AVR-BED EMBEDDED DEVELOPMENT BOARD

Peripheral Function Library Reference Manual

The AVR-BED function library supports the following peripherals concurrently:

- Timer-Counter TC1 — milliseconds timer, real-time clock, task scheduler
- LCD Module 16x2 — initialise, write command, write character, print string
- UART — initialise, check RX data, get byte, send byte, send string
- EEPROM — read byte, write byte, read array, write array
- ADC — single conversion, 10-bit result (inputs: ADC1… ADC5)
- Push-buttons — button scan routine, detect hits, get button status
- Keypad (external) — keypad scan routine, detect hits, get key-code of hit

The library allows peripherals which share common I/O pins to operate "virtually concurrently". This is achieved by "time-sharing" of MCU resources.

Library functions must always be called from the "background process" (main loop), never from an interrupt service routine (ISR). Otherwise, contention of resources can occur.

The LED strip (8 LEDs connected to Port B) may be driven by writing directly to PORTB (output register). Other peripherals which share Port B (e.g. LCD, keypad) will not disrupt the LED display, provided that other functions are not called excessively frequently.
(See notes elsewhere on LCD and keypad operation.)

Pin PD6/OC0A may be used for external GPIO or PWM output. There is a jumper option to connect PD6/OC0A to the LCD backlight switch for PWM control of brightness. Pin PD5/OC0B may also be used for an external PWM output, or GPIO, provided that neither button D nor the 4x4 keypad are needed.

PD7 is dedicated to the on-board "debug" LED, typically used for a "heart-beat" (1Hz pulse).

The potentiometer output may be jumpered to PC1/ADC1 or PC3/ADC3, or disconnected.

Pins PC1, PC2 & PC3 may be used for analogue (ADC) input. PC4 and PC5 may also be used as ADC inputs, or GPIO, provided the IIC peripheral bus is not needed.

The on-board USB-serial adapter module (FT232RL) allows the AVR-BED to be connected to a host PC for serial data communications. Typical applications are data logging with the PC, or to provide a command-line user-interface (CLI) running in a terminal app (e.g. “PuTTY”).

Timer Functions

```
/*
 * The timer library uses a real-time interrupt generated by Timer TC1 to provide
 * a time-base for a real-time clock and a general-purpose millisecond timer.
 *
 * The ISR also incorporates a primitive task scheduler which sets 'Task Flags' at
 * periodic intervals of 5ms, 50ms and 500ms.  These flags may be accessed by the
 * application program to invoke periodic tasks.  See function: isTaskPending_5ms().
 * ------------------------------------------------------------------------------
 *
 * Function:  TC1_initialize()
 *
 * This function initializes Timer-Counter TC1 to generate a periodic interrupt
 * request (IRQ) every millisecond precisely.
 *
 * The timer "Output Compare" feature is used to make the timing automatic.
 * When the counter register (TMR1) reaches the "TOP count" (OCR1A value), the
 * count register is automatically reset (zeroed) and an IRQ flag is raised.
 *
 * The timer clock pre-scaler is set to divide the CPU clock frequency by 8.
 * Assuming the CPU clock freq. is 8 MHz, the counter clock will be 1 MHz.
 *
 * TC1_initialize() must be called from main() before enabling global interrupts.
 * Global interrupts must be enabled for the timer functions to work.
 */
void  TC1_initialize();

/*
 * Function:  milliseconds()
 *
 * This function returns the value of a free-running 32-bit counter variable,
 * incremented every millisecond by Timer/Counter TC1 interrupt handler (ISR).
 * It's purpose is to implement "non-blocking" time delays and event timers.
 * Typical usage:
 *
 *    static unsigned long eventStartTime;
 *                    :
 *    eventStartTime = milliseconds();  // capture the starting time
 *                    :
 *    if (milliseconds() >= (eventStartTime + EVENT_DURATION))  // time's up!
 *    {
 *        // Do what needs to be done TIME_DURATION ms after eventStartTime
 *    }
 *
 * A program can implement many independent event timers, simply by declaring
 * a unique eventStartTime (variable) and a unique EVENT_DURATION (constant)
 * for each independent "event" or delay to be timed.
 *
 * Be sure to declare each eventStartTime as 'static' (permanent) so that it
 * will be kept between multiple calls to the function in which it is defined.
 */
unsigned long  milliseconds();

/*
 * Function:  isTaskPending_??ms()
 *
 * These three functions return TRUE if their respective Task Flag is raised;
 * otherwise they return FALSE.  The Task Flag is cleared before the function exits,
 * so that on subsequent calls it will return FALSE, until the next task period ends.
 */
BOOL  isTaskPending_5ms();

BOOL  isTaskPending_50ms();

BOOL  isTaskPending_500ms();
```

LCD Functions

```
//
// LCD Controller Command bytes
//
#define LCD_FS_8BIT_2LINES  0b00111000  // DL = 1 (8 bits), N = 1 (2 lines), F = 0
#define LCD_OFF             0b00001000  // D=0 (off), C=0 (cursor off), B=0 (no blink)
#define LCD_CLR             0b00000001  // Clears entire display
#define LCD_HOME            0b00000010  // Return cursor to home posn, DDRAM addr = 0
#define LCD_EM_INC          0b00000110  // Increment cursor position, no dispaly shift
#define LCD_ON              0b00001110  // D=1 (on), C=1 (cursor on), B=0 (no blink)
#define LCD_CURSOR_OFF      0b00001100  // Display ON, cursor OFF (hidden)
#define LCD_DDRAM_1ST_LINE  0b10000000  // Char positions on 1st line: 0x00 to 0x0F
#define LCD_DDRAM_2ND_LINE  0b11000000  // Char positions on 2nd line: 0x40 to 0x4F
#define LCD_CGRAM_SET       0b01000000  // Set CGRAM address

// Macro to set cursor position to a given row and column,
// where row is 0 (top line) or 1 (bottom line),  col is 0..15
#define lcd_cursor_posn(row, col)  lcd_command(0x80 + (row * 0x40) + col)

// Alias for lcd_initialise(), for legacy AVR-Pad library compatibility
#define initialise_LCD()  lcd_initialise()

/*
 * Initialise the LCD controller (HD44780)...
 * LCD mode is set to: 2 lines, char 5x8 dots, cursor on
 */
extern void  lcd_initialise(void);

/*
 * Output a command byte to the LCD controller.
 * Refer to command definitions (macros) above.
 *
 * Entry arg:  cmd = LCD command code (byte)
 */
extern void  lcd_command( BYTE cmd );

/*
 * Display a single ASCII character.
 * Before calling this function, set cursor position using
 *    lcd_cursor_posn(row, col)
 * where row is 0 (top line) or 1 (bottom line),  col is 0..15
 * The cursor position will be advanced one place to the right on exit.
 *
 * Entry arg:  c = ASCII character code (printable)
 */
extern void  lcd_write_char( char c );

/*
 * Function to display a NUL-terminated string.
 * Before calling this function, set cursor position using
 *    lcd_cursor_posn(row, col)
 * where row is 0 (top line) or 1 (bottom line),  col is 0..15
 * The cursor position will be advanced N places to the right on exit,
 * where N = number of characters in the input string (not incl. NUL terminator).
 *
 * Entry arg:  str = address of string (constant or variable)
 *
 * Usage examples:  lcd_print_string("Hello, world.");  // string constant
 *                  lcd_print_string(buff);  // where buff is an array of chars
 */
extern void  lcd_print_string( char *str );
```

Push-button Functions

```
/*
 * Function ButtonScan() must be called periodically from the application program
 * (main loop) at intervals of about 50ms for reliable "de-bounce" operation.
 *
 * It's main purpose is to detect "button hit" events, i.e. transition from "no button
 * pressed" to "button pressed" and to raise a status flag to signal the event.
 *
 * The entry argument (nButts) specifies the number of buttons (1..4) to be serviced.
 * For example, if nButts is 1, only Button_A is serviced;  if nButts is 3, then 3
 * buttons (Button_A, Button_B and Button_C) will be serviced by the scan routine.
 *
 * Note: The function (re-)configures the required port D pins (PD2..PD5) as inputs;
 *       and enables pull-ups on respective pins;  all other I/O pins are unaffected.
 */
void  ButtonScan(unsigned char nButts);

/*
 * Function button_hit() returns the Boolean value (TRUE or FALSE) of a flag indicating
 * whether or not a "button hit" event occurred since the previous call to the function.
 *
 * Entry argument 'button_ID' is an ASCII code identifying one of 4 buttons to check,
 * which must be one of: 'A', 'B', 'C' or 'D', otherwise the function will return FALSE.
 * If the given button is not serviced by ButtonScan(), button_hit() will return FALSE.
 *
 * The flag (internal static variable) is cleared "automatically" by the function so that
 * on subsequent calls the function will return FALSE (until the next button hit occurs).
 */
BOOL  button_hit(char  button_ID);

/*
 * Function button_pressed() returns the Boolean value (TRUE or FALSE) of a flag telling
 * whether or not a given button is currently pressed, i.e. held down.
 *
 * Entry argument 'button_ID' is an ASCII code identifying one of 4 buttons to check,
 * which must be one of: 'A', 'B', 'C' or 'D', otherwise the function will return FALSE.
 * If the given button is not serviced, button_pressed() will return FALSE.
 */
BOOL  button_pressed(char  button_ID);
```

ADC Function

```
/*
 * Function ADC_ReadInput() starts a one-off conversion on the given input, waits for
 * the conversion cycle to complete, then returns the 10-bit result.
 *
 * Entry arg:  muxsel = ADC MUX input select:  1 = ADC1, 2 = ADC2, ... 7 = ADC7
 *             (ADC0, ADC8..13 N/A, 14 = 1.1V internal ref, 15 = GND/0V)
 *
 * Note:  The function assumes the selected ADC port pin is already configured as an
 *        input and that its internal pull-up resistor is disabled.
 */
unsigned  ADC_ReadInput( BYTE muxsel );
```

EEPROM Functions

```
/*
* Function to read and return a single byte from EEPROM
* at the given address (arg1), range:  0 <= addr <= 511
*/
BYTE  EEPROM_ReadByte(unsigned addr);

/*
* Function to write a single byte to EEPROM at the given address (arg1).
*  ATmega88:  0 <= addr <= 511
*/
void EEPROM_WriteByte(unsigned addr, BYTE bDat);

/*
* Function to read a specified number of bytes (nbytes) from EEPROM
* and to copy them to an array in data memory (pdata).
* Example of usage:
* ````````````````````
*     unsigned char  dest_array[64];   // ensure size >= nbytes
*         :
*         :
*     EEPROM_ReadArray(dest_array, 40);
*/
void EEPROM_ReadArray(BYTE *pdata, int nbytes);

/*
* Function to write a specified number of bytes (nbytes) into EEPROM,
* copied from an array in data memory (pdata).
* Example of usage:
* ````````````````````
*     unsigned char  source_array[64];   // ensure size >= nbytes
*         :
*         :
*     EEPROM_WriteArray(source_array, 40);
*/
void EEPROM_WriteArray(BYTE *pdata, int nbytes);
```

UART Functions

```
// Macros to enable and disable UART receiver interrupt requests:
#define UART_IRQ_ENABLE()    SET_BIT(UCSR0B, RXCIE0)
#define UART_IRQ_DISABLE()   CLEAR_BIT(UCSR0B, RXCIE0)

// Macros to check TX and RX status:
#define UART_RxDataAvail()   (TEST_BIT(UCSR0A, RXC0) != 0)
#define UART_TX_Ready()      (TEST_BIT(UCSR0A, UDRE0) != 0)

/*
*   UART_init() ...
*   Initialise USART0 for RX and TX with the specified baudrate.
*   Must be called by the application program before using USART0.
*   Entry arg (baudrate) is bits per second (e.g. 300, 1200, 9600, 38400, 57600).
*   Note:  F_OSC_Hz is defined in UART_drv.h
*/
void   UART_init( unsigned baudrate );

/*
*   UART_GetByte() ...
*   Fetch next byte from USART0 RX buffer.
*   The function DOES NOT WAIT for data available in the input buffer;
*   the caller must first check using the macro UART_RxDataAvail().
*   If there is no data available, the function returns NUL (0).
*   The input char is NOT echoed back to the UART output stream.
*
*   Returns:  Byte from USART0 RX buffer (or 0, if buffer is empty).
*/
BYTE  UART_GetByte( void );

/*
*   UART_PutByte() ... UART Transmit Byte.
*   Writes a data byte into the UART TX Data register to be transmitted.
*   The function waits for the TX register to be cleared first.
*/
void   UART_PutByte( BYTE b );

/*
*   UART_PutString() ...  Transmit a NUL-terminated string.
*   Newline (ASCII 0x0A) is expanded to CR + LF (0x0D + 0x0A).
*   Note: This function delays the calling task by whatever time it takes to
*   transmit the string (= strlen(s) x 10000 / baudrate;  msec. approx.)
*/
void   UART_PutString( char *str );

/*
*   If the UART receiver is interrupt-driven, the application program must
*   provide an interrupt "call-back" function named: UART_RX_IRQ_Handler().
*
*   The IRQ signals that a byte has been received by the UART;
*   the byte must be read out of the UART RX data register) and stored in a
*   buffer in RAM (circular FIFO) by this function.
*/
#ifdef UART_RX_INTERRUPT_DRIVEN
extern void  UART_RX_IRQ_Handler( void );
#endif
```

Keypad Functions

```
/*
 * Function KeypadScan() must be called periodically from the application program
 * (main loop) at intervals of about 50ms for reliable de-bounce operation.
 * The function assumes only one key at a time (or none) is pressed.
 * Its purpose is to detect a "key hit" event, i.e. a transition from "no key pressed"
 * to "any key pressed" and to record the key code of the last pressed key.
 *
 * Note: The function exits with all port B pins set as outputs;  4 port D pins
 *       PD2..PD5 set as inputs;  all other I/O pins unaffected.
 */
void  KeypadScan(void);

/*
 * Function key_hit() returns the Boolean value (TRUE or FALSE) of a flag indicating
 * whether or not a "key hit" event has occurred since the previous call to the function.
 * The flag (internal static variable) is cleared "automatically" by the function so that
 * on subsequent calls the function will return FALSE (until the next key hit).
 */
BOOL  key_hit(void);

/*
 * Function key_code() returns the ASCII character code corresponding to the key last
 * pressed, i.e. when the last "key hit" event was detected.
 * Note: The ASCII codes for digits '0' to '9' are 0x30 to 0x39 (resp).
 */
char  key_code(void);
```