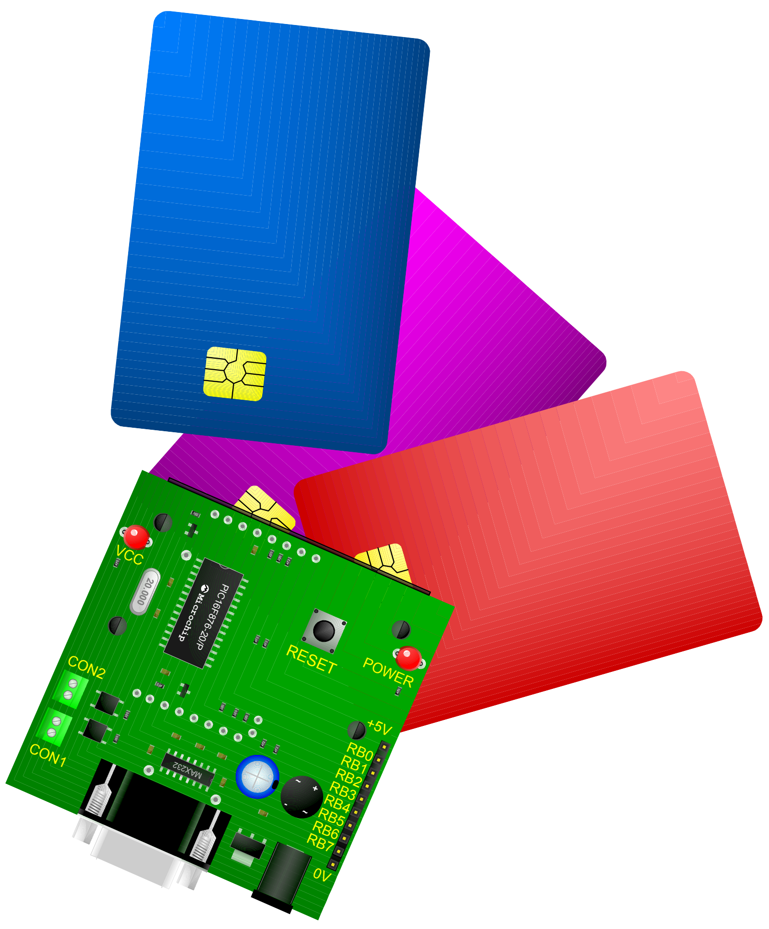
SMART Proton

Smart Card development Interface for PICmicro(TM) MCU's



Please Note.

First Published by Crownhill Associates Ltd.
June 2003

Table of Contents.

Introduction

What is a Smart card? In general a smart card is a plastic card, the size of a credit card, with an embedded microprocessor containing an operating system and erasable non-volatile memory. Physical protection against unauthorized tampering with the card is provided by integrating the microprocessor and memory in a single chip with special attention to ensure that there are no data paths that can be monitored or probed. This secure chip is connected to a thin circuit board and encapsulated with an epoxy. The "module" is then glued within a well routed into the plastic card. This prohibits physical access to the microprocessor and provides a more durable medium than more traditional magnetic stripe cards. In a smart card a microprocessor operates under the control of a integral program called an operating system. A serial interface - which makes it impossible to access the memory directly - is employed to communicate with the card. An ISO (International Standards Organization) protocol is used to exchange commands and data with the card.

Smart Cards come in two types: chip cards and chipless cards. Chip cards contain an integrated circuit (IC) chip that gives the card "smartness" or the ability to process data and make decisions about data.
The following questions and answers are from the web site for the Smart Card Forum.

> Q: What is a contactless card?
> A: There are two types of contactless cards. The first is a contactless proximity card in which the card is read by inserting it in a special reader. The second is a remote contactless card, in which the card can be read from a distance such as at a toll both.
>
> Q: How is a chip card different from the magnetic stripe cards that I carry in my wallet or purse?
> A: Existing magnetic stripe cards usually access an on-line data base. A chip card carries more information than can be accommodated
> on a magnetic stripe card. A chip card can make a decision; it has relatively powerful processing capabilities that allow it to do more than a magnetic stripe card, e.g. data encryption.
>
> Q: You will hear the terms "chip card," "integrated circuit card" and "smart card" used to refer to a plastic card with a chip. Are these different types of technology?
> A: No, a chip card is the same as an integrated circuit card. There are three types of integrated circuit cards:

Over the last decade, the use of smart cards has increased dramatically, to the extent that few industries have not seen their introduction at some level.

Why smart cards? They resemble the familiar credit card in appearance, but they can do a whole lot more than the simple magnetic strip cards. Smart cards contain integrated circuits that give them the ability to retain and process data. As a result, they have several benefits over the familiar magnetic strip card:

- Without the magnetic strip, they are more secure, they don’t lose their data during normal use
- They can store significantly more data than a magnetic strip card.
- They can be used for more than one application.
- They can be extremely secure, or not, as the application demands.

Currently the most common use for the smart card is for authentication, whether GSM, PayTV, credit/debit cards, loyalty cards, however they are also popular in areas such as, event logging cards, building security access cards, and identification cards, telephone calling cards and vending machine purchase cards.

Smart cards make excellent security devices. Where magnetic strip cards used to be deployed smart cards are now becoming the norm, being more user friendly and exhibiting a higher degree of security whilst being particularly difficult to copy and at the same time possessing the capability of retaining a larger amount of information, the Smart card is the natural choice for security conscious applications.

Security is not the only asset of the smart card, memory is also a big factor in its wide spread adoption. Where data needs collecting, storing or transporting Smart cards make the ideal medium, being user friendly, robust and reliable.

As technology moves forward Smart cards adopt and exploit the available technology, Smart cards are now autonomously processing data. They are able to receive and execute a program from the machine with which they are communicating. These cards are able to interpret the program downloaded to them and perform the appropriate processing, without machine or user intervention.

Why Proton-SMART, because Proton-SMART provides a comprehensive introduction to using the embedded micro-controller in a smart card environment, starting with memory, leading to protected memory, then in module two and three, specialist secure memory through to GSM and SCOS. Proton-SMART will become an invaluable building block and essential tool in your   venture into this evolving technology.

Smart Card use in the embedded environment is usually only half of the total package. Often smart cards are used with computers and terminals to allow graphical user interfaces to be utilised to make the whole user experience less alien.
Crownhill also supply a package to assist with Smart Card application development, the “ChipDrive starter pack” , there are several different manufac-

turers of smartcards, terminals, and drivers. There are also many industry standards for card protocols. The Crownhill ChipDrive starter pack aims to make the integration of smartcards and terminals into your application as easy as possible.

The ChipDrive starter pack is supplied with an Intelligent Smart Card reader writer terminal, a selection of Smart Cards, example application Source code in C, Delphi and VB for the Windows environment.

The Starter pack makes extensive use of the cardserver.dll API to ease access to and use of many different cards. The ChipDrive card terminal is available with Serial or USB interfaces and is compliant with PCSC as well as proprietary communications protocols.

The ChipDrive starter pack has been utilized to create many very successful commercial applications, integrating PC's and embedded solutions, one example of a ChipDrive application, created with the starter pack can be found at www.edsim2000.com

The CHIPDRIVE StarterPack enables effective development of customized applications and systems using smart cards, card terminals, and PCs. Incorporate smart card technology into your current application using the Proton Smart development system and the ChipDrive starter pack, Develop systems specific to your needs as well as those of your clients.

For more information on the ChipDrive Starter pack contact lester@Crownhill.co.uk

Interfacing a 14C02 Memory Card.

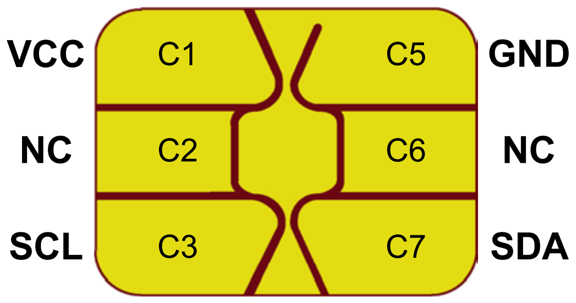
The 14C02 card (sometimes supplied as 24C02) contains 2kbits of re-programmable eeprom memory, which relates to 256 bytes. It is capable of working from 3 to 5 Volts, and has an endurance of 1 million write cycles, with a minimum data retention of 10 years. It uses a standard I^2C interface for communications, and is extremely easy to use with the PROTON+ Compiler.

In this era of multi megabyte media-cards, 256 bytes of memory is often frowned upon, or is not seen as being useful. But not all applications require megabytes of storage, or even kilobytes. For transporting a few bytes of code from one application to another, the 14C02 is an ideal solution.

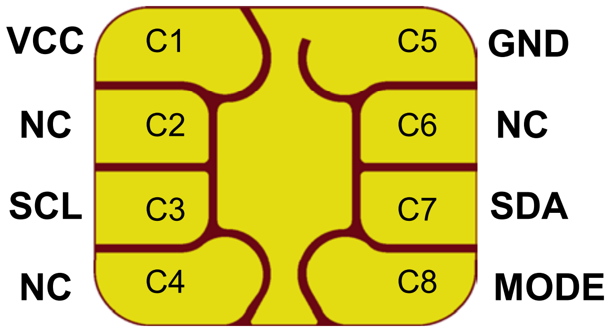
Physical characteristics of the 14C02.

Before we look at programming for the 14C02 memory card, we must take some time out to look at the physical makeup of the card, or indeed, any smartcard. Most smartcards follow a standard named ISO7816, (some more loosely than others). This, among other things, dictates the physical appearance of the card. i.e. size, and contact placements etc.

The 14C02 has a 6 pin (M3), or 8 pin (M4) footprint (shown below). These are electrically identical, but the MODE pin is brought out for use on the M4 type. The MODE pin enables/disables write protection, and is often left floating (which internally disables write protection). Hence the M3 footprint. It's an ironic twist that the official datasheet for the 14C02 card shows an M4 footprint, but as of this date, no 14C02 card has ever been produced with an M4 type contact pattern.



14Cxxx M3 Footprint.



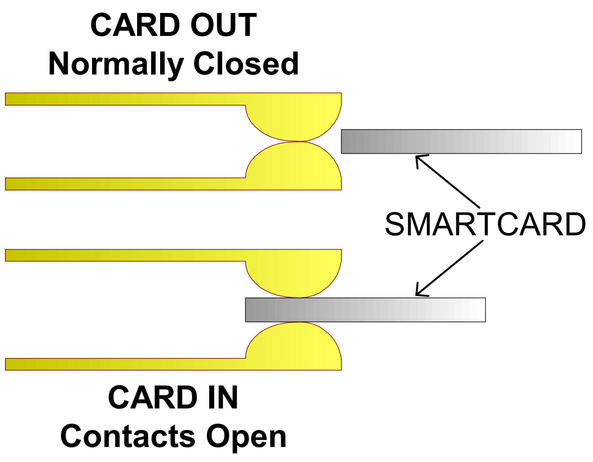
14Cxxx M4 Footprint.

As you can see, the card uses the standard I/O lines associated with the I^2C bus, namely SCL (clock), and SDA (data). One criteria of the ISO7816 standard, is the placement of the VCC, GND, and I/O lines on the card's contact footprint. All cards that follow this standard will have pad C1 as VCC, C5 as GND, and C3, C7 as clock and data. The other pads are used for various interface pins on differing types of card that require a different communications protocol. We'll see this later in the document.

Step by Step Guide.

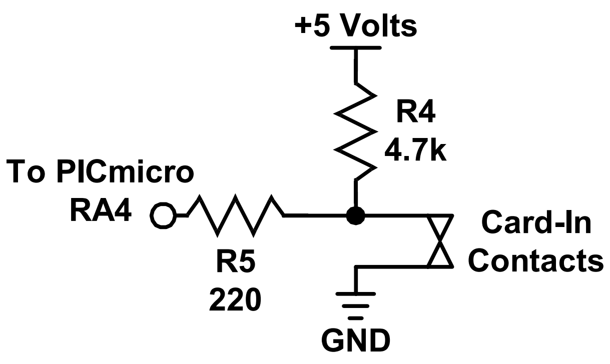
In this first application using a memory card, we'll look at each part of the development step by step. Later applications in the document will assume that this section has been read and understood.

We'll start with a simple, but very important program, to detect if a card is inserted into the card socket. The ISO7816 socket used in the PROTON SMART has two separate contacts that open when a card is inserted fully, this means that they are normally closed when not being used. This is important to remember, as the code relies on this fact for correct operation. The illustration below shows this more clearly.



ISO7816 Socket. Card-In Contacts.

One of the contacts is connected to the PICmicro's RA4 pin (PORTA.4), via a pull-up, and current limiting resistor. The other is connected to common ground. Shown below.



Card-In Contacts circuit.

This means that the PICmicro sees a LOGIC 0 with no card present, and a LOGIC 1 when a card is inserted.

NOTE: Not all ISO7816 card sockets have normally closed contacts, some have normally open types. This is important to remember, as the program code will require altering slightly.

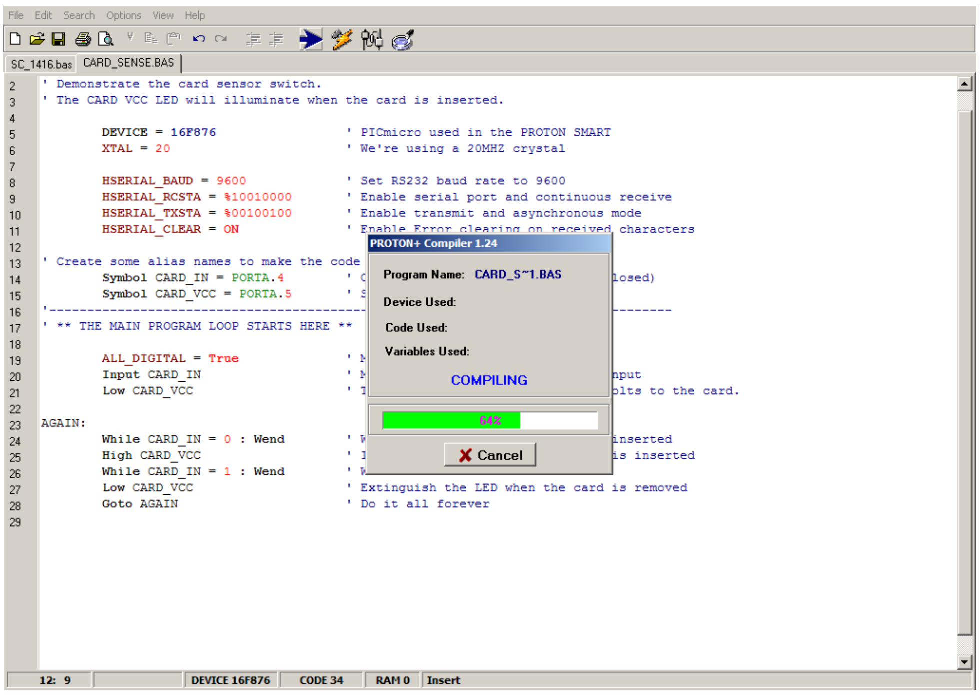
I feel a Presence.

We'll now put this knowledge into practice, and write a small program to demonstrate the action of the card sensor. This will also allow you to familiarise yourself with the PROTON+ compiler.

Run the compiler, and load the program **CARD_SENSE.BAS**. You will find this on the CDROM, inside the **SAMPLES** folder. The program is also shown below.

```
' Program CARD_SENSE.BAS
' Demonstrate the card sensor switch.
' The CARD VCC LED will illuminate when the card is inserted.

Device = 16F876                    ' PICmicro used in the PROTON SMART
XTAL = 20                          ' We're using a 20MHz crystal

' Create some alias names to make the code more readable
Symbol CARD_IN = PORTA.4           ' CARD in sensor switch (normally closed)
Symbol CARD_VCC = PORTA.5          ' Supplies the card with 5 Volts
' ** THE MAIN PROGRAM LOOP STARTS HERE **
ALL_DIGITAL = True                 ' Make PORTA all digital IO
Input CARD_IN                      ' Make the CARD IN contact pin an input
Low CARD_VCC                       ' Turn OFF the VCC LED, and the 5 Volts to the card.

AGAIN:
While CARD_IN = 0 : Wend           ' Wait in a loop until the card is inserted
High CARD_VCC                      ' Illuminate the LED when the card is inserted
While CARD_IN = 1 : Wend           ' Wait for the card to be removed
Low CARD_VCC                       ' Extinguish the LED when the card is removed
Goto AGAIN                         ' Do it all forever
```

Compile the program, by clicking on the **COMPILE** icon located on the toolbar, and you should see the screen shown below.



```
' Demonstrate the card sensor switch.
' The CARD VCC LED will illuminate when the card is inserted.

        DEVICE = 16F876                 ' PICmicro used in the PROTON SMART
        XTAL = 20                       ' We're using a 20MHZ crystal

        HSERIAL_BAUD = 9600             ' Set RS232 baud rate to 9600
        HSERIAL_RCSTA = %10010000       ' Enable serial port and continuous receive
        HSERIAL_TXSTA = %00100100       ' Enable transmit and asynchronous mode
        HSERIAL_CLEAR = ON              ' Enable Error clearing on received characters

' Create some alias names to make the code
        Symbol CARD_IN = PORTA.4        ' C                        losed)
        Symbol CARD_VCC = PORTA.5       ' S
'-----------------------------------------                       --------
' ** THE MAIN PROGRAM LOOP STARTS HERE **

        ALL_DIGITAL = True              ' M
        Input CARD_IN                   ' M                        nput
        Low CARD_VCC                    ' T                        olts to the card.

AGAIN:
        While CARD_IN = 0 : Wend        ' W                        inserted
        High CARD_VCC                   ' I                        is inserted
        While CARD_IN = 1 : Wend        ' W
        Low CARD_VCC                    ' Extinguish the LED when the card is removed
        Goto AGAIN                      ' Do it all forever
```

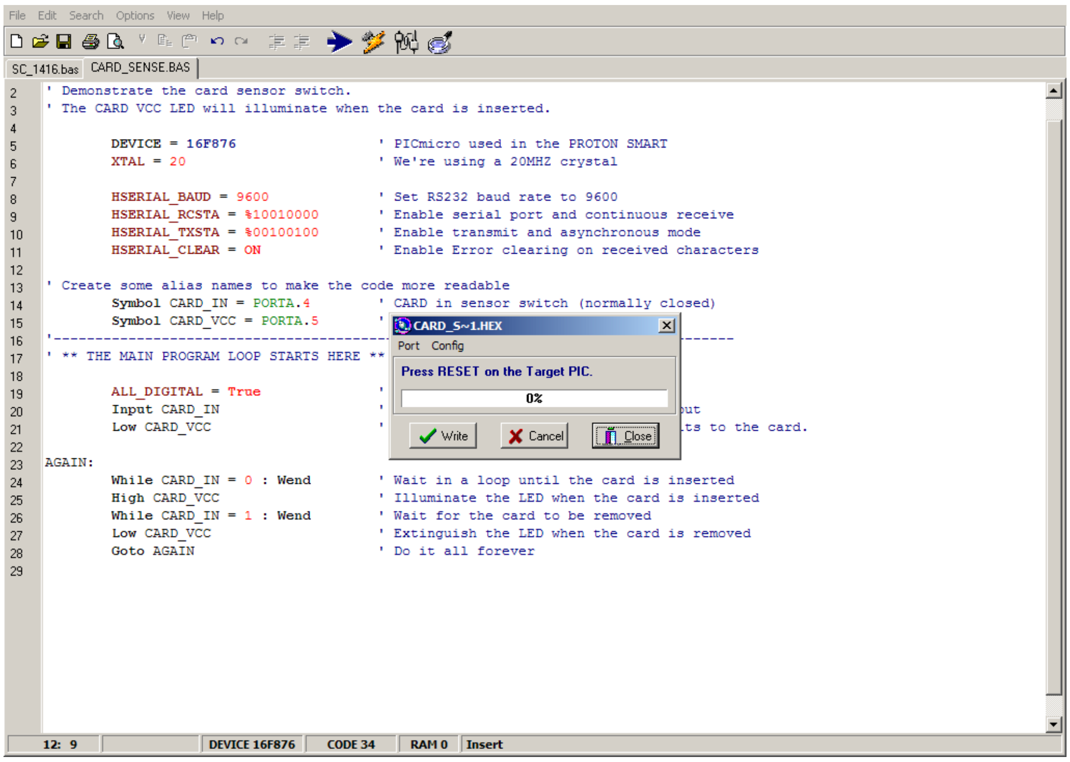
```
PROTON+ Compiler 1.24
Program Name:   CARD_S~1.BAS
Device Used:
Code Used:
Variables Used:
COMPILING
64%
Cancel
```

```
12: 9 | DEVICE 16F876 | CODE 34 | RAM 0 | Insert
```

If no errors were produced while compiling, the program is ready for downloading to the PROTON SMART board. The PROTON SMART has a serial bootloader incorporated into the on-board PICmicro. This allows it to be programmed using a standard RS232 serial cable connected to a COM port of the PC (more details of this can be found in the compiler's documentation, and in the electrical specifications located at the end of this document.

Connect a 9 to 12 Volt power supply to the PROTON SMART board, and a suitable 9-pin serial cable to a free COM port on the PC, and the PROTON SMART. The POWER LED should now be illuminated. If not, check the power supply, and its connections before proceeding.

Click on the **DOWNLOAD** icon, also located on the toolbar, and a small window will appear on the screen, shown below.



```
' Demonstrate the card sensor switch.
' The CARD VCC LED will illuminate when the card is inserted.

        DEVICE = 16F876                 ' PICmicro used in the PROTON SMART
        XTAL = 20                       ' We're using a 20MHZ crystal

        HSERIAL_BAUD = 9600             ' Set RS232 baud rate to 9600
        HSERIAL_RCSTA = %10010000       ' Enable serial port and continuous receive
        HSERIAL_TXSTA = %00100100       ' Enable transmit and asynchronous mode
        HSERIAL_CLEAR = ON              ' Enable Error clearing on received characters

' Create some alias names to make the code more readable
        Symbol CARD_IN = PORTA.4        ' CARD in sensor switch (normally closed)
        Symbol CARD_VCC = PORTA.5       '
'-----------------------------------------                   -------
' ** THE MAIN PROGRAM LOOP STARTS HERE **

        ALL_DIGITAL = True              '
        Input CARD_IN                   '                    put
        Low CARD_VCC                    '                    ts to the card.

AGAIN:
        While CARD_IN = 0 : Wend        ' Wait in a loop until the card is inserted
        High CARD_VCC                   ' Illuminate the LED when the card is inserted
        While CARD_IN = 1 : Wend        ' Wait for the card to be removed
        Low CARD_VCC                    ' Extinguish the LED when the card is removed
        Goto AGAIN                      ' Do it all forever
```

Press RESET on the Target PIC.

0%

12: 9 | DEVICE 16F876 | CODE 34 | RAM 0 | Insert

If a COM port other than COM 1 was used for the serial connection, click on the PORT menu located in the downloader's window, and change the port (this will be saved so that the chosen port will always be default).

Press the **RESET** button on the PROTON SMART board, and the program will begin downloading to the on-board PICmicro. If the screen stays the same, or an error message appears, then check the serial cable, and COM port for errors. A common error is in the choice of serial cables. There are two types of serial cable available, a NULL MODEM, and a STRAIGHT CONNECTION. A null modem cable has the pins internally reversed, and is therefore not suitable for the downloading process.

Assuming that all has gone well, and the program is now situated in the PICmicro, we can test it.

Insert a card into the socket, and watch the CARD VCC LED Illuminate. Remove the card, and the LED will extinguish. At this stage it doesn't matter which way the card is oriented when inserted into the socket, or which type of card is used, as we are not actually accessing the card, we're simply sensing it's presence. If the LED behaves as it should, then well done! You've now programmed the PROTON SMART correctly. It really is simple isn't it ?

We'll use this method of card sensing throughout all the future applications. Maybe with a little twist, or tweak here and there, but it will remain the same principal.

Storing data in the 14C02 memory card.

The previous application may seem trivial, however being able to tell if the card is in its socket is a crucial lesson for any code to work successfully. So we'll put it to good use now, and actually write, and read some data from the 14C02 card. As we've already seen, the I^{2}C memory cards follow the well documented protocols of the Philips I^{2}C bus. However, not all I^{2}C devices are accessed the same. One of the main differences is the SLAVE address, as each I^{2}C device has a different 4-bit value incorporated into the slave address, that signifies what it is on the two-wire bus. For serial eeproms, and this includes an I^{2}C memory cards, the address is binary 1010. Also, some devices require a 16-bit address, while others only require 8-bits. There are other small differences, but these are well documented in the datasheets provided on the CDROM. Make sure you read and understand these before attempting any practical applications.

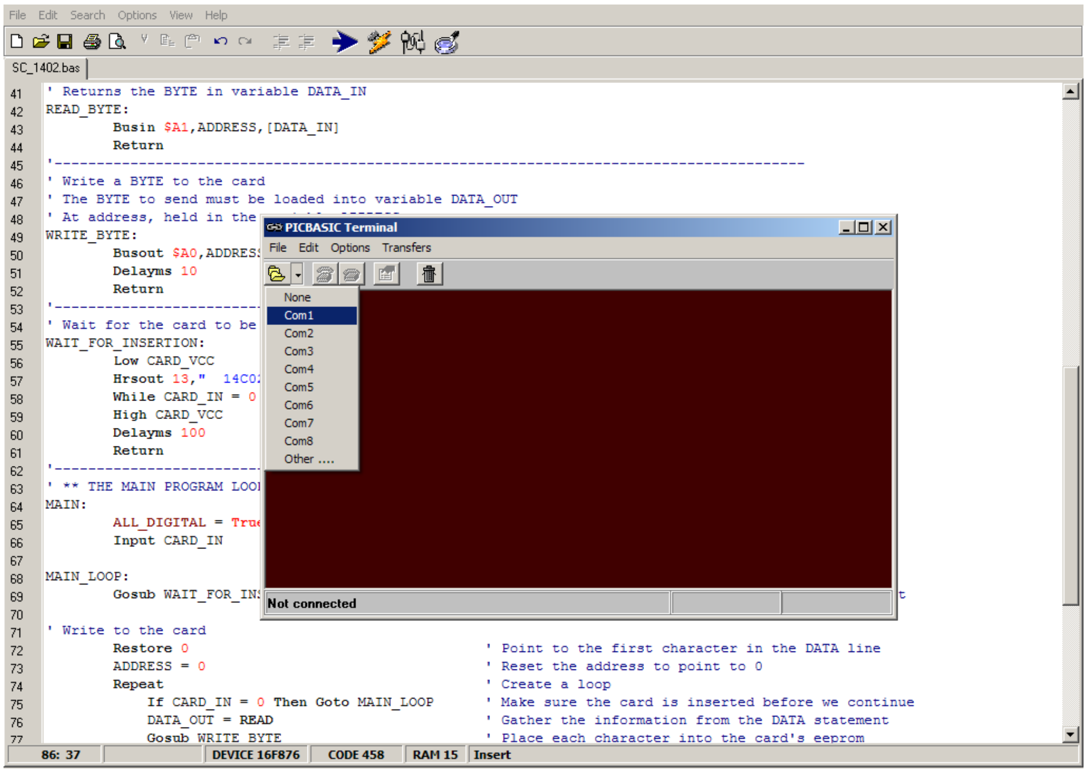
Load the program **MC_14C02.BAS**, and compile it. The BASIC code is shown below.

```
' Program MC_14C02.BAS
' Access a 14C02 (2K bits) 12C memory card

Device = 16F876                    ' PICmicro used in the PROTON SMART
XTAL = 20                          ' We're using a 20MHZ crystal
SCL_PIN = PORTC.3                  ' Assign the compiler's SCL pin for BUSIN/BUSOUT
SDA_PIN = PORTC.4                  ' Assign the compiler's SDA pin for BUSIN/BUSOUT
HSERIAL_BAUD = 9600                ' Set baud rate to 9600
HSERIAL_RCSTA = %10010000          ' Enable serial port and continuous receive
HSERIAL_TXSTA = %00100100          ' Enable transmit and asynchronous mode
HSERIAL_CLEAR = ON                 ' Enable Error clearing on received characters
'---------------------------------------------------------------------------------
' Declare some variables
Dim DATA_BYTE as BYTE              ' Declare a BYTE type variable for data transfer
Dim ADDRESS as BYTE                ' Declare a BYTE type variable for an address
Dim DATA_IN as DATA_BYTE.Lowbyte   ' Alias DATA_IN to the lowbyte of DATA_BYTE
Dim DATA_OUT as DATA_BYTE.Lowbyte ' Also alias DATA_OUT to the lowbyte of DATA_BYTE
```

```
'-------------------------------------------------------------------------------------
' Define some aliases
Symbol CARD_VCC = PORTA.5              ' Supplies the card with 5 Volts
Symbol CARD_IN = PORTA.4               ' CARD in sensor switch (normally closed)
Symbol SCL = PORTC.3                   ' Card's CLK line
Symbol SDA = PORTC.4                   ' Card's IO line
'-------------------------------------------------------------------------------------
' Create some data to write to the card
DATA "THIS HAS BEEN WRITTEN TO, AND READ FROM THE 14C02 MEMORY CARD." , 13 , 0

Delayms 500                            ' Wait for the power supply to fully stabilise
Hrsout 1                               ' Clear the serial terminal's screen before we start
Goto Main                              ' Then jump over the subroutines to the main program
loop
'-------------------------------------------------------------------------------------
' Read a BYTE from the card
' From address, held in the variable ADDRESS
' Returns the BYTE in variable DATA_IN
READ_BYTE:
Busin $A1,ADDRESS,[DATA_IN]
Return
'-------------------------------------------------------------------------------------
' Write a BYTE to the card
' The BYTE to send must be loaded into variable DATA_OUT
' At address, held in the variable ADDRESS
WRITE_BYTE:
Busout $A0,ADDRESS,[DATA_OUT]          ' Write each location
Delayms 10                             ' Delay 10ms after each write
Return
'-------------------------------------------------------------------------------------
' Wait for the card to be inserted into the socket before continuing
WAIT_FOR_INSERTION:
Low CARD_VCC                           ' Disable the card's VCC (5 Volts)
Hrsout 13,"  14C02 ACCESS",13,"PLEASE INSERT CARD",13
While CARD_IN = 0 : Wend               ' Wait for card insertion
High CARD_VCC                          ' Enable the card's VCC (5 Volts)
Delayms 100                            ' Wait for the card to fully power up
Return
'-------------------------------------------------------------------------------------
' ** THE MAIN PROGRAM LOOP STARTS HERE **
MAIN:
ALL_DIGITAL = True                     ' Make PORTA all digital IO
Input CARD_IN                          ' Make the card sensor pin an input

MAIN_LOOP:
Gosub WAIT_FOR_INSERTION               ' Wait for the card to be inserted into the socket
' Write to the card
Restore 0                              ' Point to the first character in the DATA line
ADDRESS = 0                            ' Reset the address to point to 0
Repeat                                 ' Create a loop
If CARD_IN = 0 Then MAIN_LOOP          ' Make sure the card is inserted before we continue
DATA_OUT = READ                        ' Gather the information from the DATA statement
Gosub WRITE_BYTE                       ' Place each character into the card's eeprom
```

```
Inc ADDRESS                                 ' Point to the next memory element
Until DATA_OUT = 0                          ' Loop until a zero value is found
' Read and display data from the card
ADDRESS = 0                                 ' Reset the address to point to 0
Repeat                                      ' Create a loop
If CARD_IN = 0 Then MAIN_LOOP               ' Make sure the card is inserted before we continue
Gosub READ_BYTE                             ' Read a byte from the card's eeprom
Hrsout DATA_IN                              ' Display the characters on the serial terminal
Delayms 50                                  ' Delay between characters displayed
Inc ADDRESS                                 ' Point to the next memory element
Until DATA_IN = 0                           ' Loop until a zero value is found

While CARD_IN = 1 : Wend                    ' Wait for the card to be removed before proceeding
Goto MAIN_LOOP                              ' Got look for another card insertion
```

The above program, writes a text message to the card, then reads it back and displays it on the Serial Terminal (which is part of the compiler's IDE). Download the program to the PROTON SMART using the serial bootloader (discussed earlier). And initiate the Serial Terminal by clicking VIEW->SERIAL TERMINAL. The COM port for the serial terminal must now be chosen, this will need to be the same com port as used by the serial bootloader. This is shown below.

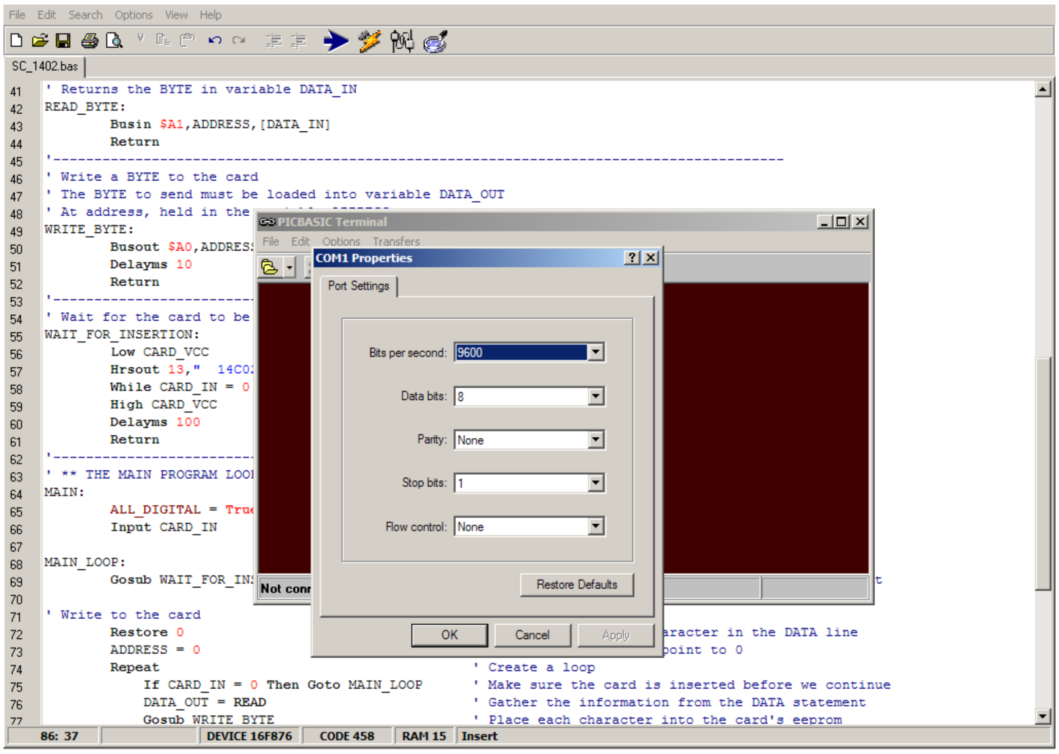


```
' Returns the BYTE in variable DATA_IN
READ_BYTE:
        Busin $A1,ADDRESS,[DATA_IN]
        Return
'------------------------------------------------------------------------------------------
' Write a BYTE to the card
' The BYTE to send must be loaded into variable DATA_OUT
' At address, held in the
WRITE_BYTE:
        Busout $A0,ADDRES
        Delayms 10
        Return
'-------------------------
' Wait for the card to be
WAIT_FOR_INSERTION:
        Low CARD_VCC
        Hrsout 13,"  14C0
        While CARD_IN = 0
        High CARD_VCC
        Delayms 100
        Return
'-------------------------
' ** THE MAIN PROGRAM LOO
MAIN:
        ALL_DIGITAL = True
        Input CARD_IN

MAIN_LOOP:
        Gosub WAIT_FOR_IN                                          t

' Write to the card
        Restore 0                                          ' Point to the first character in the DATA line
        ADDRESS = 0                                        ' Reset the address to point to 0
        Repeat                                             ' Create a loop
            If CARD_IN = 0 Then Goto MAIN_LOOP             ' Make sure the card is inserted before we continue
            DATA_OUT = READ                                ' Gather the information from the DATA statement
            Gosub WRITE_BYTE                               ' Place each character into the card's eeprom
```

Not connected

86: 37 | DEVICE 16F876 | CODE 458 | RAM 15 | Insert

The COM port must now be configured to match the RS232 data sent by the PROTON SMART board. This is 9600 baud, 1 Stop bit, 8 data bits, and no flow control.



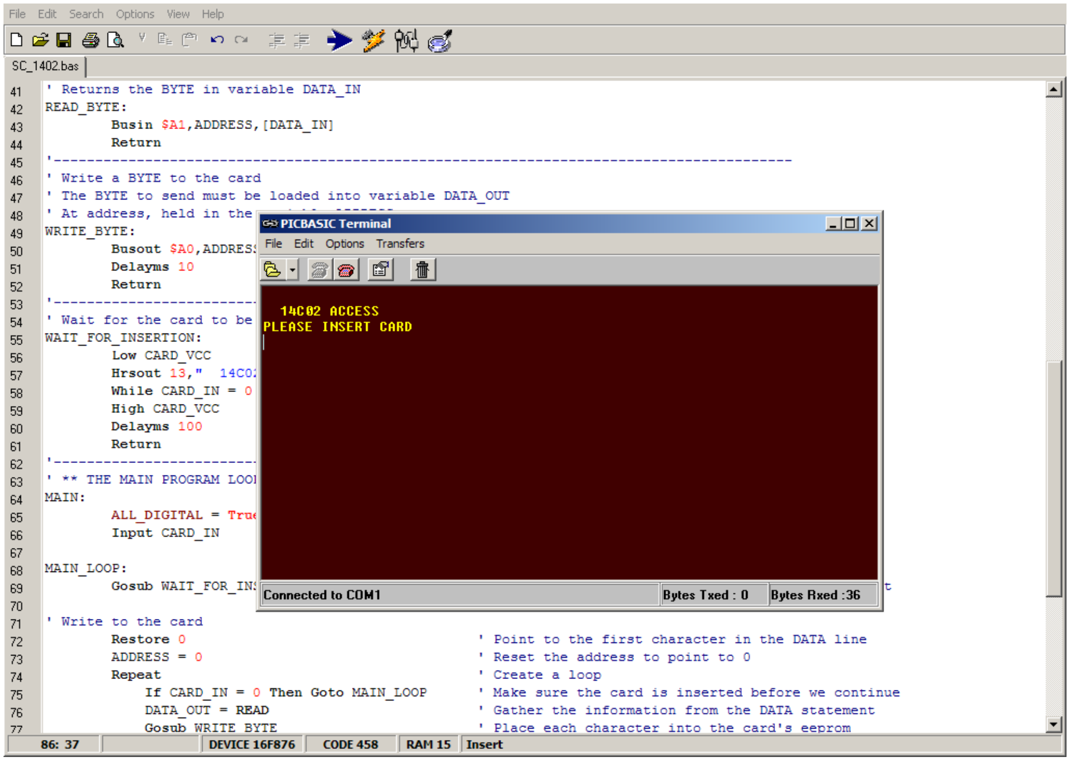
```
' Returns the BYTE in variable DATA_IN
READ_BYTE:
        Busin $A1,ADDRESS,[DATA_IN]
        Return
'-----------------------------------------------------------------------------------------
' Write a BYTE to the card
' The BYTE to send must be loaded into variable DATA_OUT
' At address, held in the
WRITE_BYTE:
        Busout $A0,ADDRES
        Delayms 10
        Return
'-------------------------
' Wait for the card to be
WAIT_FOR_INSERTION:
        Low CARD_VCC
        Hrsout 13,"  14C0
        While CARD_IN = 0
        High CARD_VCC
        Delayms 100
        Return
'-------------------------
' ** THE MAIN PROGRAM LOO
MAIN:
        ALL_DIGITAL = Tru
        Input CARD_IN

MAIN_LOOP:
        Gosub WAIT_FOR_IN

' Write to the card
        Restore 0                                       aracter in the DATA line
        ADDRESS = 0                                     oint to 0
        Repeat                                          ' Create a loop
            If CARD_IN = 0 Then Goto MAIN_LOOP          ' Make sure the card is inserted before we continue
            DATA_OUT = READ                             ' Gather the information from the DATA statement
            Gosub WRITE_BYTE                            ' Place each character into the card's eeprom
```

86: 37 | DEVICE 16F876 | CODE 458 | RAM 15 | Insert

COM1 Properties

Bits per second: 9600
Data bits: 8
Parity: None
Stop bits: 1
Flow control: None

Once this has been setup, and the computer has not complained about open com errors etc, press **RESET** on the PROTON_SMART board to re-initialise the program held within it. The serial terminal screen should display: -


File Edit Search Options View Help
SC_1402.bas
PICBASIC Terminal
File Edit Options Transfers

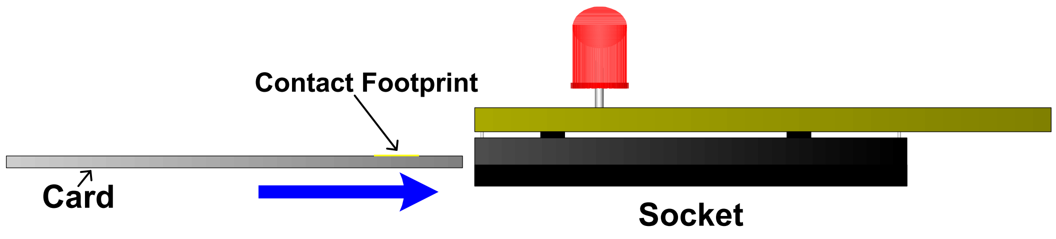
```
' Returns the BYTE in variable DATA_IN
READ_BYTE:
        Busin $A1,ADDRESS,[DATA_IN]
        Return
'-----------------------------------------------------------------------------------------
' Write a BYTE to the card
' The BYTE to send must be loaded into variable DATA_OUT
' At address, held in the
WRITE_BYTE:
        Busout $A0,ADDRES
        Delayms 10
        Return
'-------------------------
' Wait for the card to be
WAIT_FOR_INSERTION:
        Low CARD_VCC
        Hrsout 13,"  14C0
        While CARD_IN = 0
        High CARD_VCC
        Delayms 100
        Return
'-------------------------
' ** THE MAIN PROGRAM LOO
MAIN:
        ALL_DIGITAL = Tru
        Input CARD_IN

MAIN_LOOP:
        Gosub WAIT_FOR_IN

' Write to the card
        Restore 0                                       ' Point to the first character in the DATA line
        ADDRESS = 0                                     ' Reset the address to point to 0
        Repeat                                          ' Create a loop
            If CARD_IN = 0 Then Goto MAIN_LOOP          ' Make sure the card is inserted before we continue
            DATA_OUT = READ                             ' Gather the information from the DATA statement
            Gosub WRITE_BYTE                            ' Place each character into the card's eeprom
```

```
 14C02 ACCESS
PLEASE INSERT CARD
```

Connected to COM1 | Bytes Txed : 0 | Bytes Rxed :36

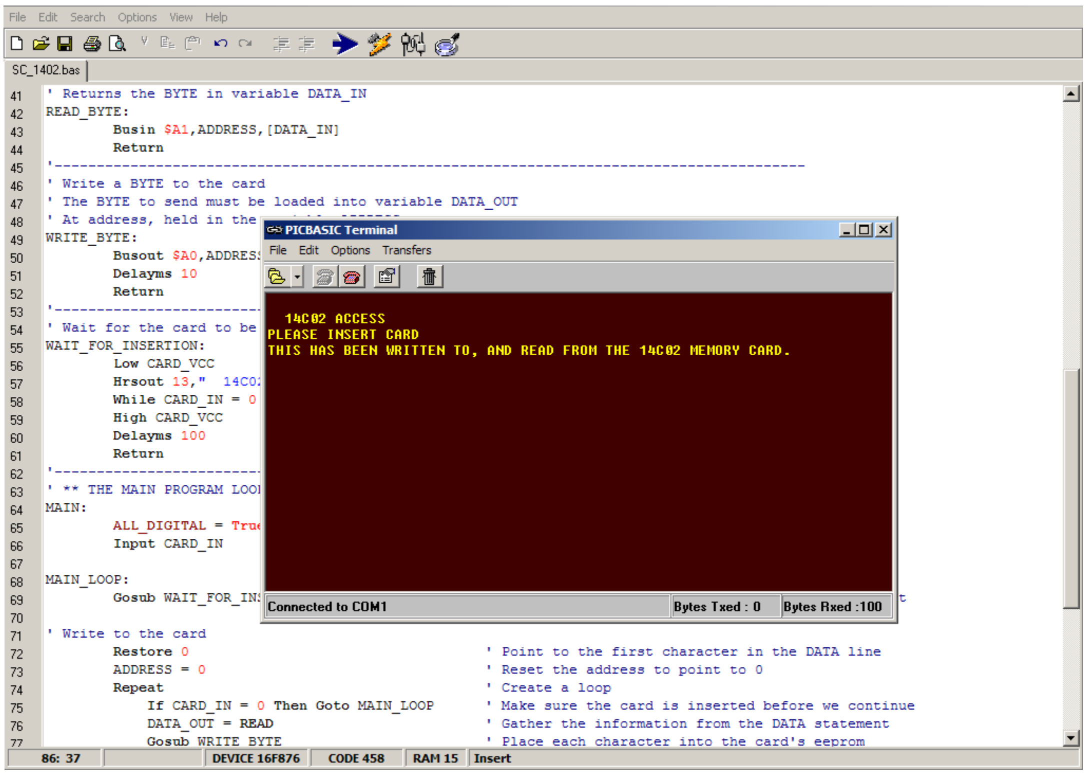
86: 37 | DEVICE 16F876 | CODE 458 | RAM 15 | Insert

Insert a 14C02 memory card into the PROTON SMART's socket, making sure that it is oriented correctly, with the contact footprint facing up towards the RESET button (shown on next page).



Card Orientation for the PROTON SMART board.

The program senses the card's presence (which was discussed earlier), and writes characters to the card using the WRITE_BYTE subroutine. It then sits in a loop reading characters from the card using the READ_BYTE subroutine, and outputs the characters serially using the PICmicro's internal USART. The serial terminal should look like the image below: -



```
' Returns the BYTE in variable DATA_IN
READ_BYTE:
        Busin $A1,ADDRESS,[DATA_IN]
        Return
'-----------------------------------------------------------------------------------------
' Write a BYTE to the card
' The BYTE to send must be loaded into variable DATA_OUT
' At address, held in the
WRITE_BYTE:
        Busout $A0,ADDRES
        Delayms 10
        Return
'-------------------------
' Wait for the card to be
WAIT_FOR_INSERTION:
        Low CARD_VCC
        Hrsout 13,"  14C0
        While CARD_IN = 0
        High CARD_VCC
        Delayms 100
        Return
'-------------------------
' ** THE MAIN PROGRAM LOO
MAIN:
        ALL_DIGITAL = True
        Input CARD_IN

MAIN_LOOP:
        Gosub WAIT_FOR_IN

' Write to the card
        Restore 0                                    ' Point to the first character in the DATA line
        ADDRESS = 0                                  ' Reset the address to point to 0
        Repeat                                       ' Create a loop
            If CARD_IN = 0 Then Goto MAIN_LOOP       ' Make sure the card is inserted before we continue
            DATA_OUT = READ                          ' Gather the information from the DATA statement
            Gosub WRITE_BYTE                         ' Place each character into the card's eeprom
```

```
  14C02 ACCESS
PLEASE INSERT CARD
THIS HAS BEEN WRITTEN TO, AND READ FROM THE 14C02 MEMORY CARD.
```

You will have noticed in the code listing, that we've kept basically the same test for card insertion, but wrapped it in a subroutine named WAIT_FOR_INSERTION. Two other tests are made for the card's presence, one in the write loop, and one in the read loop. This is an important feature in all card applications because it tests for the card before it is accessed. It also removes the socket's VCC (5 Volts) when a card is not inserted.

The rest of the applications in this document will follow the same procedures as this one. i.e. Compile program, download program, view the results on the serial terminal. So these actions will not be explained in detail, and we can get on with discussing the actual coding.

Interfacing a 24C16 Memory Card.

The 24C16 I^{2}C memory card, has 8 times as much memory as the 14C02, in that it contains 16kbits which relates to 2kbytes of storage capacity. And has a retention time of 40 years. Accessing the card is primarily the same as accessing the 14C02, but with one important difference, it's SLAVE byte (sometimes known as its CONTROL byte).

The 24C16 memory card, can be thought of as being eight 14C02 packages in a single wafer of silicon, each 2k block of data can be accessed separately. The 24C16 uses an 8-bit address, which poses a problem in accessing a memory element over 255. The way round this is to use 3 bits of the CONTROL byte to indicate which block of 2k you're talking to.

For example, a standard CONTROL byte for an eeprom write, in binary looks like 10100000. The last four bits denote which type of device we require on the bus. And the first bit indicates read or write. Bits 1 to 3 are commonly known as ADDRESS bits A0, A1, and A2 (not to be confused with the address byte), and are used to indicate which of the same type of device attached to the bus we want to talk to. And as the 24C16 is actually eight 14C02 devices, these bits control which part within the card we are communicating with.

These extra three bits are extracted from the actual memory address we need to access. For example, lets say we need to access address 1000 within the card. The binary for 1000 is 1111101000. We'll ignore the lower 8-bits for the moment, and we're left with binary 11 (or decimal 3), shift this value one bit to the left for alignment with bits 1 to 3 of CONTROL, MASK the bits required with an AND operator, then OR it into the CONTROL byte, which was 10100000. The CONTROL byte now has the binary value of 101000110, which will point to device 4 (remember, binary counts from 0 to 3) within the 24C16 card.

The lower 8-bits of the address that we ignored previously, which was binary 11101000 (decimal 232) is now placed as the standard 8-bit address byte. This means we're accessing address 232 from device 4 within the card.

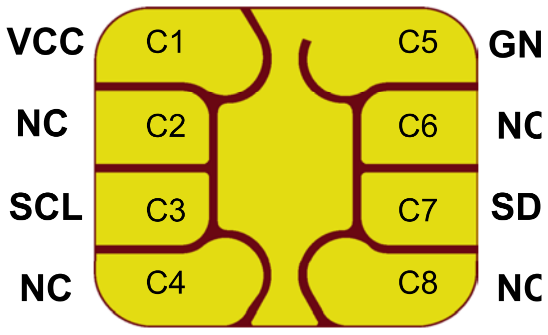
This sounds complex, but ends up as one line of BASIC code, shown below: -

```
CONTROL = %10100000 | ((ADDRESS.HIGHBYTE << 1) & %00001110)
```

And the BUS commands looks like: -

```
Busin CONTROL,ADDRESS.LOWBYTE,[DATA_IN]          ' Read a byte of data from the card
Busout CONTROL,ADDRESS.LOWBYTE,[DATA_OUT]        ' Write a byte of data to the card
```

There is another difference that may cause some concern, this time in the actual architecture of the 24C16 card. Some cards have an extra pin named WC. This is the WRITE CONTROL pin, and enables or disables write protect for the card. Leaving this floating (unconnected), or setting it high disables write protection, while pulling it to ground enables write protection. The BASIC code configures this pin as an input, so as not to interfere with cards that contain this pin, and those that don't. The contact footprint for the 24C16 is shown below.

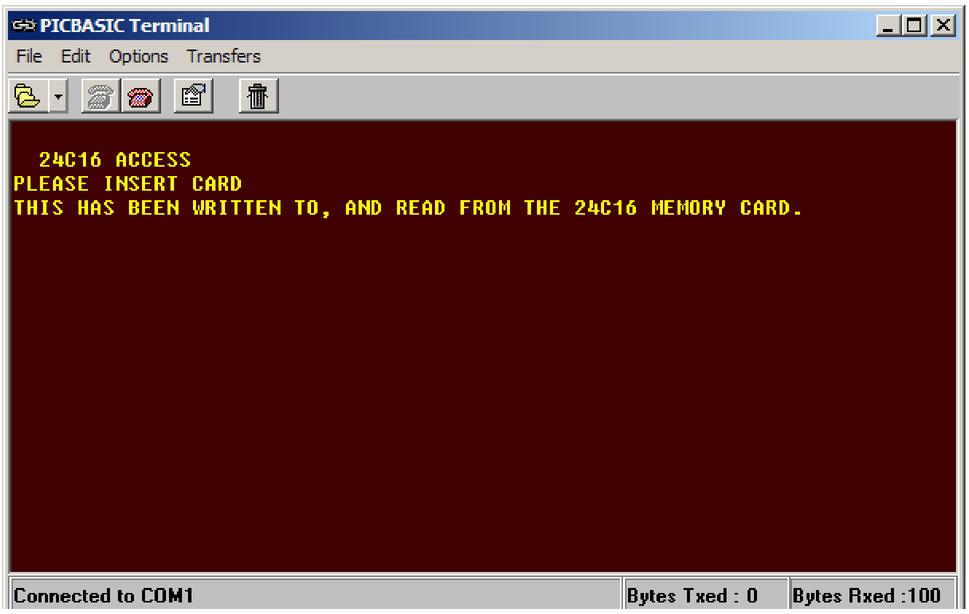


The footprint pattern may be different on the card supplied with the PROTON SMART, but the functionality and positioning of the contacts remains the same.

Now that we have that out of the way, we can look at the code for accessing the 24C16 memory card. Load the program **MC_24C16.BAS**, and compile it. You might have noticed the similarity to the previous 14C02 program. This is because both devices require similar coding. With the differences for the address handling in 3 subroutines, shown below.

```
'-------------------------------------------------------------------------------------
' Calculate the address required, and adjust the CONTROL byte for the card
' By moving bits 8 to 10 of the 16-bit ADDRESS, to bits 1 to 3 of CONTROL
CALCULATE_ADDRESS:
CONTROL = %10100000 | ((ADDRESS.HIGHBYTE << 1) & %00001110)
Return
'-------------------------------------------------------------------------------------
' Read a BYTE from the card
' From address, held in the variable ADDRESS
' Returns the BYTE in variable DATA_IN
READ_BYTE:
Gosub CALCULATE_ADDRESS                        ' Arrange the address for the 24C16
Busin CONTROL,ADDRESS.LOWBYTE,[DATA_IN]        ' Read a byte of data from the card
Return
'-------------------------------------------------------------------------------------
' Write a BYTE to the card
' The BYTE to send must be loaded into variable DATA_OUT
' At address, held in the variable ADDRESS
WRITE_BYTE:
Gosub CALCULATE_ADDRESS                        ' Arrange the address for the 24C16
Busout CONTROL,ADDRESS.LOWBYTE,[DATA_OUT]     ' Write a byte of data to the card
Delayms 5                                      ' Delay 5ms after each write
Return
```

Download the program to the PROTON SMART board, and open the Serial Terminal window (discussed earlier). After pressing RESET on the PROTON SMART board, and inserting a card. You should be greeted with: -



```
  24C16 ACCESS
PLEASE INSERT CARD
THIS HAS BEEN WRITTEN TO, AND READ FROM THE 24C16 MEMORY CARD.
```

Connected to COM1 | Bytes Txed : 0 | Bytes Rxed :100

Interfacing a 24C256 Memory Card.

The 24C256 I^{2}C memory card, has 128 times as much memory as the 14C02, in that it contains 256kbits which relates to 32kbytes of storage capacity. And has a retention time of 40 years. Accessing the card is again, primarily the same as accessing the 14C02, but this card accepts a 16-bit ADDRESS. This allows all 32768 bytes of memory to be accessed without having to manipulate the CONTROL byte. The contact footprint for this card is the same as the 24C16.

Load the program **MC_24C256.BAS**, and you will see that there is very little difference between it and the **MC_14C02.BAS** program. The only real differences are that the ADDRESS variable is now defined as a WORD (16-bit) type, and the WC pin is left floating to enable writing to a card that has this pin, just in case the card used requires this pin.

There's not much more that can be said for the code, other than compile it, then download it to the PROTON SMART board, and view the results on the Serial Terminal window.

Something to think about.

We have only scratched the surface of accessing these cards, there are other ways of writing to the cards that have not been discussed. These are called PAGE writes, and allow up to 64 bytes of data to be written to the card in a single operation. The PAGE size differs from card to card, but the information is well documented in the datasheets supplied on the CDROM. I'll leave the coding of page writes up to you, but I will give you some pointers as to what commands and operators to use.

Start by creating a byte array to hold the page size required.
Use the STR directive to send and receive data from the card.
You must calculate the next address based on the page size written or read, and the current address.

Good luck, and I look forward to seeing any further code you produce on the forum. **www.picbasic.org/forum**

Accessing the SLE4442 smart card.

The Siemens SLE4442 Intelligent eeprom card, is well on it's way to becoming what is commonly thought of as a smartcard. It has protected memory, secure memory, and a PIN (personal identification number). It also has an internal counter that renders the card invalid if the PIN is not entered correctly within a set number of attempts (three to be exact).

Operation of the SLE4442 card.

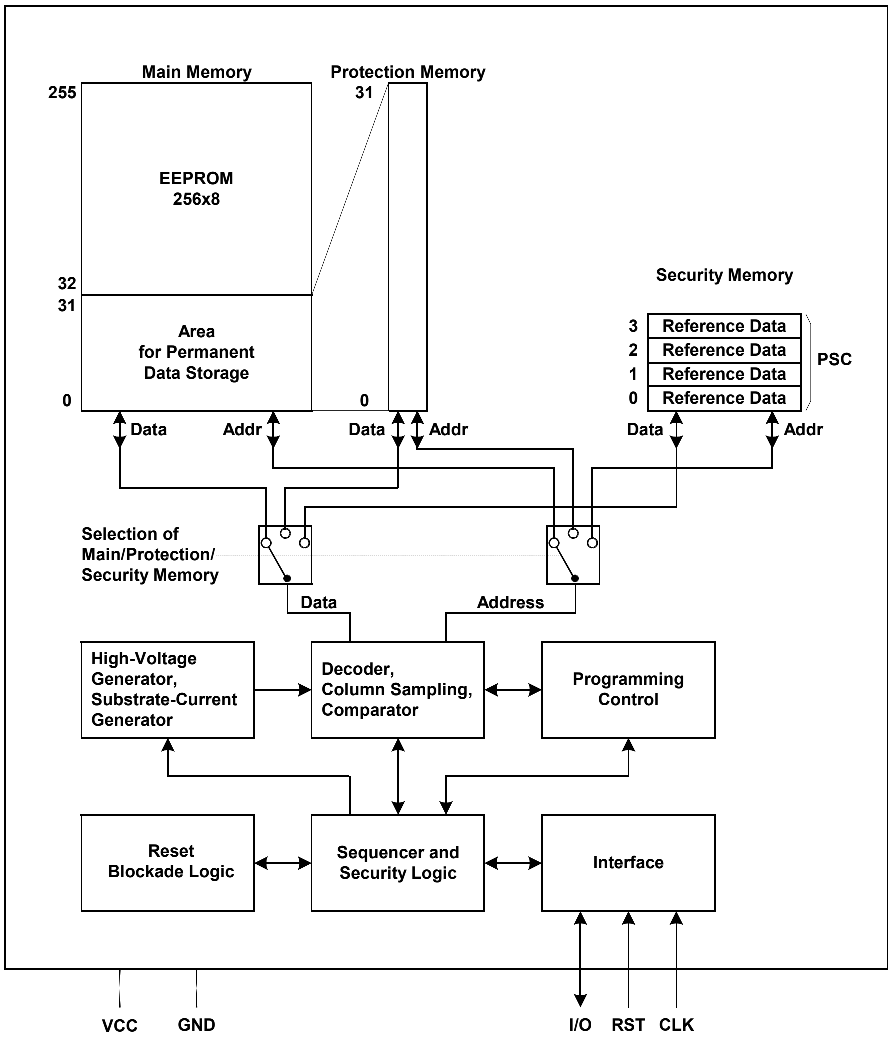
Because of the more complex nature of this card, you will need to understand its internal operations and requirements before any code is written. More so than the previous I^2C memory cards, because the I^2C bus is a well documented, and readily understood protocol. So bear with me, because this does get rather involved. You may need to read the following section a few times before an understanding is reached, but it is worth persevering.

The SLE4442 consists of 256 bytes of eeprom main memory, and a 32-bit (4 byte) protection memory with PROM functionality. The main memory is erased and written on a byte to byte basis. When erased, all 8-bits of a data cell are set to logic one (hex FF). When written, the information in the individual eeprom cells is, according to the input data, altered bit by bit to logical zeros (logical AND between the old and the new data in the eeprom). Normally, a data change consists of an erase and write procedure. It depends on the contents of the data byte already contained in the main memory and the new data byte as to whether the eeprom is really erased and/or written. If none of the 8 bits in the addressed byte require a zero-to-one transition, the erase access will be suppressed. Vice versa the write access will be suppressed if no one-to-zero transition is necessary. The write and erase operations take a minimum of 2.5ms each.

Each of the first 32 bytes of memory can be irreversibly protected against data change by writing the corresponding bit in the protection memory. Each data byte in this address range is assigned to one bit of the protection memory, and has the same address as the data byte in the main memory which it's assigned to. Once written, the protection bit cannot be erased, this is the PROM functionality of the card.

The SLE4442 also includes security code logic, which controls write/erase access to the memory. For this purpose, the SLE4442 contains a 4 byte security memory area with an Error Counter EC (bit-0 to bit-2), and 3 bytes of reference data. These 3 bytes as a whole are called Programmable Security Code (PSC).

After power on, the whole memory, except for the reference data can only be read. Only after a successful comparison of the PIN, can the memory be erased, or written. After three UNSUCCESSFUL comparisons, the Error Counter blocks any subsequent attempts, and hence any possibility to write or erase, which renders the card virtually useless. A block diagram illustrating the internal operations of the SLE4442 is shown overleaf.



SLE4442 Intelligent eeprom card, internal diagram.

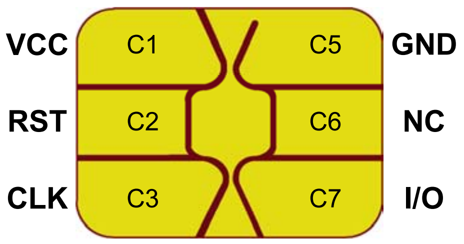
The protocol used by the SLE4442 is a two wire interface, loosely based on an I^{2}C model, in that it has a START, and STOP condition. All data changes on the I/O are initiated by the falling edge of the clock line.

The transmission protocol consists of 4 modes: -

- Reset, and Answer-to-Reset.
- Command Mode.
- Outgoing Data Mode.
- Processing Mode.

It's should be pointed out at this stage that the I/O pin is open drain, and therefore requires a pull-up resistor to achieve a high level. This is already incorporated in the PROTON SMART board, but is important to remember when a stand-alone application is created.

The contact footprint for the SLE4442 card is an M3 type, shown below.



Reset and Answer-to-Reset (ATR).

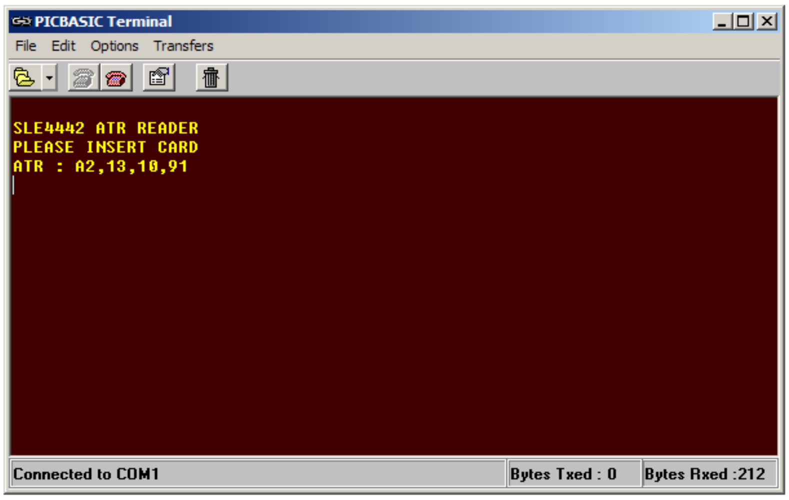
Answer-to-Reset occurs as soon as the card is powered up, and a reset is implemented. ATR is described in ISO7816 part 3. The reset can be given at any time during operation. When the card is reset, the address counter is cleared to zero together with a clock pulse, and the first data bit (LSB) is output to I/O then the RST line is pulled from logic HIGH to logic LOW. Under a continuous input of additional 31 clock pulses, the contents of the first 4 eeprom addresses is read out. The 33rd clock pulse switches I/O to high impedance, and finishes the ATR procedure.

The format of the 4 bytes of ATR sent by the card are shown below.

Byte 1	Byte 2	Byte 3	Byte 4
$D0_7 \ldots D0_0$	$D0_{15} \ldots D0_8$	$D0_{23} \ldots D0_{16}$	$D0_{31} \ldots D0_{24}$

The 4 byte response from an Answer-to-Reset is set when the card is manufactured, and each card has a different sequence. The meaning of each byte is somewhat long winded, and follows the ISO7816 part 4 standard. More information is contained in the SLE4442 datasheet on the CDROM. The important information for us is that it should contain the hex values A2, 13, 10, and 91.

Before we go any further, we'll look at a program that reads the ATR from the SLE4442 card. Load the program **READ_ATR.BAS** located in the SLE4442 folder. Compile and download it to the PROTON SMART board. Open the serial terminal, and press RESET on the board, and insert an SLE4442 card. You should be greeted with the display below: -



```
SLE4442 ATR READER
PLEASE INSERT CARD
ATR : A2,13,10,91
```

Connected to COM1   Bytes Txed : 0   Bytes Rxed :212

The program is based around two key subroutines, SEND_RESET, and READ_ATR. As explained earlier, a reset is accomplished by pulling the RST line from high to low, while pulsing the clock line. This is shown below: -

```
' Send a RESET condition by:-
' Pulling the RST line HIGH-LOW while the CLK line is toggled HIGH-LOW
SEND_RESET:
High RST                    ' Bring the RESET line HIGH (to RESET the card)
High CLK                    ' Bring the CLOCK line HIGH
Low CLK                     ' Pull the CLOCK line LOW
Low RST                     ' Pull the RESET line LOW (to release the card from RESET)
Return
```

The subroutine to actually read the ATR is shown below: -

```
' Read the 32 bit ATR (ANSWER-TO-RESET)
' The DWORD variable ATR_MEM holds the 32 bit ATR result
READ_ATR:
Clear ERROR_CODE
 If CARD_IN = 0 Then ERROR_CODE = 1 : Low CARD_VCC : Return   ' Indicate if card removed
Gosub SEND_RESET                                              ' Send a RESET
Shiftin
SIO,CLK,LSBPRE,[ATR_MEM.BYTE3,ATR_MEM.BYTE2,ATR_MEM.BYTE1,ATR_MEM.BYTE0]
Return
```

The **SHIFTIN** command reads each byte of the 4-byte response, into a DWORD (32-bit) variable. The subroutine is also responsible for checking if the card is inserted into the socket before it attempts a read. If not, then it disables the VCC to the socket, and indicates an error (held in the variable ERROR_CODE). This is a theme that will be implemented in the rest of the demonstration programs.

The full program for reading the ATR is shown below: -

```
' Program READ_ATR.BAS
' Read the ATR (ANSWER-TO-RESET) from a SIEMENS SLE4442 Smartcard.

DEVICE = 16F876                   ' PICmicro used in the PROTON SMART
XTAL = 20                         ' We're using a 20MHZ crystal
HSERIAL_BAUD = 9600               ' Set baud rate to 9600
HSERIAL_RCSTA = %10010000         ' Enable serial port and continuous receive
HSERIAL_TXSTA = %00100100         ' Enable transmit and asynchronous mode
HSERIAL_CLEAR = ON                ' Enable Error clearing on received characters
'-------------------------------------------------------------------------------
' Declare some variables
Dim ERROR_CODE  as BIT            ' Returns NON-ZERO if an error occurs in any process
Dim ATR_MEM as DWORD              ' Holds the 4 bytes of ATR data
'-------------------------------------------------------------------------------
' Define some aliases to make the code more readable
Symbol CARD_VCC = PORTA.5         ' Supplies the card with 5 Volts
Symbol CARD_IN = PORTA.4          ' CARD in sensor switch (normally closed)
Symbol CLK = PORTC.3              ' Card's CLK line
Symbol SIO = PORTC.4                    ' Card's IO line
Symbol RST = PORTA.3                    ' Card's RESET line
Delayms 500                             ' Wait for the power supply to fully stabilise
Hrsout 1                                ' Clear the serial terminal's screen before we start
Goto MAIN                               ' Then jump over the subroutines to the main program
loop
'-----------------------------------------------------------------------------------
' Send a RESET condition by:-
' Pulling the RST line HIGH-LOW
' While the CLK line is toggled HIGH-LOW
SEND_RESET:
High RST                                ' Bring the RESET line HIGH (to RESET the card)
High CLK                                ' Bring the CLOCK line HIGH
Low CLK                                 ' Pull the CLOCK line LOW
Low RST                                 ' Pull the RESET line LOW (to release the card from RE-
SET)
Return
'-----------------------------------------------------------------------------------
' Read the 32 bit ATR (ANSWER-TO-RESET)
' The DWORD variable ATR_MEM holds the 32 bit ATR result
READ_ATR:
Clear ERROR_CODE
If CARD_IN = 0 Then ERROR_CODE = 1 : Low CARD_VCC : Return    ' Indicate if card removed
Gosub SEND_RESET                                              ' Send a RESET
Shiftin
SIO,CLK,LSBPRE,[ATR_MEM.BYTE3,ATR_MEM.BYTE2,ATR_MEM.BYTE1,ATR_MEM.BYTE0]
Return
'-----------------------------------------------------------------------------------
' Wait for the card to be inserted into the socket before continuing
WAIT_FOR_INSERTION:
Low CARD_VCC                                        ' Disable the card's VCC (5 Volts)
Hrsout 13,"SLE4442 ATR READER",13,"PLEASE INSERT CARD",13
While CARD_IN = 0 : Wend                            ' Wait for card insertion
High CARD_VCC                                       ' Enable the card's VCC (5 Volts)
Delayms 100                                         ' Wait for the card to fully power up
Return
'-----------------------------------------------------------------------------------
' *** MAIN PROGRAM LOOP STARTS HERE ***
MAIN:
ALL_DIGITAL = True                        ' Make PORTA all digital IO
Input CARD_IN                             ' Make the card sensor pin an input

CARD_LOOP:
Gosub WAIT_FOR_INSERTION                  ' Wait for the card to be inserted into the socket

Gosub READ_ATR                            ' Read the ATR (ANSWER-TO-RESET) of the
card
If ERROR_CODE != 0 Then CARD_LOOP         ' Check for card insertion error

' Display the ATR value on the serial terminal
Hrsout "ATR : ",HEX2 ATR_MEM.BYTE3,",",HEX2 ATR_MEM.BYTE2,",",HEX2 ATR_MEM.BYTE1
,",",HEX2 ATR_MEM.BYTE0,13

While CARD_IN = 1 : Delayms 10 : Wend     ' Wait for the card to be removed before proceed-
ing
Goto CARD_LOOP
```

Operational Modes.

After the Answer-to-Reset, the card waits for a command. Every command begins with a Start Condition, includes a 3 byte command entry, followed by an additional clock pulse, and ends with a Stop Condition.

A Start Condition consists of a Falling edge on I/O during CLK high.

```
Input SIO                    ' Keep the DATA line HIGH
Delayus 1                    ' Wait for 1 microsecond (us)
High CLK                     ' Bring the CLOCK line HIGH
Delayus 1                    ' Wait for 1 microsecond (us)
Low SIO                      ' Pull the DATA line LOW
```

A Stop Condition consists of a Rising edge on I/O during CLK high.

```
Low SIO                      ' Keep the DATA line LOW
Delayus 1                    ' Wait for 1 microsecond (us)
High CLK                     ' Bring the CLOCK line HIGH
Delayus 1                    ' Wait for 1 microsecond (us)
Input SIO                    ' Bring the DATA line HIGH
```

After the reception of a command, there are two possible modes: -

- Outgoing data mode for Reading.
- Processing mode for Writing, or Erasing.

Outgoing Data Mode.

In this mode, the card sends data to the PROTON SMART board. The first bit becomes valid on I/O after the first falling edge on CLK. After the last data bit, an additional clock pulse is necessary in order to set the I/O to high impedance, and to prepare the card for a new command. During this mode, any Start or Stop condition is ignored.

Processing Mode.

In this mode, the card processes internally. The card has to be clocked continuously until I/O (which was switched to high after the falling edge of CLK) , is set to high impedance. Any Start and Stop condition is discarded during this mode.

Commands.

Each command consists of three bytes: -

MSB			Control				LSB
B7	B6	B5	B4	B3	B2	B1	B0

MSB			Address				LSB
A7	A6	A5	A4	A3	A2	A1	A0

MSB			Data				LSB
D7	D6	D5	D4	D3	D2	D1	D0

The table below shows the effect of each command byte, and its requirements for address, and data.

Byte 1 CONTROL								Byte 2 ADDRESS	Byte 3 DATA	OPERATION	MODE
B7	B6	B5	B4	B3	B2	B1	B0	A7 - A0	D7 - D0		
0	0	1	1	0	0	0	0	Address	No Effect	READ MAIN MEMORY	Outgoing Data
0	0	1	1	1	0	0	0	Address	Input Data	UPDATE MAIN MEMORY	Processing
0	0	1	1	0	1	0	0	No Effect	No Effect	READ PROTECTION MEMORY	Outgoing Data
0	0	1	1	1	1	0	0	Address	Input Data	WRITE PROTECTION MEMORY	Processing
0	0	1	1	0	0	0	1	No Effect	No Effect	READ SECURITY MEMORY	Outgoing Data
0	0	1	1	1	0	0	1	Address	Input Data	UPDATE SECURITY MEMORY	Processing
0	0	1	1	0	0	1	1	Address	Input Data	COMPARE VERIFICATION DATA	Processing

Sending a command to the SLE4442 is simplicity itself, as the subroutine below illustrates: -

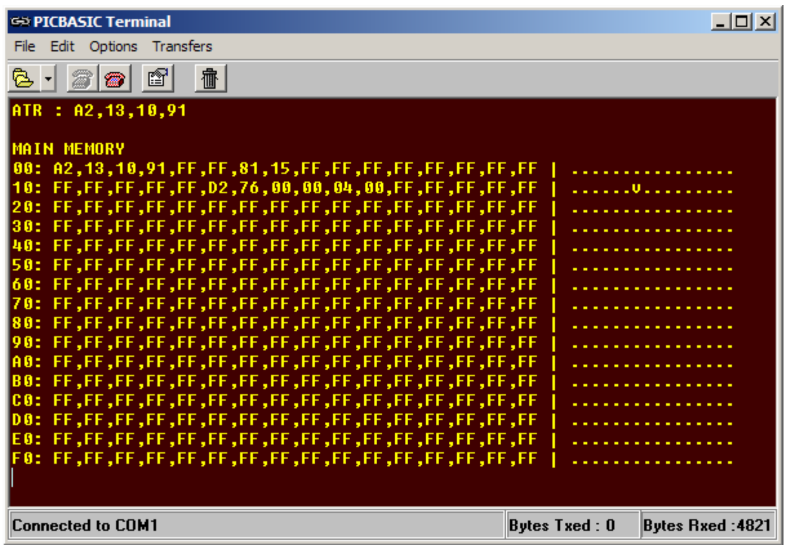
```
' Send a command to the card
' The command to send is held in variable COMMAND
' The address (if any) is held in variable ADDRESS
' The data to send (if any) is held in variable DATA_BYTE
SEND_COMMAND:
ERROR_CODE = 0                                              ' Default to no errors
If CARD_IN = 0 Then ERROR_CODE = 1 : Low CARD_VCC : Return  ' Indicate an error if card
removed
Gosub SEND_START                                            ' Send a START condition
Shiftout SIO , CLK , LSBFIRST , [COMMAND,ADDRESS,DATA_BYTE] ' Shift out the command
byte
Gosub SEND_STOP                                             'Send a STOP condition
Return
```

The **SEND_COMMAND** subroutine (above), uses the subroutines we looked at earlier namely, **SEND_START**, and **SEND_STOP**. It also indicates if the card has been removed, and disables the card's VCC if so.

The **SEND_COMMAND** subroutine is of paramount importance, as no access to or from the card is possible without calling it first.

Reading MAIN Memory.

We'll now look at a program for reading all 255 bytes of MAIN MEMORY from the SLE4442 card. Load and compile program **READ_MAIN.BAS**, then download it to the PROTON SMART. After opening the serial terminal window, and inserting a card into the socket, you should be greeted with the following display.



```
ATR : A2,13,10,91

MAIN MEMORY
00: A2,13,10,91,FF,FF,81,15,FF,FF,FF,FF,FF,FF,FF,FF | ................
10: FF,FF,FF,FF,FF,D2,76,00,00,04,00,FF,FF,FF,FF,FF | ......v.........
20: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
30: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
40: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
50: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
60: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
70: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
80: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
90: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
A0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
B0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
C0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
D0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
E0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
F0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
```

Connected to COM1 | Bytes Txed : 0 | Bytes Rxed :4821

The full program for reading the MAIN MEMORY is shown below.

```
' Program READ_MAIN.BAS
' Read, and display the MAIN MEMORY from a SIEMENS SLE4442 Smartcard.

DEVICE = 16F876                          ' PICmicro used in the PROTON SMART
XTAL = 20                                ' We're using a 20MHZ crystal
HSERIAL_BAUD = 9600                      ' Set baud rate to 9600
HSERIAL_RCSTA = %10010000                ' Enable serial port and continuous receive
HSERIAL_TXSTA = %00100100                ' Enable transmit and asynchronous mode
HSERIAL_CLEAR = ON                       ' Enable Error clearing on received characters
'-------------------------------------------------------------------------------------
' Declare some variables

Dim CLEARED_ONCE as Bit             ' Toggle flag for clearing the serial terminal only once in a
loop
Dim ERROR_CODE  as Bit                   ' Returns NON-ZERO if an error occurs in any
process
Dim LOOP          as Byte
Dim LOOP2         as Byte
Dim TEMP          as Byte
Dim COMMAND       as Byte                ' Command to send to the card
Dim ADDRESS       as Byte                ' The card address to read and write
Dim DATA_BYTE     as Byte                ' Data byte used for some commands
Dim ATR_MEM       as Dword               ' Holds the 4 bytes of ATR data
Dim MEMORY[255]   as Byte                ' 255 bytes of MAIN memory
'-------------------------------------------------------------------------------------
' Define some aliases to make the code more readable

Symbol CARD_VCC = PORTA.5           ' Supplies the card with 5 Volts
Symbol CARD_IN = PORTA.4            ' CARD in sensor switch (normally closed)
Symbol CLK = PORTC.3                ' Card's CLK line
Symbol SIO = PORTC.4                ' Card's IO line
Symbol RST = PORTA.3                ' Card's RESET line
' Card commands
Symbol READ_MAIN_MEM = %00110000
```

```
'-------------------------------------------------------------------------------------
Delayms 500                                   ' Wait for the power supply to fully stabilise
ALL_DIGITAL = True                            ' Make PORTA all digital IO
Hrsout 1                                      ' Clear the serial terminal's screen before we start
Goto Main                                     ' Then jump over the subroutines to the main program
loop
'-------------------------------------------------------------------------------------
' Send a START condition by:-
' Falling Edge of IO line, while CLK is HIGH
Send_Start:
Input SIO                                     ' Keep the DATA line HIGH
Delayus 1                                     ' Wait for 1 microsecond (us)
High CLK                                      ' Bring the CLOCK line HIGH
Delayus 1                                     ' Wait for 1 microsecond (us)
Low SIO                                       ' Pull the DATA line LOW
Return
'-------------------------------------------------------------------------------------
' Send a STOP condition by:-
' Rising Edge of IO line, while CLK is HIGH
Send_Stop:
Low SIO                                       ' Keep the DATA line LOW
Delayus 1                                     ' Wait for 1 microsecond (us)
High CLK                                      ' Bring the CLOCK line HIGH
Delayus 1                                     ' Wait for 1 microsecond (us)
Input SIO                                     ' Bring the DATA line HIGH
Return
'-------------------------------------------------------------------------------------
' Send a RESET condition by:-
' Pulling the RST line HIGH-LOW
' While the CLK line is toggled HIGH-LOW
SEND_RESET:
High RST                                      ' Bring the RESET line HIGH (to RESET the card)
High CLK                                      ' Bring the CLOCK line HIGH
Low CLK                                       ' Pull the CLOCK line LOW
Low RST                                       ' Pull the RESET line LOW (to release the card from RE-
SET)
Return
'-------------------------------------------------------------------------------------
' Send a command to the card
' The command to send is held in variable COMMAND
' The address (if any) is held in variable ADDRESS
' The data to send (if any) is held in variable DATA_BYTE
SEND_COMMAND:
ERROR_CODE = 0                                                 ' Default to no error
If CARD_IN = 0 Then ERROR_CODE = 1 : Low CARD_VCC : Return     ' Indicate an error if card
removed
Gosub SEND_START                                               ' Send a START condition
Shiftout SIO , CLK , LSBFIRST , [COMMAND,ADDRESS,DATA_BYTE]    ' Shift out the command
byte
Gosub SEND_STOP                                                ' Send a STOP condition
Return
```

```
'-------------------------------------------------------------------------------------
' READ MAIN MEMORY
' Reads all of memory into array MEMORY
' Reads from address 0 to 255
READ_MAIN_MEMORY:
ERROR_CODE = 0                               ' Default to no error
ADDRESS = 0                                  ' Start at address 00
COMMAND = READ_MAIN_MEM                      ' Set up for a READ MAIN MEMORY
command
Gosub SEND_COMMAND                           ' Send the READ MAIN MEMORY com-
mand
Gosub SEND_START                             ' Send a START condition
LOOP = 0                                     ' Clear the loop variable before we start
Repeat                                       ' Create a loop
If CARD_IN = 0 Then ERROR_CODE = 1 : Low CARD_VCC : Return    ' Indicate an error if card
removed
MEMORY[LOOP] = Shiftin SIO , CLK , LSBPRE , 8
Inc LOOP                                     ' Point to the next data byte
Until LOOP = 0                               ' 0 is byte sized 256
High CLK : Delayus 1 : Low CLK               ' Send an extra clock
Gosub SEND_STOP                              ' Send a STOP condition
Return
'-------------------------------------------------------------------------------------
' Read the 32 bit ATR (ANSWER-TO-RESET)
' The DWORD variable ATR_MEM holds the 32 bit ATR result
READ_ATR:
Clear ERROR_CODE
If CARD_IN = 0 Then ERROR_CODE = 1 : Low CARD_VCC : Return    ' Indicate an error if card
removed
Gosub SEND_RESET                                              ' Send a RESET
Shiftin
SIO,CLK,LSBPRE,[ATR_MEM.BYTE3,ATR_MEM.BYTE2,ATR_MEM.BYTE1,ATR_MEM.BYTE0]
High CLK : Delayus 1 : Low CLK                                ' Send an extra clock
Return
'-------------------------------------------------------------------------------------
' Wait for the card to be inserted into the socket before continuing
WAIT_FOR_INSERTION:
While CARD_IN = 0                       ' Wait for the card to be inserted into the socket
Low CARD_VCC
If CLEARED_ONCE = 0 Then                ' Make sure CLS is carried out only once in the
loop
Hrsout 1,"  SLE4442 READER",13,"PLEASE INSERT CARD",13
Set CLEARED_ONCE
Endif
Wend
High CARD_VCC                           ' Enable the card's VCC (5 Volts)
If CLEARED_ONCE = 1 Then Hrsout 1 : CLEARED_ONCE = 0
Delayms 200                             ' Wait for the card to fully power up
Return
'-------------------------------------------------------------------------------------
' Check if the card is the right type, and is inserted correctly
' i.e. not upside down, or back to front
CHECK_CARD_TYPE:
ERROR_CODE = 0
' Check if the card in the socket correctly. Will return all $FF if not
If ATR_MEM.BYTE3 == $FF AND ATR_MEM.BYTE2 == $FF Then
Hrsout 1,"CARD INSERTION ERROR",13
```

```
While CARD_IN = 1 : Delayms 10 : Wend           ' Wait for the card to be removed before proceed-
ing
ERROR_CODE = 1                                  ' Indicate an error
Return
Endif
' Check if the correct type of card is inserted
' By examining the first byte of the ATR
If ATR_MEM.BYTE3 != $A2 Then                    ' Is the first byte equal to $A2 ?
Hrsout 1,"INVALID CARD TYPE",13
While CARD_IN = 1 : Delayms 10 : Wend           ' Wait for the card to be removed before proceed-
ing
ERROR_CODE = 1                                  ' Indicate an error
Endif
Return
'-----------------------------------------------------------------------------------------
' Display the MAIN MEMORY via RS232
DISPLAY_MAIN_MEMORY:
Hrsout "MAIN MEMORY",13
For LOOP = 0 to 255 Step 16
Hrsout HEX2 LOOP , ": "
For LOOP2 = 0 to 15                             ' Display HEX
Hrsout HEX2 MEMORY[LOOP  + LOOP2]
If LOOP2 < 15 Then Hrsout ","
Next
Hrsout " | "
For Loop2 = 0 to 15                             ' Display ASCII
TEMP = MEMORY[LOOP + LOOP2]
If TEMP > 32 AND TEMP < 127 Then
Hrsout TEMP
Else
Hrsout "."
Endif
Next
Hrsout 13
Next
Return
'-----------------------------------------------------------------------------------------
' *** MAIN PROGRAM LOOP STARTS HERE ***
MAIN:
Clear                                           ' Clear all user RAM
CLEARED_ONCE = 0
Input CARD_IN
Delayms 50                                      ' A small delay
Gosub WAIT_FOR_INSERTION                        ' Wait for the card to be inserted into the socket
Gosub Read_ATR                                  ' Read the ATR (ANSWER-TO-RESET) of the
card
If ERROR_CODE != 0 Then MAIN                    ' Check for card insertion error
Gosub CHECK_CARD_TYPE
If ERROR_CODE != 0 Then MAIN

' Display the ATR value on the serial terminal
Hrsout "ATR : ",HEX2 ATR_MEM.BYTE3,",",HEX2 ATR_MEM.BYTE2,",",HEX2
ATR_MEM.BYTE1,",",HEX2 ATR_MEM.BYTE0,13,13

Gosub READ_MAIN_MEMORY                          ' Read all of MAIN memory into an array
If ERROR_CODE != 0 Then MAIN                    ' Check for card insertion error
Gosub DISPLAY_MAIN_MEMORY
If ERROR_CODE != 0 Then MAIN
```

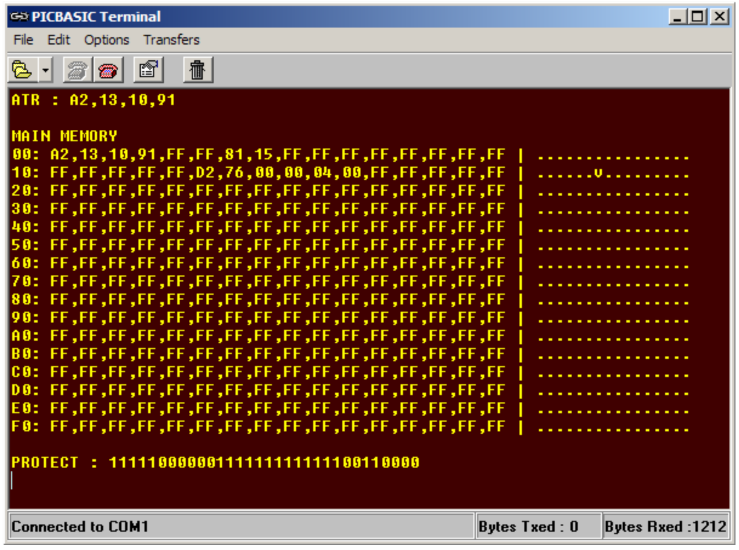
```
While CARD_IN = 1 : Delayms 10 : Wend          ' Wait for the card to be removed before proceed-
ing
Goto MAIN
```

To read a byte from the SLE card, we must send a READ_MAIN_MEMORY command (hex 30) using the **SEND_COMMAND** subroutine, along with the starting address we require (0 in the program above). After the command entry, the program has to supply sufficient clock pulses for the amount of bytes required. The number of clock pulses is (256 – ADDRESS) x 8 + 1. The data is read out LSB (least significant bit) first. To read the main memory, a PIN is not required.

Reading PROTECTION Memory.

The first 32 bytes of MAIN memory can be set to READ ONLY by use of a 32-bit block of memory known as PROTECTION MEMORY. Each bit in this block of memory corresponds to the read/write attribute of the first 32 bytes of MAIN memory. Setting a bit to ZERO in PROTECTION MEMORY will mean that the relevant byte in MAIN memory is READ ONLY.

To illustrate more clearly this principal, load and compile the program **READ_PROT.BAS**. As usual, download the program to the PROTON SMART, then open the serial terminal window, insert the card, and view the results. You should see the display to the right.



```
ATR : A2,13,10,91

MAIN MEMORY
00: A2,13,10,91,FF,FF,81,15,FF,FF,FF,FF,FF,FF,FF,FF | ................
10: FF,FF,FF,FF,FF,D2,76,00,00,04,00,FF,FF,FF,FF,FF | ......v.........
20: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
30: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
40: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
50: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
60: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
70: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
80: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
90: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
A0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
B0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
C0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
D0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
E0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
F0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................

PROTECT : 11111000000111111111111100110000

Connected to COM1    Bytes Txed : 0    Bytes Rxed :1212
```

The program is fundamentally the same as **READ_MAIN.BAS**, except that the PROTECTION memory is now read and displayed, along with the MAIN memory. The relevant parts of the display are:

```
MAIN MEMORY
00: A2,13,10,91,FF,FF,81,15,FF,FF,FF,FF,FF,FF,FF,FF |
................
10: FF,FF,FF,FF,FF,D2,76,00,00,04,00,FF,FF,FF,FF,FF |
......v.........

PROTECT : 11111000000111111111111100110000
```

As you can see by the data pre-programmed my the manufacturer, some bytes are already write protected. Namely, bits 0 to 3, bits 6 and 7, bits 21 to 26. Examining the binary string of PROTECT, you will see that these bits are all set to ZERO, indicating that they are read only.

It must be noted at this point that the PROTECTION memory area cannot be written to directly, as will be demonstrated in later programs.

To read the PROTECTION memory, the program uses the subroutine READ_PROT_MEMORY, shown below.

```
' READ PROTECTION MEMORY
' Reads the 4-bytes of memory into variable P_MEM
' ADDRESS, and DATA_BYTE are ignored by the card, so do not need initialising
READ_PROT_MEMORY:
COMMAND = READ_PROT_MEM
Gosub SEND_COMMAND                ' Send the READ PROTECTED MEMORY com-
mand
If ERROR_CODE != 0 Then Return
Gosub SEND_START                  ' Send a START condition
If CARD_IN = 0 Then ERROR_CODE = 1 : Low CARD_VCC : Return ' Indicate error ?
Shiftin SIO , CLK , MSBPRE ,
[P_MEM.BYTE3,P_MEM.BYTE2,P_MEM.BYTE1,P_MEM.BYTE0]
High CLK : Delayus 1 : Low CLK    ' Send an extra clock
Gosub SEND_STOP                   ' Send a STOP condition
Return
```

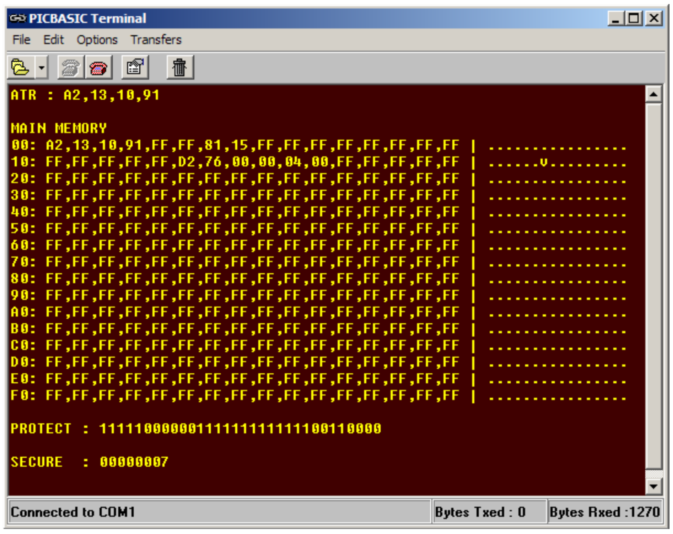
The command READ_PROT_MEM (hex 34) transfers the protection bits under a continuous input of 32 clock pulses. The reading of PROTECTION memory does not require a PIN to be entered, but writing to it does.

Before we look at writing to the card, there is one more piece of memory that we need to read. This is the SECURITY MEMORY, which holds the PIN, and PIN attempts counter.

Reading SECURITY Memory.

The SECURITY memory is of paramount importance when the card is in everyday use, as it helps inhibit illegal writes to the card, unless a valid PIN is entered. Reading the SECURITY memory will not reveal the PIN stored in the card, until a valid PIN is entered first. But it will show the amount of attempts left if an invalid PIN is entered. SECURITY memory consists of 4 bytes, the first 3 bytes hold the PIN (not visible at this point), while the last byte holds the attempts counter.

Load and compile the program **READ_SECURE.BAS**. Then follow the standard procedures of downloading to the PROTON SMART, opening the serial terminal, pressing RESET, and inserting a card. You should be greeted with the following display.



```
ATR : A2,13,10,91

MAIN MEMORY
00: A2,13,10,91,FF,FF,81,15,FF,FF,FF,FF,FF,FF,FF,FF | ................
10: FF,FF,FF,FF,FF,D2,76,00,00,04,00,FF,FF,FF,FF,FF | ......v.........
20: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
30: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
40: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
50: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
60: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
70: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
80: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
90: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
A0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
B0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
C0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
D0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
E0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
F0: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................

PROTECT  : 11111000000111111111111100110000

SECURE   : 00000007
```

Connected to COM1 | Bytes Txed : 0 | Bytes Rxed :1270

Reading the SECURITY memory is almost exactly the same as reading PROTECTION memory, with the exception of a different command, and receptor variable. The subroutine is shown below.

```
' READ SECURITY MEMORY
' Reads the 4-bytes of memory into variable S_MEM
' ADDRESS, and DATA_BYTE are ignored by the card, so do not need initialising
READ_SECURE_MEMORY:
COMMAND = READ_SECURE_MEM
Gosub SEND_COMMAND                        ' Send the READ SECURITY MEMORY com-
mand
If ERROR_CODE != 0 Then Return                        ' Was there an error ?
Gosub SEND_START                                      ' Send a START condition
If CARD_IN = 0 Then ERROR_CODE = 1 : Low CARD_VCC : Return    ' Indicate an error if card
removed   Shiftin SIO , CLK , LSBPRE ,
[S_MEM.BYTE0,S_MEM.BYTE1,S_MEM.BYTE2,S_MEM.BYTE3]
Gosub SEND_STOP                                       ' Send a STOP condition
Return
```

The rest of the program is essentially the same as the previous one, but now reads and displays the SECURITY memory of the card.

The 4 bytes of SECURITY memory look like: -

PIN Byte 2	PIN Byte 1	PIN Byte 0	Error Counter
Byte 3	Byte 2	Byte 1	Byte 0
$D0_{31}$ ... $D0_{24}$	$D0_{23}$ ... $D0_{16}$	$D0_{15}$ ... $D0_{8}$	$D0_{7}$ ... $D0_{0}$

SECURITY MEMORY

Byte 0 of the SECURITY memory is the ERROR COUNTER. This is decremented in the event of an incorrect PIN being entered, thus indicating the amount of attempts left.

ERROR COUNTER

0	0	0	0	0	D2	D1	D0

The next 3 bytes hold the PIN, which will read as 0's until a valid value is entered.

We can now read all areas of the card, but we cannot write to it yet. For this we need a different set of subroutines, but we still require the card reading subroutines. We'll now look at writing to the various memory locations of the card, and also entering a valid PIN.

Open Wide Please.

Before we can write or erase the SLE4442 card, we must enter a valid 3 byte PIN (Personal Identification Number). This will open all areas of the memory for writing, except those areas that have been set to read only by the PROTECTION memory. When the card is first supplied, it contains a PIN set to hex FFFFFF. This makes it easy to work with for now, and we can change the PIN later.

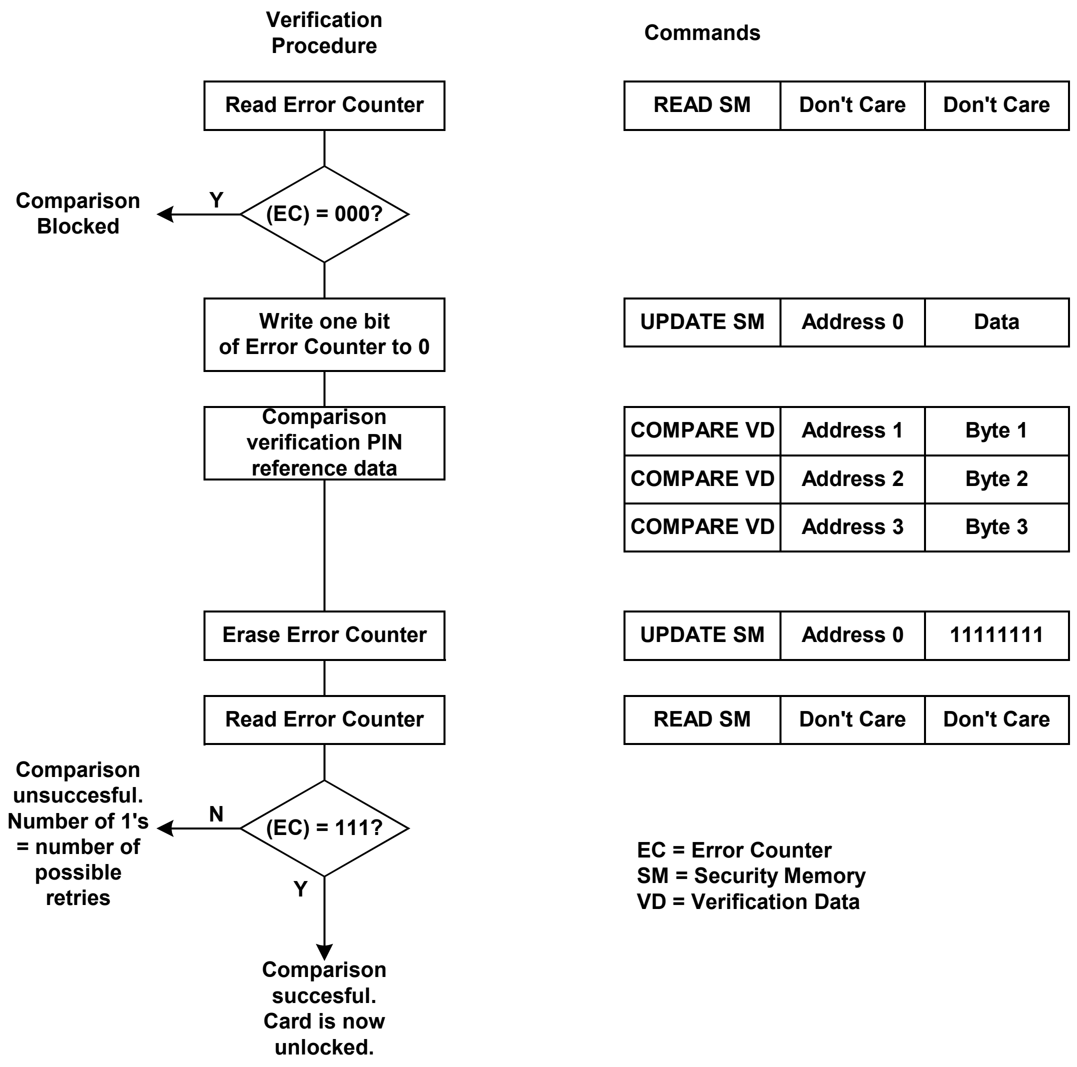
Opening the card for writing, must follow a set sequence of events, with the correct PIN. The following procedure has to be carried out exactly as described. Any variation will leads to a failure, so that a write/erase access will not be achieved.

At first, the ERROR COUNTER bit has to be written to 0 by an UPDATE command (see the flowchart overleaf) followed by three COMPARE VERIFICATION DATA commands beginning with byte 1 of the reference data (PIN bytes). A successful conclusion of the whole procedure can be recognised by being able to erase the error counter, which is not automatically erased. Now write/erase to all areas of memory is possible as long as the card's VCC is applied. This also applies to the 3 PIN bytes, which allows them to be changed. In case of error, the whole procedure can be repeated, as long as the ERROR COUNTER byte contains 1's. The ERROR COUNTER byte starts off with the first three bits set to 1. Each invalid PIN entry, decrements one bit.

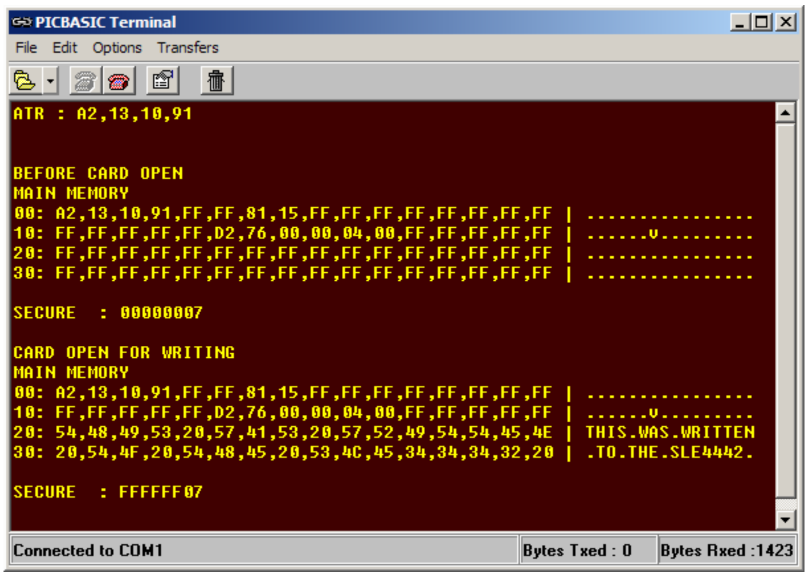
The following table gives an overview of the necessary commands for the PIN entry, and memory opening.

Command	Control	Address	Data	Remarks
	B7…B0	A7…A0	D7…D0	
Read Security Memory	31_H	No Effect	No Effect	Check Error Counter
Update Security Memory	39_H	00_H	Input Data	Write free bit in Error Counter 0000 0ddd binary
Compare Verification Data	33_H	01_H	Input Data	PIN byte 1
Compare Verification Data	33_H	02_H	Input Data	PIN byte 2
Compare Verification Data	33_H	03_H	Input Data	PIN byte 3
Update Security Memory	33_H	00_H	FF_H	Erase Error Counter
Read Security Memory	31_H	No Effect	No Effect	Check Error Counter

The card remembers the ERROR COUNTER even if power is removed. And will only be reset when a valid PIN is entered successfully.


Verification Procedure
Commands
Read Error Counter
READ SM | Don't Care | Don't Care
Comparison Blocked
Y
(EC) = 000?
Write one bit of Error Counter to 0
UPDATE SM | Address 0 | Data
Comparison verification PIN reference data
COMPARE VD | Address 1 | Byte 1
COMPARE VD | Address 2 | Byte 2
COMPARE VD | Address 3 | Byte 3
Erase Error Counter
UPDATE SM | Address 0 | 11111111
Read Error Counter
READ SM | Don't Care | Don't Care
Comparison unsuccesful. Number of 1's = number of possible retries
N
(EC) = 111?
Y
Comparison succesful. Card is now unlocked.
EC = Error Counter
SM = Security Memory
VD = Verification Data

To demonstrate the above procedure, load the program **OPEN_MEMORY.BAS**, and compile it. Download it to the PROTON SMART board, but do NOT insert a card until the serial terminal is opened, and RESET is pressed. After inserting the card you should be greeted with the display shown below: -



```
ATR : A2,13,10,91


BEFORE CARD OPEN
MAIN MEMORY
00: A2,13,10,91,FF,FF,81,15,FF,FF,FF,FF,FF,FF,FF,FF | ................
10: FF,FF,FF,FF,FF,D2,76,00,00,04,00,FF,FF,FF,FF,FF | ......v.........
20: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................
30: FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF | ................

SECURE  : 00000007

CARD OPEN FOR WRITING
MAIN MEMORY
00: A2,13,10,91,FF,FF,81,15,FF,FF,FF,FF,FF,FF,FF,FF | ................
10: FF,FF,FF,FF,FF,D2,76,00,00,04,00,FF,FF,FF,FF,FF | ......v.........
20: 54,48,49,53,20,57,41,53,20,57,52,49,54,54,45,4E | THIS.WAS.WRITTEN
30: 20,54,4F,20,54,48,45,20,53,4C,45,34,34,34,32,20 | .TO.THE.SLE4442.

SECURE  : FFFFFF07
```

Connected to COM1 | Bytes Txed : 0 | Bytes Rxed :1423

What the program is doing is reading the MAIN memory, and displaying only 64 bytes of it. Then opening the memory, and writing a small sentence to address 32 onwards. Then reading and displaying the results. It also displays the SECURITY memory, before, and after the card is opened.

Opening the memory is accomplished by the subroutine **OPEN_MEMORY**, shown below.

```
' OPEN_MEMORY for write
' Opens the card for writing by entering the correct PIN number
' The PIN is loaded into a DWORD sized variable named PIN
' SECURITY MEMORY byte 0 must be decremented before the card is unlocked
' The CARD remains unlocked as long as VCC power is applied
OPEN_MEMORY:
Gosub READ_SECURE_MEMORY                ' Read all of SECURE memory
If ERROR_CODE != 0 Then Return          ' Return if an error occurred during the
READ_SECURE_MEMORY process
If S_MEM.BYTE0 = 0 Then                 ' Indicate NO ATTEMPTS left
ERROR_CODE = 2                          ' Indicate we have an error opening the card
Delayms 50
Hrsout 1,1
Delayms 50
Hrsout 13,"THE CARD HAS BEEN DE-ACTIVATED!",13
Return                                  ' Return from the subroutine prematurely
Endif

' Update SECURE memory, at address 0
COMMAND = UPDATE_SECURE_MEM             ' Setup the command to UPDATE SECURITY
ADDRESS = 0                             ' Point to SECURE memory address 0
DATA_BYTE = S_MEM.BYTE0 - 1        ' Decrement the attempts counter. Required before
open
Gosub SEND_COMMAND                 ' Send the command to UPDATE SECURITY MEM-
ORY
If ERROR_CODE != 0 Then Return     ' Return if an error occurred during the process
Gosub START_WRITE_PROCESS          ' Start of PROCESSING CONDITION.

' Send PIN byte 1.
COMMAND = COMPARE_SECURE_MEM            ' Setup the command for COMPARE SECU-
RITY
Inc ADDRESS                             ' ADDRESS now points to $01
DATA_BYTE = PIN.Byte0                   ' Load the FIRST PIN number
Gosub SEND_COMMAND                 ' Send the COMPARE SECURITY MEMORY com-
mand
If ERROR_CODE != 0 Then Return     ' Return if an error occurred during the process
Gosub START_WRITE_PROCESS          ' Start of PROCESSING CONDITION.

' Send PIN byte 2.
Inc ADDRESS                        ' ADDRESS now points to $02
DATA_BYTE = PIN.Byte1              ' Load the SECOND PIN number
Gosub SEND_COMMAND                 ' Send the COMPARE SECURITY MEMORY com-
mand
```

```
If ERROR_CODE != 0 Then Return      ' Return if an error occurred during the process
Gosub START_WRITE_PROCESS           ' Start of PROCESSING CONDITION.
' Send PIN byte 3.
Inc ADDRESS                         ' ADDRESS now points to $03
DATA_BYTE = PIN.Byte2               ' Load the THIRD PIN number
Gosub SEND_COMMAND                  ' Send the COMPARE SECURITY MEMORY com-
mand
If ERROR_CODE != 0 Then Return      ' Return if an error occurred during the process
Gosub START_WRITE_PROCESS           ' Start of PROCESSING CONDITION.

' Try and erase the ERROR COUNTER
ADDRESS = $00                       ' Point to SECURE memory address 0
DATA_BYTE = $FF                     ' Write $FF to address of SECURITY memory
COMMAND = UPDATE_SECURE_MEM ' Setup the command to UPDATE SECURITY
MEMORY
Gosub SEND_COMMAND                  ' Send the UPDATE SECURITY MEMORY com-
mand
If ERROR_CODE != 0 Then Return      ' Return if an error occurred during the process
Gosub START_WRITE_PROCESS           ' Start of PROCESSING CONDITION.
Gosub READ_SECURE_MEMORY            ' Read the SECURITY memory again

If S_MEM.BYTE0 <> 7 Then            ' Has the ERROR COUNTER been decremented ?
ERROR_CODE = 2                      ' Indicate we have an error opening the card
Delayms 50
Hrsout 1,1
Delayms 50
Hrsout "WRONG PIN ENTERED",13
Endif
If S_MEM.BYTE0 = 6 Then
Hrsout "WARNING! ONLY TWO ATTEMPTS LEFT AT ENTERING THE CORRECT PIN",13
Endif
If S_MEM.BYTE0 = 4 Then
Hrsout "WARNING! ONLY ONE ATTEMPT LEFT AT ENTERING THE CORRECT PIN",13
Endif
Return
```

Notice on the display, that the SECURITY memory shows 00000007 before the card is opened, and FFFFFF07 after the card is opened. The FFFFFF bytes correspond to the PIN. The 07 corresponds to the amount of attempts left, which is seven (bin 00000111) in this case because the correct PIN was entered.

Now we'll see what happens when a wrong PIN is entered.

WARNING. Incorrect usage of the following changes in the program could render the card as invalid. Read and understand the next few paragraphs before attempting any code changes.

Change line **345** of the **OPEN_MEMORY.BAS** program from: -

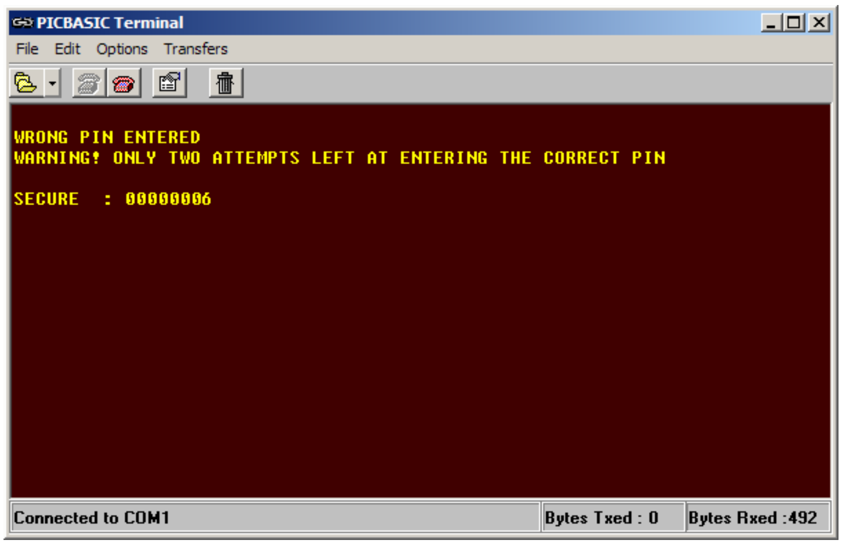
```
PIN = $FFFFFF                       ' Set the PIN number to default $FFFFFF
```

to: -

```
PIN = $FFFF00                                  ' Set the PIN number to $FFFF00
```

Make sure there is **NO** card in the PROTON SMART board, and re-compile the program, then download it. Open the serial terminal, and RESET the board.

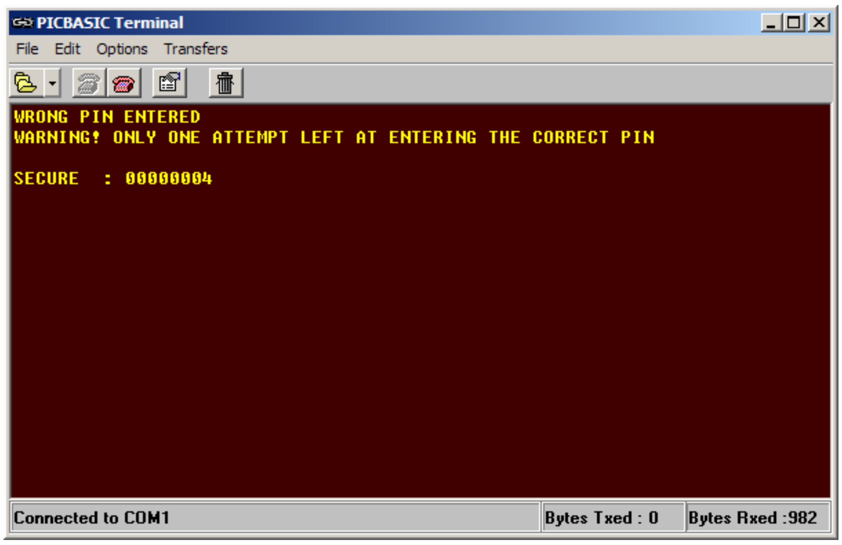
Now insert the card into the PROTON SMART's socket, and you will be greeted with the following display: -



```
WRONG PIN ENTERED
WARNING! ONLY TWO ATTEMPTS LEFT AT ENTERING THE CORRECT PIN

SECURE   : 00000006
```

```
Connected to COM1        Bytes Txed : 0    Bytes Rxed :492
```

Notice that the SECURITY memory still only displays 000000 because an incorrect PIN has been entered, but the ERROR COUNTER has been decremented by one bit (from 00000111 to 00000110). Remove the card from its socket and re-insert it. **WARNING**. This will decrement the ERROR COUNTER once again. If you do not feel brave enough to attempt this, then skip this part. Upon re-entering the card, you will be greeted (if that's the correct terminology) with the display below.



```
WRONG PIN ENTERED
WARNING! ONLY ONE ATTEMPT LEFT AT ENTERING THE CORRECT PIN

SECURE   : 00000004
```

```
Connected to COM1        Bytes Txed : 0    Bytes Rxed :982
```

Notice, that the ERROR counter has decremented once again because of the invalid PIN (from 00000110 to 00000100). NOW REMOVE THE CARD, AND **DO NOT** RE-INSERT IT until changes have been made to the program.

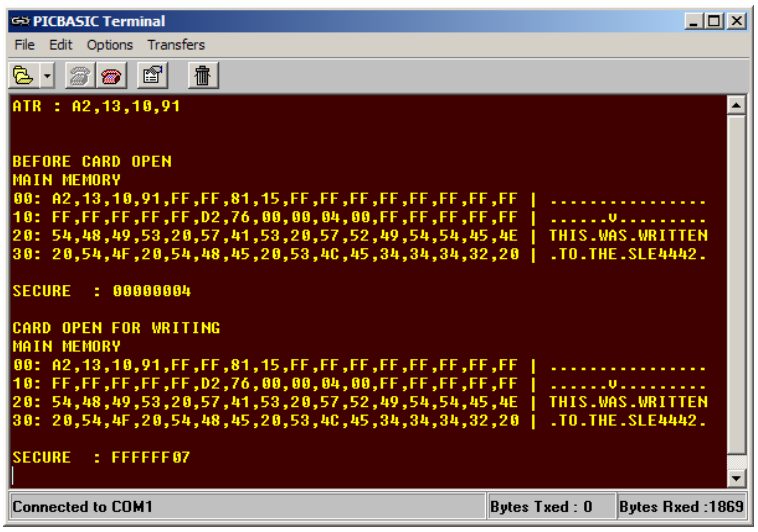
We now need to reset the ERROR COUNTER in order for the card not to be invalidated. Change line **345** of the **OPEN_MEMORY.BAS** program from: -

```
PIN = $FFFF00                    ' Set the PIN number to $FFFF00
```

to: -

```
PIN = $FFFFFF                    ' Set the PIN number to default $FFFFFF
```

Re-compile the program, and download it to the PROTON SMART board. Open the serial terminal, and press RESET. Only now can the card be re-inserted into the socket. This will enter the correct PIN value of FFFFFF, thus resetting the ERROR COUNTER to 07. See display below: -



```
ATR : A2,13,10,91


BEFORE CARD OPEN
MAIN MEMORY
00: A2,13,10,91,FF,FF,81,15,FF,FF,FF,FF,FF,FF,FF,FF | ................
10: FF,FF,FF,FF,FF,D2,76,00,00,04,00,FF,FF,FF,FF,FF | ......v.........
20: 54,48,49,53,20,57,41,53,20,57,52,49,54,54,45,4E | THIS.WAS.WRITTEN
30: 20,54,4F,20,54,48,45,20,53,4C,45,34,34,34,32,20 | .TO.THE.SLE4442.

SECURE  : 00000004

CARD OPEN FOR WRITING
MAIN MEMORY
00: A2,13,10,91,FF,FF,81,15,FF,FF,FF,FF,FF,FF,FF,FF | ................
10: FF,FF,FF,FF,FF,D2,76,00,00,04,00,FF,FF,FF,FF,FF | ......v.........
20: 54,48,49,53,20,57,41,53,20,57,52,49,54,54,45,4E | THIS.WAS.WRITTEN
30: 20,54,4F,20,54,48,45,20,53,4C,45,34,34,34,32,20 | .TO.THE.SLE4442.

SECURE  : FFFFFF07

Connected to COM1            Bytes Txed : 0     Bytes Rxed :1869
```

Notice that before the card was opened, the SECURITY memory still read as 00000004, but after the card was presented with a valid PIN, the SECURITY memory reads FFFFFF07. We're safe now, the card has 3 attempts again.

Writing to MAIN memory.

The program **OPEN_MEMORY.BAS**, also writes to the MAIN memory of the card using the subroutine WRITE_MAIN_MEMORY. Shown below.

```
' WRITE MAIN MEMORY
WRITE_MAIN_MEMORY:
RESTORE 0                        ' Point to beginning of DATA statement
COMMAND = UPDATE_MAIN_MEM        ' Setup the correct command
ADDRESS = 32                     ' Write from address 32 of the card
Repeat                           ' Create a loop
DATA_BYTE = READ                 ' Read the data from the DATA state-
ment
Gosub SEND_COMMAND               ' Write the data to the card
```

```
If ERROR_CODE != 0 Then Return                    ' Was there an error ?
Gosub START_ERASE_WRITE_PROCESS                   ' Start of PROCESSING CONDITION.
Inc ADDRESS                                       ' Point to the next address location within the
card
Until DATA_BYTE = 0 OR LOOP = 0                   ' Loop until finished
Return
```

The WRITE_MAIN_MEMORY subroutine uses the command UPDATE MAIN memory, and another subroutine named START_ERASE_WRITE_PROCESS. So we'll look at the UPDATE MAIN memory command first.

This command programs the addressed eeprom byte with the data byte transmitted. Depending on the old and new data, one of the following sequences will take place during the processing mode: -

Erase and Write	(5ms)	Corresponding to m = 255 clock pulses.
Write without Erase	(2.5ms)	Corresponding to m = 124 clock pulses.
Erase without Write	(2.5ms)	Corresponding to m = 124 clock pulses.

The subroutine START_ERASE_WRITE_PROCESS, performs the task that its name suggests, in that it starts the process for erasing a memory cell before data is written to it. The subroutine is shown below: -

```
' START of ERASE_AND_WRITE PROCESSING CONDITION.
' Falling Edge of IO line, while CLK is LOW
START_ERASE_WRITE_PROCESS:
Input SIO                                   ' Keep the DATA line HIGH (via pullup resis-
tor)
Low CLK                                     ' Bring the CLOCK line HIGH
Low SIO                                     ' Pull the DATA line LOW
Clear LOOP2
Repeat
High CLK : Delayus 1 : Low CLK              ' Send 255 dummy clocks
Inc LOOP2
Until LOOP2 = 255
Input SIO                                   ' Bring the DATA line HIGH
Return
```

The subroutine START_WRITE_PROCESS, allows two tasks to be performed, Write without Erase, or Erase without Write, depending on what is placed in the variable DATA_BYTE prior to sending the UPDATE_MAIN_MEM command. If hex FF is loaded into DATA_BYTE, then an Erase without Write will be performed, any other value will perform Write without Erase. This subroutine is shown below: -

```
' START of WRITE PROCESSING CONDITION.
' Falling Edge of IO line, while CLK is LOW
START_WRITE_PROCESS:
Input SIO                                   ' Keep the DATA line HIGH (via pullup resis-
tor)
```

```
Low CLK                                ' Bring the CLOCK line HIGH
Low SIO                                ' Pull the DATA line LOW
Clear LOOP2
Repeat
High CLK : Delayus 1 : Low CLK         ' Send 124 dummy clocks
Inc LOOP2
Until LOOP2 = 124
Input SIO                              ' Bring the DATA line HIGH
Return
```

These two subroutines are the key to writing data to the SLE4442 card, and are used throughout the rest of the programs. The program **OPEN_MEMORY.BAS** already demonstrates writing to the card, so we need not create another demonstration program for this. But instead, we'll take a look at write protecting parts of the MAIN memory.

As was discussed earlier, the first 32 bytes of MAIN memory can be write protected, which effectively creates a PROM area, as opposed to EEPROM. To accomplish write protection we must issue the WRITE PROTECTION MEMORY command (hex 3C), with the address required (0 to 31), and the data byte containing the value that is already written to the memory cell at that address.

For example, suppose we wish to write protect address 4 of MAIN memory. The sequence of events is: -

- Perform a conventional WRITE to address 4, using the UPDATE MAIN MEMORY command.
- Read the value from address 4.
- Re-write this value into address 4, using the WRITE PROTECTION MEMORY command, instead of the UPDATE MAIN MEMORY command.

If the MAIN memory cell is to be write protected along with being written, then step two of the above list may be omitted, as the value is already known.

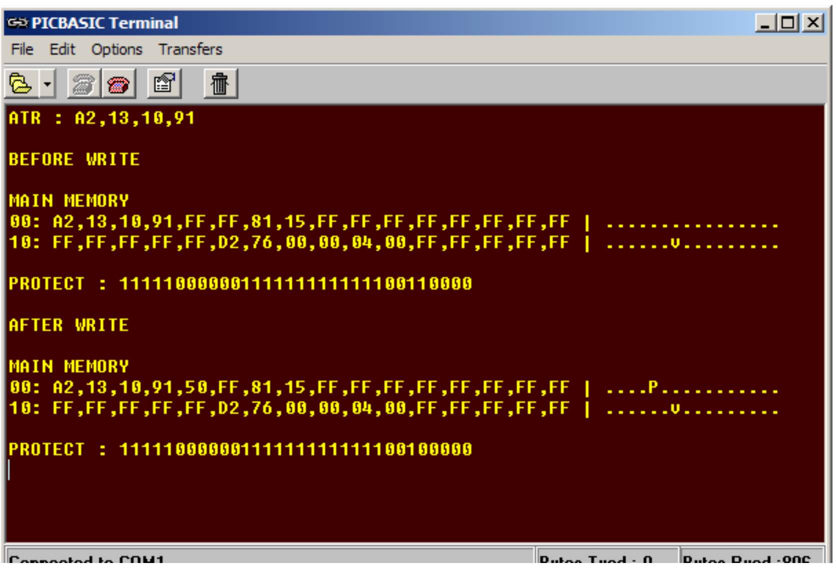
This breaks down into the following BASIC code: -

```
' Write the value $50 (ASCII P) to address 4 of the card
COMMAND = UPDATE_MAIN_MEM                  ' Setup the correct command
ADDRESS = 4                                ' Write from address 4 of the card
DATA_BYTE = $50                            ' Setup for a value of $50 to be written
Gosub SEND_COMMAND                         ' Write the data to the card
If ERROR_CODE != 0 Then MAIN               ' Was there an error ?
Gosub START_ERASE_WRITE_PROCESS            ' Start of PROCESSING CONDITION.

' Now WRITE Protect the value at address 4
' By loading variable ADDRESS with the memory address required for write protect (0 - 31)
' And placing the same value ($50) into DATA_BYTE
' These are already pre-loaded by the code above.
COMMAND = WRITE_PROT_MEM          ' Setup to write to protected memory
Gosub SEND_COMMAND                ' Send the WRITE PROTECTION MEMORY com-
mand
If ERROR_CODE != 0 Then MAIN       ' Was there an error ?
Gosub START_WRITE_PROCESS         ' Start of WRITE PROCESSING CONDITION.
```

A program to illustrate write protection is included on the CDROM, named **WRITE_PROT.BAS**. Load the program, and follow the standard compilation and downloading, but DO NOT insert the card yet.

Once, the serial terminal has been opened, and RESET has been pressed, you can insert the card. **WARNING! THIS WILL WRITE AND PROTECT A SINGLE CELL IN MAIN MEMORY. THIS CELL WILL NOT BE ABLE TO BE ERASED OR REWRITTEN WITH ANOTHER VALUE**.

When the card is inserted, you will be greeted with the display below: -



```
ATR : A2,13,10,91

BEFORE WRITE

MAIN MEMORY
00: A2,13,10,91,FF,FF,81,15,FF,FF,FF,FF,FF,FF,FF,FF | ................
10: FF,FF,FF,FF,FF,D2,76,00,00,04,00,FF,FF,FF,FF,FF | ......v.........

PROTECT : 11111000000111111111111100110000

AFTER WRITE

MAIN MEMORY
00: A2,13,10,91,50,FF,81,15,FF,FF,FF,FF,FF,FF,FF,FF | ....P...........
10: FF,FF,FF,FF,FF,D2,76,00,00,04,00,FF,FF,FF,FF,FF | ......v.........

PROTECT : 11111000000111111111111100100000
```

What the program does is reads the card and displays the MAIN memory, and the PROTECTION memory. Then writes ASCII P to address 4 and write protects it.

As you can see, the PROTECTION memory before the write is: -

```
PROTECT : 11111000000111111111111100110000
```

But after the write, and write protection, the PROTECTION memory is: -

```
PROTECT : 11111000000111111111111100100000
```

Bit 4 of the PROTECTION memory is now set to 0, which indicates that address 4 (containing ASCII P), is now write protected, and cannot be erased, or written to, even with a valid PIN.

The full code listing of **WRITE_PROT.BAS:**

```
' Program WRITE_PROT.BAS
' Opens the SLE4442 memory for writing or erasing, by entering a valid PIN.
' Then writes information to MAIN memory, and locks a cell to read only.

' WARNING! THIS CODE HAS THE POTENTIAL TO RENDER A CARD USELESS IF AL-
TERED

DEVICE = 16F876                         ' PICmicro used in the PROTON SMART
XTAL = 20                               ' We're using a 20MHZ crystal

HSERIAL_BAUD = 9600                     ' Set baud rate to 9600
HSERIAL_RCSTA = %10010000               ' Enable serial port and continuous receive
HSERIAL_TXSTA = %00100100               ' Enable transmit and asynchronous mode
HSERIAL_CLEAR = ON                      ' Enable Error clearing on received characters
'-------------------------------------------------------------------------------------
' Declare some variables
Dim CLEARED_ONCE        as Bit          ' Toggle flag for clearing the terminal only once in a
loop
Dim ERROR_CODE          as Byte         ' Returns NON-ZERO if an error occurs in any proc-
ess
Dim LOOP                as Byte
Dim LOOP2               as Byte
Dim TEMP                as Byte
Dim COMMAND             as Byte         ' Command to send to the card
Dim ADDRESS             as Byte         ' The card address to read and write
Dim DATA_BYTE           as Byte         ' Data byte used for some commands
Dim P_MEM               as Dword        ' 32 bits (4 bytes) of PROTECTION memory
Dim ATR_MEM             as Dword        ' Holds the 4 bytes of ATR data
Dim PIN                 as Dword        ' Holds the 3 byte PIN
Dim S_MEM               as Dword        ' Holds the 4 bytes of SECURITY memory
Dim MEMORY[255]         as Byte         ' 255 bytes of MAIN memory
'-------------------------------------------------------------------------------------
' Define some aliases to make the code more readable

Symbol CARD_VCC = PORTA.5               ' Supplies the card with 5 Volts
Symbol CARD_IN = PORTA.4                ' CARD IN sensor switch (normally closed)
Symbol CLK = PORTC.3                    ' Card's CLK line
Symbol SIO = PORTC.4                    ' Card's IO line
Symbol RST = PORTA.3                    ' Card's RESET line

' Card commands
Symbol READ_MAIN_MEM = %00110000
Symbol UPDATE_MAIN_MEM = %00111000
Symbol READ_PROT_MEM = %00110100
Symbol WRITE_PROT_MEM = %00111100
Symbol READ_SECURE_MEM = %00110001
Symbol UPDATE_SECURE_MEM = %00111001
Symbol COMPARE_SECURE_MEM = %00110011
'-------------------------------------------------------------------------------------
Delayms 500                             ' Wait for the power supply to fully stabilise
ALL_DIGITAL = True                      ' Make PORTA all digital IO
```

```
Hrsout 1                            ' Clear the serial terminal's screen before we start
Goto Main                           ' Then jump over the subroutines to the main pro-
gram loop

'-----------------------------------------------------------------------------------
' Send a START condition by:-
' Falling Edge of IO line, while CLK is HIGH
Send_Start:
Input SIO                           ' Keep the DATA line HIGH
Delayus 1                           ' Wait for 1 microsecond (us)
High CLK                            ' Bring the CLOCK line HIGH
Delayus 1                           ' Wait for 1 microsecond (us)
Low SIO                             ' Pull the DATA line LOW
Return
'-----------------------------------------------------------------------------------
' Send a STOP condition by:-
' Rising Edge of IO line, while CLK is HIGH
Send_Stop:
Low SIO                             ' Keep the DATA line LOW
Delayus 1                           ' Wait for 1 microsecond (us)
High CLK                            ' Bring the CLOCK line HIGH
Delayus 1                           ' Wait for 1 microsecond (us)
Input SIO                           ' Bring the DATA line HIGH
Return
'-----------------------------------------------------------------------------------
' Send a RESET condition by:-
' Pulling the RST line HIGH-LOW
' While the CLK line is toggled HIGH-LOW
SEND_RESET:
High RST                        ' Bring the RESET line HIGH (to RESET the card)
High CLK                        ' Bring the CLOCK line HIGH
Low CLK                         ' Pull the CLOCK line LOW
Low RST                         ' Pull the RESET line LOW (to release the card from RE-
SET)
Return
'-----------------------------------------------------------------------------------
' START of WRITE PROCESSING CONDITION.
' Falling Edge of IO line, while CLK is LOW
' Send 125 clock pulses
START_WRITE_PROCESS:
Input SIO                           ' Keep the DATA line HIGH (via pullup resistor)
Low CLK                             ' Bring the CLOCK line HIGH
Low SIO                             ' Pull the DATA line LOW
Clear LOOP2
Repeat
High CLK : Delayus 1 : Low CLK      ' Send 124 dummy clocks
Inc LOOP2
Until LOOP2 = 125
Input SIO                           ' Bring the DATA line HIGH
Return
'-----------------------------------------------------------------------------------
' START of ERASE_AND_WRITE PROCESSING CONDITION.
' Falling Edge of IO line, while CLK is LOW
```

```
START_ERASE_WRITE_PROCESS:
Input SIO                              ' Keep the DATA line HIGH (via pullup resistor)
Low CLK                                ' Bring the CLOCK line HIGH
Low SIO                                ' Pull the DATA line LOW
Clear LOOP2
Repeat
High CLK : Delayus 1 : Low CLK         ' Send 255 dummy clocks
Inc LOOP2
Until LOOP2 = 0
Input SIO                              ' Bring the DATA line HIGH
Return
'-------------------------------------------------------------------------------------
' Send a command to the card
' The command to send is held in variable COMMAND
' The address (if any) is held in variable ADDRESS
' The data to send (if any) is held in variable DATA_BYTE
SEND_COMMAND:
ERROR_CODE = 0                                     ' Default to no error
If CARD_IN = 0 Then ERROR_CODE = 1 : Return        ' Indicate an error if card removed
Gosub SEND_START                                   ' Send a START condition
Shiftout SIO , CLK , LSBFIRST , [COMMAND,ADDRESS,DATA_BYTE] ' Shift out the com-
mand
Gosub SEND_STOP                                    ' Send a STOP condition
Return
'-------------------------------------------------------------------------------------
' READ SECURITY MEMORY
' Reads the 4-bytes of memory into variable S_MEM
' ADDRESS, and DATA_BYTE are ignored by the card, so do not need initialising
READ_SECURE_MEMORY:
COMMAND = READ_SECURE_MEM
Gosub SEND_COMMAND                            ' Send the READ SECURITY MEMORY
command
If ERROR_CODE != 0 Then Return                ' Was there an error ?
Gosub SEND_START                              ' Send a START condition
If CARD_IN = 0 Then ERROR_CODE = 1 : Return ' Indicate an error if card removed
Shiftin SIO , CLK , LSBPRE ,
[S_MEM.BYTE0,S_MEM.BYTE1,S_MEM.BYTE2,S_MEM.BYTE3]
High CLK : Delayus 1 : Low CLK                ' Send an extra clock
Gosub SEND_STOP                               ' Send a STOP condition
Return
'-------------------------------------------------------------------------------------
' READ PROTECTION MEMORY
' Reads the 4-bytes of memory into variable P_MEM
' ADDRESS, and DATA_BYTE are ignored by the card, so do not need initialising
READ_PROT_MEMORY:
COMMAND = READ_PROT_MEM
Gosub SEND_COMMAND                     ' Send the READ PROTECTED MEMORY com-
mand
If ERROR_CODE != 0 Then Return
Gosub SEND_START                       ' Send a START condition
If CARD_IN = 0 Then ERROR_CODE = 1 : Return   ' Indicate an error if card removed
Shiftin SIO , CLK , LSBPRE ,
[P_MEM.BYTE0,P_MEM.BYTE1,P_MEM.BYTE2,P_MEM.BYTE3]
```

```
High CLK : Delayus 1 : Low CLK         ' Send an extra clock
Gosub SEND_STOP                        ' Send a STOP condition
Return
'-----------------------------------------------------------------------------------
' READ MAIN MEMORY
' Reads all of memory into array MEMORY
' Reads from address 0 to 255
READ_MAIN_MEMORY:
ADDRESS = 0
COMMAND = READ_MAIN_MEM
Gosub SEND_COMMAND                          ' Send the READ MAIN MEMORY command
If ERROR_CODE != 0 Then Return
Gosub SEND_START                            ' Send a START condition
LOOP = 0
Repeat
If CARD_IN = 0 Then ERROR_CODE = 1 : Return    ' Indicate an error if card removed
MEMORY[LOOP] = Shiftin SIO , CLK , LSBPRE , 8
Inc LOOP
Until LOOP = 0                              ' 0 is byte sized 256
High CLK : Delayus 1 : Low CLK              ' Send an extra clock
Gosub SEND_STOP                             ' Send a STOP condition
Return
'-----------------------------------------------------------------------------------
' Read the 32 bit ATR (ANSWER-TO-RESET)
' The DWORD variable ATR_MEM holds the 32 bit ATR result
READ_ATR:
ERROR_CODE = 0                              ' Default to no error
If CARD_IN = 0 Then ERROR_CODE = 1 : Return    ' Indicate an error if card removed
Gosub SEND_RESET                            ' Send a RESET
Shiftin
SIO,CLK,LSBPRE,[ATR_MEM.BYTE3,ATR_MEM.BYTE2,ATR_MEM.BYTE1,ATR_MEM.BYTE0]
High CLK : Delayus 1 : Low CLK              ' Send an extra clock
Return
'-----------------------------------------------------------------------------------
' OPEN_MEMORY for write
' Opens the card for writing by entering the correct PIN number
' The PIN is loaded into a DWORD sized variable named PIN
' SECURITY MEMORY byte 0 must be decremented before the card is unlocked
' The CARD remains unlocked as long as VCC power is applied
OPEN_MEMORY:
Gosub READ_SECURE_MEMORY               ' Read all of SECURE memory into variable S_MEM
If ERROR_CODE != 0 Then Return              ' Return if an error occurred during the proc-
ess
If S_MEM.BYTE0 = 0 Then                     ' Indicate NO ATTEMPTS left
ERROR_CODE = 2                              ' Indicate we have an error opening the card
Delayms 50
Hrsout 1,1
Delayms 50
Hrsout 13,"THE CARD HAS BEEN DE-ACTIVATED!",13
Return                                      ' Return form the subroutine prematurely
Endif

' Update SECURE memory, at address 0
```

```
COMMAND = UPDATE_SECURE_MEM ' Setup the command to UPDATE SECURITY
MEMORY
ADDRESS = 0                         ' Point to SECURE memory address 0
DATA_BYTE = S_MEM.BYTE0 - 1       ' Decrement the attempts counter. Required before
open
Gosub SEND_COMMAND                ' Send the command to UPDATE SECURITY MEM-
ORY
If ERROR_CODE != 0 Then Return    ' Return if an error occurred during the process
Gosub START_WRITE_PROCESS         ' Start of PROCESSING CONDITION.

' Start sending the PIN number. $FF $FF $FF in this case
' Send PIN byte 1.
COMMAND = COMPARE_SECURE_MEM ' Setup command COMPARE SECURITY MEM-
ORY
Inc ADDRESS                         ' ADDRESS now points to $01
DATA_BYTE = PIN.Byte0               ' Load the FIRST PIN number
Gosub SEND_COMMAND                ' Send the COMPARE SECURITY MEMORY com-
mand
If ERROR_CODE != 0 Then Return      ' Return if an error occurred during the proc-
ess
Gosub START_WRITE_PROCESS           ' Start of PROCESSING CONDITION.
' Send PIN byte 2.
Inc ADDRESS                         ' ADDRESS now points to $02
DATA_BYTE = PIN.Byte1               ' Load the SECOND PIN number
Gosub SEND_COMMAND                ' Send the COMPARE SECURITY MEMORY com-
mand
If ERROR_CODE != 0 Then Return    ' Return if an error occurred during the process
Gosub START_WRITE_PROCESS           ' Start of PROCESSING CONDITION.
' Send PIN byte 3.
Inc ADDRESS                         ' ADDRESS now points to $03
DATA_BYTE = PIN.Byte2               ' Load the THIRD PIN number
Gosub SEND_COMMAND                ' Send the COMPARE SECURITY MEMORY com-
mand
If ERROR_CODE != 0 Then Return      ' Return if an error occurred during the proc-
ess
Gosub START_WRITE_PROCESS           ' Start of PROCESSING CONDITION.

' Try and erase the ERROR COUNTER
ADDRESS = $00                       ' Point to SECURE memory address 0
DATA_BYTE = $FF                     ' Write $FF to address of SECURITY memory
COMMAND = UPDATE_SECURE_MEM ' Setup the command to UPDATE SECURITY
MEMORY
Gosub SEND_COMMAND                ' Send the UPDATE SECURITY MEMORY com-
mand
If ERROR_CODE != 0 Then Return      ' Return if an error occurred during the proc-
ess
Gosub START_WRITE_PROCESS           ' Start of PROCESSING CONDITION.
Gosub READ_SECURE_MEMORY            ' Read the SECURITY memory again
If S_MEM.BYTE0 <> 7 Then            ' Houston, we have a problem!
ERROR_CODE = 2                      ' Indicate we have an error opening the card
Delayms 50
Hrsout 1,1
Delayms 50
```

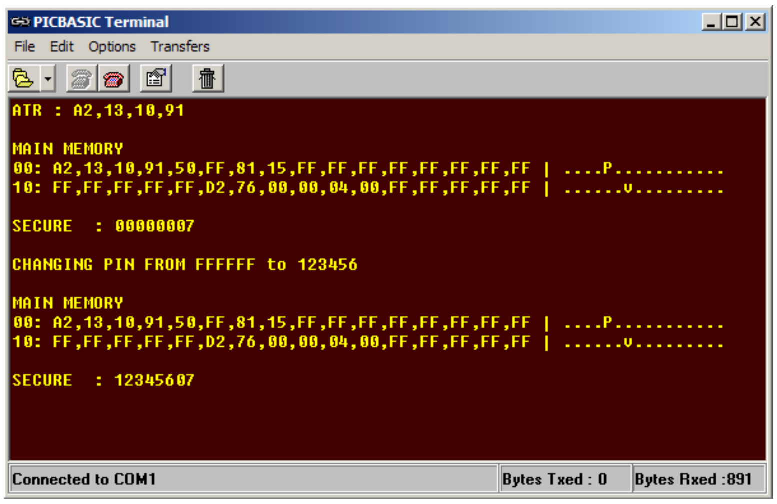
```
Hrsout "WRONG PIN ENTERED",13
Endif
If S_MEM.BYTE0 = 6 Then
Hrsout "WARNING! ONLY TWO ATTEMPTS LEFT AT ENTERING THE CORRECT PIN",13
Endif
If S_MEM.BYTE0 = 4 Then
Hrsout "WARNING! ONLY ONE ATTEMPT LEFT AT ENTERING THE CORRECT PIN",13
Endif
Return
'-------------------------------------------------------------------------------------
' Wait for the card to be inserted into the socket before continuing
WAIT_FOR_INSERTION:
While CARD_IN = 0                         ' Wait for the card to be inserted into the
socket
Low CARD_VCC
If CLEARED_ONCE = 0 Then                  ' Make sure CLS is carried out only once in
the loop
Hrsout 1,"  SLE4442 READER",13,"PLEASE INSERT CARD",13
Set CLEARED_ONCE
Endif
Wend
High CARD_VCC                             ' Enable the card's VCC (5 Volts)
If CLEARED_ONCE = 1 Then Hrsout 1 : CLEARED_ONCE = 0
Delayms 200                               ' Wait for the card to fully power up
Return
'-------------------------------------------------------------------------------------
' Check if the card is the right type, and is inserted correctly
' i.e. not upside down, or back to front
CHECK_CARD_TYPE:
ERROR_CODE = 0
' Check if the card in the socket correctly
' Will return all $FF if not
If ATR_MEM.BYTE3 = $FF AND ATR_MEM.BYTE2 = $FF Then
Hrsout 1,"CARD INSERTION ERROR",13
While CARD_IN = 1 : Delayms 10 : Wend     ' Wait for the card to be removed before pro-
ceeding
ERROR_CODE = 1
Return
Endif

' Check if the correct type of card is inserted
' By examining the first byte of the ATR
If ATR_MEM.BYTE3 != $A2 Then              ' Is the first byte equal to $A2 ?
Hrsout 1,"INVALID CARD TYPE",13
While CARD_IN = 1 : Delayms 10 : Wend     ' Wait for the card to be removed before pro-
ceeding
ERROR_CODE = 1
Endif
Return
'-------------------------------------------------------------------------------------
' Display the PROTECTED part of MAIN MEMORY (first 32 bytes)
DISPLAY_MAIN_MEMORY:
Hrsout "MAIN MEMORY",13
```

```
For LOOP = 0 to 31 Step 16
Hrsout HEX2 LOOP , ": "
For LOOP2 = 0 to 15                                          ' Display HEX
Hrsout HEX2 MEMORY[LOOP  + LOOP2]
If LOOP2 < 15 Then Hrsout "," : Else Hrsout " | "
Next
For Loop2 = 0 to 15                                          ' Display ASCII
TEMP = MEMORY[LOOP  + LOOP2]
If TEMP > 32 AND TEMP < 127 Then
Hrsout TEMP
Else
Hrsout "."
Endif
Next
Hrsout 13
Next
Return
'-------------------------------------------------------------------------------------
' *** MAIN PROGRAM LOOP STARTS HERE ***
MAIN:
Clear                                ' Clear all user RAM
CLEARED_ONCE = 0
Input CARD_IN
PIN = $FFFFFF                        ' Set the PIN number to default $FFFFFF

Gosub WAIT_FOR_INSERTION             ' Wait for the card to be inserted into the
socket
Gosub Read_ATR                       ' Read the ATR (ANSWER-TO-RESET) of the
card
If ERROR_CODE != 0 Then MAIN         ' Check for card insertion error
Gosub CHECK_CARD_TYPE                ' Make sure it's the right type of card
If ERROR_CODE != 0 Then MAIN         ' Check for card insertion error

' Display the ATR value on the serial terminal
Hrsout "ATR : ",HEX2 ATR_MEM.BYTE3,",",HEX2 ATR_MEM.BYTE2,",",HEX2
ATR_MEM.BYTE1,",",HEX2 ATR_MEM.BYTE0,13,13

' Read and Display MAIN and PROTECTION memory before writing to it
Hrsout "BEFORE WRITE",13,13
Gosub READ_MAIN_MEMORY               ' Read all of MAIN memory into an array
If ERROR_CODE != 0 Then MAIN         ' Check for card insertion error
Gosub READ_PROT_MEMORY               ' Read the 4 bytes of PROTECTION memory
If ERROR_CODE != 0 Then MAIN         ' Check for card insertion error
Gosub DISPLAY_MAIN_MEMORY            ' Display 32 bytes of MAIN memory
If ERROR_CODE != 0 Then MAIN         ' Check for card insertion error
Hrsout 13,"PROTECT : ",BIN32 P_MEM,13  ' Display the PROTECTION memory

Gosub OPEN_MEMORY                    ' Open the memory, ready for writing
If ERROR_CODE = 1 Then MAIN          ' The card has been removed ?
If ERROR_CODE = 2 Then               ' There was an error with opening the card ?
Hrsout 13,"SECURE  : ",HEX8 S_MEM,13 ' Display the SECURITY memory area
While CARD_IN = 1 : Wend             ' Wait for the card to be removed
Goto MAIN                            ' Then start again
```

```
Endif

' Write the value $50 (ASCII P) to address 4 of the card
COMMAND = UPDATE_MAIN_MEM              ' Setup the correct command
ADDRESS = 4                            ' Write from address 4 of the card
DATA_BYTE = "P"                        ' Setup for a value of $50  to be written
Gosub SEND_COMMAND                     ' Write the data to the card
If ERROR_CODE != 0 Then MAIN           ' Was there an error ?
Gosub START_ERASE_WRITE_PROCESS        ' Start of PROCESSING CONDITION.
' Now WRITE Protect the value at address 4
' Load variable ADDRESS with the memory address required for write protect (0 - 31)
' This is already pointed to by previous code above.
COMMAND = WRITE_PROT_MEM               ' Setup to write to protected memory
Gosub SEND_COMMAND                 ' Send the WRITE PROTECTION MEMORY com-
mand
If ERROR_CODE != 0 Then MAIN           ' Was there an error ?
Gosub START_WRITE_PROCESS              ' Start of WRITE PROCESSING CONDITION.

' Read and Display MAIN, and PROTECTION memory after writing to it
Hrsout 13,"AFTER WRITE",13,13
Gosub READ_MAIN_MEMORY                 ' Read all of MAIN memory into an array
If ERROR_CODE != 0 Then MAIN           ' Check for card insertion error
Gosub READ_PROT_MEMORY                 ' Read the 4 bytes of PROTECTION memory
If ERROR_CODE != 0 Then MAIN           ' Check for card insertion error
Gosub DISPLAY_MAIN_MEMORY              ' Display 32 bytes of MAIN memory
If ERROR_CODE != 0 Then MAIN           ' Check for card insertion error
Hrsout 13,"PROTECT : ",BIN32 P_MEM,13  ' Display the PROTECTION memory
While CARD_IN = 1 : Delayms 10 : Wend  ' Wait for the card to be removed before pro-
ceeding
Goto MAIN
```

Let's recap on what we can now do with the SLE4442 card. We can read all of it's memory, as well as write all and write protect parts of the MAIN memory, using a PIN. What we need to examine now is a means of changing the PIN from the default value of FFFFFF to whatever 3 byte combination is desired.

Remove any card that's inserted into the socket, and load the program **CHANGE_PIN.BAS**, compile, and download it to the PROTON SMART board, then open the serial terminal window. Press RESET on the PROTON SMART to re-initialise the program, and insert the card. You will be greeted with the display below:



```
ATR : A2,13,10,91

MAIN MEMORY
00: A2,13,10,91,50,FF,81,15,FF,FF,FF,FF,FF,FF,FF,FF | ....P...........
10: FF,FF,FF,FF,FF,D2,76,00,00,04,00,FF,FF,FF,FF,FF | ......v.........

SECURE  : 00000007

CHANGING PIN FROM FFFFFF to 123456

MAIN MEMORY
00: A2,13,10,91,50,FF,81,15,FF,FF,FF,FF,FF,FF,FF,FF | ....P...........
10: FF,FF,FF,FF,FF,D2,76,00,00,04,00,FF,FF,FF,FF,FF | ......v.........

SECURE  : 12345607
```

Connected to COM1 | Bytes Txed : 0 | Bytes Rxed :891

As can be seen from the above display, the PIN has been changed from the default value FFFFFF$_H$, to the new value of 123456$_H$. To ensure that the PIN really has been changed, remove the card, and re-insert it into the socket. The display will inform you that an incorrect PIN has been entered. This is because the card requires a valid PIN before it can be changed, and as the card now contains a different PIN, the card does not now recognise the original PIN.

The key elements to the **CHANGE_PIN.BAS** program are shown below:

On lines **349** and **350** of the program, there are these two statements.

```
PIN = $FFFFFF                       ' Set the PIN value to default $FFFFFF
NEW_PIN = $123456                   ' The new pin value for the card
```

The variable PIN holds the default (or original) PIN of the card, which is FFFFFF in this case. The variable NEW_PIN contains the new PIN value that we require. The subroutine that actually changes the PIN is named CHANGE_PIN, and is shown below:

```
' Write a new PIN to the card
' The card must already be opened by a valid PIN,
' and the new PIN value required must be loaded into variable NEW_PIN
CHANGE_PIN:
COMMAND = UPDATE_SECURE_MEM         ' Setup command UPDATE SECURITY
MEMORY
ADDRESS = 1                         ' ADDRESS points to $01
' Send PIN byte 0.
DATA_BYTE = NEW_PIN.Byte0           ' Load the FIRST NEW PIN value
Gosub SEND_COMMAND              ' Send the COMPARE SECURITY MEMORY com-
mand
If ERROR_CODE != 0 Then Return      ' Return if an error occurred during the proc-
ess
Gosub START_ERASE_WRITE_PROCESS     ' Start of PROCESSING CONDITION.
' Send PIN byte 1.
Inc ADDRESS                         ' ADDRESS now points to $02
DATA_BYTE = NEW_PIN.Byte1           ' Load the SECOND NEW PIN value
Gosub SEND_COMMAND              ' Send the UPDATE SECURITY MEMORY com-
mand
If ERROR_CODE != 0 Then Return      ' Return if an error occurred during the proc-
ess
Gosub START_ERASE_WRITE_PROCESS     ' Start of PROCESSING CONDITION.
' Send PIN byte 2.
Inc ADDRESS                         ' ADDRESS now points to $03
DATA_BYTE = NEW_PIN.Byte2           ' Load the THIRD NEW PIN value
Gosub SEND_COMMAND              ' Send the UPDATE SECURITY MEMORY com-
mand
If ERROR_CODE != 0 Then Return      ' Return if an error occurred during the proc-
```

```
ess
Gosub START_ERASE_WRITE_PROCESS    ' Start of PROCESSING CONDITION.
Return
```

To change the PIN value back to its default value of FFFFFF, change lines 349 and 350 to:

```
PIN = $123456                      ' Set the PIN value to 123456
NEW_PIN = $FFFFF                   ' The new pin value for the card
```

Download the program to the PROTON SMART board, and re-insert the card. The PIN is now back to it's original value.

The CHANGE_PIN subroutine is similar in appearance to the OPEN_MEMORY subroutine that validates the PIN, except that it now issues the UPDATE SECURITY MEMORY command, instead of the COMPARE SECURITY command.

That's it! We've come to the end of the functionality explanations of the SLE4442 card. We can now read, write, enter, and change the PIN value. It's up to you now!

One thing you must take into consideration when using the SLE4442 card, or any of the memory cards, is that they can be read without using a PIN value. Therefore any delicate or secure data placed on the card should use some sort of encoding. This can be as simple as an XOR with the previous value, or a full blown DES or Triple DEC encoding scheme. The internet has a wealth of resources regarding such encryption techniques, and some have even been implemented on the PIC microcontroller.

The possible applications for smartcards, or memory cards is endless, and far too involved for a tutorial such as this. Some applications include: -

- Customer/Employee ID.
- Automatic Door Entry.
- Remote Data Logging.
- Firmware upgrades using removable memory cards.
- Hardware configuration changes using a memory card.

The list goes on......

PROTON SMART Electrical and Software Specifications.

The PROTON SMART board offers a flexible approach to developing hardware and software solutions for the all kinds of smartcards. 8K Bytes of program space in the PROTON SMART, means that large, or complex programs may be written without worrying too much about running out of memory.

The PROTON SMART also offers an exceedingly easy to use programming method using a serial bootloader. Simply connect the board to a spare serial port on the PC.

Writing code for the PROTON SMART is independent of the language used. i.e. Assembler, C, or BASIC etc. However, for this document, we'll assume the language of BASIC for simplicity, and clarity.

The BASIC language used is the PROTON+ Compiler Version 2.0 onwards from Crownhill Associates. Not only is this language flexible and easy to use, but it incorporates a bootloader within the Windows IDE. Two clicks of the mouse button can compile and download the newly created source code into the PROTON SMART's on-board PICmicro.

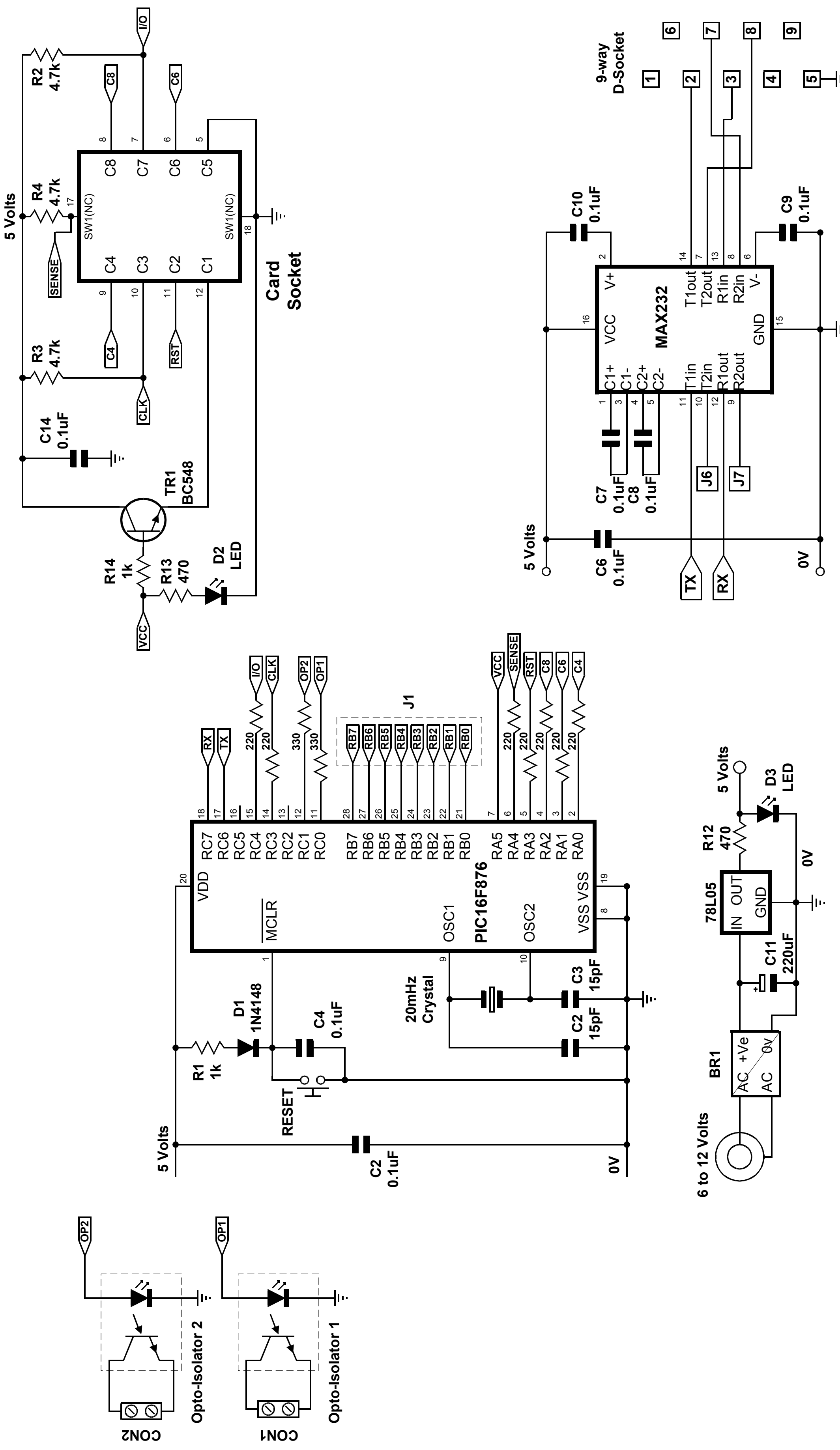
The same serial link can be used to communicate with the PC when developing new software, using the Compiler's serial terminal window.

The PROTON SMART is based around a PIC16F876 Microcontroller from Microchip. This device has 8K of program space, 368 bytes of user ram for storage of variables etc, and 256 bytes of eeprom space for storage of data. A 20MHz crystal oscillator ensures fast, accurate timing over a large range of temperatures and environment changes. Eight digital I/O lines allow external devices to be connected to the PROTON SMART, and are fully TTL level compatible, with each line having a pullup resistor associated with it, which may be enabled/disabled through software.

There are also two fully isolated output lines for connection to higher powered devices, such as relays, solenoids etc. Or circuits that have a higher voltage requirement, such as an H-Bridge.

The power requirements for the PROTON board are also extremely flexible, and robust. Being either an AC or DC, 6 to 12 Volts supply at approx 500mA.

The full circuit of the PROTON SMART is shown overleaf.



PROTON SMART Circuit Diagram.

What is a bootloader?

The PIC16F87x range of devices have a unique feature, in that they can program their own FLASH code space while running. All other PICmicros must be programmed by a device programmer, such as the PICSTART or EPIC programmers. This self-modifying feature allows a PIC16F87x device to run a program named a bootloader.

A bootloader is a program that resides in the code space of the target PIC. It can be activated to allow additional program code to be written to and read from that same target PIC. A bootloader consists of 2 elements, connected by a serial cable.

The first part of the bootloader is a program resident on the PICmicro. This program occupies the upper 256 words of the FLASH code space. This small program must be placed into the PICmicro using a conventional programmer.

The program resident in the PICmicro communicates with the second element of the loader over a serial connection. This second program is the bootloader window within the compiler's IDE and is the user interface. It allows the compiled BASIC code to be programmed.

Only the code space and data space may be read and programmed on the target PIC. The ID space and CONFIGURATION fuses are not accessible to the bootloader. The configuration fuses must be set at the time the actual loader program is programmed into the PICmicro. Once they are set, they cannot be changed by the bootloader.

The bootloader software resident in the PICmicro, intercepts the reset vector. When the PIC powers up, it enters the loader's boot supervisor, this watches the USART's serial input pin for a start bit for 200 milliseconds (ms). If it sees activity during this period, it enters the communications section of the software to download a program. If it does not see any activity during the 200ms, it starts the user program in the PICmicro.

The interception of the reset vector is accomplished by automatically relocating the first 4 user program words from address's 0 to 3 to a reserved place in the bootloader's code space (within the top 256 words). A jump to the bootloader is then placed at locations 0 to 3. When the loader software running on the PC reads or writes these addresses the values seen are as if the bootloader was not resident and the reset code had not been moved.

The USART's serial pins used by the bootloader are only required when the loader is actually programming the PIC. They are unattached while the downloaded user program is running on the target and may be assigned to any other task or serial baud rate, with one exception. As we've already discussed, the bootloader checks the serial input pin at power up and reset, to determine if it should run, or if the user (downloaded) program should run.

Therefore, for the user program to run, this pin must not be in the start bit condition when the device is powered up.

Make sure the serial in pin (PORTC.7) is in the idle state when the PIC is first powered up.

The serial communication speed is set at 19200 baud. The bootloader program resident in the PICmicro can easily communicate at this speed with an oscillator frequency from 4MHz to 20MHz. This oscillator frequency is determined at the time the loader code is programmed into the target PIC. The target PIC must then only be run at this frequency in order to be able to communicate with its matching part running on the PC. The bootloader uses no PICmicro resources while the user program is running. All the data memory, RAM, and I/O pins are available to the user program.

However, there are a few considerations that should be noted when writing programs that will be loaded by the bootloader. The first is that the bootloader takes over at power up and any subsequent resets. Any time the program vectors through the reset address, the loader becomes active and watches the RX pin (PORTC.7) for any activity. If there is any action on this pin, the loader will start, and the user program will not execute. Even if there is no activity on this pin, the start of the user program will be delayed by the 200 milliseconds while the bootloader is watching the RX pin.

Another consideration is the fact that the configuration fuses are not alterable by the bootloader. If some programs require the use of the Watchdog Timer and others don't, then separate PICmicros will be necessary. One PIC programmed with the WDT on and one with it off. The same is true for the Power up Timer, Brownout Detect Enable and Oscillator type. The standard programmed defaults are WATCHDOG TIMER ON, POWERUP TIMER ON, BROWNOUT DETECT ENABLE ON and HS OSCILLATOR.

The configuration fuses for code protection CANNOT be enabled. The bootloader needs to be able to freely read and write to the PIC's code and data space. Therefore, the device cannot be code protected. The bootloader is primarily aimed at development work, any final products that require code protection must be programmed in the conventional way.

The bootloader software occupies the last 256 words of code space. A compiled program is written starting at location 0 and grows upward so the loader's position in memory is not noticeable. You must make sure that the program code does not attempt to enter the upper 256 words of code space. The bootloader inserts its own code at the reset vector and automatically relocates the user's reset code to an area reserved within the top 256 words of memory. Normally, these locations contain a jump to the start-up routine for the user program. However, since the user code is no longer situated at these locations, the user program should not attempt to jump to, or call any routine within the code area between 0 and 3.

NOTES