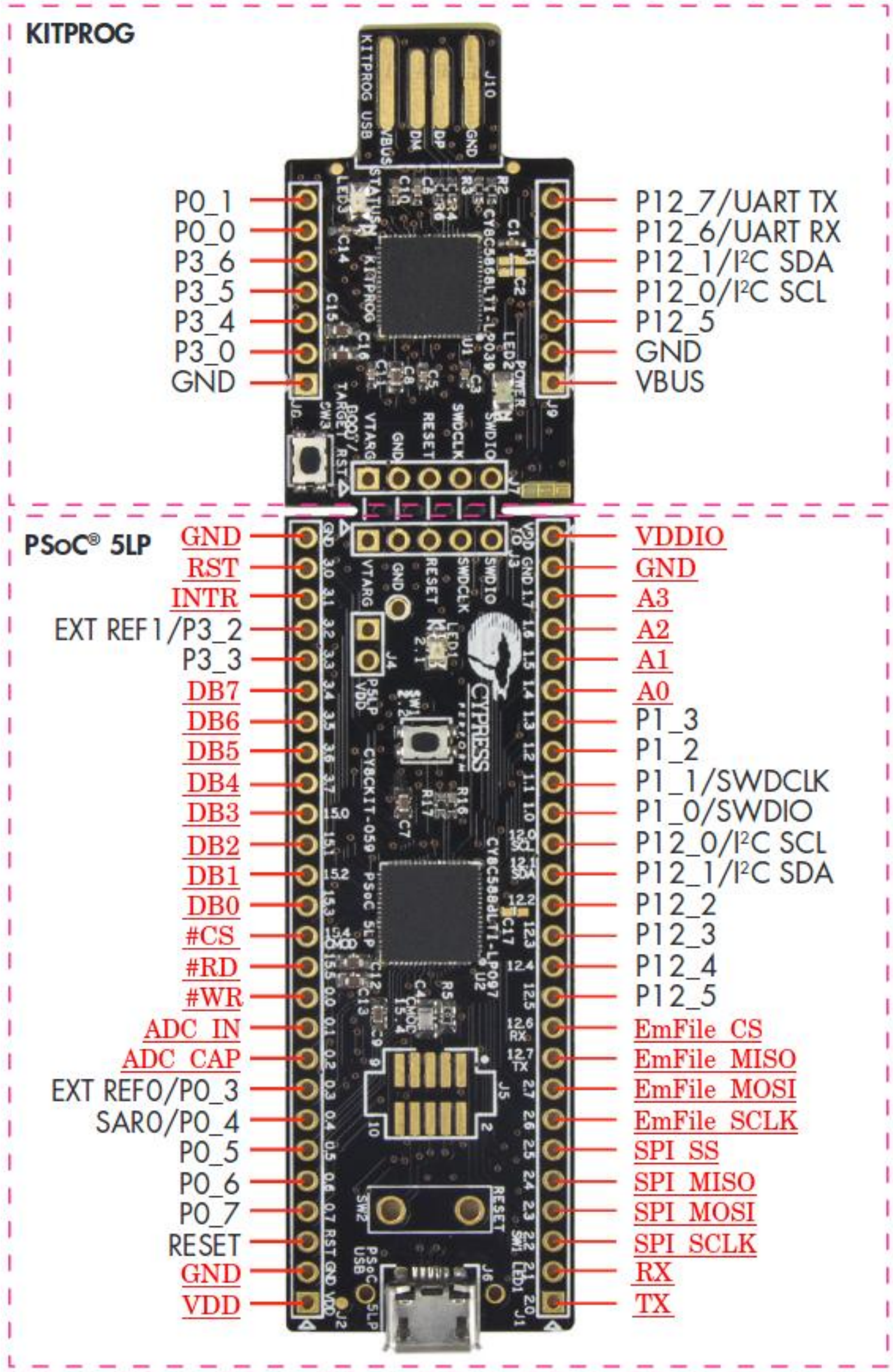
The 8000 Series Swiss Army Knife Manual

Matthew Burns | Eric Ponce
miburns@mit.edu

August 2017



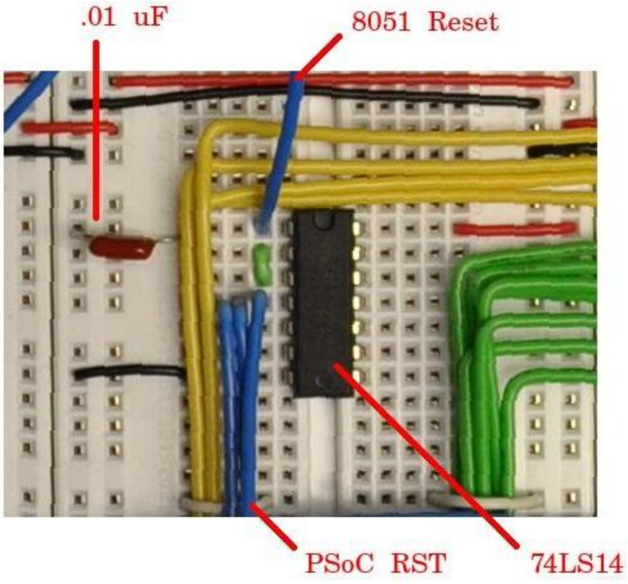
1 Overview

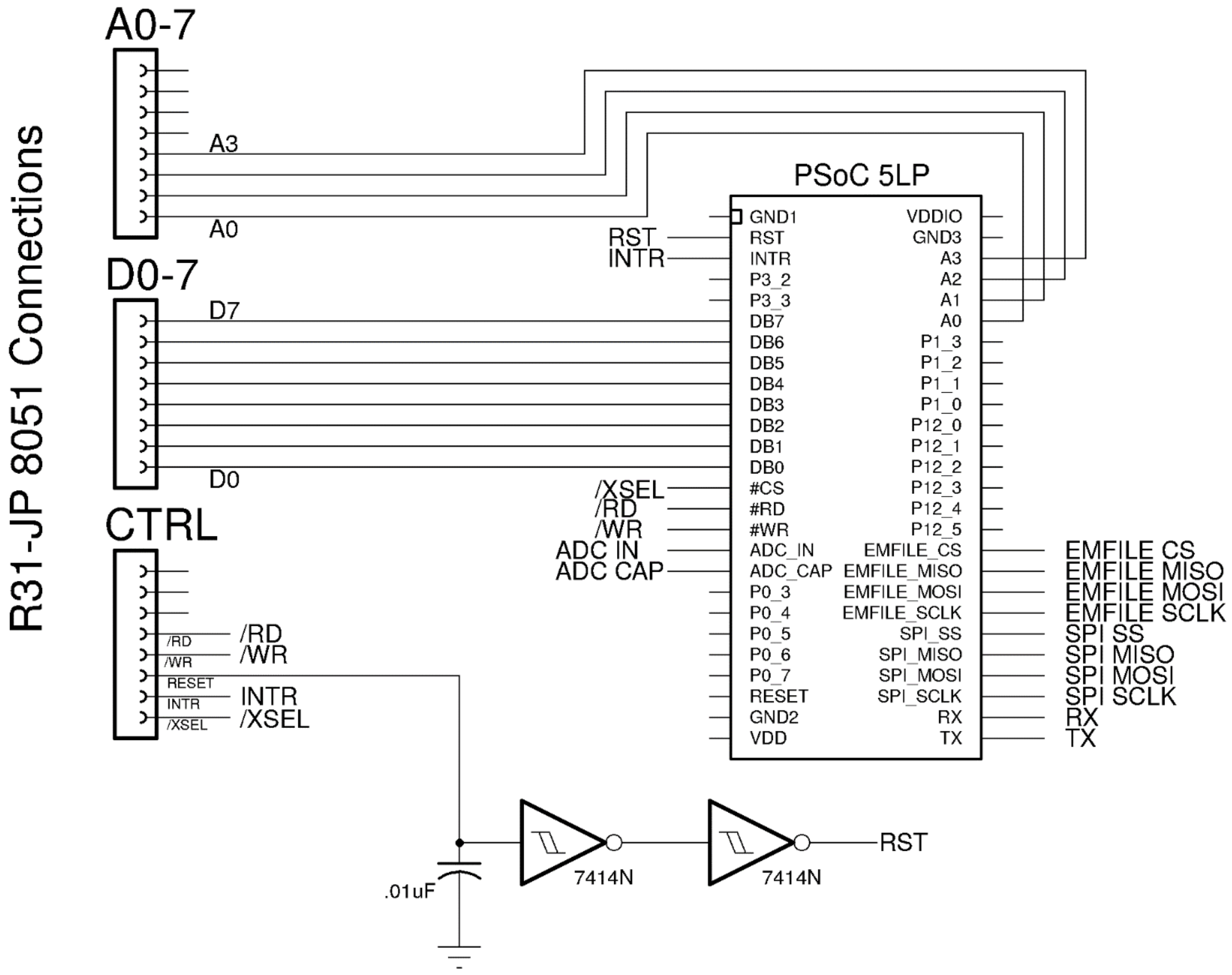
The 8000 Series Swiss Army Knife is intended to be a user configurable replacement to most 8000 series peripherals. It has a data bus, an address bus, and all other supporting pins to make this PSoC communicate with an 8051 as if it were an 8000 series peripheral with 16 registers. Several of the registers already have dedicated functions controlling components such as a Universal Asynchronous Receiver/Transmitter (UART) component, a Serial Peripheral Interface Master (SPIM) component, an Analog to Digital Converter (ADC) component, and an SD card communication (EmFile) component. Several other registers are left unused so that the user can implement their own 8000 series peripherals.

2 Hardware

This 8000 Series Swiss Army Knife requires only a CY8CKIT-059 PSoC 5LP. Many of the pins on this PSoC are not (by default) used for this peripheral replacement solution and are labeled in black on the pinout shown on page one of this manual. The pins that are used for this replacement solution are labeled with their function in red.

This PSoC is fitted with an 8-bit data bus, a 4-bit address bus, read (/RD) and write (/WR) lines, a chip select (/CS) line, and an interrupt (INTR) line to communicate with the 8051 as if it were an 8000 series peripheral. It also has a reset (RST) line which controls the software reset of the entire PSoC replacement. This RST line should be connected to the 8051 reset pin through two 7414 inverters with a .01 uF capacitance on the 8051 reset line.





*Note: Power (5V) and ground (0V) connections were intentionally left off of the PSoC in the schematic for clarity.

In addition to connecting all standard lines for 8000 series communication, the three GND pins of the PSoC should connect to the ground of the 8051 system. VDD and VDDIO should be tied to +5VDC.

3 Software

UART:

Wiring:

The TX and RX pins of the UART module are accessible through P2.0 and P2.1 on the PSoC respectively.

Interface:

The UART module makes use of the first three of the 16 addressable registers on the PSoC: address 0x0, address 0x1, and address 0x2.

Address 0x0 is the control register for the UART module. It allows the user to turn the UART module on and off, set the baud rate, and manipulate the receive / transmit flags and the receive / transmit interrupt enable bits.

Address 0x1 is the UART write register. Writing to this register causes the UART module to transmit the byte that was written to this register provided that the UART module is on and is not currently transmitting a byte.

Address 0x2 is the UART read register. Reading this register returns the byte that was most recently received by the UART module.

All of these registers have a reset value of 0x00.

Table of 16C450 Registers

Address:	Register Name:	Reset Value:	Function:
0x0	UART Control Register	0x00	The control register allows you to turn the UART module on and off, set the baud rate, and control the UART interrupts.
0x1	UART Transmit Buffer	0x00	The transmit buffer is the register you write to when you wish to transmit a byte.
0x2	UART Receive Buffer	0x00	The receive buffer holds the value of the UART module’s most recently received byte.

Control Register:

7	6	5	4	3	2	1	0
TF	RF	TIE	RIE	M3	M2	M1	M0

The bits of the control register (0x0) are set up as follows:

Bit 7: Transmit Flag. This bit is automatically set when the UART component finishes transmitting a byte. Both this bit and the receive flag (bit 6) need to be cleared to reset the interrupt line if this UART component sent a falling edge interrupt to the 8051.

Bit 6: Receive Flag. This bit is automatically set when the UART component receives a byte. Both this bit and the transmit flag (bit 7) need to be cleared to reset the interrupt line if this UART component sent a falling edge interrupt to the 8051.

Bit 5: Transmit Interrupt Enable. If this bit is set, the PSoC will send a falling edge interrupt to the 8051 upon completion of a UART transmit.

Bit 4: Receive Interrupt Enable. If this bit is set, the PSoC will send a falling edge interrupt to the 8051 upon receiving a byte.

Bit 3: The lower nibble of this control register is used to turn on and off the UART module and set the baud rate. (M3)

Bit 2: (M2)

Bit 1: (M1)

Bit 0: (M0)

The UART mode is used to control the baud rate of the UART module and is determined by the lower four bits of the UART control register. Below are the possible nibble values to control the UART mode:

M3:	M2:	M1:	M0:		Mode:
0	0	0	0		UART off
0	0	0	1		300 Baud
0	0	1	0		1200 Baud
0	0	1	1		2400 Baud
0	1	0	0		4800 Baud
0	1	0	1		9600 Baud
0	1	1	0		19200 Baud
0	1	1	1		38400 Baud
1	0	0	0		57600 Baud
1	0	0	1		115200 Baud
1	0	1	0		230400 Baud
1	0	1	1		9600 Baud
1	1	0	0		9600 Baud
1	1	0	1		9600 Baud
1	1	1	0		9600 Baud
1	1	1	1		9600 Baud

Once the control word is set up in the control register, the UART module is set up to receive and write bytes. In order to keep track of the state of the UART module, the user can either poll the receive / transmit flags, or the user can set up an interrupt. If the PSoC generates an interrupt on the INTR line back to the 8051, the line will drop low until the user clears the interrupt. The interrupt can be cleared by clearing the TF and RF bits of the control register.

SPIM:

Wiring:

The SPIM module makes use of the standard four wire SPI interface that includes a serial clock (SCLK) line, a slave select (SS) line, a master-in slave-out (MISO) line, and a master-out slave-in (MOSI) line. All four of these connections are located between P2.2 and P2.5 on the PSoC and are labeled as SPI pins on the pinout found on the first page of this document.

Interface:

The SPIM module makes use of the fourth and fifth of the 16 addressable registers on the PSoC: address 0x3, and address 0x4.

Address 0x3 is the control register for the SPIM module. It allows the user to turn the SPIM module on and off, set the bit rate, and manipulate the shift flag and the shift interrupt enable bit.

Address 0x4 is the SPIM buffer. Writing to this register causes the SPIM module to transmit the byte that was written to this register provided that the SPIM module is on and is not currently shifting a byte. Reading from this register returns the value that was most recently shifted into the SPIM module.

Both of these registers have a reset value of 0x00.

Table of SPIM Registers

Address:	Register Name:	Reset Value:	Function:
0x3	SPIM Control Register	0x00	The control register allows you to turn the SPIM module on and off, set the bit rate, and control the SPIM interrupt.
0x4	SPIM Buffer	0x00	The shift buffer is the register you write to when you wish to shift out or read in a byte. Reading the value of this register after writing to it returns the value that was just shifted into this buffer.

Control Register:

7	6	5	4	3	2	1	0
SF	SIE	X	X	X	M2	M1	M0

The bits of the control register (0x3) are set up as follows:

Bit 7: Shift Flag. This bit is automatically set when the SPIM component finishes shifting out a byte. This bit needs to be cleared to reset the interrupt line if this SPIM component sent a falling edge interrupt to the 8051.

Bit 6: Shift Interrupt Enable. If this bit is set, the PSoC will send a falling edge interrupt to the 8051 upon completion of a SPIM shift.

Bit 5: Not used

Bit 4: Not used

Bit 3: Not used

Bit 2: The lower three bits of this control register are used to turn the SPIM module on and off as well as to set the bit rate. (M2)

Bit 1: (M1)

Bit 0: (M0)

The SPIM mode is used to control the bit rate of the SPIM module and is determined by the lower three bits of the SPIM control register. Below are the possible values of the lower three bits that control the SPIM mode:

M2:	M1:	M0:		Mode:
0	0	0		SPIM off
0	0	1		50k Bit Rate
0	1	0		100k Bit Rate
0	1	1		200k Bit Rate
1	0	0		500k Bit Rate
1	0	1		1M Bit Rate
1	1	0		2M Bit Rate
1	1	1		5M Bit Rate

Once the control word is set up in the control register, the SPIM module is ready to receive and write bytes. In order to keep track of the state of the SPIM module, the user can either poll the shift flag, or the user can set up an interrupt. If the PSoC generates an interrupt on the INTR line back to the 8051, the line will drop low until the user clears the interrupt. The interrupt can be cleared by clearing the SF bit of the control register.

ADC:

Wiring:

The ADC makes use of P0.1 and P0.2 on the PSoC. P0.1 is the analog input to the ADC; it is the voltage level on this line that is converted to a digital value between 0x00 and 0xFF. P0.2 is the internal ADC reference bypass pin; a .1uF capacitor can be placed between this pin and ground in order to reduce noise for the ADC.

Interface:

The ADC module makes use of the 16th addressable register on the PSoC – address 0xF.

Beginning a conversion:

Writing any value to register 0xF other than 0x00 will begin an ADC conversion. Specifically writing 0x01 to this register starts a conversion **and** enables the external interrupt to the 8051. This external interrupt line falls low when a conversion completes and data is ready to be read. Writing any value to this register, including 0x00, clears the external interrupt line.

Reading the result:

After the conversion is complete, reading from this register returns the value of the analog to digital conversion.

EmFile SD Card Communication:

Overview:

EmFile is an embedded file system that allows you write to and read from files within an SD card. In order to run EmFile on the PSoC, libraries must be downloaded from the SEGGER website and linked to the PSoC project. The libraries are already included with the 8000_Series_Swiss_Army_Knife PSoC project workspace.

By default, the linked EmFile libraries allow access to FAT16 file systems and do not allow for file names that are longer than 8 characters. This can be configured by changing which EmFile libraries are linked to the project. See the PSoC EmFile data sheet at http://www.cypress.com/file/135136/download for more information on which libraries to link.

Wiring:

The EmFile component makes use of four pins of the PSoC to drive the SD card as an SPI slave component. The pins are located between P2.6 and P12.6 on the PSoC and are labeled as EmFile pins in the pinout on the first page of this document.

Interface:

The EmFile module makes use of the 14th and 15th of the 16 addressable registers on the PSoC: address 0xD, and address 0xE.

Address 0xD is the control register for the EmFile module. It allows the user to control the state of the EmFile module.

Address 0xE is the EmFile data register. Writing to this register does different things depending on the state of the EmFile component. For example, if the EmFile component is in the "Enter File Name" state (see table below), writing to address 0xE appends characters to the desired file name of the file to be created or opened. If the EmFile component is in the "Write" state, writing to address 0xE writes data to the open file on the SD card. This register can also be used to read data from the SD card file if the EmFile component is in the "Read" state.

Both of these registers have a reset value of 0x00.

Table of EmFile Registers

Address:	Register Name:	Reset Value:	Function:
0xD	EmFile State Control Register	0x00	The control register allows you to control the state of the EmFile Component.
0xE	EmFile Data Transfer Register	0x00	The data register allows you to send data to and from the SD card as well as specify file names to be opened or created.

Control Register:

The control register for the EmFile component on the PSoC 8000 Series Swiss Army Knife controls the state of the EmFile component. The following write values of the control register (0xD) implement the following changes within the EmFile component.

Write Value:	State:	Function:
0x1	Enter File Name	Writing 0x1 to the EmFile control register sets up the EmFile component to receive the name of the file to be accessed in ascii hex through writing to the data register 0xEh. For example... after writing 0x1 to the control register, one would sequentially write (0x50, 0x41, 0x54, 0x54, 0x45, 0x52, 0x4E, 0x2E, 0x74, 0x78, 0x74) in order to access the file "PATTERN.txt". The maximum file name size is 8 characters (bytes).
0x2	Write	Writing 0x2 to the EmFile control register sets up the EmFile component to begin receiving data through register 0xE and writing it to the file that was specified while in the "Enter File Name" state. Before sending 0x2 to the control register, one must first write 0x1 and enter an appropriate file name.
0x3	Read	Writing 0x3 to the EmFile control register sets up the EmFile component to begin sending out data from the file that was specified while in the "Enter File Name" state through register 0xE. Before sending 0x3 to the control register, one must first write 0x1 and enter an appropriate file name.
0x4	Close File	Writing 0x4 to the EmFile control register closes the open file that was specified while in "Enter File Name" mode.
0x0	Clear Interrupt	Writing this value to the control register is the only value that can be written to either the control register (0xD) or the data register (0xE) that does not set the finished flag - bit 6 within the EmFile control register. Bit 7 of the EmFile control register is the EmFile interrupt enable bit. Therefore, in order to reset the interrupt line from a previous interrupt and enable future EmFile interrupts, one would write 0x80 to the EmFile control register. In order to reset the interrupt line from a previous interrupt and turn off future EmFile interrupts, one would write 0x00 to the EmFile control register. The critical thing is that the 6th bit is cleared on this write to reset the interrupt line.

*Note: The upper nibble of the write value to the EmFile State Control Register was not specified because its value depends on how the user sets the interrupts.

*Note: Writing to a file that already exists will overwrite the file. However, the previous file will not be deleted before the overwrite occurs. This means that if you overwrite a file with another file of a smaller size, data from the previous file will be left over at the end of the newly written file.

Bit 7 of the EmFile State Control Register is the EmFile Interrupt Enable Bit. Setting this bit causes a falling edge interrupt on the INTR line to occur whenever the EmFile Finished Flag is set. Bit 6 of the EmFile State Control Register is the EmFile Finished Flag. The EmFile finished flag sets after every operation involving a write to either the state control or the data register (except writing 0x0 to the state control register) completes.

Data Register:

The data register is used for passing data into and out of the EmFile system and the SD card. It is used to name the file to open (while in "Enter File Name" mode) by writing the filename to this register byte by byte. It is used to write data into a file within the SD card byte by byte while within the "Write" mode. This register is also used to read data from a file within the SD card while within the "Read" mode. A byte read is completed by writing any value to this register and then reading the value within this register. By writing to this register while in "Read" mode, we are telling EmFile to fetch the next byte from the SD card and stuff it into this register so that we can read it.

Overview of Internal Operation:

The PSoC's internal PLL output is set to 48 MHz. This is used to drive the system MASTER_CLK clock. Setting the clock to 48 MHz ensures that the PSoC is able to perform its functions fast enough to complete its task without interfering too much with the 8051.

The PSoC emulates memory mapped registers using DMA accessed memory locations. Given a command – read to or write from a particular register – DMA block's move data to and from the appropriate memory locations. The registers that hold the 16 accessible locations exist within a uint8 array labeled 'Reg'. This replacement uses an 8-bit wide control register to write data back out to the data bus and status registers to read data into the 'Reg' array as well as to read the address into the 'Addr' variable.

Once the value of the 'Addr' variable is changed by a data write, the main loop of the PSoC code implements the appropriate changes to the programmed peripherals. Having the main loop control the peripheral hardware makes the code understandable while having the DMA blocks control the 8000 series communication ensures the timing specification is not violated.

4 UART Example

To demonstrate this 8000 Series Swiss Army Knife UART module, we will attach it to an Amulet module and write some software for the 8051 so that every time a button is pressed on the Amulet module, the character is displayed on the R31-JP Port 1 LED's as well as on the monitor's display.

After wiring the 8000 series communication portion of this chip to the 8051, all that is left to do is attach the PSoC's TX line to the Amulet's Rx line and the PSoC's RX line to the Amulet's TX line. Be sure to connect the ground of the Amulet module to the ground of the R31-JP kit and PSoC. Assemble and load the 8051 system with the following assembly code:

```
.org   000h
ljmp   start

.org   003h
ljmp   isr

.org   100h
start:
     lcall init
main:
     sjmp  main

init:
; Set up serial communication to the computer
     mov   tmod, #20h          ; set timer 1 for auto reload - mode 2
     mov   tcon, #41h          ; run counter 1 and set edge trig ints
     mov   th1,  #0fdh         ; set 9600 baud with xtal=11.059mhz
     mov   scon, #50h          ; set serial control reg for 8 bit data
                               ; and mode 1

     mov   IE,   #81h          ; Fully enable the edge triggered interrupt

     mov   dptr, #0xFE00       ; Set up PSoC UART flags for 9600 baud
     mov   a, #0x15            ; communication
     movx  @dptr, a

     ret

isr:
     mov   dptr, #0xFE02       ; Read in the byte from the PSOC
     movx  a, @dptr

     mov   P1, a

     ; Here I send the byte to the PC
     clr  scon.1               ; clear the tx  buffer full flag.
     mov  sbuf,a               ; put chr in sbuf
txloop:
     jnb  scon.1, txloop       ; wait till chr is sent

     mov   dptr, #0xFE00       ; Clear the PSoC UART flags thus clearing
                               ; the external interrupt
     mov   a, #0x15
     movx  @dptr, a

     reti                      ; Return from interrupt
```

Once this code is loaded onto and running on the 8051 system, the monitor should be responsive to presses on the Amulet module.

5 SPIM Example

To demonstrate this 8000 Series Swiss Army Knife SPIM module, we will attach it to an SD card and read in 16 bytes of raw data located within sector one (the second grouping of 512 bytes) of the SD card. We will then sequencially flash the 8051's P1 LED's with the 16 different one-byte values that were read from sector one.

After wiring the 8000 series communication portion of this chip to the 8051, all that is left to do is wire the SD card to the SPI port. The easiest way to do this is to use a breakout board like the one found at http://store.linksprite.com/sd-card-breakout-board/. This breakout board is labeled with VDD, GND, and all four of the SPI connections. SD cards are 3.3V devices and will be destroyed if you connect them to a 5 volt supply. However, since this board handles all of the necessary level shifting, we can attach VDD to 5VDC and wire the SPI connections directly to the SPI port on the PSoC. Once everything is wired together, assemble and load the 8051 system with the following assembly code:

```
.org   000h
ljmp   start

.org   003h
ljmp   isr

.org   080h
.db    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF    ; Buffer
.db    0x40, 0x00, 0x00, 0x00, 0x00, 0x95    ; CMD0
.db    0x77, 0x00, 0x00, 0x00, 0x00, 0x00    ; CMD55
.db    0x69, 0x00, 0x00, 0x00, 0x00, 0x00    ; ACMD41
.db    0x51, 0x00, 0x00, 0x00, 0x00, 0x00    ; CMD17

; CMD17 reads the first 16 bytes of the block at SD card address
; located at 0xR7, 0xR6, 0xR5, 0xR4 to the scratch-pad memory at 0x30

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                                                                     ;;;
;;;                      Main Code Written Below                        ;;;
;;;                                                                     ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

.org   300h
start:
     lcall init
main:
     mov   R0,  #30h
     mov   R1,  #10h
     mainLoop:
          mov   a,  @R0
          mov   P1, a

          mov    R2, #00h         ; Wait a second or two so we can see the
          mov    R3, #00h         ; pattern on the LED’s
          mov    R4, #02h
          delay:
               djnz   R2, delay
               djnz   R3, delay
               djnz   R4, delay

          inc   R0
          djnz  R1, mainLoop
     sjmp  main
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                                                                     ;;;
;;;                      Main Code Written Above                        ;;;
;;;                                                                     ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                                                                     ;;;
;;;                   Initialization Written Below                      ;;;
;;;                                                                     ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
init:
; Set up serial communication to the computer
    mov   tmod, #20h       ; set timer 1 for auto reload - mode 2
    mov   tcon, #41h       ; run counter 1 and set edge trig ints
    mov   th1,  #0fdh      ; set 9600 baud with xtal=11.059mhz
    mov   scon, #50h       ; set serial control reg for 8 bit data
                           ; and mode 1

    mov   IE,   #81h       ; Fully enable the edge triggered interrupt

    mov   dptr, #0xFE03    ; Set the PSOC SPI to 200kBS communication and
    mov   a, #0x43         ; turn it on
    movx  @dptr, a

    lcall SDinit

    mov   R7, #00h            ; Read in the block starting at address
    mov   R6, #00h            ; 0x00000200h - Sector 1
    mov   R5, #02h
    mov   R4, #00h            ; It reads the first 16 bytes into scratch
                              ; pad memory starting at 0x30
    lcall CMD17mod            ; And dumps the rest!

    ret

SDinit:
    mov   R1, #80h         ; Here we send 74+ clock pulses to ready the SD
                           ; card
    lcall SDSend
    mov   R1, #80h
    lcall SDSend

    lcall CMD0             ; Here we send CMD0 to put the SD card in IDLE
                           ; mode for a software reset.

    lcall ACMD41           ; Here we send ACMD41 to initialize the SD card
                           ; and take it out of IDLE mode.
                           ; After this, the card is ready to use.

    ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                                                                     ;;;
;;;                   Initialization Written Above                      ;;;
;;;                                                                     ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                                                                     ;;;
;;;                         ISR Written Below                           ;;;
;;;                                                                     ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

isr:
    mov   dptr, #0xFE03    ; Clear the PSoC SPI flag thus clearing the
                           ; external interrupt
```

```
    mov   a, #0x43
    movx  @dptr, a

    setb  00h
    reti                   ; Return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                                                                     ;;;
;;;                            ISR Written Above                        ;;;
;;;                                                                     ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                                                                     ;;;
;;;               SD Card Support Subroutines Written Below             ;;;
;;;                                                                     ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

SDSendByte:
; This subroutine sends the value of a over the SPI port and waits till the
; shift is complete.
    mov   dptr,  #0xFE04
    movx  @dptr, a
    clr   00h
    Hold:
        jnb   00h, Hold
    ret

SDSend:
; This subroutine sends 6 bytes of data to the SD card starting
; With the byte stored in 8051 memory location 0x00XX where XX is
; Determined by the value in R1
    mov   R2, #0x06
    SDSendLoop:
        mov   dph,   #00h
        mov   dpl,   R1
        movx  a,     @dptr
        lcall SDSendByte
        inc   R1
        djnz  R2, SDSendLoop
    ret

SDSendScratch:
; This subroutine sends 6 bytes of data to the SD card starting
; With the byte stored in 8051 scratch-pad memory location starting
; at 0x40
    mov   R2, #0x06
    mov   R1, #0x40
    SDSendLoopS:
        mov   a,     @R1
        lcall SDSendByte
        inc   R1
        djnz  R2, SDSendLoopS
    ret
```

```
SDRead:
; This subroutine searches for a response from the SD card following a
; SDSend command.  It returns with either the first byte of the response or
; a failed response value of 0xFF stored in R2.
    mov   R2, #0xFF
    SDReadLoop:
        mov   dptr,  #0xFE04
        mov   a, #0xFF
        lcall SDSendByte
        mov   dptr,  #0xFE04
        movx  a,     @dptr
        cjne  a, #0xFF, SDReadDone
        djnz  R2, SDReadLoop
        mov   R2, 0xFF
        ret

    SDReadDone:
        mov   R2, a
        ret

CMD0:
; This subroutine sends the CMDO command to the SD card.  It makes use of
; SDSend and SDRead to accomplish this.  If this command fails, the LED's
; on P1 will show 0x00.  If this command succeeds, the LED's will be left
; as 0xFF - as they were at start-up.
    mov   R3, #0xFF
    CMD0Loop:
        mov   R1, #86h
        lcall SDSend
        lcall SDRead
        cjne  R2, #01h, CMD0Error
        sjmp  CMD0Done
        CMD0Error:
            mov   R1, #80h
            lcall SDSend
            djnz  R3, CMD0Loop

            mov   P1, #00h
            lcall SDError
    CMD0Done:
        mov   R1, #80h
        lcall SDSend
        ret

ACMD41:
; This subroutine sends the ACMD41 command to the SD card.  It makes use of
; SDSend and SDRead to accomplish this.  If this command fails, the LED's
; on P1 will show 0x41.  If this command succeeds, the LED's will be left
; as 0xFF - as they were at start-up.

    mov   R3, #0xFF
    ACMD41Loop:
        mov   R1, #8Ch
        lcall SDSend
```

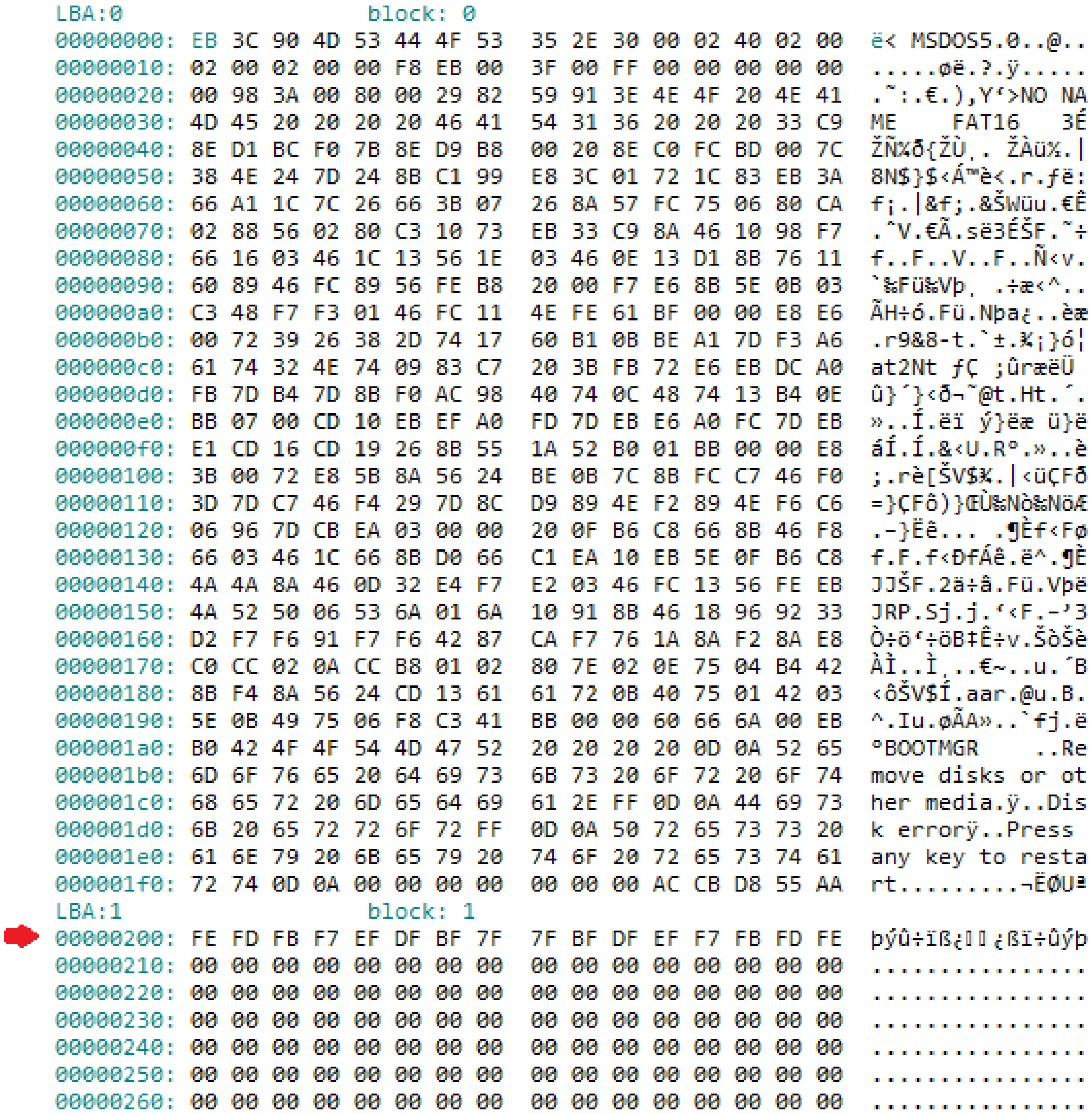
```
        lcall SDRead
        mov   R1, #80h
        lcall SDSend
        mov   R1, #92h
        lcall SDSend
        lcall SDRead
        cjne  R2, #00h, ACMD41Error
        sjmp  ACMD41Done
        ACMD41Error:
            mov   R1, #80h
            lcall SDSend
            djnz  R3, ACMD41Loop

            mov   P1, #41h
            lcall SDError
    ACMD41Done:
        mov   R1, #80h
        lcall SDSend
        ret

CMD17mod:
; This subroutine sends the CMD17 command to the SD card.  The address
; argument it uses is the values in R7, R6, R5, R4.  It saves the read
; values to the buffer space 0x100 to 0x2FF within RAM.  It makes use of
; SDSend and SDRead to accomplish this.  If this command fails, the LED's
; on P1 will show 0x17.  If this command succeeds, the LED's will be left
; as 0xFF - as they were at start-up.


    mov   dptr,  #0x0098
    mov   R0,    #40h
    movx  a,     @dptr
    mov   @R0,   a
    inc   R0

    mov   a,     R7
    mov   @R0, a
    inc   R0

    mov   a,     R6
    mov   @R0, a
    inc   R0

    mov   a,     R5
    mov   @R0, a
    inc   R0

    mov   a,     R4
    mov   @R0, a
    inc   R0

    mov   dptr,  #0x009D
    movx  a,     @dptr
    mov   @R0,   a
```

```
    mov   R3, #0xFF
    CMD17Loop:
        lcall SDSendScratch
        lcall SDRead
        cjne  R2, #00h, CMD17Error
        sjmp  CMD17Done
        CMD17Error:
            mov   R1, #80h
            lcall SDSend
            djnz  R3, CMD17Loop

            mov   P1, #17h
            lcall SDError
    CMD17Done:
; This part of CMD17 reads data in from the SD card.  If this fails, the P1
; LED's will show 0xF0
        lcall SDRead
        cjne  R2, #0xFE, ReadError

        clr   01h
        mov   R4, #01h
        mov   R5, #00h

        mov   R7, #00h
        mov   R6, #02h
        ReadLoop:
            mov   dptr,  #0xFE04
            mov   a,     #0xFF
            lcall SDSendByte
            mov   dptr,  #0xFE04
            movx  a,     @dptr
            jb    01h,   modSkip
            mov   b,     a
            mov   a,     #30h
            add   a,     R5
            mov   R0,    a
            mov   a,     b
            mov   @R0,   a
            modSkip:
            inc   R5
            cjne  R5, #10h, modSkip2
            setb  01h
            modSkip2:
            cjne  R5, #00h, ReadContinue
            inc   R4
            ReadContinue:
                djnz  R7, ReadLoop
                djnz  R6, ReadLoop
            ret
        ReadError:
            mov   P1, #0xF0
            ljmp  SDError
```

```
SDError:
    sjmp  SDError

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;                                                                     ;;;
;;;             SD Card Support Subroutines Written Above               ;;;
;;;                                                                     ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Once this code is loaded onto and running on the 8051 system, we need to set up and attach the SD card so that we can see meaningful data on the P1 LED's. This assembly code for SD initialization is geared for SD cards that are under 2GB in size. Higher capacity SD cards may require a slightly different initialization process.

Once you have chosen your SD card, plug it into your computer and run the DMDE hex editor software provided on the course website. From the **Drive** drop-down menu, select **Select Drive**. Then choose the SD card to open. You should be able to see the raw data on the SD card in hexadecimal format. The data is separated into blocks of 512 bytes. Your screen should be similar to that of the image below.

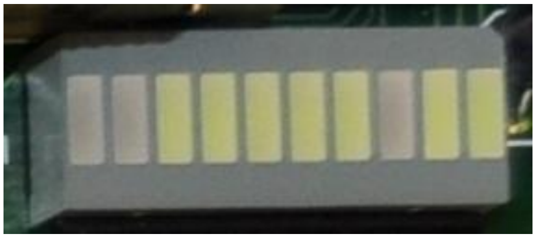


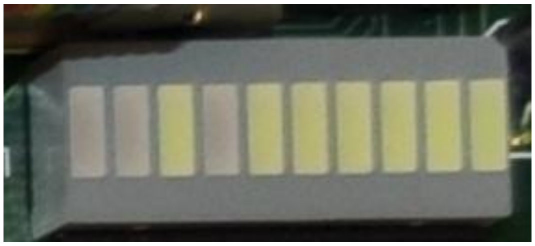
```
LBA:0                   block: 0
00000000: EB 3C 90 4D 53 44 4F 53  35 2E 30 00 02 40 02 00  ë< MSDOS5.0..@..
00000010: 02 00 02 00 00 F8 EB 00  3F 00 FF 00 00 00 00 00  .....øë.?.ÿ.....
00000020: 00 98 3A 00 80 00 29 82  59 91 3E 4E 4F 20 4E 41  .˜:.€.)‚Y‘>NO NA
00000030: 4D 45 20 20 20 20 46 41  54 31 36 20 20 20 33 C9  ME    FAT16   3É
00000040: 8E D1 BC F0 7B 8E D9 B8  00 20 8E C0 FC BD 00 7C  ŽÑ¼ð{ŽÙ¸. ŽÀü½.|
00000050: 38 4E 24 7D 24 8B C1 99  E8 3C 01 72 1C 83 EB 3A  8N$}$‹Á™è<.r.ƒë:
00000060: 66 A1 1C 7C 26 66 3B 07  26 8A 57 FC 75 06 80 CA  f¡.|&f;.&ŠWüu.€Ê
00000070: 02 88 56 02 80 C3 10 73  EB 33 C9 8A 46 10 98 F7  .ˆV.€Ã.së3ÉŠF.˜÷
00000080: 66 16 03 46 1C 13 56 1E  03 46 0E 13 D1 8B 76 11  f..F..V..F..Ñ‹v.
00000090: 60 89 46 FC 89 56 FE B8  20 00 F7 E6 8B 5E 0B 03  `‰Fü‰Vþ¸ .÷æ‹^..
000000a0: C3 48 F7 F3 01 46 FC 11  4E FE 61 BF 00 00 E8 E6  ÃH÷ó.Fü.Nþa¿..èæ
000000b0: 00 72 39 26 38 2D 74 17  60 B1 0B BE A1 7D F3 A6  .r9&8-t.`±.¾¡}ó¦
000000c0: 61 74 32 4E 74 09 83 C7  20 3B FB 72 E6 EB DC A0  at2Nt ƒÇ ;ûræëÜ 
000000d0: FB 7D B4 7D 8B F0 AC 98  40 74 0C 48 74 13 B4 0E  û}´}‹ð¬˜@t.Ht.´.
000000e0: BB 07 00 CD 10 EB EF A0  FD 7D EB E6 A0 FC 7D EB  »..Í.ëï ý}ëæ ü}ë
000000f0: E1 CD 16 CD 19 26 8B 55  1A 52 B0 01 BB 00 00 E8  áÍ.Í.&‹U.R°.»..è
00000100: 3B 00 72 E8 5B 8A 56 24  BE 0B 7C 8B FC C7 46 F0  ;.rè[ŠV$¾.|‹üÇFð
00000110: 3D 7D C7 46 F4 29 7D 8C  D9 89 4E F2 89 4E F6 C6  =}ÇFô)}ŒÙ‰Nò‰NöÆ
00000120: 06 96 7D CB EA 03 00 00  20 0F B6 C8 66 8B 46 F8  .–}Ëê... .¶Èf‹Fø
00000130: 66 03 46 1C 66 8B D0 66  C1 EA 10 EB 5E 0F B6 C8  f.F.f‹ÐfÁê.ë^.¶È
00000140: 4A 4A 8A 46 0D 32 E4 F7  E2 03 46 FC 13 56 FE EB  JJŠF.2ä÷â.Fü.Vþë
00000150: 4A 52 50 06 53 6A 01 6A  10 91 8B 46 18 96 92 33  JRP.Sj.j.‘‹F.–’3
00000160: D2 F7 F6 91 F7 F6 42 87  CA F7 76 1A 8A F2 8A E8  Ò÷ö‘÷öB‡Ê÷v.ŠòŠè
00000170: C0 CC 02 0A CC B8 01 02  80 7E 02 0E 75 04 B4 42  ÀÌ..Ì¸..€~..u.´B
00000180: 8B F4 8A 56 24 CD 13 61  61 72 0B 40 75 01 42 03  ‹ôŠV$Í.aar.@u.B.
00000190: 5E 0B 49 75 06 F8 C3 41  BB 00 00 60 66 6A 00 EB  ^.Iu.øÃA»..`fj.ë
000001a0: B0 42 4F 4F 54 4D 47 52  20 20 20 20 0D 0A 52 65  °BOOTMGR    ..Re
000001b0: 6D 6F 76 65 20 64 69 73  6B 73 20 6F 72 20 6F 74  move disks or ot
000001c0: 68 65 72 20 6D 65 64 69  61 2E FF 0D 0A 44 69 73  her media.ÿ..Dis
000001d0: 6B 20 65 72 72 6F 72 FF  0D 0A 50 72 65 73 73 20  k errorÿ..Press 
000001e0: 61 6E 79 20 6B 65 79 20  74 6F 20 72 65 73 74 61  any key to resta
000001f0: 72 74 0D 0A 00 00 00 00  00 00 00 AC CB D8 55 AA  rt.........¬ËØUª
LBA:1                   block: 1
00000200: FE FD FB F7 EF DF BF 7F  7F BF DF EF F7 FB FD FE  þýû÷ïß¿▯▯¿ßï÷ûýþ
00000210: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000220: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000230: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000240: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000250: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000260: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

As you can see, the line indicated by the red arrow has already been edited for this example. In order to change the first 16 bytes of block one, you need to select **Edit Mode** from the **Edit** drop-down menu. This will enable you to select the bytes you want to edit and type in their desired values.

When you are finished entering the desired byte values, select **Apply Changes** from the **Drive** drop-down menu. Then, you can exit the software and remove your SD card.

Plug your SD card into the SD card socket and run the code that you loaded onto your 8051 system. You should be able to see the HEX values that you loaded onto the SD card flashing sequencially on the P1 LED's of the R31-JP.





6 EmFile Example

To demonstrate this 8000 Series Swiss Army Knife EmFile component, we will read in data from a file called “PATTERN.txt” stored within an SD card. We will then sequencially flash the 8051’s P1 LED’s with the 16 different one-byte values that were read from the file on the SD card.

After wiring the 8000 series communication portion of this chip to the 8051, all that is left to do is wire the SD card to the EmFile SPI port. The easiest way to do this is to use a breakout board like the one found at http://store.linksprite.com/sd-card-breakout-board/. This breakout board is labeled with VDD, GND, and all four of the SPI connections. Since this board handles all of the necessary level shifting, we can attach VDD to 5VDC and wire the SPI connections directly to the EmFile SPI port on the PSoC. Once everything is wired together, assemble and load the 8051 system with the following assembly code:

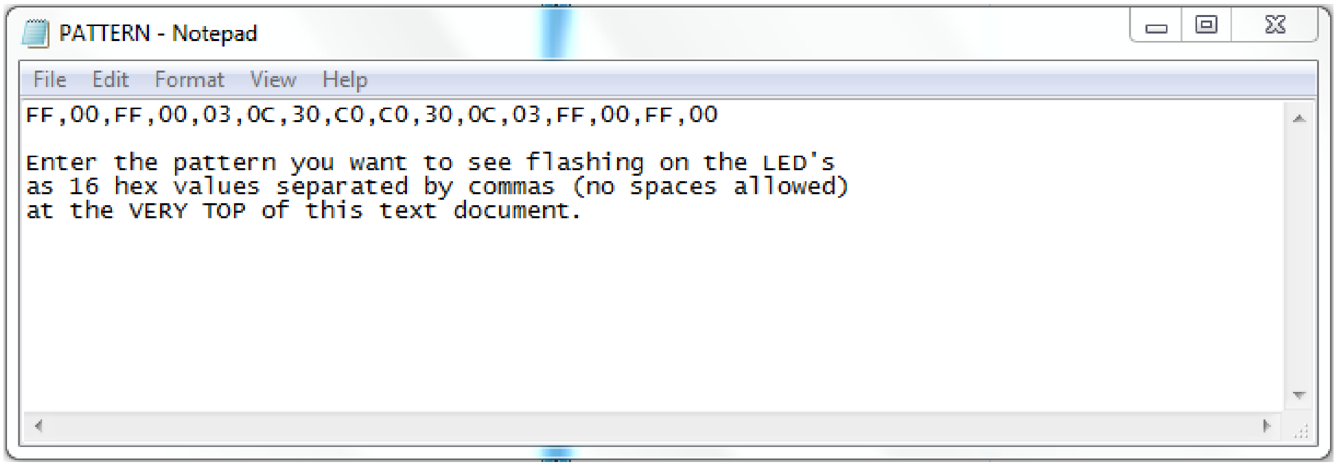
```
.org   000h
ljmp   start

.org   003h
ljmp   isr

.org   080h
.db    0x50, 0x41, 0x54, 0x54, 0x45, 0x52, 0x4E, 0x2E, 0x74, 0x78, 0x74

; These bytes are ascii for the file name to be read "Pattern.txt"
;================================================================

.org   100h
start:
    lcall init
main:
    mov   R0,  #30h
    mov   R1,  #10h
    mainLoop:
        mov   a,  @R0
        mov   P1, a

        mov    R2, #00h            ; Wait a second or two
        mov    R3, #00h
        mov    R4, #02h
        delay:
            djnz   R2, delay
            djnz   R3, delay
            djnz   R4, delay

        inc   R0
        djnz  R1, mainLoop
    sjmp  main

;================================================================

init:
    mov   tcon, #01h       ; Enable edge triggered interrupts

    mov   IE,   #81h       ; Set the global interrupt enable flag

    mov   dptr, #0xFE0D    ; Set up PSoC EmFile to accept a file name and
                           ; enable the PSoC external interrupt
    mov   a, #0x81
    clr   00h
    movx  @dptr, a

    lcall  hold

    mov   R0,   #80h
    mov   R1,   #0Bh
```

```
    nameSend:
    ; This sends the name of the file to be read to the SD card -
    ; "PATTERN.txt"
        mov   dpl,  R0
        mov   dph,  #00h
        movx  a,    @dptr

        mov   dptr, #0xFE0E
        clr   00h
        movx  @dptr, a

        lcall  hold

        inc    R0
        djnz   R1, nameSend

    mov   dptr, #0xFE0D    ; Set up PSoC EmFile to read data
    mov   a, #0x83
    clr   00h
    movx  @dptr, a

    lcall  hold

    mov   R0,   #30h
    mov   R1,   #10h
    dataRead:
; This reads the relevant data from the SD card and stuffs it into the
; scratch-pad memory starting at 0x30
        mov    dptr, #0xFE0E
        mov    a,    #0xFF
        clr    00h
        movx   @dptr, a

        lcall  hold

        mov    dptr,  #0xFE0E     ; Reads in the first byte of a typed hex
                                  ; number
        movx   a,     @dptr
        lcall  ascbin
        swap   a
        mov    R2,    a

        mov    dptr, #0xFE0E
        mov    a,    #0xFF
        clr    00h
        movx   @dptr, a

        lcall  hold

        mov    dptr,  #0xFE0E     ; Reads in the second byte of a typed hex
                                  ; number
        movx   a,     @dptr
        lcall  ascbin
        orl    a,     R2
```

```
        mov    @R0,   a          ; Stores the byte into the scratchpad
                                 ; memory
        inc    R0

        mov    dptr, #0xFE0E     ; Ignores the following byte to ignore
                                 ; the commas
        mov    a,    #0xFF
        clr    00h
        movx   @dptr, a

        lcall  hold

        djnz   R1,   dataRead

    mov   dptr, #0xFE0D          ; Closes the file
    mov   a, #0x84
    clr   00h
    movx  @dptr, a

    lcall  hold

    ret

;================================================================

hold:
    jnb    00h, hold
    ret

;================================================================

isr:
    mov   dptr, #0xFE0D         ; Clear the PSoC EmFile flag thus clearing
                                ; the external interrupt
    mov   a, #0x80
    movx  @dptr, a

    setb  00h

    reti                        ; Return

;================================================================
; Subroutine Ascbin
; This routine takes the ascii character passed to it in the
; acc and converts it to a 4 bit binary number which is returned
; in the acc.  If an error occurs, 0 will be stuffed in the acc.
;================================================================
ascbin:
    clr   c
    add   a,  #0d0h        ; if chr < 30 then error
    jnc   notnum
    clr   c                ; check if chr is 0-9
    add   a,  #0f6h        ; adjust it
```

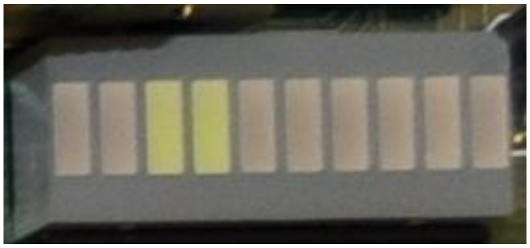
```
    jc    hextry            ; jmp if chr not 0-9
    add   a,  #0ah          ; if it is then adjust it
    ret
    hextry:
    clr   acc.5             ; convert to upper
    clr   c                 ; check if chr is a-f
    add   a,  #0f9h         ; adjust it
    jnc   notnum            ; if not a-f then error
    clr   c                 ; see if char is 46 or less.
    add   a,  #0fah         ; adjust acc
    jc    notnum            ; if carry then not hex
    anl   a,  #0fh          ; clear unused bits
    ret
notnum:
    mov   a,  #00h
    ret
```

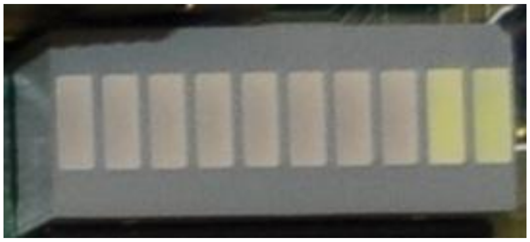
Once this code is loaded onto and running on the 8051 system, we need to set up and attach the SD card so that we can see meaningful data on the P1 LED's. Plug the SD card into your computer and create a new text document with the name "PATTERN" in the root directory of your SD card. On the first line of your "PATTERN.txt" document, enter 16 hexadecimal values separated by commas. What comes after that is not important and is ignored by the PSoC and 8051 program. Therefore, your text document can look something like this.



```
FF,00,FF,00,03,0C,30,C0,C0,30,0C,03,FF,00,FF,00

Enter the pattern you want to see flashing on the LED's
as 16 hex values separated by commas (no spaces allowed)
at the VERY TOP of this text document.
```

Plug your SD card into the SD card socket and run the code that you loaded onto your 8051 system. You should be able to see the HEX values that you loaded into the "PATTERN.txt" file flashing sequencially on the P1 LED's of the R31-JP.

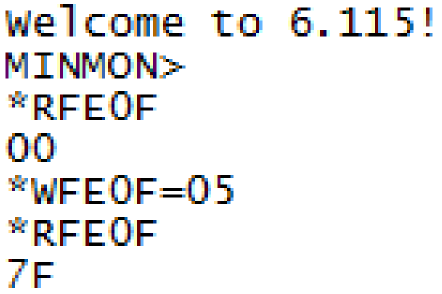




6 ADC Example

To demonstrate this 8000 Series Swiss Army Knife ADC component, we will use MINMON to verify the operation of the analog to digital converter.

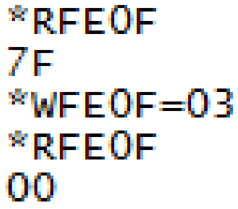
After wiring the 8000 series communication portion of this chip to the 8051, all that is left to do is connect the analog voltage to be measured to P0.1 of the PSoC. For this example, we will attach a 1000 ohm resistor between ADC IN and 5VDC and another 1000 ohm resistor between ADC IN and GND. This should keep the ADC IN pin held at roughly 2.5 volts. It is recommended, though not necessary, that you attach a .1 uF capacitance between P0.2 on the PSoC and ground. P0.2 is the Bypass pin for the analog to digital converter. Placing a capacitor on this pin can reduce the noise in the signals we read in through the ADC. Once everything is wired together, run MINMON on your 8051 system and enter the following commands:



```
Welcome to 6.115!
MINMON>
*RFE0F
00
*WFE0F=05
*RFE0F
7F
```

Here, we can verify the reset value of the ADC register by reading from 0xF before writing anything to it. Then, we write some value – the value is not important as long as it is neither 0x00 nor 0x01. Writing this value – in our case, 0x05 – begins a conversion. To get the result of the conversion, we subsequently read from the ADC register. We can see that the ADC is working properly by the fact that it returns a result (0x7F) which corresponds to the 2.5 volts presented at its input by the resistor divider.

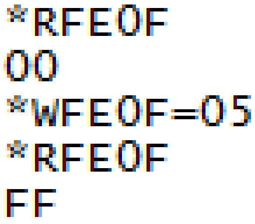
Next, I removed the resistor pulling the ADC input to 5VDC. This means that the ADC IN pin is at 0 volts. Again, we run the following commands:



```
*RFE0F
7F
*WFE0F=03
*RFE0F
00
```

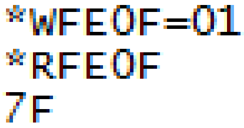
By reading the value of the ADC register (0xF) before writing anything else, we can see that the value stored in the ADC register does not change until write to the ADC register. We can also see that it does not matter what value is written to the ADC to start a conversion without interrupts as long as the value is neither 0x00 nor 0x01. In this case, 0x03 was written to 0xF to begin the conversion. As expected, the value returned from the conversion is 0x00.

Next, I repeat the previous experiment, except I re-insert the resistor pulling the ADC IN pin to 5VDC and remove the resistor pulling the ADC IN pin to GND. The behavior of the ADC run from MINMON is exactly as expected.



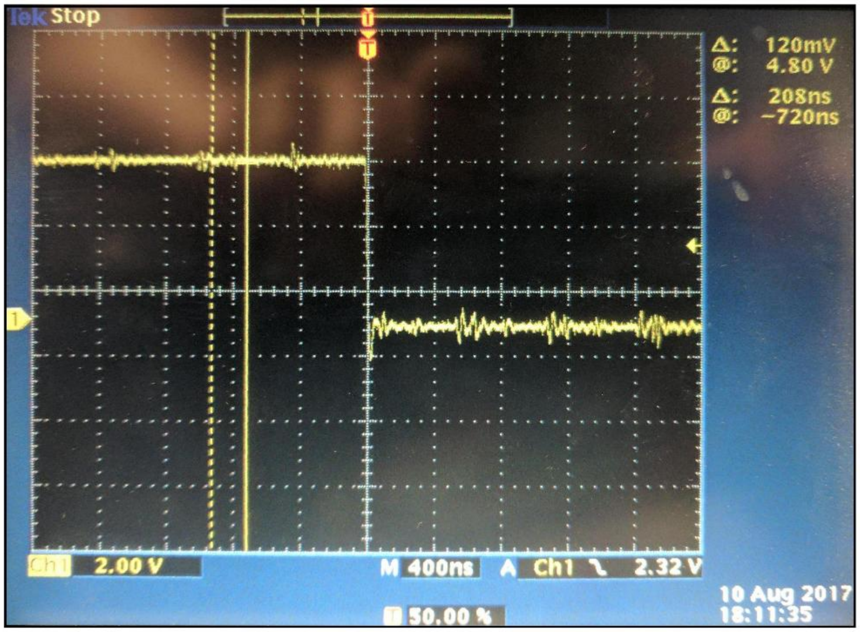
```
*RFE0F
00
*WFE0F=05
*RFE0F
FF
```

Lastly, we'll test the interrupt of the ADC. For this test, I placed both resistors in their original configuration holding the ADC IN pin at roughly 2.5 volts. To begin an analog to digital conversion enabling the falling edge interrupt to the 8051 on the INTR line, we need to write 0x01 to the ADC register (0xF).



```
*WFE0F=01
*RFE0F
7F
```

Here we can see that the analog to digital conversion worked properly. The only difference between writing 0x01 to the ADC and writing any other greater value to the ADC is that writing 0x01 enables the falling edge interrupt on the INTR line. Upon completion of this analog to digital conversion, the INTR line drops low.



Writing 0x00 to the ADC register clears the interrupt and does not begin a conversion. You can see that when the write command is issued, the INTR line jumps high again.

```
*WFE0F=00
```