

```

***** ##HEADER-START## *****/
;
; ORIGINAL RELEASE DATE & AUTHOR:
; 07-11-03 M. Karas
;
; CURRENT RELEASE NUMBER: 1
;
; SOFTWARE ID: 1.3
;
; FILENAME: loader.a51
;
; DATE OF LATEST CHANGE: 7/16/03
;
; ABSTRACT:
; In application FLASH Reloader Module
;
; DESCRIPTION:
; This file contains all code associated with a program loader
; function which takes control of the processor power on start
; time and determines if a startup re-programming request is
; being requested via a special 8 byte handshake that is sent to
; here via a firmware re-programming tool running on a PC
; computer connected to the UART port of the processor. If no
; loader request is observed then this loader will cease operation
; and control will be passed onto the base of the target application
; program.
;
; Also included within this module are interrupt handlers for the
; download UART port which are used while the loader is in use.
; A physical to logical interrupt re-vectoring table is present
; which re-targets most of the hardware interrupts to the normal
; C program vector table. Any interrupt shared for use between the
; loader and main application code vectors to a local code block
; here and uses a global flag to control the steering of the
; interrupt. the UART interrupt is used in this manner. The global
; flag that manages the interrupt steering is the F1 bit in the
; PSW.1 bit. This comes up cleared out of reset and after the loader
; completes this bit is set to 1 to force shared interrupts to be
; steered off to the C applications code.
;
; The product has support for two UARTs and the selection of which
; UART is used is based upon the assembly time equate UART which is
; set either to a Zero value to select UART 0 or to a NonZero value
; to select UART1.
;
; Also present here is a command packet interperter that supports
; a rudimentary protocol with a PC host to support hardware
; parameter initializations in the system EEPROM, These commands
; will be simple commands that are built as an extension of the
; FLASH programming and verification commands.
;
; CHANGE LOG:
; DATE AUTHOR SDR DESCRIPTION
; 11-jul-03 M.Karas ---- Initial implementation to support
; FLASH re-programming for the Cygnal F126
; via the UART serial port. The initial
; implementation also has support for access
; to the EEPROMU.
***** ##HEADER-END## *****/
;
; $NOMOD51
; $NOCOND
; NAME ?LOADER
;
; #include <C8051F120.h> ; register definition file.
;
; UART_SEL EQU 1 ; set 0 for loader on UART0
; ; !0 for loader on UART1
;
; ;
; ; define constants used within the UART driver routine
; ;
; QUE_SIZ EQU 0x10 ; size of the sio queues
; ; must be power of 2 for index masking
; IDX_MSK EQU 0x0F ; mask must match queue size -1
; IF (UART_SEL = 0)
; UART_DIV EQU 65524 ; uart0 divisor using timer 3
; ; with sysclk @ 22118400 Hz

```

```

; div=(65536-(sysclk/(16*baudrate)))
; div=(65536-(22118400/(16*115200)))
ELSE
  UART_DIV EQU 248 ; uart1 divisor using timer 1
; with sysclk @ 221184 Hz
; div=(256-(sysclk/(12*2*baudrate)))
; div=(256-(22118400/(12*2*115200)))
ENDIF

;
;
; define constants used within the TIMRO driver routine
;
TIMRO_DIV EQU 43418 ; timer 0 divisor using SysClk @ 22118400 Hz
; div=(65536-(22118400/(1000)))
;
;
; define the constants associated with the packet handler.
;
PACKET_SYNC EQU 0xAA ; packet lead in byte
PACKET_ECHO EQU 0x00 ; ID for Echo packet
PACKET_ACK EQU 0x01 ; ID for ACK packet
PACKET_NACK EQU 0x02 ; ID for NACK packet
PACKET_LSIZ EQU 0x03 ; ID for LSIZE packet
PACKET_READ EQU 0x04 ; ID for READ packet
PACKET_ERAS EQU 0x05 ; ID for ERASE packet
PACKET_WRIT EQU 0x06 ; ID for WRITE packet
PACKET_EEWR EQU 0x07 ; ID for EE_WRITE packet
PACKET_EERD EQU 0x08 ; ID for EE_READ packet
;
PACKET_MIN EQU 5 ; minimum with <sync><id><len><crch><crcl>
PACKET_MAX EQU PACKET_MIN+35 ; maximum length of a packet with 32 bytes data
; ; and 3 bytes address

;
; define the offset that is used to locate up the normal application code beyond
; the end of the loader. This offset is programmed in multiples of the Cygnal
; 2K byte flash bank size.
;
APP_OFFSET equ 0x1000
;
;
; Physical reset and interrupt vector table
;
CSEG AT 0x0000 ; Hardware Reset Vector
LJMP LOADER_BASE ; go to the loader entry point
;
CSEG AT 0x0003 ; Intr # 0 - External Interrupt 0 (/INT0)
LJMP $+APP_OFFSET
;
CSEG AT 0x000B ; Intr # 1 - Timer 0 Overflow
LJMP TIMRO_ISR ; to local interrupt handler
; unless loader function is not active
;
CSEG AT 0x0013 ; Intr # 2 - External Interrupt 1 (/INT1)
LJMP $+APP_OFFSET
;
CSEG AT 0x001B ; Intr # 3 - Timer 1 Overflow
LJMP $+APP_OFFSET
;
CSEG AT 0x0023 ; Intr # 4 - UART0
IF (UART_SEL = 0)
LJMP UART_ISR ; to local loader interrupt handler
; unless loader function is not active
ELSE
LJMP $+APP_OFFSET
ENDIF
;
CSEG AT 0x002B ; Intr # 5 - Timer 2
LJMP $+APP_OFFSET
;
CSEG AT 0x0033 ; Intr # 6 - Serial Peripheral Interface
LJMP $+APP_OFFSET
;
CSEG AT 0x003B ; Intr # 7 - SMBus Interface
LJMP SMBus_ISR ; to local handler for interrupt. This
; stays active as application runs and
; application can use SMBus driver code.

```

```

;
CSEG    AT 0x0043                ; Intr # 8 - ADC0 Window Comparator
LJMP    $+APP_OFFSET
;
CSEG    AT 0x004B                ; Intr # 9 - PCA 0
LJMP    $+APP_OFFSET
;
CSEG    AT 0x0053                ; Intr # 10 - Comparator 0
LJMP    $+APP_OFFSET
;
CSEG    AT 0x005B                ; Intr # 11 - Comparator 0
LJMP    $+APP_OFFSET
;
CSEG    AT 0x0063                ; Intr # 12 - Comparator 1
LJMP    $+APP_OFFSET
;
CSEG    AT 0x006B                ; Intr # 13 - Comparator 1
LJMP    $+APP_OFFSET
;
CSEG    AT 0x0073                ; Intr # 14 - Timer 3
LJMP    $+APP_OFFSET
;
CSEG    AT 0x007B                ; Intr # 15 - ADC0 End of Conversion
LJMP    $+APP_OFFSET
;
CSEG    AT 0x0083                ; Intr # 16 - Timer 4
LJMP    $+APP_OFFSET
;
CSEG    AT 0x008B                ; Intr # 17 - ADC2 Window Comparator
LJMP    $+APP_OFFSET
;
CSEG    AT 0x0093                ; Intr # 18 - ADC2 End of Conversion
LJMP    $+APP_OFFSET
;
CSEG    AT 0x00A3                ; Intr # 20 - UART1
IF (UART_SEL = 0)
LJMP    $+APP_OFFSET
ELSE
LJMP    UART_ISR                ; to local loader interrupt handler
                                ; unless loader function is not active
ENDIF
;
;
; define the bit memory flag that is used by the loader to
; control the interrupt direction switching. This is placed
; as the first bit BIT memory area. The linker will fit
; the other application bit variables around this bit. This also
; allocates a number of bits used by the SMBus interface routines
;
BSEG    AT 0x20.0
LDR_FLAG: DBIT    1                ; interrupt vector switching. Cleared
                                ; for application mode. set for loader mode.
SM_RW:    DBIT    1                ; R/W command bit. 1=READ, 0=WRITE
SM_BUSY:  DBIT    1                ; SMBus Busy flag (kept in software)
;
;
; define the global memory locations used by the loader for
; support of the SMBus driver to communicate with the EEPROM
;
DSEG    AT 0x30
EE_XMIT: DS      1                ; holds the byte to transmit to EEPROM
EE_RECV: DS      1                ; holds byte received from EEPROM
EE_ADDR: DS      1                ; holds the address of the EEPROM
;
;
; define the local memory locations used by the loader.
; these include those for the UART serial interrupt
; handler. Note that all of the loader data allocations are
; handled by direct address equates so that the memory in use
; can be completely reclaimed by the normal applications program
; after the loader has completed all of its functions. The
; normal data allocations are determined by the linker.
;
;
RAM_PTR SET $                    ; base of locally used DATA RAM after above
                                ; hard allocated DSEG data
;
; UART handler variables
;
RX_EMPTY EQU RAM_PTR            ; receive buffer empty flag

```

```

RAM_PTR    SET RAM_PTR+1
;
RX_WR_IDX  EQU RAM_PTR          ; receive buffer write index
RAM_PTR    SET RAM_PTR+1
;
RX_RD_IDX  EQU RAM_PTR          ; receive buffer read index
RAM_PTR    SET RAM_PTR+1
;
TX_EMPTY   EQU RAM_PTR          ; transmit buffer empty flag
RAM_PTR    SET RAM_PTR+1
;
TX_WR_IDX  EQU RAM_PTR          ; transmit buffer write index
RAM_PTR    SET RAM_PTR+1
;
TX_RD_IDX  EQU RAM_PTR          ; transmit buffer read index
RAM_PTR    SET RAM_PTR+1
;
; TIMER handler variables
;
TIM_CNT    EQU RAM_PTR          ; timer interrupt divide by 10 counter
RAM_PTR    SET RAM_PTR+1
;
TIMER1     EQU RAM_PTR          ; software timer 1 @ 100 Hz
RAM_PTR    SET RAM_PTR+1
;
TIMER2     EQU RAM_PTR          ; software timer 2 @ 100 Hz
RAM_PTR    SET RAM_PTR+1
;
TIMER3     EQU RAM_PTR          ; software timer 3 @ 100 Hz
RAM_PTR    SET RAM_PTR+1
;
TIMER4     EQU RAM_PTR          ; software timer 4 @ 100 Hz
RAM_PTR    SET RAM_PTR+1
;
; wake up sequence variables
;
WAKE_CNT   EQU RAM_PTR          ; counter for the wakeup bytes received
RAM_PTR    SET RAM_PTR+1
;
; packet handler variables
;
R_STATE    EQU RAM_PTR          ; receive packet state byte
RAM_PTR    SET RAM_PTR+1
;
R_CNT      EQU RAM_PTR          ; receive packet count so far
RAM_PTR    SET RAM_PTR+1
;
R_LENGTH   EQU RAM_PTR          ; receive packet length from host
RAM_PTR    SET RAM_PTR+1
;
; loader buffers (Note these accessed indirectly only as may very well
; extend up to addresses over the 0x7F directly addressable boundary
;
RX_BUFFER  EQU RAM_PTR          ; uart receive buffer queue
RAM_PTR    SET RAM_PTR+QUE_SIZ
;
TX_BUFFER  EQU RAM_PTR          ; uart transmit buffer queue
RAM_PTR    SET RAM_PTR+QUE_SIZ
;
R_PACKET   EQU RAM_PTR          ; receive packet buffer
RAM_PTR    SET RAM_PTR+PACKET_MAX
;
S_PACKET   EQU RAM_PTR          ; send packet buffer
RAM_PTR    SET RAM_PTR+PACKET_MAX
;
;
; setup the stack to grow upwards from the end of the allocated data
;
LDR_STACK  EQU RAM_PTR
;
;
;
;
; setup the base of the actual loader module
;
CSEG      AT      0x100          ; start our actual program code here
;
;
; subroutine to perform configuration of the target hardware and
; SFR initialization. Note: This code superceeds the code from the

```

```
; standard CONFIG function normally produced with the Cygnal configuration
; wizard and as such if changes are made they need to be done here as opposed
; to in the configuration wizard.
```

```
CONFIG:
```

```
; Watchdog Timer Configuration
```

```
; WDTCN.[7:0]: WDT Control
```

```
; Writing 0xA5 enables and reloads the WDT.
; Writing 0xDE followed within 4 clocks by 0xAD disables the WDT
; Writing 0xFF locks out disable feature.
```

```
; WDTCN.[2:0]: WDT timer interval bits
```

```
; NOTE! When writing interval bits, bit 7 must be a 0.
```

```
; Bit 2 | Bit 1 | Bit 0
```

```
-----
; 1 | 1 | 1 Timeout interval = 1048576 x Tsysclk
; 1 | 1 | 0 Timeout interval = 262144 x Tsysclk
; 1 | 0 | 1 Timeout interval = 65636 x Tsysclk
; 1 | 0 | 0 Timeout interval = 16384 x Tsysclk
; 0 | 1 | 1 Timeout interval = 4096 x Tsysclk
; 0 | 1 | 0 Timeout interval = 1024 x Tsysclk
; 0 | 0 | 1 Timeout interval = 256 x Tsysclk
; 0 | 0 | 0 Timeout interval = 64 x Tsysclk
-----
```

```
MOV WDTCN, #07H ; Watchdog Timer Control Register
MOV WDTCN, #0DEH ; Disable WDT
MOV WDTCN, #0ADH
```

```
CROSSBAR REGISTER CONFIGURATION
```

```
; NOTE: The crossbar register should be configured before any
; of the digital peripherals are enabled. The pinout of the
; device is dependent on the crossbar configuration so caution
; must be exercised when modifying the contents of the XBR0,
; XBR1, and XBR2 registers. For detailed information on
; Crossbar Decoder Configuration, refer to Application Note
; AN001, "Configuring the Port I/O Crossbar Decoder".
```

```
; The Crossbar configuration results in the
; following port pinout assignment:
```

```
Port 0
; P0.0 = UART0 TX (Push-Pull Output)(Digital)
; P0.1 = UART0 RX (Open-Drain Output/Input)(Digital)
; P0.2 = SPI Bus SCK (Push-Pull Output)(Digital)
; P0.3 = SPI Bus MISO (Open-Drain Output/Input)(Digital)
; P0.4 = SPI Bus MOSI (Push-Pull Output)(Digital)
; P0.5 = SMBus SDA (Open-Drain Output/Input)(Digital)
; P0.6 = SMBus SCL (Open-Drain Output/Input)(Digital)
; P0.7 = UART1 TX (Push-Pull Output)(Digital)
```

```
Port 1
; P1.0 = UART1 RX (Open-Drain Output/Input)(Digital)
; P1.1 = /INT0 (Open-Drain Output/Input)(Digital)
; P1.2 = /INT1 (Open-Drain Output/Input)(Digital)
; P1.3 = T2 (Push-Pull Output)(Digital)
; P1.4 = T2EX (Open-Drain Output/Input)(Digital)
; P1.5 = GP I/O (Open-Drain Output/Input)(Digital)
; P1.6 = GP I/O (Open-Drain Output/Input)(Digital)
; P1.7 = GP I/O (Open-Drain Output/Input)(Digital)
```

```
Port 2
; P2.0 = GP I/O (Open-Drain Output/Input)(Digital)
; P2.1 = GP I/O (Open-Drain Output/Input)(Digital)
; P2.2 = GP I/O (Open-Drain Output/Input)(Digital)
; P2.3 = GP I/O (Open-Drain Output/Input)(Digital)
; P2.4 = GP I/O (Open-Drain Output/Input)(Digital)
; P2.5 = GP I/O (Open-Drain Output/Input)(Digital)
; P2.6 = GP I/O (Open-Drain Output/Input)(Digital)
; P2.7 = GP I/O (Open-Drain Output/Input)(Digital)
```

```
Port 3
; P3.0 = GP I/O (Push-Pull Output)(Digital)
; P3.1 = GP I/O (Push-Pull Output)(Digital)
; P3.2 = GP I/O (Push-Pull Output)(Digital)
; P3.3 = GP I/O (Push-Pull Output)(Digital)
; P3.4 = GP I/O (Open-Drain Output/Input)(Digital)
```

```

; P3.5 = GP I/O          (Push-Pull Output)(Digital)
; P3.6 = GP I/O          (Open-Drain Output/Input)(Digital)
; P3.7 = GP I/O          (Push-Pull Output)(Digital)
;
MOV    SFRPAGE, #CONFIG_PAGE    ; point to the config page
MOV    XBR0, #007H
MOV    XBR1, #074H
MOV    XBR2, #044H
;
; Select Pin I/O
;
; NOTE: Some peripheral I/O pins can function as either inputs or
; outputs, depending on the configuration of the peripheral. By default,
; the configuration utility will configure these I/O pins as push-pull
; outputs.
; Port configuration (1 = Push Pull Output)
;
MOV    SFRPAGE, #CONFIG_PAGE    ; point to the config page
MOV    P0, #0FFH                ; port 0 output latch bits to 1's
MOV    P1, #01FH                ; port 1 latch bits except for output bits
; P1.7 P1.6 & P1.5 which will be a software
; test points
MOV    P2, #0FFH                ; port 2 output latch bits to 1's
MOV    P3, #0FDH                ; all 1's except for P3.1 (LCD_RST/) which
; we hold low
MOV    P4, #0FFH                ; port 4 output latch bits to 1's
MOV    P5, #0FFH                ; port 5 output latch bits to 1's
MOV    P6, #0FFH                ; port 6 output latch bits to 1's
MOV    P7, #0FFH                ; port 7 output latch bits to 1's
;
MOV    POMDOUT, #095H           ; Output configuration for P0
MOV    P1MDOUT, #0F8H           ; Output configuration for P1
; (P1.7-6-5 & P1.3 are set as outputs)
MOV    P2MDOUT, #000H           ; Output configuration for P2
MOV    P3MDOUT, #0AFH           ; Output configuration for P3
MOV    P4MDOUT, #0C0H           ; Output configuration for P4
; Push Pull for EMIF controls
MOV    P5MDOUT, #0FFH           ; Output configuration for P5
; Push Pull for EMIF
MOV    P6MDOUT, #0FFH           ; Output configuration for P6
; Push Pull for EMIF
MOV    P7MDOUT, #0FFH           ; Output configuration for P7
; Push Pull for EMIF
;
MOV    P1MDIN, #0FFH            ; Input configuration for P1
;
; initialize the external memory inteface
;
MOV    SFRPAGE, #000H
MOV    EMI0CF, #038H            ; External Memory Configuration Register
; (Internal / external split mode)
MOV    EMI0TC, #045H            ; External Memory Timing
; (SYSCLK=45.2 ns.
; AddSet=1 clk PulWid=2 clk AddHold=1 clk)
;
; Comparators Register Configuration
;
; Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0
; -----
; R/W   |   R   |  R/W  |  R/W  |  R/W  |  R/W  |  R/W  |  R/W
; -----
; Enable | Output | Rising | Falling | Positive | Negative
;         | State  | Edge   | Edge   | Hysterisis | Hysterisis
;         | Flag  | Int.   | Int.   | 00: Disable | 00: Disable
;         |       | Flag   | Flag   | 01: 5mV    | 01: 5mV
;         |       |       |       | 10: 10mV   | 10: 10mV
;         |       |       |       | 11: 20mV   | 11: 20mV
; -----
;
MOV    SFRPAGE, #CPT0_PAGE
MOV    CPT0MD, #030H            ; Comparator 0 Configuration Register
MOV    CPT0CN, #000H            ; Comparator 0 Control Register
;
MOV    SFRPAGE, #CPT1_PAGE
MOV    CPT1MD, #030H            ; Comparator 1 Configuration Register
MOV    CPT1CN, #000H            ; Comparator 1 Control Register
;
; Oscillator Configuration
;
MOV    SFRPAGE, #CONFIG_PAGE

```

```

MOV    OSCXCN, #067H          ; EXTERNAL Oscillator Control Register
;
MOV    R0, #0                 ; initialize the XTLVLD blanking interval (>1ms)
MOV    R1, #0
CONFIG_1:
INC    R1
CJNE  R1, #00H, CONFIG_2
INC    R0
CONFIG_2:
MOV    A, R1
CPL   A
ORL   A, R0
JNZ   CONFIG_1
;
; wait for xtal osc to start up
CONFIG_3:
MOV    A, OSCXCN              ; wait till OSCXCN.7 becomes 1
JNB   ACC.7, CONFIG_3
;
MOV    CLKSEL, #01H          ; Oscillator Clock Selector
; (Select external osc)
CLR   A                      ; Internal Oscillator Control Register
MOV   OSCICN,A               ; This disables internal oscillator to lower Icc
;
MOV   FLSCL, #080H           ; FLASH Memory Control
MOV   PLL0DIV, #000H         ; PLL pre-divide Register
MOV   PLL0MUL, #01H          ; PLL Clock scaler Register
MOV   PLL0FLT, #031H         ; PLL Filter Register
MOV   PLL0CN, #000H          ; PLL Control Register
;
; Reference Control Register Configuration
;
MOV   SFRPAGE, #ADC0_PAGE
MOV   REFOCN, #000H
;
; ADC0 Configuration
;
MOV   SFRPAGE, #ADC0_PAGE
MOV   AMX0CF, #000H          ; AMUX Configuration Register
MOV   AMX0SL, #000H         ; AMUX Channel Select Register
MOV   ADC0CF, #0F8H         ; ADC Configuration Register
MOV   ADC0CN, #000H         ; ADC Control Register
;
MOV   ADC0LTH, #000H        ; ADC Less-Than High Byte Register
MOV   ADC0LTL, #000H       ; ADC Less-Than Low Byte Register
MOV   ADC0GTH, #0FFH       ; ADC Greater-Than High Byte Register
MOV   ADC0GTL, #0FFH       ; ADC Greater-Than Low Byte Register
;
; ADC2 Configuration
;
MOV   SFRPAGE, #ADC2_PAGE
MOV   AMX2SL, #000H         ; AMUX Channel Select Register
MOV   AMX2CF, #000H         ; AMUX Configuration Register
MOV   ADC2CF, #0F8H         ; ADC Configuration Register
MOV   ADC2LT, #0FFH         ; ADC Less-Than Register
MOV   ADC2GT, #0FFH         ; ADC Greater-Than Register
MOV   ADC2CN, #000H         ; ADC Control Register
;
; DAC Configuration
;
MOV   SFRPAGE, #DAC0_PAGE
MOV   DAC0L, #000H          ; DAC0 Low Byte Register
MOV   DAC0H, #000H          ; DAC0 High Byte Register
MOV   DAC0CN, #000H         ; DAC0 Control Register
;
MOV   SFRPAGE, #DAC1_PAGE
MOV   DAC1L, #000H          ; DAC1 Low Byte Register
MOV   DAC1H, #000H          ; DAC1 High Byte Register
MOV   DAC1CN, #000H         ; DAC1 Control Register
;
; SPI Configuration
;
MOV   SFRPAGE, #SPI0_PAGE
MOV   SPI0CFG, #070H        ; SPI Configuration Register
MOV   SPI0CKR, #002H        ; SPI Clock Rate Register
ORL   SPI0CN, #001H
MOV   SPI0CN, #001H         ; SPI Control Register
;
; UART0 Configuration
;
MOV   SFRPAGE, #UART0_PAGE

```

```

MOV    SADENO, #000H      ; Serial 0 Slave Address Enable
MOV    SADDR0, #000H     ; Serial 0 Slave Address Register
MOV    SSTA0, #000H     ; UART0 Status and Clock Selection Register
MOV    SCON0, #050H     ; Serial Port Control Register
ANL    SCON0, #0FCH     ; clear interrupt pending flags
;
MOV    PCON, #000H      ; Power Control Register
;
; UART1 Configuration
;
MOV    SFRPAGE, #UART1_PAGE
MOV    SCON1, #010H     ; Serial Port 1 Control Register
ANL    SCON1, #0FCH     ; clear interrupt pending flags
;
; SMBus Configuration
;
MOV    SFRPAGE, #SMB0_PAGE
MOV    SMB0CN, #040H    ; SMBus Control Register
MOV    SMB0ADR, #000H   ; SMBus Address Register
MOV    SMB0CR, #000H    ; SMBus Clock Rate Register
;
; PCA Configuration
;
MOV    SFRPAGE, #PCA0_PAGE
MOV    PCA0MD, #000H    ; PCA Mode Register
MOV    PCA0CN, #000H    ; PCA Control Register
MOV    PCA0L, #000H     ; PCA Counter/Timer Low Byte
MOV    PCA0H, #000H     ; PCA Counter/Timer High Byte
;
MOV    PCA0CPM0, #000H  ; PCA Capture/Compare Register 0
MOV    PCA0CPL0, #000H ; PCA Counter/Timer Low Byte
MOV    PCA0CPH0, #000H ; PCA Counter/Timer High Byte
;
MOV    PCA0CPM1, #000H  ; PCA Capture/Compare Register 1
MOV    PCA0CPL1, #000H ; PCA Counter/Timer Low Byte
MOV    PCA0CPH1, #000H ; PCA Counter/Timer High Byte
;
MOV    PCA0CPM2, #000H  ; PCA Capture/Compare Register 2
MOV    PCA0CPL2, #000H ; PCA Counter/Timer Low Byte
MOV    PCA0CPH2, #000H ; PCA Counter/Timer High Byte
;
MOV    PCA0CPM3, #000H  ; PCA Capture/Compare Register 3
MOV    PCA0CPL3, #000H ; PCA Counter/Timer Low Byte
MOV    PCA0CPH3, #000H ; PCA Counter/Timer High Byte
;
MOV    PCA0CPM4, #000H  ; PCA Capture/Compare Register 4
MOV    PCA0CPL4, #000H ; PCA Counter/Timer Low Byte
MOV    PCA0CPH4, #000H ; PCA Counter/Timer High Byte
;
MOV    PCA0CPM5, #000H  ; PCA Capture/Compare Register 5
MOV    PCA0CPL5, #000H ; PCA Counter/Timer Low Byte
MOV    PCA0CPH5, #000H ; PCA Counter/Timer High Byte
;
; Timers Configuration
;
MOV    SFRPAGE, #TMR01_PAGE
MOV    CKCON, #010H    ; Clock Control Register
MOV    TL0, #000H      ; Timer 0 Low Byte
MOV    TL1, #000H      ; Timer 1 Low Byte
MOV    TH0, #000H      ; Timer 0 High Byte
MOV    TH1, #000H      ; Timer 1 High Byte
MOV    TMOD, #020H     ; Timer Mode Register
MOV    TCON, #000H     ; Timer Control Register
;
MOV    SFRPAGE, #TMR2_PAGE
MOV    TMR2CF, #000H   ; Timer 2 Configuration
MOV    RCAP2L, #000H   ; Timer 2 Reload Register Low Byte
MOV    RCAP2H, #000H   ; Timer 2 Reload Register High Byte
MOV    TMR2L, #000H    ; Timer 2 Low Byte
MOV    TMR2H, #000H    ; Timer 2 High Byte
MOV    TMR2CN, #000H   ; Timer 2 CONTROL
;
MOV    SFRPAGE, #TMR3_PAGE
MOV    TMR3CF, #000H   ; Timer 3 Configuration
MOV    RCAP3L, #000H   ; Timer 3 Reload Register Low Byte
MOV    RCAP3H, #000H   ; Timer 3 Reload Register High Byte
MOV    TMR3H, #000H    ; Timer 3 High Byte
MOV    TMR3L, #000H    ; Timer 3 Low Byte
MOV    TMR3CN, #000H   ; Timer 3 Control Register
;

```

```

MOV    SFRPAGE, #TMR4_PAGE
MOV    TMR4CF, #000H          ; Timer 4 Configuration
MOV    RCAP4L, #000H         ; Timer 4 Reload Register Low Byte
MOV    RCAP4H, #000H         ; Timer 4 Reload Register High Byte
MOV    TMR4H, #000H          ; Timer 4 High Byte
MOV    TMR4L, #000H          ; Timer 4 Low Byte
MOV    TMR4CN, #000H         ; Timer 4 Control Register

```

```

;
; Reset Source Configuration
;

```

```

; Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0

```

```

;-----
; R | R/W | R/W | R/W | R | R | R/W | R
;-----
; JTAG | Convert | Comp.0 | S/W | WDT | Miss. | POR | HW
; Reset | Start | Reset/ | Reset | Reset | Clock | Force | Pin
; Flag | Reset/ | Enable | Force | Flag | Detect | & | Reset
; | Enable | Flag | & | | Flag | Flag | Flag
; | Flag | | Flag | | | | |
;-----

```

```

; NOTE! : Comparator 0 must be enabled before it is enabled as a
; reset source.
;

```

```

; NOTE! : External CNVSTR must be enabled through the crossbar, and
; the crossbar enabled prior to enabling CNVSTR as a reset source
;

```

```

MOV    SFRPAGE, #LEGACY_PAGE
MOV    RSTSRC, #04H          ; Reset Source Register

```

```

;
; Interrupt Configuration
;

```

```

MOV    IE, #000H            ; Interrupt Enable
MOV    IP, #000H            ; Interrupt Priority
MOV    EIE1, #000H          ; Extended Interrupt Enable
MOV    EIE2, #000H          ; Extended Interrupt Enable 2
MOV    EIP1, #000H          ; Extended Interrupt Priority 1
MOV    EIP2, #000H          ; Extended Interrupt Priority 2

```

```

;
RET

```

```

;
; routine to initialize UART for operation in full duplex manner
; with interrupts and buffer queues. This routine will also
; setup the baud rate utilizing a timer as the baud rate clock for
; both the transmit and receive sides of the UART. The initialized
; baud rate is a fixed rate at 115200 baud.
;

```

```

UART_INIT:

```

```

>SETB   TR1                ; let timer 1 run
MOV     SFRPAGE, #UART1_PAGE ; setup SFR page for uart 1
MOV     SCON1, #030H        ; enable uart mode
ENDIF
;
CLR     A
MOV     RX_RD_IDX, A        ; initialize the buffer control variables
MOV     RX_WR_IDX, A
MOV     TX_RD_IDX, A
MOV     TX_WR_IDX, A
MOV     RX_EMPTY, #01H      ; mark receive queue as empty
MOV     TX_EMPTY, #01H      ; mark transmit queue as empty

```

```

;
IF (UART_SEL = 0)
    ORL     IE, #010H        ; enable the uart interrupt
ELSE
    ORL     EIE2, #040H      ; enable the uart interrupt
ENDIF
RET

```

```

;
; routine to disable the UART interrupts and stop the companion timer
; baud rate register. This will leave the timer and UART registers back
; at their CONFIG function values.
;

```

```

UART_RESET:

```

```

IF (UART_SEL = 0)
    ANL     IE, #(NOT 010H)  ; disable uart interrupt
    MOV     SFRPAGE, #TMR3_PAGE ; activate the timer 3 page
    CLR     TR3              ; stop the counter
    MOV     TMR3CF, #000H    ; Timer 3 Configuration
    MOV     RCAP3L, #000H    ; Timer 3 Reload Register Low Byte

```

```

MOV    RCAP3H, #000H          ; Timer 3 Reload Register High Byte
MOV    TMR3H, #000H          ; Timer 3 High Byte
MOV    TMR3L, #000H          ; Timer 3 Low Byte
MOV    TMR3CN, #000H         ; Timer 3 Control Register
;
MOV    SFRPAGE, #UART0_PAGE  ; setup the SFR page for uart 0
MOV    SADENO, #000H          ; Serial 0 Slave Address Enable
MOV    SADDR0, #000H          ; Serial 0 Slave Address Register
MOV    SSTA0, #000H          ; UART Status and Clock Selection Register
MOV    SCON0, #050H          ; Serial Port Control Register
ANL    SCON0, #0FCH          ; clear interrupt pending flags
ELSE
ANL    EIE2, #(NOT 040H)      ; disable the uart interrupt
MOV    SFRPAGE, #TMR01_PAGE
CLR    TR1                    ; stop the counter
MOV    CKCON, #010H          ; Clock Control Register
MOV    TL1, #000H            ; Timer 1 Low Byte
MOV    TH1, #000H            ; Timer 1 High Byte
MOV    TMOD, #020H           ; Timer Mode Register
MOV    TCON, #000H           ; Timer Control Register
;
MOV    SFRPAGE, #UART1_PAGE
MOV    SCON1, #010H          ; Serial Port 1 Control Register
ANL    SCON1, #0FCH          ; clear interrupt pending flags
ENDIF
RET
;
;
; UART Transmit Data send routine. This will transfer the caller
; specified data to the transmit queue. The data is the entry R1
; argument. The return value from the function is setup such that
; a non zero value indicates that the data was successfully placed
; into the transmit queue. Zero return if the data could not be
; put into the transmit queue. Return value is given back in the
; A register.
;
UART_SEND:
MOV    A, TX_EMPTY           ; check if transmit queue is empty
JNZ    UART_SEND1            ; queue is empty
;
MOV    A, TX_WR_IDX           ; test for rd and wr indexes being equal.
XRL    A, TX_RD_IDX           ; if equal then the queue is FULL
JNZ    UART_SEND1            ;
CLR    A                       ; exit with the queue full error
RET
;
UART_SEND1:
CLR    EA                     ; disable interrupts during pointer updates
MOV    A, #TX_BUFFER          ; make pointer to the transmit buffer insert point
ADD    A, TX_WR_IDX
MOV    R0, A                   ; put pointer into R0
MOV    A, R1                   ; get byte being queued
MOV    @R0, A                  ; place it into queue
;
INC    TX_WR_IDX               ; bump & wrap queue index
ANL    TX_WR_IDX, #IDX_MSK
;
MOV    A, TX_EMPTY           ; check if queue was initially empty
JZ     UART_SEND2             ; it was not empty so no kick start is needed
;
MOV    TX_EMPTY, #0           ; set buffer flag to show not empty
IF (UART_SEL = 0)
MOV    SFRPAGE, #UART0_PAGE  ; force an initial empty -> not empty interrupt
SETB   TIO
ELSE
MOV    SFRPAGE, #UART1_PAGE  ; force an initial empty -> not empty interrupt
SETB   TI1
ENDIF
;
UART_SEND2:
SETB   EA                     ; reenale interrupts
MOV    A, #01H                ; show that data was accepted
RET
;
;
; UART Receive Data read routine. This will fetch a possible
; character from the receive 0 queue and place in the return R1
; register. The A register will return a non zero value if there
; is data returned in the R1 register. If the queue was empty then
; this will exit with A equal to zero to show that no data is returned

```

```

; in R1.
;
UART_READ:
    MOV    A, RX_EMPTY           ; test if the queue is empty
    JZ     UART_READ1           ; queue is not empty
    CLR    A                     ; exit with the queue empty error
    RET
;
UART_READ1:
    CLR    EA                   ; disable UART 0 to prevent interrupts
    MOV    A, #RX_BUFFER        ; make pointer to the receive buffer
    ADD    A, RX_RD_IDX
    MOV    R0, A                ; put pointer into R0
    MOV    A, @R0               ; get byte from the buffer
    MOV    R1, A                ; place it into R7 return register
    INC    RX_RD_IDX            ; increment & wrap index
    ANL    RX_RD_IDX, #IDX_MSK
;
    MOV    A, RX_RD_IDX         ; if wr and rd indexes equal then
    CJNE   A, RX_WR_IDX, UART_READ2 ; buffer has gone empty
;
    MOV    RX_EMPTY, #01H      ; mark buffer as empty
UART_READ2:
    SETB   EA                   ; show data has been returned
    MOV    A, #01H
    RET
;
;
; UART Interrupt Service routine. If the loader is not active then this
; will pass control on to the normal applications interrupt handler.
; Locally this will handle both a possible receive character and a pending
; transmit character. On receive if the queue is full the newest byte
; received is discarded.
;
UART_ISR:
    PUSH   PSW
    JB     LDR_FLAG, UART_ISR0   ; If flag set process here in the loader
    POP    PSW
    IF (UART_SEL = 0)
        LJMP APP_OFFSET+0x0023   ; go to the handler in the normal applications code
    ELSE
        LJMP APP_OFFSET+0x00A3   ; go to the handler in the normal applications code
    ENDIF
;
UART_ISR0:
    PUSH   ACC
    MOV    A, R0                ; preserve register R0
    PUSH   ACC
    MOV    A, R1                ; preserve register R1
    PUSH   ACC
;
    IF (UART_SEL = 0)
        MOV    SFRPAGE, #UART0_PAGE ; setup the SFR page for uart 0
        JNB    RI0, UART_ISR3       ; test if receive data interrupt pending
    ELSE
        MOV    SFRPAGE, #UART1_PAGE ; setup the SFR page for uart 1
        JNB    RI1, UART_ISR3       ; test if receive data interrupt pending
    ENDIF
;
UART_ISR1:
; process a receive interrupt
    IF (UART_SEL = 0)
        MOV    R1, SBUF0          ; get the received byte
        CLR    RI0                ; clear the received interrupt flag
    ELSE
        MOV    R1, SBUF1          ; get the received byte
        CLR    RI1                ; clear the received interrupt flag
    ENDIF
;
    MOV    A, RX_EMPTY           ; check if the buffer empty
    JNZ    UART_ISR2            ; empty so go ahead and queue data
    MOV    A, RX_WR_IDX         ; test for rd and wr indexes being equal.
    XRL    A, RX_RD_IDX         ; if equal then the queue is FULL
    JZ     UART_ISR3            ; full so discard the received data
;
UART_ISR2:
    MOV    A, #RX_BUFFER        ; make pointer to the receive buffer insert point
    ADD    A, RX_WR_IDX
    MOV    R0, A                ; put pointer into R0
    MOV    A, R1                ; get byte to be queued
    MOV    @R0, A               ; place it into queue

```

```

;
INC    RX_WR_IDX                ; bump & wrap queue index
ANL    RX_WR_IDX, #IDX_MSK
MOV    RX_EMPTY, #00H          ; show receive buffer not empty
;
UART_ISR3:
IF (UART_SEL = 0)
JNB    TI0, UART_ISR5          ; if transmit empty interrupt
;
CLR    TI0                      ; clear flag now that we see it was set
ELSE
JNB    TI1, UART_ISR5          ; if transmit empty interrupt
;
CLR    TI1                      ; clear flag now that we see it was set
ENDIF
MOV    A, TX_RD_IDX             ; the transmit queue has data in it if
XRL    A, TX_WR_IDX             ; the wr index != rd index
JZ     UART_ISR4
;
MOV    A, #TX_BUFFER           ; make pointer to the transmit buffer
ADD    A, TX_RD_IDX
MOV    R0, A                    ; put pointer into R0
MOV    A, @R0                  ; get byte from the buffer
IF (UART_SEL = 0)
MOV    SBUF0, A                ; and send it to the transmitter
ELSE
MOV    SBUF1, A                ; and send it to the transmitter
ENDIF
INC    TX_RD_IDX               ; bump & wrap queue index
ANL    TX_RD_IDX, #IDX_MSK
JMP    UART_ISR5               ; exit from transmit handling
;
UART_ISR4:
MOV    TX_EMPTY, #01H          ; show transmit queue is empty
;
UART_ISR5:
POP    ACC                     ; restore register 1
MOV    R1, A
POP    ACC
MOV    R0, A                    ; restore register 0
POP    ACC                     ; restore pre-interrupt context
POP    PSW
RETI
;
;
; routine to initialize timer 0 to generate an interrupt every 1 milliseconds.
;
TIMRO_INIT:
MOV    SFRPAGE, #TMR01_PAGE    ; setup page to timer 0
MOV    A, CKCON                 ; setup timer 0 to use SYSCLK
ANL    A, #0F7H
ORL    A, #008H
MOV    CKCON, A
;
MOV    A, TMOD                  ; set T/CO Mode
ANL    A, #0F0H
ORL    A, #001H
MOV    TMOD, A
;
MOV    A, #LOW_TIMRO_DIV        ; setup the 1 msec period
MOV    TLO, A
MOV    A, #HIGH_TIMRO_DIV
MOV    TH0, A
;
MOV    TIM_CNT, #10            ; init the timer counter to 10 msec
;
SETB   ETO                      ; enable timer 0 interrupts
SETB   TRO                      ; start timer 0 running
RET
;
;
; routine to disable the loader usage of the timer 0 and restore
; all its registers back to the CONFIG settings.
;
TIMRO_RESET:
MOV    SFRPAGE, #TMR01_PAGE    ; setup page to timer 0
CLR    ETO                      ; disable the timer 0 interrupts
CLR    TRO                      ; stop timer 0 from running
MOV    CKCON, #010H            ; Clock Control Register
MOV    TLO, #000H              ; Timer 0 Low Byte

```

```

MOV     TL1, #000H           ; Timer 1 Low Byte
MOV     TMOD,#020H          ; Timer Mode Register
MOV     TCON, #000H         ; Timer Control Register
RET
;
;
; local loader timer 0 interrupt used to generate a periodic 10 msec interrupt
; for the software times management. If the loader is not active then this
; will pass control on to the normal applications interrupt handler.
; Locally this will handle both a possible receive character and a pending
; transmit character. On receive if the queue is full the newest byte
; received is discarded.
;
TIMRO_ISR:
    PUSH    PSW
    JB     LDR_FLAG, TIMRO_ISR0 ; If flag set process here in the loader
    POP     PSW
    LJMP   APP_OFFSET+0x000B    ; go to the handler in the normal applications code
;
TIMRO_ISR0:
    PUSH    ACC
    MOV     SFRPAGE, #TMR01_PAGE ; setup page to timer 0
;
    CLR     TR0
    MOV     A, #LOW TIMRO_DIV    ; setup the next 1 msec period
    MOV     TLO, A
    MOV     A, #HIGH TIMRO_DIV
    MOV     TH0, A
    SETB   TR0                  ; allow the timer to run again
;
    DJNZ   TIM_CNT, TIMRO_ISR5  ; exit if not 10 msec yet
    MOV     TIM_CNT, #10        ; reset counter for next 10 msec
;
;
; at each 10 msec decrement any of the software timers that are non-zero
;
TIMRO_ISR1:
    MOV     A, TIMER1           ; get/test timer 1
    JZ     TIMRO_ISR2
    DEC    TIMER1
;
TIMRO_ISR2:
    MOV     A, TIMER2           ; get/test timer 2
    JZ     TIMRO_ISR3
    DEC    TIMER2
;
TIMRO_ISR3:
    MOV     A, TIMER3           ; get/test timer 3
    JZ     TIMRO_ISR4
    DEC    TIMER3
;
TIMRO_ISR4:
    MOV     A, TIMER4           ; get/test timer 4
    JZ     TIMRO_ISR5
    DEC    TIMER4
;
TIMRO_ISR5:
    POP     ACC
    POP     PSW
    RETI
;
;
; CRC-16 coefficient lookup table. This table produces a CRC-16
; with the generator polynomial as follows:
;
;           16   15   2
;           G(x) = x  + x  + x  + 1
;
;
CRC_TABLE:
    DW     00000H,0C0C1H,0C181H,00140H,0C301H,003C0H,00280H,0C241H
    DW     0C601H,006C0H,00780H,0C741H,00500H,0C5C1H,0C481H,00440H
    DW     0CC01H,00CC0H,00D80H,0CD41H,00F00H,0CF41H,0CE81H,00E40H
    DW     00A00H,0CAC1H,0CB81H,00B40H,0C901H,009C0H,00880H,0C841H
    DW     0D801H,018C0H,01980H,0D941H,01B00H,0DBC1H,0DA81H,01A40H
    DW     01E00H,0DEC1H,0DF81H,01F40H,0DD01H,01DC0H,01C80H,0DC41H
    DW     01400H,0D4C1H,0D581H,01540H,0D701H,017C0H,01680H,0D641H
    DW     0D201H,012C0H,01380H,0D341H,01100H,0D1C1H,0D081H,01040H
    DW     0F001H,030C0H,03180H,0F141H,03300H,0F3C1H,0F281H,03240H
    DW     03600H,0F6C1H,0F781H,03740H,0F501H,035C0H,03480H,0F441H
    DW     03C00H,0FCC1H,0FD81H,03D40H,0FF01H,03FC0H,03E80H,0FE41H

```



```

MOV     DPH, A           ; now DPTR points into table.
;
MOV     A, #01H         ; get low byte of crc table entry
MOVC   A, @A+DPTR
XRL    A, R4           ; XOR it with previous high byte
MOV    R5,A           ; and save as new low byte
;
CLR     A               ; get high byte of crc table entry
MOVC   A, @A+DPTR
MOV    R4, A          ; save the new high byte from table
;
INC     R3              ; increment the loop counter
JMP    GEN_CRC1
;
GEN_CRC2:
MOV     A, R2           ; move the CRC bytes to packet at
DEC     A               ; [length-1] & [length-2]
ADD     A, R1
MOV     R0, A
MOV     A, R5
MOV     @R0, A
DEC     R0
MOV     A, R4
MOV     @R0, A
RET
;
;
; routine to check the CRC of the entry packet for the length
; specified. the return value here will be a non zero value
; if the packet has the correct CRC value and zero if there is an
; error in the packet.
;
; entry:
;   R1 = packet pointer
;   R2 = packet length
; uses:
;   R0   for indirect pointer to packet buffer
;   R4::R5 for the compare CRC accumulator
;   +others used in GEN_CRC
; return:
;   A    = 0 if packet CRC is good
;        != 0 if packet CRC is bad
;
CHK_CRC:
MOV     A, R2           ; get existing crc bytes from packet
DEC     A               ; at [length-1] & [length-2]
ADD     A, R1
MOV     R0, A
MOV     A, @R0         ; get the len-1 byte
PUSH   ACC             ; save it on stack
DEC     R0
MOV     A, @R0         ; get the len-2 byte
PUSH   ACC             ; save on the stack
MOV     A, R0          ; save pointer on stack too
PUSH   ACC
;
CALL    GEN_CRC        ; generate a CRC based on packet data
;
POP     ACC
MOV     R0, A          ; restore pointer to length-2
;
POP     ACC             ; get the orig crc high byte
MOV     R4, A          ; save orig in R4
XCHD   A, @R0         ; swap orig with new high byte
XRL    A, R4           ; compare the high bytes
MOV     R4, A          ; save compare results in R4
;
INC     R0              ; pointer to length-1
POP     ACC             ; get the orig crc low byte
MOV     R5, A          ; save orig in R5
XCHD   A, @R0         ; swap orig with new low byte
XRL    A, R5           ; compare the low bytes
MOV     R5, A          ; save compare results in R5
;
ORL    A, R4           ; check if we had a compare
RET     ; returns A = 0 if crc was good
;
;
; routine to send a packet to the UART serial queue. This accepts a
; pointer to the packet buffer and a length. This will insert length and

```

```

; proper CRC bytes to the end of the packet and will then transmit the
; data on to the serial line via the UART queueing routine. This will
; poll on the queue to wait should the queue become full.
;
; entry:
;   R1 = packet pointer
;   R2 = packet length
; uses:
;   R0   for indirect pointer to packet buffer
;   +others as used in GEN_CRC
; return:
;   none
;
SEND_PKT:
    MOV     A, R1
    ADD     A, #2           ; index to length location
    MOV     R0, A
    MOV     A, R2           ; place length into packet
    MOV     @R0, A
;
    CALL    GEN_CRC        ; insert the proper CRC value
;
    MOV     A, R1           ; get R0 as pointer
    MOV     R0, A
SEND_PKT1:
    MOV     A, @R0         ; get the packet data byte
    MOV     R1, A         ; pass to UART routine in R1
;
    MOV     A, R0           ; save pointer
    PUSH    ACC
    MOV     A, R2           ; save loop counter
    PUSH    ACC
SEND_PKT2:
    CALL    UART_SEND      ; send the data via UART 0 queue
    JZ     SEND_PKT2       ; poll till queue took the data
    POP     ACC
    MOV     R2, A           ; restore the loop counter
    POP     ACC
    MOV     R0, A           ; restore buffer pointer
;
    INC     R0              ; increment the buffer pointer
    DJNZ   R2, SEND_PKT1   ; decrement and loop till packet is complete
    RET
;
;
; receive packet support routines.
; These routines operate as a state machine to receive packets from the
; UART port.
;
; definition of the receive packet state numbers.
;
R_STATE_SYNC EQU 0           ; state to scan for SYNC byte
R_STATE_ID   EQU 1           ; state to scan for ID byte
R_STATE_LEN  EQU 2           ; state to scan for length byte
R_STATE_DATA EQU 3           ; state to scan in packet data
R_STATE_HOLD EQU 4           ; hold onto packet till used/copied
;
;
; routine called to initialize the receive packet state machine
;
RECV_PKT_INIT:
    MOV     R_STATE, #R_STATE_SYNC ; reset the state machine
    MOV     R_CNT, #0
    RET
;
;
; routine called to scan for a packet coming in from the UART port.
; this will return a zero value if there is not a packet ready. If a
; packet has been received then this will return a non-zero value which
; is the length of the packet received. This routine will enter a hold mode
; after first reporting a packet as available so that the packet can be processed.
; A call to the RECV_PKT_RPLY routine will free the receive buffer and dispatch
; a ACK/NACK reply back to the host based on the results of the packet received.
;
; This routine operates as a state machine and as such is needs to be called
; in a repetitive manner to.
;
; receive packet state table. Entries here must be ordered to match the state
; numbers given above.
;

```

```

R_STATE_TABLE:
    DW    RECV_SYNC          ; pointer to receive sync routine
    DW    RECV_ID            ; pointer to receive id byte routine
    DW    RECV_LEN           ; pointer to receive length routine
    DW    RECV_DATA         ; pointer to receive data routine
    DW    RECV_HOLD         ; pointer to receive hold routine
R_STATE_CNT EQU    ($ - R_STATE_TABLE)/2 ; number of states
;
RECV_PKT_SCAN:
    MOV    A, R_STATE        ; fetch the current state number
    CJNE   A, #R_STATE_CNT, $+3 ;
    JC     RECV_PS1          ; state number OK
    MOV    R_CNT, #0
    MOV    R_STATE, #R_STATE_SYNC ; reset to SYNC state if invalid
;
RECV_PS1:
    MOV    DPTR, #R_STATE_TABLE ; index to the state handling routine
    CLR    C
    RLC    A                  ; multiply * 2 for word access
    MOV    R0, A              ; save a copy of index
    INC    A                  ; increment index to the high byte
    MOVC   A, @A+DPTR        ; low byte
    PUSH   ACC                ; onto stack
    MOV    A, R0
    MOVC   A, @A+DPTR        ; high byte
    PUSH   ACC                ; onto stack
    RET                      ; direct branch to the subroutine
; note that the return of state processing
; routines goes directly back to caller to here
;
;
; receive state processing to wait until a packet lead in sync byte is
; fetched from the UART input.
;
RECV_SYNC:
    CALL   UART_READ         ; check if a byte ready
    JZ     RECV_SYNC1        ; exit with if no byte ready
;
    CJNE   R1, #PACKET_SYNC, RECV_SYNC1 ; exit if not the SYNC byte
;
    MOV    R0, #R_PACKET     ; point to start of buffer
    MOV    A, R1
    MOV    @R0, A             ; save at head of packet buffer
    MOV    R_CNT, #1         ; show have 1 byte now
    MOV    R_STATE, #R_STATE_ID ; set the next state number
;
RECV_SYNC1:
    CLR    A                  ; exit to show no packet ready
    RET
;
;
; receive state processing to get in the packet ID byte as
; fetched from the UART input.
;
RECV_ID:
    CALL   UART_READ         ; check if a byte ready
    JZ     RECV_ID1         ; exit with if no byte ready
;
    MOV    A, #R_PACKET     ; index to next packet location
    ADD    A, R_CNT
    MOV    R0, A
    MOV    A, R1             ; put the new byte into buffer as
    MOV    @R0, A            ; the ID byte
    INC    R_CNT             ; bump the byte count
    MOV    R_STATE, #R_STATE_LEN ; set next state to receive the length
;
RECV_ID1:
    CLR    A                  ; exit to show no packet ready
    RET
;
;
; receive state processing to get in the packet length byte as
; fetched from the UART input.
;
RECV_LEN:
    CALL   UART_READ         ; check if a byte ready
    JZ     RECV_LEN2        ; exit with if no byte ready
;
    MOV    A, #R_PACKET     ; index to next packet location
    ADD    A, R_CNT

```

```

MOV     R0, A
MOV     A, R1                ; put the new byte into buffer as
MOV     @R0, A               ; the length byte
INC     R_CNT                ; bump the byte count
CJNE   R1, #PACKET_MIN, $+3 ; check for at least minimum length
JC     RECV_LEN1            ; too short so reset to SYNC state
CJNE   R1, #PACKET_MAX+1, $+3 ; check if bigger than maximum size
JNC    RECV_LEN1            ; too long so reset to SYNC state
;
MOV     R_LENGTH, R1         ; save the length for next states
MOV     R_STATE, #R_STATE_DATA ; set next state to receive the data
JMP    RECV_LEN2
;
RECV_LEN1:
MOV     R_STATE, #R_STATE_SYNC ; reset the state machine
MOV     R_CNT, #0
;
RECV_LEN2:
CLR     A                    ; exit to show no packet ready
RET
;
;
; receive state processing to get in the specified number of bytes
; fetched from the UART input. R_LENGTH contains the total packet
; length to expect (which includes the CRC bytes too).
;
RECV_DATA:
MOV     A, R_CNT              ; sanity check to make sure no
CJNE   A, #PACKET_MAX, $+3   ; possible buffer overflow
JC     RECV_DATA1            ; branch if OK
;
MOV     R_STATE, #R_STATE_SYNC ; reset the state machine
MOV     R_CNT, #0            ; if sanity check failed
JMP    RECV_DATA2
;
RECV_DATA1:
CALL   UART_READ             ; check if a byte ready
JZ     RECV_DATA3            ; exit with if no byte ready
;
MOV     A, #R_PACKET         ; index to next packet location
ADD    A, R_CNT
MOV     R0, A
MOV     A, R1                ; put the new byte into buffer as
MOV     @R0, A               ; a data or CRC byte
INC     R_CNT                ; bump the byte count
;
RECV_DATA2:
MOV     A, R_CNT              ; check if desired count have
CJNE   A, R_LENGTH, $+3     ; been received yet
JC     RECV_DATA3            ; still more to come yet
;
MOV     R_STATE, #R_STATE_HOLD ; mark next as holding state
MOV     A, R_LENGTH          ; return the size received
JMP    RECV_DATA4
;
RECV_DATA3:
CLR     A                    ; exit to show no packet ready
RECV_DATA4:
RET
;
;
; receive state processor to hold the current packet in the buffer
; until the reply routine is called and this state is released.
;
RECV_HOLD:
MOV     A, R_LENGTH          ; keep returning the size received
RET
;
;
; routine to reply to the host as the response to a received packet.
; The entry A value indicates if the ACK or the NACK packet will be
; returned. A=0 indicates a good packet so send back an ACK ID packet
; A != 0 indicates a bad packet so send back a NACK ID packet
;
RECV_PKT_RPLY:
CJNE   A, #0, RECV_PR1       ; branch if to setup the NACK packet
;
MOV     R0, #S_PACKET
MOV     @R0, #PACKET_SYNC    ; setup ACK reply
INC     R0

```

```

MOV    @R0, #PACKET_ACK
JMP    RECV_PR2
;
RECV_PR1:
MOV    R0, #S_PACKET
MOV    @R0, #PACKET_SYNC        ; setup NACK reply
INC    R0
MOV    @R0, #PACKET_NACK
;
RECV_PR2:
MOV    R1, #S_PACKET            ; queue the reply packet
MOV    R2, #PACKET_MIN         ; as a minimum length packet
CALL   SEND_PKT
;
MOV    R_STATE, #R_STATE_SYNC   ; release the hold state
RET
;
;
; AT24C01 Support Routines. This is in support of a 128 byte I2C type
; serial EEPROM that is connected to the SBus of the processor as
; follows:
; P0.6 -> SBus SCL to AT24C01 Pin 6 SCL
; P0.5 -> SBus SDA to AT24C01 Pin 5 SDA
;
; routine to support the writing of a byte into the EEPROM.
; Interface to this routine is done so as to directly support calls from
; C code in the main application program as well as calls from the packet
; level driver in this loader module.
;
; Entry:
; R7 = EEPROM Address (0x00 -> 0x7F)
; R5 = EEPROM Data (0x00 -> 0xFF)
; Exit:
; None
; C Function Prototype:
; void EE_WRITE(unsigned char e_addr, unsigned char e_data);
;
PUBLIC  _EE_WRITE
_EE_WRITE:
PUSH   ACC
;
_EE_WRITE1:
JB     SM_BUSY, _EE_WRITE1      ; loop here till previous transaction
; is completed
MOV    A, R7
CLR    C                        ; shift the address left by 1 and make RW
RLC    A                        ; be a zero for WRITE
MOV    EE_ADDR, A
CLR    SM_RW                    ; also clear the RW mode bit to Write
;
MOV    A, R5                    ; setup the data byte
MOV    EE_XMIT, A               ; into the send out location
;
SETB   SM_BUSY                 ; set busy for new transaction
MOV    SFRPAGE, #SMB0_PAGE
SETB   STA                      ; start the sequence
;
POP    ACC
RET
;
;
; routine to support reading of a byte from the EEPROM.
; Interface to this routine is done so as to directly support calls from
; C code in the main application program as well as calls from the packet
; level driver in this loader module.
;
; Entry:
; R7 = EEPROM Address (0x00 -> 0x7F)
; Exit:
; R7 = EEPROM Data (0x00 -> 0xFF)
;
; C Function Prototype:
; unsigned char EE_READ(unsigned char e_addr);
;
PUBLIC  _EE_READ
_EE_READ:
PUSH   ACC
;
_EE_READ1:
JB     SM_BUSY, _EE_READ1      ; loop here till previous transaction

```

```

; is completed
MOV    A, R7
SETB  C                ; shift the address left by 1 and make RW
RLC   A                ; be a one for READ
MOV   EE_ADDR, A
SETB  SM_RW           ; also set the RW mode bit to Read
;
;
SETB  SM_BUSY        ; set busy for new transaction
MOV   SFRPAGE, #SMB0_PAGE
SETB  STA            ; start the sequence
;
;
_EE_READ2:
JB    SM_BUSY, _EE_READ2 ; loop here till read completes
;
MOV   R7, EE_RECV    ; return data from the receive location
;
POP   ACC
RET
;
;
; routine to configure the SMBus controller on the processor
;
; - Configures and enables the SMBus.
; - Sets SMBus clock rate.
; - Enables SMBus interrupt.;
; - Clears SM_Busy flag for first transfer.
;
SMBus_INIT:
MOV   SFRPAGE, #SMB0_PAGE
MOV   SMB0CN, #0x04    ; Configure SMBus to send ACKs on acknowledge cycle
MOV   SMB0CR, #210    ; SMBus clock rate approx 56 kHz
;                               ; Tlow = 4*(256-SMB0CR)/22118400
;                               ; Thigh = 4*(258-SMB0CR)/22118400 + 625 nSec
;                               ; SMBClk = 1/(Thigh + Tlow)
ORL   SMB0CN, #0x40    ; Enable SMBus
;
ORL   EIE1, #0x02     ; Enable SMBus interrupts
CLR   SM_BUSY
RET
;
;
; interrupt service support routine for the SMBus sequencer.
;
; Implemented as a state table lookup, with the SMBus status register as the index.
; SMBus status codes are multiples of 8.
;
SMBus_ISR:
PUSH  PSW              ; save registers during interrupt
PUSH  ACC
PUSH  DPH
PUSH  DPL
PUSH  B
;
MOV   A, #LOW SMBus_ISR1 ; make pseudo RET address on stack
PUSH  ACC
MOV   A, #HIGH SMBus_ISR1
PUSH  ACC
;
MOV   SFRPAGE, #SMB0_PAGE
MOV   A, SMB0STA      ; Load accumulator with current SMBus state.
ANL   A, #0xF8        ; convert the SMBus status from *8 index into *2 index
RR    A
RR    A
;
MOV   B, A
INC   A
MOV   DPTR, #SMBus_TABLE ; make pointer into decoding table
MOVC  A, @A+DPTR      ; get low byte of branch address
PUSH  ACC              ; onto stack
MOV   A, B
MOVC  A, @A+DPTR      ; high byte
PUSH  ACC              ; onto stack
RET                    ; "branch" to the routine
;
; State processor function subroutine will do a RET to here
;
;
SMBus_ISR1:
CLR   SI              ; clear the SMBus interrupt flag
POP   B                ; restore main tread context
POP   DPL

```

```
POP DPH
POP ACC
POP PSW
RETI
```

```
;
;
; SMBus Interrupt State Processing Routines
;
```

```
SMBus_TABLE:
```

```
DW SMBus_BUS_ERROR ; 0x00 (all modes) BUS ERROR
DW SMBus_START ; 0x08 (MT & MR) START transmitted
DW SMBus_RP_START ; 0x10 (MT & MR) repeated START
DW SMBus_MTADDACK ; 0x18 (MT) Slave address + W transmitted;
; ACK received
DW SMBus_MTADDNACK ; 0x20 (MT) Slave address + W transmitted;
; NACK received
DW SMBus_MTDBACK ; 0x28 (MT) data byte transmitted; ACK rec'vd
DW SMBus_MTDBNACK ; 0x30 (MT) data byte transmitted; NACK rec'vd
DW SMBus_MTARBLOST ; 0x38 (MT) arbitration lost
DW SMBus_MRADDACK ; 0x40 (MR) Slave address + R transmitted;
; ACK received
DW SMBus_MRADDNACK ; 0x48 (MR) Slave address + R transmitted;
; NACK received
DW SMBus_MRDBACK ; 0x50 (MR) data byte rec'vd; ACK transmitted
DW SMBus_MRDBNACK ; 0x58 (MR) data byte rec'vd; NACK transmitted
;
DW SMBus_SROADACK ; 0x60 (SR) SMB's own slave address + W rec'vd;
; ACK transmitted
DW SMBus_SROARBLOST ; 0x68 (SR) SMB's own slave address + W rec'vd;
; arbitration lost
DW SMBus_SRGADACK ; 0x70 (SR) general call address rec'vd;
; ACK transmitted
DW SMBus_SRGARBLOST ; 0x78 (SR) arbitration lost when transmitting
; slave addr + R/W as master; general
; call address rec'vd; ACK transmitted
DW SMBus_SRODBACK ; 0x80 (SR) data byte received under own slave
; address; ACK returned
DW SMBus_SRODBNACK ; 0x88 (SR) data byte received under own slave
; address; NACK returned
DW SMBus_SRGDBACK ; 0x90 (SR) data byte received under general
; call address; ACK returned
DW SMBus_SRGDBNACK ; 0x98 (SR) data byte received under general
; call address; NACK returned
DW SMBus_SRSTOP ; 0xA0 (SR) STOP or repeated START received
; while addressed as a slave
DW SMBus_STOADACK ; 0xA8 (ST) SMB's own slave address + R rec'vd;
; ACK transmitted
DW SMBus_STOARBLOST ; 0xB0 (ST) arbitration lost in transmitting
; slave address + R/W as master; own
; slave address rec'vd; ACK transmitted
DW SMBus_STDBACK ; 0xB8 (ST) data byte transmitted; ACK rec'ed
DW SMBus_STDBNACK ; 0xC0 (ST) data byte transmitted; NACK rec'ed
DW SMBus_STDBLAST ; 0xC8 (ST) last data byte transmitted (AA=0);
; ACK received
DW SMBus_SCLHIGHTO ; 0xD0 (ST & SR) SCL clock high timer per
; SMOOCR timed out (FTE=1)
DW SMBus_NOSTATE_D8 ; 0xD8 (none) Non existant state
DW SMBus_NOSTATE_E0 ; 0xE0 (none) Non existant state
DW SMBus_NOSTATE_E8 ; 0xE8 (none) Non existant state
DW SMBus_NOSTATE_F0 ; 0xF0 (none) Non existant state
DW SMBus_IDLE_F8 ; 0xF8 (all modes) Idle
```

```
;
;
; Slave mode states not used in this application so they just leave immediately
;
```

```
SMBus_SROADACK: ; 0x60 (SR) SMB's own slave address + W rec'vd;
; ACK transmitted
SMBus_SROARBLOST: ; 0x68 (SR) SMB's own slave address + W rec'vd;
; arbitration lost
SMBus_SRGADACK: ; 0x70 (SR) general call address rec'vd;
; ACK transmitted
SMBus_SRGARBLOST: ; 0x78 (SR) arbitration lost when transmitting
; slave addr + R/W as master; general
; call address rec'vd; ACK transmitted
SMBus_SRODBACK: ; 0x80 (SR) data byte received under own slave
; address; ACK returned
SMBus_SRODBNACK: ; 0x88 (SR) data byte received under own slave
; address; NACK returned
SMBus_SRGDBACK: ; 0x90 (SR) data byte received under general
; call address; ACK returned
```

```

SMBus_SRGNACK:      ; 0x98 (SR) data byte received under general
                    ;                call address; NACK returned
SMBus_SRSTOP:       ; 0xA0 (SR) STOP or repeated START received
                    ;                while addressed as a slave
SMBus_STOACK:        ; 0xA8 (ST) SMB's own slave address + R rec'vd;
                    ;                ACK transmitted
SMBus_STOARBLOST:   ; 0xB0 (ST) arbitration lost in transmitting
                    ;                slave address + R/W as master; own
                    ;                slave address rec'vd; ACK transmitted
SMBus_STDBACK:       ; 0xB8 (ST) data byte transmitted; ACK rec'ed
SMBus_STDBNACK:     ; 0xC0 (ST) data byte transmitted; NACK rec'ed
SMBus_STDBLAST:     ; 0xC8 (ST) last data byte transmitted (AA=0);
                    ;                ACK received
SMBus_SCLHIGHTO:    ; 0xD0 (ST & SR) SCL clock high timer per
                    ;                SMBOCR timed out (FTE=1)
    RET
;
;
; Undedined and Idle Mode State should never occur but they are here
; just in case will leave immediately
;
SMBus_NOSTATE_D8:    ; 0xD8 (none) Non existant state
SMBus_NOSTATE_E0:    ; 0xE0 (none) Non existant state
SMBus_NOSTATE_E8:    ; 0xE8 (none) Non existant state
SMBus_NOSTATE_F0:    ; 0xF0 (none) Non existant state
SMBBus_IDLE_F8:      ; 0xF8 (all modes) Idle
;
    RET
;
;
; SMBus State Routine
; 0x00 (all modes) BUS ERROR
;
;     Reset the hardware by setting the STOP bit
;
SMBus_BUS_ERROR:
    SETB    STO
    RET
;
;
; SMBus State Routine
; 0x08 (MT & MR) START transmitted
;
;     The R/W bit will determine if we are reading or writing
;     Here we simply send out the address byte because it gets sent
;     in both modes and the RW bit in the byte has been preset
;
SMBus_START:

    MOV     SMBODAT, EE_ADDR      ; transfer the Address Byte out
    CLR     STA                   ; manually clear the start bit
    RET
;
;
; SMBus State Routine
; 0x10 (MT & MR) repeated START
;
;     This state does not occur when communicating to the AT24C01 EEPROM
;
SMBus_RP_START:
    CLR     STA                   ; manually clear the start bit
    RET
;
;
; SMBus State Routine
; 0x18 (MT) Slave address + W transmitted; ACK received
;
SMBus_MTADDACK:
    MOV     SMBODAT, EE_XMIT      ; initiate sending the data out
    RET
;
;
; SMBus State Routine
; 0x20 (MT) Slave address + W transmitted; NACK received
;
;     The slave is not responding. Try again with acknowledge
;     polling. Send STOP + START.
;
SMBus_MTADDNACK:

```

```

SETB   STO
SETB   STA
RET
;
;
;
; SMBus State Routine
; 0x28 (MT) data byte transmitted; ACK rec'vd
;
;   The write transaction is complete so send stop and clear the busy
;
SMBus_MTDDBACK:
SETB   STO
CLR    SM_BUSY           ; clear the busy bit
RET
;
;
; SMBus State Routine
; 0x30 (MT) data byte transmitted; NACK rec'vd
;
;   The slave is not responding. Try again with acknowledge
;   polling. Send STOP + START.
;
SMBus_MTDDBNACK:
SETB   STO
SETB   STA
RET
;
;
; SMBus State Routine
; 0x38 (MT) arbitration lost
;
;   Should not happen in this EEPROM case, but if it does then
;   restart the transfer
;
SMBus_MTARBLOST:
SETB   STO
SETB   STA
RET
;
;
; SMBus State Routine
; 0x40 (MR) Slave address + R transmitted; ACK received
;
;   In this case ready to receive a byte from the EEPROM
;   set to transmit a NACK after the transfer since we are
;   doing only single byte transfers.
;
SMBus_MRADDACK:
CLR    AA
RET
;
;
; SMBus State Routine
; 0x48 (MR) Slave address + R transmitted; NACK received
;
;   Attempt the transfer again
;
SMBus_MRADDNACK:
SETB   STO
SETB   STA
RET
;
;
; SMBus State Routine
; 0x50 (MR) data byte rec'vd; ACK transmitted
;
;   This state should not occur because the AA bit was cleared
;   prior to send a MACK. Just send the stop condition if it
;   does happen.
;
SMBus_MRDBACK:
SETB   STO
RET
;
;
; SMBus State Routine
; 0x58 (MR) data byte rec'vd; NACK transmitted
;
;   This state signals the completion of the READ transaction
;   So read the data register and send the stop condition

```

```

;
SMBus_MRDBNACK:
    MOV     EE_RECV, SMBODAT        ; retrieve the data
    SETB   STO
    SETB   AA                      ; set AA for next transfer
    CLR    SM_BUSY                  ; show transaction complete
    RET
;
;
; startup entry point for the loader function. This will take entry from the
; hardware reset vector and then initialize the hardware to the minimum level
; that is needed to permit the loader function to operate over the UART serial
; port.
;
LOADER_BASE:
    MOV     R0, #0xFF               ; clear all of RAM from top to bottom
LOADER_ZERO:
    MOV     @R0, #0                  ; clear a byte
    DJNZ   R0, LOADER_ZERO         ; final dec R0 leaves RAM 0 also zero !
;
    MOV     SP, #LDR_STACK          ; setup the stack pointer in internal RAM
    SETB   LDR_FLAG                ; force local interrupt handling.
;
    CALL   CONFIG                   ; perform all the SFR initialization
;
    CALL   UART_INIT                ; initialize UART for local loader control.
;
    CALL   TIMRO_INIT               ; initialize the timer interrupts
;
    CALL   SMBus_INIT               ; initialize the SMBus controller
;
    SETB   EA                       ; loader interrupts
;
;
; loader wakeup logic. The loader will wakeup through receipt of an
; eight byte sequence received from the UART port provided those
; bytes arrive within 1 second of startup. If a timeout is detected
; on receipt of any byte then the loader wakeup sequence is aborted.
; If any of the bytes has the wrong value then the wakeup sequence
; is aborted. As each correct byte of the wakeup sequence is detected
; the loader will complement the bits and send the byte back to the
; host to inform the host that the "loader is here" as a handshake.
;
; The wakeup byte sequence used is in the following table.
;
LOADER_WAKE:
    JMP    LOADER_WAKE1             ; branch over the wakeup bytes
;
; loader wakeup byte table
;
WAKE_BYTES:
    DB     0x4E, 0xA3, 0x2C, 0xED, 0x12, 0xD3, 0x5C, 0xB1
WAKE_BYTE_CNT equ ($ - WAKE_BYTES) ; total bytes in wakeup sequence
;
; setup to receive the eight byte wakeup sequence. A 1 second
; timer is used to timeout on any given byte. Any error in a byte
; received or a byte timeout will cease the polling loop.
;
LOADER_WAKE1:
    MOV    WAKE_CNT, #0              ; loop counter for the wakeup bytes
LOADER_WAKE2:
    MOV    TIMER1, #100             ; set timer 1 for 1 second
LOADER_WAKE3:
    CALL   UART_READ                ; see if a byte came in.
    JNZ   LOADER_WAKE4              ; got a byte!
    MOV    A, TIMER1                 ; check for a timeout on byte
    JNZ   LOADER_WAKE3              ; still timer active period
    JMP   LOADER_WAKE5              ; timeout so abort wakeup sequence
;
LOADER_WAKE4:                      ; here if got a byte in from UART
    MOV    A, WAKE_CNT               ; check byte against the string
    MOV    DPTR, #WAKE_BYTES
    MOVC  A, @A+DPTR                ; fetch the byte from known string
    XRL   A, R1                     ; compare to received byte
    JZ    LOADER_WAKE6              ; compare so continue wakeup sequence
;
LOADER_WAKE5:
    JMP   LOADER_DONE                ; bale out of wakeup sequence
;
LOADER_WAKE6:

```

```

MOV    A, #0xFF                ; complement the byte and echo it
XRL    A, R1
MOV    R1, A
CALL   UART_SEND                ; echo the complement of the byte
;
INC    WAKE_CNT                 ; increment the wakeup byte counter
MOV    A, WAKE_CNT              ; see if have surveyed all the bytes
CJNE   A, #WAKE_BYTE_CNT, LOADER_WAKE2 ; loop for next wakeup byte handling
;
; here the wakeup sequence has completed successfully so start to enter packet
; control mode. In this mode packets are received in a slave mode. As each is
; processed and the replied to. Every intact packet is actually replied to
; including those with improper CRC values. Note that the receive packet
; scanner routine will filter out all packets that are ill formed, too short,
; too long etc. Packets are processed here in a continuous loop until either
; a power down or a processor reset.
;
LOADER_RUN:
CALL   RECV_PKT_SCAN           ; look to see if a packet has arrived
JZ     LOADER_RUN              ; loop till one comes
;
MOV    R2, A                    ; check the packets CRC
MOV    R1, #R_PACKET
CALL   CHK_CRC
JZ     LOADER_EXEC             ; if A = 0 CRC was good
;
CALL   RECV_PKT_RPLY           ; A != 0 - send back a NACK
JMP    LOADER_RUN              ; go try for a new packet
;
;
; packet type header table. This table provides a listing of the characteristics
; for each packet type and a pointer to the code subroutine used to process that
; packet type after it has been validated against the attributes specified in the
; table. The table is indexed according to the packet ID type.
;
; equates for offsets to the item elements in each entry
;
ITEM_ID EQU    0x00              ; offset of item ID
ITEM_MIN EQU    0x01            ; offset to minimum length of packet
ITEM_MAX EQU    0x02            ; offset to maximum length of packet
ITEM_FUNC_H EQU  0x03           ; offset to the high byte of function pointer
ITEM_FUNC_L EQU  0x04           ; offset to the low byte of function pointer
;
PACKET_TABLE:
DB     0x00                    ; ID code - Echo Packet
DB     PACKET_MIN, PACKET_MIN  ; minimum and maximum length of packet
DW     LOADER_ECHO_FUNC        ; just send echo packet back
;
PACKET_ITEM_SIZE EQU    ($ - PACKET_TABLE) ; size of each table item
DB     0x01                    ; ID Code - Ack Packet
DB     PACKET_MIN, PACKET_MIN  ; minimum and maximum length of packet
DW     LOADER_ACK_FUNC        ; normal not received...just ACK it
;
DB     0x02                    ; ID Code - Nack Packet
DB     PACKET_MIN, PACKET_MIN  ; minimum and maximum length of packet
DW     LOADER_ACK_FUNC        ; normal not received...just ACK it
;
DB     0x03                    ; ID Code - Loader Size Request
DB     PACKET_MIN, PACKET_MIN  ; size of packet
DW     LOADER_SIZE_FUNC       ; report loader size in bytes
;
DB     0x04                    ; ID Code - Read Flash Request
DB     PACKET_MIN+4, PACKET_MIN+4 ; size of request packet
DW     LOADER_READ_FUNC       ; send some bytes to the host
;
DB     0x05                    ; ID Code - Erase Flash Request
DB     PACKET_MIN+3, PACKET_MIN+3 ; size of erase packet
DW     LOADER_ERAS_FUNC       ; erase a flash block
;
DB     0x06                    ; ID Code - Write Flash Request
DB     PACKET_MIN+3, PACKET_MAX ; size range for write block
DW     LOADER_WRIT_FUNC
;
DB     0x07                    ; ID Code - Write EEPROM Request
DB     PACKET_MIN+2, PACKET_MIN+2 ; size of EE_WRITE packet
DW     LOADER_EEWR_FUNC
;
DB     0x08                    ; ID Code - Read EEPROM Request
DB     PACKET_MIN+1, PACKET_MIN+1 ; size od EE_READ packet
DW     LOADER_EERD_FUNC

```

```

PACKET_ID_MAX EQU ($ - PACKET_TABLE)/PACKET_ITEM_SIZE ; max ID code
;
;
; Here a valid packet was received so look it up in the table
; and process it.
;
LOADER_EXEC:
    MOV     R0, #R_PACKET+1           ; check for an echo packet
    MOV     A, @R0                    ; get the packet ID byte
    CJNE   A, #PACKET_ID_MAX, $+3     ; check if ID within table range
    JC     LOADER_EXEC1              ; ID code within table range
;
LOADER_EXECO:                        ; in here to send back a NACK
    MOV     A, #1                     ; A != 0 - send back a NACK
    CALL   RECV_PKT_RPLY              ; for packet with unknown ID
    JMP     LOADER_RUN                 ; go try for a new packet
;
LOADER_EXEC1:
    MOV     B, #PACKET_ITEM_SIZE      ; compute packet table index
    MUL     AB
    MOV     DPTR, #PACKET_TABLE       ; make pointer into decoding table
    ADD     A, DPL
    MOV     DPL, A
    MOV     A, B
    ADDC   A, DPH
    MOV     DPH, A                    ; DPTR has table pointer
;
    MOV     A, #ITEM_MIN
    MOVC   A, @A+DPTR                 ; fetch the shortest packet length allowed
    MOV     R1, A                      ; R1 is min length allowed
    MOV     R0, #R_PACKET+2           ; point to length byte in the packet
    MOV     A, @R0                    ; get the value to A
    CLR     C
    SUBB   A, R1                      ; compute length - min NC means len >= min
    JC     LOADER_EXECO               ; go NACK packet that is too short
;
    MOV     A, @R0                    ; get length byte from packet
    MOV     R1, A
    MOV     A, #ITEM_MAX
    MOVC   A, @A+DPTR                 ; fetch the longest packet length allowed
    CLR     C
    SUBB   A, R1                      ; compute max - length NC means max >= len
    JC     LOADER_EXECO               ; go NACK packet that is too long
;
    MOV     A, #LOW_LOADER_EXEC2      ; make pseudo RET address on stack
    PUSH   ACC
    MOV     A, #HIGH_LOADER_EXEC2
    PUSH   ACC
;
    MOV     A, #ITEM_FUNC_L           ; fetch the packet subroutine address
    MOVC   A, @A+DPTR                 ; low byte
    PUSH   ACC                        ; onto stack
    MOV     A, #ITEM_FUNC_H
    MOVC   A, @A+DPTR                 ; high byte
    PUSH   ACC                        ; onto stack
    RET                                  ; "branch" to the routine
; DPTR points to decode table entry
; R_PACKET is packet data
; packet processor function subroutine will do a RET to here
; will pass back A = 0 if reply already handled
; A = 1 if ACK should be sent
; A = -1 if NACK should be sent
LOADER_EXEC2:
    JNZ    LOADER_EXEC3              ; need to handle a reply here
;
    MOV     R_STATE, #R_STATE_SYNC    ; release the hold state
    JMP     LOADER_RUN                 ; loop for a new receive packet
;
LOADER_EXEC3:
    CJNE   A, #1, LOADER_EXECO        ; go to process a NACK reply
;
    CLR     A
    CALL   RECV_PKT_RPLY              ; A = 0 send back an ACK packet
    JMP     LOADER_RUN                 ; loop back for next receive packet
;
;
; come here when the loader is complete with its activities and enter
; the normal application program.
;
LOADER_DONE:

```


This function cannot allow operations above
0x1FBFD which is the extend of the FLASH

Request Packet Format:

0 1 2 3 4 5 6 7 8
<Sync><ID-0x04><PktLen=9><AdrHi><AdrMid><AdrLo><BytCnt><CksHi><CksLo>

Reply Packet Format

<Sync><ID-0x04><PktLen=BytCnt+5><Data0>...<DataN><CksHi><CksLo>

LOADER_READ_FUNC:

```
MOV R0, #R_PACKET+3 ; get parameters from the request
MOV A, @R0 ; get addr high byte to R5
MOV R5, A
INC R0
MOV A, @R0 ; get addr mid byte to R6
MOV R6, A
INC R0
MOV A, @R0 ; get addr low byte to R7
MOV R7, A
INC R0
MOV A, @R0 ; get byte count to R4
MOV R4, A
;
CJNE R4, #0, LDR_READ1 ; request of 0 bytes illegal
LDR_READ0:
MOV A, #-1 ; send NACK for bad request
RET
;
LDR_READ1:
CJNE R4, #(PACKET_MAX-PACKET_MIN+1), $+3 ; JC set is len < max allowed
JNC LDR_READ0 ; show request too large
;
CLR C ; make sure start address not over top of flash
MOV A, R7
SUBB A, #0xFE
MOV A, R6
SUBB A, #0xFB
MOV A, R5
SUBB A, #0x01
JNC LDR_READ0 ; NACK is address >= 0x01FBFE
;
MOV A, R4 ; compute addr + length to R1::R2::R3
ADD A, R7
MOV R3, A
MOV A, #0
ADDC A, R6
MOV R2, A
MOV A, #0
ADDC A, R5
MOV R1, A
;
CLR C ; make sure address + cnt not over top of flash
MOV A, R3
SUBB A, #0xFF
MOV A, R2
SUBB A, #0xFB
MOV A, R1
SUBB A, #0x01
JNC LDR_READ0 ; NACK if address + cnt >= 0x01FBFF
;
; here to read the bytes of the flash image to the S_Buffer R5::R6::R7 == ADDR
; R4 = byte count
;
; This loop works by using MOVC opcode to read the bytes of flash. The flash
; is banked in the DPTR address range of 0x8000 to 0xFFFF and the four banks
; are selected via the bits 5::1 of the PSBANK register.
;
MOV R0, #S_PACKET ; point at the packet buffer
MOV @R0, #PACKET_SYNC
INC R0
MOV @R0, #PACKET_READ ; plug in the return ID
INC R0 ; increment over the length byte
INC R0 ; R0 = pointer to place the read data
;
MOV A, R4
MOV R1, A ; use R1 for a loop counter
;
MOV A, R7
MOV DPL, A ; setup start DPTR
MOV A, R6
```

```

SETB ACC.7 ; force DPTR to high map space
MOV DPH, A
;
MOV A, R6 ; make up the 2 bank select bits
RLC A ; upper bit R4 to CARRY
MOV A, R5
RLC A
ANL A, #0x03 ; Bank bits in bits 1::0
SWAP A ; now in 5::4
;
ANL PSBANK, #0xCF
ORL PSBANK, A
LDR_READ2:
CLR A
MOVC A, @A+DPTR ; read flash byte
MOV @R0, A ; stuff into packet
INC R0
INC DPTR ; bump pointers
MOV A, DPH ; see if DPTR went to ZERO need to bump to next bank
ORL A, DPL
JNZ LDR_READ3
;
MOV DPH, #0x80 ; reset to next bank space
MOV A, PSBANK
ADD A, #0x10 ; bump bank number too
ANL A, 0x3F ; limit bank select to 2 bits
MOV PSBANK, A
LDR_READ3:
DJNZ R1, LDR_READ2 ; loop to copy all the bytes
;
MOV A, R4 ; get full packet length
ADD A, #PACKET_MIN
MOV R2, A
MOV R1, #S_PACKET
CALL SEND_PKT ; send the echo packet
CLR A ; show we handled it all here
RET
;
;
; LOADER ERASE REQUEST HANDLER This is a request to erase a block of FLASH
; bytes out of the device back to the host.
; This function cannot allow operations above
; 0x1F7FF which is the extent of the FLASH that
; can be erased via software access routines.
; This also specifically disallows the erasure
; within the loader area.
;
; Request Packet Format:
; 0 1 2 3 4 5 6 7
; <Sync><ID=0x05><PktLen=9><AdrHi><AdrMid><AdrLo><CksHi><CksLo>
;
LOADER_ERAS_FUNC:
MOV R0, #R_PACKET+3 ; get parameters from the request
MOV A, @R0 ; get addr high byte to R5
MOV R5, A
INC R0
MOV A, @R0 ; get addr mid byte to R6
MOV R6, A
INC R0
MOV A, @R0 ; get addr low byte to R7
MOV R7, A
INC R0
;
MOV A, R7 ; mask the address to 1K pages
ANL A, #LOW( NOT (1024-1))
MOV R7, A
MOV A, R6
ANL A, #HIGH( NOT (1024-1))
MOV R6, A
;
CLR C
MOV A, R7 ; make sure address not over top of flash
SUBB A, #0x00
MOV A, R6
SUBB A, #0xF8
MOV A, R5
SUBB A, #0x01
JC LDR_ERAS1 ; NACK if address >= 0x01F800
;
LDR_ERAS0:
MOV A, #-1 ; force a NACK

```

```

RET
;
LDR_ERAS1:
CLR    C
MOV    A, R7                ; make sure address not in the loader
SUBB  A, #LOW(APP_OFFSET AND (NOT(1024-1)))
MOV    A, R6
SUBB  A, #HIGH(APP_OFFSET AND (NOT(1024-1)))
MOV    A, R5
SUBB  A, #0x00
JC     LDR_ERAS0            ; NACK if address < APP_OFFSET
;
;
; here to erase the bytes of a flash page R5::R6::R7 == Page Base Address
;
MOV    A, R7
MOV    DPL, A                ; setup start DPTR
MOV    A, R6
SETB  ACC.7                 ; force DPTR to high map space
MOV    DPH, A
;
MOV    A, R6                ; make up the 2 bank select bits
RLC   A                    ; upper bit R4 to CARRY
MOV    A, R5
RLC   A
ANL   A, #0x03             ; Bank bits in bits 1::0
SWAP  A                    ; now in 5::4
;
ANL   PSBANK, #0xCF
ORL   PSBANK, A
;
CLR   EA                  ; disable interrupts during erase
;
MOV   SFRPAGE, #0         ; ensure page 0
ANL   PSCTL, #NOT 0x04   ; clear PSCTL.2 SFLE bit - disable scratchpad
ORL   FLSCL, #0x01       ; set FLSCL.0 FLWE bit - enable user FLASH wrt/eras
ORL   PSCTL, #0x02       ; set PSCTL.1 PSEE bit - enable flash erases
ORL   PSCRL, #0x01       ; set PSCRL.0 PSWE bit - allow MOVX to flash
CLR   A
;
MOVX  @DPTR, A            ; write a byte to erase the page
;
ANL   PSCTL, #NOT 0x02   ; clear PSCTL.1 PSEE bit - disable flash erases
ANL   PSCRL, #NOT 0x01   ; clear PSCRL.0 PSWE bit - disallow MOVX to flash
ANL   FLSCL, #NOT 0x01   ; clear FLSCL.0 FLWE bit - disable user FLASH wrt/eras
;
SETB  EA                  ; re-enable interrupts
;
MOV   A, #1               ; show to send ACK on completion
RET
;
;
; LOADER WRITE REQUEST HANDLER This is a request to write a block of FLASH
;                               bytes from the host to the device flash area..
;                               This function cannot allow operations above
;                               0x1F7FF which is the extent of the FLASH that
;                               can be written by software after erasing. It
;                               also specifically locks out the loader area.
;
; Request Packet Format:
;
;   0       1       2       3       4       5       6       N+6     N+7     N+8
;   <Sync><ID-0x05><PktLen=N+8><AdrHi><AdrMid><AdrLo><Data0>...<DataN-1><CksHi><CksLo>
;
LDR_WRIT_FUNC:
MOV    R0, #R_PACKET+2    ; get parameters from the request
MOV    A, @R0              ; get packet length
INC    R0
CLR    C
SUBB  A, #8                ; data length is packet len-8
MOV    R4, A
;
CJNE  R4, #0, LDR_WRIT1   ; request of 0 bytes illegal
LDR_WRIT0:
MOV    A, #-1              ; send NACK for bad request
RET
;
LDR_WRIT1:
MOV    A, @R0              ; get addr high byte to R5
MOV    R5, A
INC    R0

```

```

MOV    A, @R0                ; get addr mid byte to R6
MOV    R6, A
INC    R0
MOV    A, @R0                ; get addr low byte to R7
MOV    R7, A
;
CLR    C
MOV    A, R7                ; make sure start address not over top of flash
SUBB   A, #0xFF
MOV    A, R6
SUBB   A, #0xF7
MOV    A, R5
SUBB   A, #0x01
JNC    LDR_WRITO            ; NACK if address >= 0x01FBFE
;
CLR    C
MOV    A, R7                ; make sure start address above loader area
SUBB   A, #LOW(APP_OFFSET)
MOV    A, R6
SUBB   A, #HIGH(APP_OFFSET)
MOV    A, R5
SUBB   A, #0x00
JC     LDR_WRITO            ; NACK if address < APP_OFFSET
;
MOV    A, R4                ; compute addr + length to R1::R2::R3
ADD    A, R7
MOV    R3, A
MOV    A, #0
ADDC   A, R6
MOV    R2, A
MOV    A, #0
ADDC   A, R5
MOV    R1, A
;
CLR    C
MOV    A, R3                ; make sure address + length not over top of flash
SUBB   A, #0xFF
MOV    A, R2
SUBB   A, #0xFB
MOV    A, R1
SUBB   A, #0x01
JNC    LDR_WRITO            ; NACK if address + cnt >= 0x01FBFF
;
;
; here to write the bytes of the flash image to the S_Buffer R5::R6::R7 == ADDR
;                                     R4 = byte count
;
; This loop works by using MOVX opcode to write the bytes of flash. The flash
; is banked in the DPTR address range of 0x8000 to 0xFFFF and the four banks
; are selected via the bits 5::1 of the PSBANK register.
;
MOV    R0, #R_PACKET+6      ; point at the received packet buffer
;
MOV    A, R4
MOV    R1, A                ; use R1 for a loop counter
;
MOV    A, R7
MOV    DPL, A               ; setup start DPTR
MOV    A, R6
SETB   ACC.7                ; force DPTR to high map space
MOV    DPH, A
;
MOV    A, R6                ; make up the 2 bank select bits
RLC    A                    ; upper bit R4 to CARRY
MOV    A, R5
RLC    A
ANL    A, #0x03             ; Bank bits in bits 1::0
SWAP   A                    ; now in 5::4
;
ANL    PSBANK, #0xCF
ORL    PSBANK, A
;
CLR    EA                   ; disable interrupts during writing
MOV    SFRPAGE, #0          ; ensure page 0
ANL    CCHOEN, #NOT 0x01    ; clear CCHOEN.0 CHBLKW bit - disable flash block writes
ANL    PSCTL, #NOT 0x04     ; clear PSCTL.2 SFLE bit - disable scratchpad
ANL    PSCTL, #NOT 0x02     ; clear PSCTL.1 PSEE bit - disable flash erases
ORL    FLSC, #0x01          ; set FLSC.0 FLWE bit - enable user FLASH wrt/eras
ORL    PSCTL, #0x01         ; set PSCTL.0 PSWE bit - allow MOVX to flash

```


