>
<td>wt_clk_hi</td>
<td>00000079</td>
</tr>
<tr>
<td>wt_clk_lo</td>
<td>00000061</td>
</tr>
</tbody>
</table>

MEMORY USAGE MAP ('X' = Used, ' ' = Unused)

```
0000 : XXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXX
0040 : XXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXX
0080 : XXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXXXXX XXXXXXXXXXXXXXXX----- ---------------
```

All other memory blocks unused.

Program Memory Words Used: 171
Program Memory Words Free: 853

Errors : 0
Warnings : 0 reported, 0 suppressed
Messages : 0 reported, 14 suppressed
Note the following details of the code protection feature on PICmicro® MCUs.

- The PICmicro family meets the specifications contained in the Microchip Data Sheet.
- Microchip believes that its family of PICmicro microcontrollers is one of the most secure products of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the PICmicro microcontroller in a manner outside the operating specifications contained in the data sheet. The person doing so may be engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable”.
- Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our product.

If you have any further questions about this matter, please contact the local sales office nearest to you.

Information contained in this publication regarding device applications and the like is intended through suggestion only and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. No representation or warranty is given and no liability is assumed by Microchip Technology Incorporated with respect to the accuracy or use of such information, or infringement of patents or other intellectual property rights arising from such use or otherwise. Use of Microchip’s products as critical components in life support systems is not authorized except with express written approval by Microchip. No licenses are conveyed, implicitly or otherwise, under any intellectual property rights.

Note the following details of the code protection feature on PICmicro® MCUs.

- The PICmicro family meets the specifications contained in the Microchip Data Sheet.
- Microchip believes that its family of PICmicro microcontrollers is one of the most secure products of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the PICmicro microcontroller in a manner outside the operating specifications contained in the data sheet. The person doing so may be engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable”.
- Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our product.

If you have any further questions about this matter, please contact the local sales office nearest to you.

Trademarks

The Microchip name and logo, the Microchip logo, FilterLab, KEELOQ, microID, MPLAB, PIC, PICmicro, PICMASTER, PICSTART, PRO MATE, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

dsPIC, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, microPort, Migratable Memory, MPASM, MPLIB, MPLINK, MPSIM, MXDEV, PICC, PICDEM, PICDEM.net, rPIC, Select Mode and Total Endurance are trademarks of Microchip Technology Incorporated in the U.S.A.

Serialized Quick Turn Programming (SQTP) is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

© 2002, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

Printed on recycled paper.
**AMERICAS**

**Corporate Office**
2335 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200 Fax: 480-792-7277
Web Address: http://www.microchip.com

**Rocky Mountain**
2335 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7966 Fax: 480-792-7456

**Atlanta**
500 Sugar Mill Road, Suite 200B
Atlanta, GA 30350
Tel: 770-844-0034 Fax: 770-844-0307

**Boston**
2 Lan Drive, Suite 120
Westford, MA 01886
Tel: 978-692-3848 Fax: 978-692-3821

**Chicago**
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

**Dallas**
4570 Westgrove Drive, Suite 160
Addison, TX 75001
Tel: 972-818-7924 Fax: 972-818-2924

**Detroit**
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48324
Tel: 248-538-2250 Fax: 248-538-2260

**Kokomo**
2767 S. Albright Road
Kokomo, Indiana 46902
Tel: 765-864-8360 Fax: 765-864-8387

**Los Angeles**
18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-263-1888 Fax: 949-263-1338

**New York**
150 Motor Parkway, Suite 202
Hauppauge, NY 11788
Tel: 631-273-5055 Fax: 631-273-5035

**San Jose**
140 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

**Toronto**
6285 Northamer Drive, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel: 905-673-0699 Fax: 905-673-6509

**ASIA/PACIFIC**

**Australia**
Microchip Technology Australia Pty Ltd
Suite 22, 41 Rawson Street
Epping 2121, NSW
Australia
Tel: 61-2-9668-6733 Fax: 61-2-9668-6755

**China - Beijing**
Microchip Technology Consulting (Shanghai) Co., Ltd., Beijing Liaison Office
Unit 915
Bei Hui Wan Tai Bldg.
No. 6 Chaoyangmen Beidajie
Beijing, 100027, No. China
Tel: 86-10-85282100 Fax: 86-10-85282104

**China - Chengdu**
Microchip Technology Consulting (Shanghai) Co., Ltd., Chengdu Liaison Office
Rm. 2401, 24th Floor,
Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel: 86-28-6766200 Fax: 86-28-6766599

**China - Fuzhou**
Microchip Technology Consulting (Shanghai) Co., Ltd., Fuzhou Liaison Office
No. 71 Wusi Road
Fuzhou 350001, China
Tel: 86-591-7503506 Fax: 86-591-7503521

**Japan**
Microchip Technology Japan K.K.
Benex S-1 6F
3-18-20, Shinyokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, Japan
Tel: 81-45-471-6166 Fax: 81-45-471-6122

**Korea**
Microchip Technology Korea
168-1, Younbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea 135-882
Tel: 82-2-554-7200 Fax: 82-2-558-5934

**Singapore**
Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-02 Prime Centre
Singapore, 189980
Tel: 65-334-8870 Fax: 65-334-8850

**Taiwan**
Microchip Technology Taiwan
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

**EUROPE**

**Denmark**
Microchip Technology Nordic ApS
Regus Business Centre
Lautrup høj 1-3
Ballerpup DK-2750 Denmark
Tel: 45 4420 9895 Fax: 45 4420 9910

**France**
Microchip Technology SARL
Parc d’Activite du Moulin de Massy
43 Rue du Saule Trappu
Batiment A - ler Etage
91300 Massy, France
Tel: 33-1-69-33-63-20 Fax: 33-1-69-30-90-79

**Germany**
Microchip Technology GmbH
Gustav-Heinemann Ring 125
D-81739 Munich, Germany
Tel: 49-89-6275-1300 Fax: 49-89-6275-1344

**Italy**
Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Taurus 1 V. Le Colleoni 1
20041 Agerate Brianza
Milan, Italy
Tel: 39-039-65791-1 Fax: 39-039-6899883

**United Kingdom**
Arizona Microchip Technology Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5869 Fax: 44-118 921-5820

01/18/02