Cow Pi, mark 1d*

Datasheet

The mark 1d Cow Pi is a member of the Cow Pi product line from Eclectic Electronics[1] and is guaranteed to bring you minutes of fun. Whether you need a 4-function calculator to count your terror bird eggs before they hatch, an electronic combination lock to secure your wildly-dangerous experimental robots, or a motion alarm to protect yourself from zombie gourds, using the Cow Pi development board is your best option.

This document describes the use of the Cow Pi mk1d development board with the CowPi v0.4 library.

Contents

*The Cow Pi hardware design, library, example code, and documentation

[1] Eclectic Electronics is a joint subsidiary of The Pleistocene Petting Zoo, a Limited-Reality Corporation, and of Babbage's Analytic Engines, a Frankly-Figmentary Partnership.
(These are fictional companies imagined by Christopher Bohn for course assignments. You didn't think they really existed, did you? Christopher Bohn, however, really does exist and can be contacted at bohn@unl.edu.)

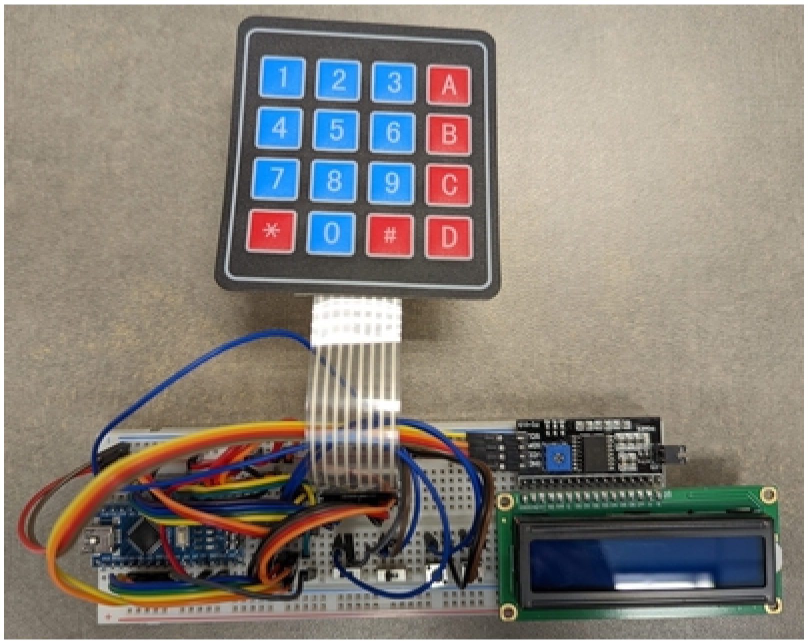


Figure 1: A mark 1d Cow Pi development board.

1 Hardware Overview

This section describes the mark 1d Cow Pi development board, describes the theory of operation for its components, and summarizes the features of its display module.

1.1 Cow Pi Development Board

The Cow Pi development board consists of a microcontroller board (Arduino Nano), a display module (2×16 LCD dot-matrix character display), a 4×4 matrix keypad, two momentary buttons, two toggleable switches, and two LEDs. The board is assembled on a solderless breadboard; see Figure 1.

The toggleable switches are referred to as the **left switch** and the **right switch**, and each can be positioned in the left or right position. When a switch is in the right position, its logic value is high, by way of a pull-up resistor. When a switch is in the left position, the switch is grounded, and its logic value is low.

The momentary buttons are referred to as the **left button** and the **right button**, and each can be pressed (alternatively, in the down position) or unpressed (alternatively, in the up position). The buttons are normally-open, and so when a button is unpressed, its logic value is high, by way of a pull-up resistor. When a button is pressed, the button is grounded, and its logic value is low.

The LEDs are referred to as the **left LED** and the **right LED**. The **left LED** is aliased to the Arduino Nano's built-in LED, and some documents may refer to it as the Cow Pi's **internal LED**. Indeed, Arduino-based mark 1 Cow Pis simply use the Arduino Nano's built-in LED as the **left LED** to reduce clutter on the solderless breadboard. Some documents may refer to the **right LED** as the Cow Pi's **external LED**. An LED will illuminate when the corresponding microcontroller output is high, and it will deluminate when the corresponding microcontroller output is low.

The matrix keypad is designed to be scanned using the conventional approach of selectively setting the rows' logic values and reading the columns' resulting logic values. Note that mark 1 Cow Pis' keypads cannot be accurately scanned if more than two keys are pressed at the same time. Further, **there are no current-limiting protections** (other than the microcontroller's fuses) in place to prevent over-current if simultaneously pressing more than one in the same column shorts a logic-high row to a logic-low row.

The microcontroller communicates with the 2×16 LCD dot-matrix character display-display module using the Inter-Integrated Circuit (I^2C or IIC) protocol, also known as the Two-Wire Interface (TWI) protocol.

Figure 2 shows which input or output is connected to each of the Arduino Nano's pins, as well as which general-purpose input/output register bit corresponds to each pin.

1.2 Matrix Keypad

1.2.1 Theory of Operation

Each key on a matrix keypad is a normally-open, momentary button that resides at the intersection of a row and a column; see Figure 3(a). When pressed, the key closes an electrical connection between that row and column. On the Cow Pi, each row is connected to an output pin on the Arduino Nano, and each column is connected to an input pin with a pull-up resistor.

Because the input pins that the columns are connected to use pull-up resistors, the logic value on these pins will normally read high (boolean 1). A column will read as logic low (boolean 0) only when it is electrically connected to a row that is set low. An application developer can take advantage of this by setting all of the rows' pins to logic low (boolean 0); see Figure 3(b). When a key is pressed, its column will then become low.

A keypress, thus, can be detected based on the values read from the columns' pins. A pin-change interrupt that is triggered by a change on the columns' pins can be used to indicate that a key has been pressed; however, for the mark 1d Cow Pi, it is probably easier to use an external interrupt that is triggered by a change in the output of the NAND gate that is also connected to the keypad's columns (see Section 5). As an alternative to using an interrupt, an application programmer can poll the four columns' pins. If, collectively, they produce the bit vector 0xF, then no key is being pressed; however, if the bit vector is anything other than 0xF, then at least one key is being pressed.

Once it has been determined that a key is pressed, code that scans the keypad should execute. If every row is made logic-high *except* for one row, then the code can determine whether the key that was pressed is in that row. For example, as shown in Figure 4(a), if the "8" key is pressed and "row4" is the only logic-low row, then the column bit vector is

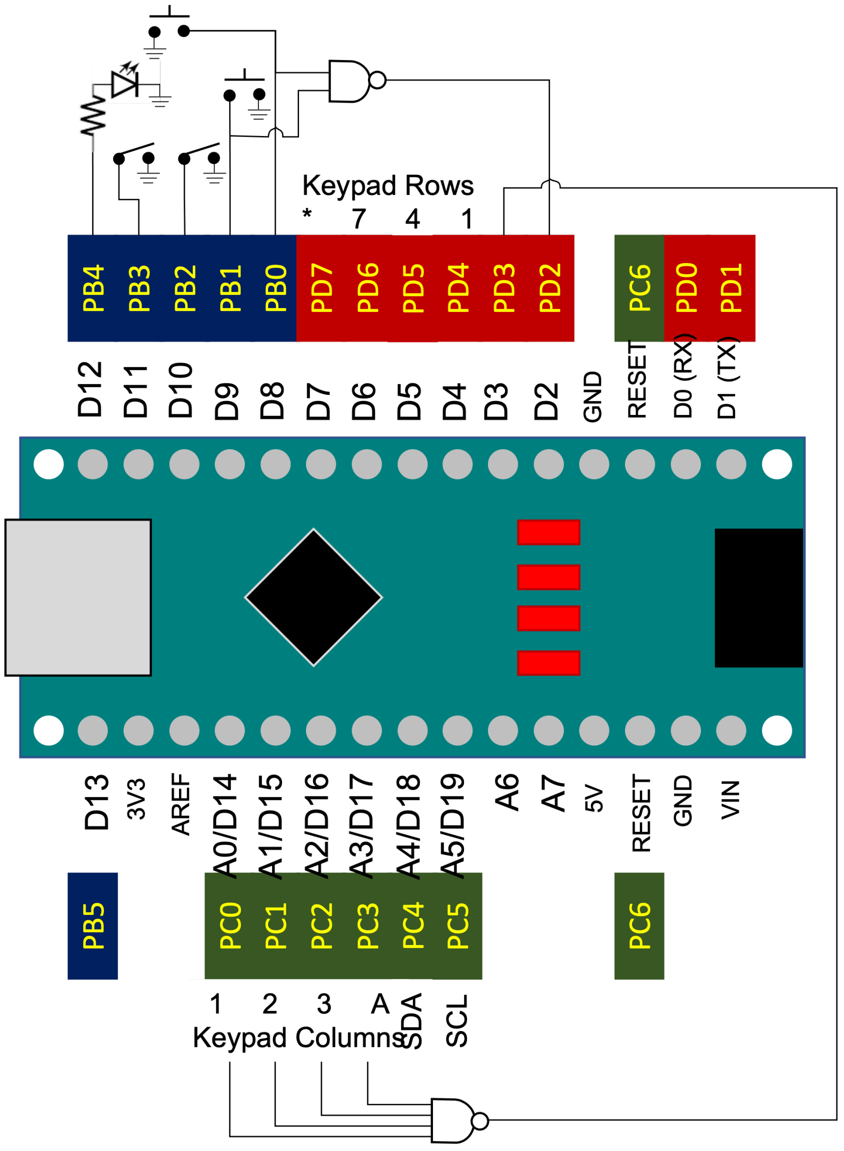
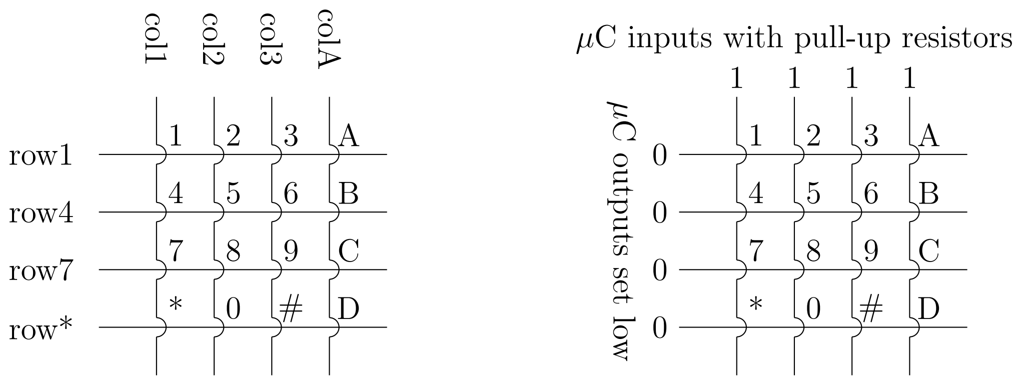


Figure 2: Pinout for a Cow Pi development board using the Arduino Nano and the I^{2}C serial communication protocol.

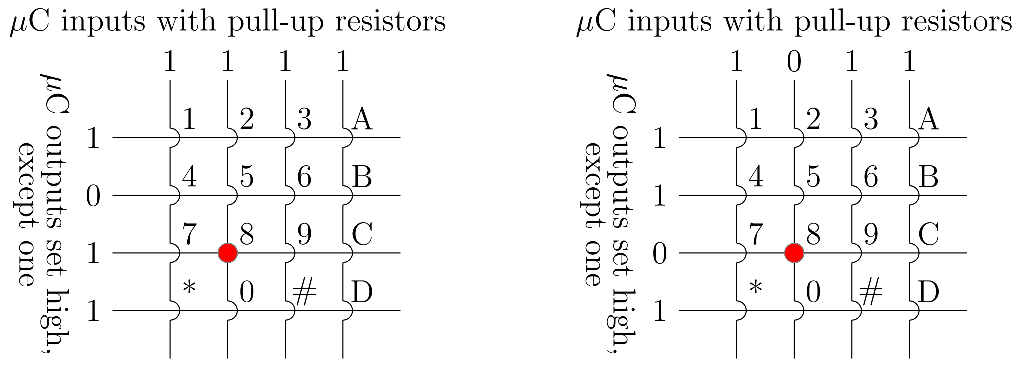


(a) Each key on the keypad is at the intersection of a row and a column.

(b) Detecting a keypress is possible by setting each row low and monitoring whether any column becomes low.

Figure 3: Preparing a matrix keypad to detect keypresses.

0xF, and so the pressed key is not in that row. But, as shown in Figure 4(b), if “row7” is the only logic-low row, then the column bit vector is not 0xF, and so the pressed key is in that row; moreover, because “col2” is now logic-low, the code can establish that the pressed key is at the intersection of “row7” and “col2,” *i.e.*, the “8” key.


μC inputs with pull-up resistors
1 1 1 1
μC outputs set high, except one
1
0
1
1
μC inputs with pull-up resistors
1 0 1 1
μC outputs set high, except one
1
1
0
1
1
2
3
A
4
5
6
B
7
8
9
C
*
0
#
D

(a) Examining a row that does not have a pressed key.

(b) Examining a row that does have a pressed key.

Figure 4: Scanning a matrix keypad. The red dot indicates which key is being pressed.

After the code has determined which row and column the pressed key is on, it can return a value or assign a value to a variable accordingly. This might be a `char` corresponding to the character on the key’s face, as is the case for `cowpi_get_keypress` (Section 2.4). Or this might be an `int` corresponding to the value of the numeral on the key’s face. Or this might even be some value unrelated to whatever is printed on the key’s face.

1.2.2 Scanning the Keypad

There are a few options for obtaining the value corresponding to a key that is pressed on the keypad. The most efficient for a simple application is to use a lookup table. For example, if you need to return a character that corresponds to the face value of the key that was pressed, then the lookup table would be:

$$keys \leftarrow \begin{pmatrix} '1' & '2' & '3' & 'A' \\ '4' & '5' & '6' & 'B' \\ '7' & '8' & '9' & 'C' \\ '*' & '0' & '\#' & 'D' \end{pmatrix}$$

If the keypad is wired to the Arduino Nano such that four contiguous output pins are connected to the rows and four contiguous input pins are connected to the columns (as is the case for the Cow Pi), then this pseudocode will scan the keypad and determine which key, if any, is pressed: Note that this pseudocode will report at most one key pressed; it would have to be modified to report multiple keys pressed. (This software limitation is not a limitation for mark 1 Cow Pis, as mark 1 Cow Pis have a hardware limitation: their keypads have no protection against shorting power to ground when two keys are pressed simultaneously.)

```
1 for each row:
2     row_bit_vector ← 0b1111      (* set all rows to 1 *)
3     row_bit_vector(row) ← 0      (* except the row we're currently examining *)
4     wait at least one microcontroller clock cycle
5     for each column:
6         if (column_bit_vector(column) = 0):
7             key_pressed ← keys(row,column)
8 row_bit_vector ← 0b0000          (* set all rows to 0 to detect the next keypress *)
```

The delay shown in line 4 is sometimes, but not always necessary. There is a slight delay between setting a pin's output value and being able to detect the change by reading a different pin's input value. Some realizations of the pseudocode attempt to read the change before it can be read reliably; this usually manifests as one of the keypad's columns not being readable. The fix is to introduce a delay of at least one clock cycle (strictly speaking, one clock cycle is more than enough, but a shorter delay is not possible). This could be managed by introducing a delay based on the microcontroller's clock cycle, using the AVR-libc `_NOP()` or `_delay_loop_1()` functions,[?][?] to be able to introduce a delay of exactly one or three clock cycles (`_NOP()` or `_delay_loop_1(1)`, respectively), or this could be managed by introducing a 1μs delay using the Arduino core library's `delayMicroseconds()` function,[?] which is portable across all devices using the Arduino toolchain.

1.3 Display Module

The display module used in the mark 1d Cow Pi is a 2×16 LCD dot-matrix character display. A full description of its control signals can be found in the HD44780U datasheet;[?] however, the CowPi library's functions handles these control signals – see Section 2.5. If a function to send a halfbyte to the display module is registered, either the CowPi library's default implementation or your own (Section 4.2.4), then you do not need to manage the control signals.

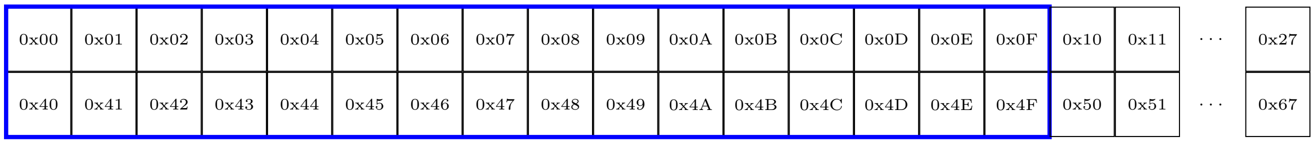


Figure 5: Initially, the characters at addresses 0x00–0x0F and 0x40–0x4F are displayed. The black squares are character memory locations; the blue rectangle shows which characters are displayed.

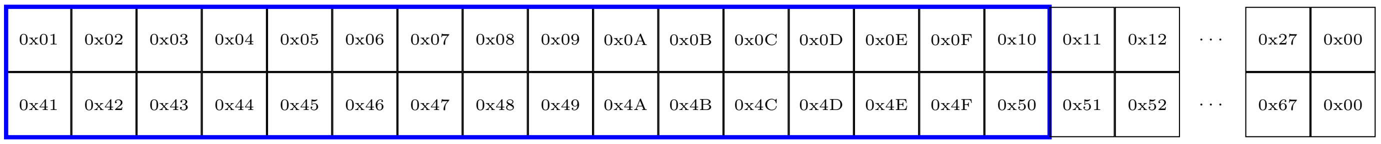


Figure 6: After shifting the display to the left, the characters at addresses 0x01–0x10 and 0x41–0x50 are displayed. The black squares are character memory locations; the blue rectangle shows which characters are displayed.

Displaying useful information on the display module does require some awareness of the dipslay module's characteristics. As previously stated, the display module can display up to 32 characters, 16 characters on each of two rows. When sending these characters using the library's `cowpi_lcd1602_place_character()` or `cowpi_lcd1602_send_character()` functions, you typically can use standard ASCII characters. Specifically, if the `character` argument is in the range 0x20–0x7D, then the corresponding ASCII character will be displayed. For the characters displayed when the argument is 0x7E, 0x7F, or in the range 0xA1–0xFF, see Table 4 of the HD44780U datasheet.[?] Up to eight custom characters can also be created; see the `cowpi_lcd1602_create_character()` function and the *lcd1602_custom_characters* example program described in Section 3.4.

While there are 16 characters displayed per row, the display module has memory addresses for 40 characters per row. The base address of the top row is 0x00, and the base address of the bottom row is 0x40. Initially, the characters at addresses 0x00–0x0F and 0x40–0x4F are displayed, as shown in Figure 5.

The displayed addresses can shift to the left or to the right. Normally the display will not shift when a character is displayed; however, the display can be configured to shift with each new character added. This is achieved by changing the entry mode with the `cowpi_lcd1602_send_command()` function. Alternatively, the display can be explicitly shifted, regardless of the entry mode. If the `cowpi_lcd1602_send_command()` function is used so send the `LCDSHIFT_DISPLAYLEFT` or the `LCD_DISPLAY_RIGHT` command, then the display will shift to the left or to the right accordingly.

For example, if the display is shifted to the left from its initial configuration, then the characters at addresses 0x01–0x10 and 0x41–0x50 are displayed, as shown in Figure 6.

If, when shifting, one "end" of the rows or the other is passed, then the display "wraps" around, displaying some characters at both ends of the rows' addresses. For example, if the display is shifted to the right from its initial configuration, then the characters at addresses 0x27, 0x00–0x0E and 0x67, 0x40–0x4E are displayed, as shown in Figure 7.

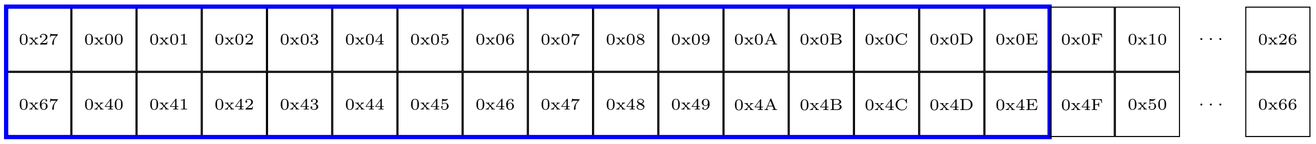

0x27 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F 0x10 ··· 0x26
0x67 0x40 0x41 0x42 0x43 0x44 0x45 0x46 0x47 0x48 0x49 0x4A 0x4B 0x4C 0x4D 0x4E 0x4F 0x50 ··· 0x66

Figure 7: When shifting, the display "wraps" around the rows' addresses. The black squares are character memory locations; the blue rectangle shows which characters are displayed.

2 Library Overview

This section describes the v0.4 CowPi library's functions relevant to the Cow Pi mk1d development board. More complete documentation can be obtained by processing the library's source code with Doxygen.

2.1 Configuring / Initializing the Board

Functions used to configure the Cow Pi hardware and library at the start of a program.

- `void cowpi_setup (unsigned int configuration)`

  Configures the microcontroller's pins for the expected hardware setup, and configures the library to work with the specified display module and communication protocol.

  parameters

  configuration bitwise disjunction of named constants, specifying the display module and protocol
  - For the Cow Pi mk1d, the typical argument will be `LCD1602|I2C`

  returns n/a

  notes
  - If the I^{2}C protocol is used, then `cowpi_set_display_i2c_address()` must be called *before* `cowpi_setup()`
  - If a non-`COWPI_DEFAULT` display dialect is used, then `cowpi_set_display_dialect()` must be called before `cowpi_setup()`
  - If `printf()` and/or `scanf()` will be used, then we recommend that `cowpi_stdio_setup()` be called before `cowpi_setup()`; however, this is not required

- `void cowpi_set_display_i2c_address (uint8_t peripheral_address)`

  Sets the I^{2}C address for an I^{2}C-driven display module.

  Because the I^{2}C protocol uses addresses to select the peripheral that the microcontroller is communicating with, the display module's address needs to be set. If the SPI protocol is being used, then there is no need to call this function.

The common PCF8574-based interfaces used with LCD1602 display modules typically have an address of 0x27, and we have encountered some interfaces with an address of 0x3F, but these addresses can readily be changed.

parameters

peripheral_address

returns n/a

notes

– If this function is called, it must be called before `cowpi_setup()` so that the display module can be properly configured.

- `void cowpi_set_display_dialect (uint8_t dialect)`

Sets the "dialect," or the mapping of protocol bits to display module bits.

Some display modules (e.g., MAX7219-based modules) have only one possible mapping, and calling this function has no effect for those modules. For other display modules, the `COWPI_DEFAULT` dialect is the default; this function does not need to be called if the `COWPI_DEFAULT` dialect will be used.

parameters

dialect a named constant specifying which mapping of protocol bits to display module bits shall be used

returns n/a

notes

– If this function is called, it must be called before `cowpi_setup()` so that the display module can be properly configured.

- `void cowpi_stdio_setup (unsigned long baud)`

Configures the CowPi library to use *stdio.h* functions.

Configures `printf()` to write to, and `scanf()` to read from, the serial interface between the microcontroller and the host computer. Calls to `printf()` and `scanf()` that occur before the call to `cowpi_stdio_setup()` will have no effect.

parameters

baud the USART signal rate

returns n/a

notes

– By default, `printf()` will not format floating point numbers

2.2 Simple Inputs

Buttons and switches that can be read directly from pins.

- `bool cowpi_left_button_is_pressed (void)`

  Reports whether the left button is pressed.

  There is no debouncing. This is a portable implementation, not a memory-mapped implementation.

  Assumes the left button is in Arduino pin D8. A pressed button grounds a pulled-high input.

  parameters n/a

  returns `true` if the button is pressed, `false` otherwise

- `bool cowpi_right_button_is_pressed (void)`

  Reports whether the right button is pressed.

  There is no debouncing. This is a portable implementation, not a memory-mapped implementation.

  Assumes the left button is in Arduino pin D9. A pressed button grounds a pulled-high input.

  parameters n/a

  returns `true` if the button is pressed, `false` otherwise

- `bool cowpi_left_switch_is_in_left_position (void)`

  Reports whether the left switch is in the left position.

  There is no debouncing. This is a portable implementation, not a memory-mapped implementation.

  Assumes the left switch is in Arduino pin A4 (D18) if SPI (but not I^{2}C) is in use or if no protocol is in use; assumes the switch is in pin D11 if I^{2}C (but not SPI) is in use. If both protocols are in use, then this function will always return `false`. A switch in the left position grounds a pulled-high input.

  parameters n/a

  returns `true` if the switch is in the left position, `false` otherwise

- `bool cowpi_left_switch_is_in_right_position (void)`

  Reports whether the left switch is in the right position.

  There is no debouncing. This is a portable implementation, not a memory-mapped implementation.

  Assumes the left switch is in Arduino pin A4 (D18) if SPI (but not I^{2}C) is in use or if no protocol is in use; assumes the switch is in pin D11 if I^{2}C (but not SPI) is in use. If both protocols are in use, then this function will always return `false`. A switch in the right position floats, allowing a pulled-high input to remain high.

  parameters n/a

  returns `true` if the switch is in the right position, `false` otherwise

- `bool cowpi_right_switch_is_in_left_position (void)`

  Reports whether the right switch is in the left position.

  There is no debouncing. This is a portable implementation, not a memory-mapped implementation.

  Assumes the right switch is in Arduino pin A5 (D19) if SPI (but not I^{2}C) is in use or if no protocol is in use; assumes the switch is in pin D10 if I^{2}C (but not SPI) is in use. If both protocols are in use, then this function will always return `false`. A switch in the left position grounds a pulled-high input.

  parameters n/a

  returns `true` if the switch is in the left position, `false` otherwise

- `bool cowpi_right_switch_is_in_right_position (void)`

  Reports whether the right switch is in the right position.

  There is no debouncing. This is a portable implementation, not a memory-mapped implementation.

  Assumes the right switch is in Arduino pin A5 (D19) if SPI (but not I^{2}C) is in use or if no protocol is in use; assumes the switch is in pin D10 if I^{2}C (but not SPI) is in use. If both protocols are in use, then this function will always return `false`. A switch in the right position floats, allowing a pulled-high input to remain high.

  parameters n/a

  returns `true` if the switch is in the right position, `false` otherwise

2.3 Simple Outputs

LEDs that can be written directly through pins.

- `void cowpi_illuminate_left_led (void)`

  Illuminates the left LED, aka the built-in LED, aka the internal LED. This is a portable implementation, not a memory-mapped implementation.

  Assumes the left LED is on Arduino pin D13.

  The Arduino semantics are that an LED illuminates when the pin is placed high.

  parameters n/a

  returns n/a

- `void cowpi_deluminate_left_led (void)`

  Deluminates the left LED, aka the built-in LED, aka the internal LED. This is a portable implementation, not a memory-mapped implementation.

  Assumes the left LED is on Arduino pin D13.

  The Arduino semantics are that an LED deluminates when the pin is placed low.

  parameters n/a

  returns n/a

- `void cowpi_illuminate_right_led (void)`

  Illuminates the right LED, aka the external LED. This is a portable implementation, not a memory-mapped implementation.

  Assumes the right LED is on Arduino pin D12.

  An LED illuminates when the pin is placed high, to match the semantics of Arduino's built-in LED.

  parameters n/a

  returns n/a

- `void cowpi_deluminate_right_led (void)`

  Deluminates the right LED, aka the external LED. This is a portable implementation, not a memory-mapped implementation.

Assumes the right LED is on Arduino pin D12.

An LED deluminates when the pin is placed low, to match the semantics of Arduino's built-in LED.

parameters n/a

returns n/a

2.4 Scanned Inputs

Keypad that is scanned through a combination of writing to pins and reading from other pins.

- `char cowpi_get_keypress (void)`

  Scans the keypad to determine which, if any, key was pressed.

  There is no debouncing. This is a portable implementation, not a memory-mapped implementation. Returns the ASCII representation of the character depicted on whichever key was pressed (0-9, A-D, *, #).

  Assumes a common 4x4 matrix keypad with the rows in Arduino pins D4-D7 and the columns in Arduino pins A0-A3 (D14-D17). A pressed key grounds a pulled-high input.

  parameters n/a

  returns ASCII character corresponding to the key that is pressed, or `NUL` if no key is pressed

2.5 Display Modules

Functions for display modules.

- `typedef void (*cowpi_lcd1602_send_halfbyte_t) (uint8_t halfbyte, bool is_command)`

  Convenience type for a pointer to a function that sends a halfbyte to the LCD1602 display.

- `void (*cowpi_lcd1602_send_halfbyte) (uint8_t halfbyte, bool is_command)`
  declared as `cowpi_lcd1602_send_halfbyte_t cowpi_lcd1602_send_halfbyte`

  Pointer to a function that sends a halfbyte to the LCD1602 display.

  The `cowpi_lcd1602` utility functions all make use of the `cowpi_lcd1602_send_halfbyte()` function pointer.

Initially, `cowpi_lcd1602_send_halfbyte()` is one of two default implementations, depending on the protocol specified in the argument to `cowpi_setup()`. The SPI implementation is a portable software-only implementation that does not use SPI hardware, nor does it use memory-mapped I/O of any form. The I^{2}C implementation currently makes use of the I^{2}C hardware using avr-libc macros.

Re-implementing this function to use SPI or I^{2}C hardware is a possible part of a memory-mapped I/O lab assignment.

SPI: Assumes the display module's data-in line is connected to the microcontroller's COPI pin (D11 on most Arduino boards), the display module's serial-clock line is connected to the microcontroller's SCK pin (D13 on most Arduino boards), and the display module's chip-select line is connected to Arduino pin D10.

I^{2}C: Assumes the display module's data line is connected to the microcontroller's SDA pin (D18 on most Arduino boards) and the display module's serial-clock line is connected to the microcontroller's SCL pin (D19 on most Arduino boards).

parameters

halfbyte the data to be sent to the display module

is_command indicates whether the data is part of a command (`true`) or part of a character (`false`)

returns n/a

notes

- Application developers cannot access this function pointer directly; they will assign the function to the function pointer through `cowpi_lcd1602_set_send_halfbyte()`, and the myriad `cowpi_lcd1602` library functions will use the assigned function.
- **TODO**: finish implementing portable software-only I^{2}C implementation.

- `void cowpi_lcd1602_set_send_halfbyte (cowpi_lcd1602_send_halfbyte_t send_halfbyte_fu`

Reassigns the `cowpi_lcd1602_send_halfbyte` function pointer to point to the specified function.

During setup, this function is used to assign one of the two default `cowpi_lcd1602_send_halfbyte_t` implementations to `cowpi_lcd1602_send_halfbyte`, unless it was previously used to assign a reimplementation. It can also later be used to assign a re-implementation.

parameters

send_halfbyte_function the function to be used to send halfbytes to the LCD1602 display module

returns n/a

notes n/a

- `void cowpi_lcd1602_create_character (uint8_t encoding, const uint8_t pixel_vector[8]`

  Creates a custom character for the LCD1602 display module.

  The encoding can be a value between 0 and 7, inclusive. Each element of the pixel vector corresponds to a row of the 5 × 8 character matrix (thus, only 5 bits of each element are used).

  parameters

  encoding the value used to represent the character

  pixel_vector identifies which LCDs are "on" or "off" for the custom character

  returns n/a

  notes n/a

- `void cowpi_lcd1602_place_character (uint8_t address, uint8_t character)`

  Places the specified character on the display at the specified location.

  The character is an ASCII or "extended-ASCII" character, or a custom character created by using `cowpi_lcd1602_create_character()`.

  parameters

  address the address of the desired location

  character the character to be displayed

  returns n/a

  notes

  - The base address of the top row is 0x00, and the base address of the bottom row is 0x40.

- `void cowpi_lcd1602_place_cursor (uint8_t address)`

  Places the cursor at the specified location without updating the display.

  While the cursor will move to the specified location, it will only be visibly there if the cursor is on.

  parameters

  address the address of the desired location

  returns n/a

  notes

  - The base address of the top row is 0x00, and the base address of the bottom row is 0x40.

- `void cowpi_lcd1602_return_home (void)`

  Places the cursor in row 0, column 0.

  If the display was shifted, the display is shifted back to its original position. The contents of each character position remain unchanged. remain unchanged.

  parameters n/a

  returns n/a

- `void cowpi_lcd1602_send_character (uint8_t character)`

  Displays a character at the cursor's location on the LCD1602 display module.

  The character is an ASCII or "extended-ASCII" character, or a custom character created by using `cowpi_lcd1602_create_character()`.

  parameters

  character the character to be displayed

  returns n/a

- `void cowpi_lcd1602_send_command (uint8_t command)`

  Sends a command to the LCD1602 display module. The command is a bitwise disjunction of named constants to specify the entry mode (cursor and text movement to occur after characters are sent), a disjunction of named constants to specify the display mode (whether the display is on, whether the underscore cursor is displayed, and whether the cursor's location blinks), or one shift command (shift display left/right or shift cursor left/right).

  parameters

  command

  returns n/a

  notes The possible commands are:

  - Entry Mode Commands (apply bitwise disjunction)

    `LCDENTRY_CURSORMOVESRIGHT` Instructs the display module to move the cursor right after a character is displayed

    * CowPi library initializes display with the cursor moving right

    `LCDENTRY_CURSORMOVESLEFT` Instructs the display module to move the cursor left after a character is displayed

- ∗ If an entry mode command is sent without specifying the direction of cursor movement, the LCD1602 defaults to the cursor moving left cursor movement, the LCD1602 defaults to cursor moving left

`LCDENTRY_TEXTSHIFTS` Instructs the display module to shift the entire display after a character is displayed

`LCDENTRY_TEXTNOSHIFT` Instructs the display module not to shift the display after a character is displayed

- ∗ CowPi library initializes display with the text not shifting
- ∗ If an entry mode command is sent without specifying whether to shift the text, the LCD1602 defaults to the text not shifting

– Display Mode Commands (apply bitwise disjunction)

`LCDONOFF_DISPLAYON` Instructs the display module to turn on the display

- ∗ CowPi library initializes display with the display on

`LCDONOFF_DISPLAYOFF` Instructs the display module to turn off the display

- ∗ If a display mode command is sent without specifying whether to turn the display on, the LCD1602 defaults to turning the display off

`LCDONOFF_CURSORON` Instructs the display module to turn on the underscore cursor

`LCDONOFF_CURSOROFF` Instructs the display module to turn off the underscore cursor

- ∗ CowPi library initializes display with the cursor off
- ∗ If a display mode command is sent without specifying whether to turn the cursor on, the LCD1602 defaults to turning the cursor off

`LCDONOFF_BLINKON` Instructs the display module to blink the cursor's position

`LCDONOFF_BLINKOFF` Instructs the display module not to blink the cursor's position

- ∗ CowPi library initializes display with the cursor not blinking
- ∗ If a display mode command is sent without specifying whether to have the cursor blink, the LCD1602 defaults to having the cursor not blink

– Shift Commands (apply only one at a time)

`LCDSHIFT_DISPLAYLEFT` Shifts the entire display to the left

`LCDSHIFT_DISPLAYRIGHT` Shifts the entire display to the right

`LCDSHIFT_CURSORLEFT` Shifts the cursor to the left

`LCDSHIFT_CURSORRIGHT` Shifts the cursor to the right

- `void cowpi_lcd1602_set_backlight (bool backlight_on)`

Turns the display module's backlight on or off.

parameters

backlight_on indicates whether the backlight should be on (`true`) or off (`false`)

returns n/a

- `void cowpi_lcd1602_clear_display (void)`

  Clears the display and places the cursor in row 0, column 0.

  parameters n/a

  returns n/a

3 Example Programs

The CowPi v0.4 library has a few example programs to demonstrate the use of the library's functions with the Cow Pi mk1d.

3.1 `stdio` Demonstration

The avr-libc library used by gcc when compiling programs for the ATmega328P does not normally have a couple of key `stdio` functions defined, making most of the rest of the `stdio` functions unusable. For most microcontroller applications, this is a reasonable decision: there usually isn't a terminal connected to an embedded system, and by omitting the `stdio` functions, the firmware uploaded to the microcontroller has a smaller footprint. (Consider, for example, that the ATmega328P has 32KB of flash memory for the firmware and 2KB of SRAM for the program stack and heap – even an increase of 128 bytes is a significant fraction of the available memory.) The Arduino toolchain provides `Serial.print()` and `Serial.println()` to overcome the absence of `printf()`, and `Serial.readXXX()` and `Serial.parseXXX()` to overcome the absence of `scanf()`.

If a developer is willing to accept a slightly larger binary, the `printf()` and `scanf()` offer significant benefits to application developers, including being functions that are familiar, and providing far more concise ways to read and write non-trivial inputs and outputs. For this reason, the CowPi library implements the missing `stdio` functions to allow application developers to choose to include `stdio` functions or not, based on their needs.

NOTES

- The CowPi library's implementation of these functions includes no software buffering. While this keeps the memory footprint down, it does mean that `printf()` and `scanf()` are slow.

- As is common in most microcontroller toolchains, `printf()` and `scanf()` do not process floating point values by default. This keeps the memory footprint down, and it is very rare that a microcontroller application would use floating point values. The avr-libc library achieves this by aliasing `printf()` to the integer-only `iprintf()`. If you need to print or scan floating point values, you can undo the aliasing with the pre-processor directive `#define printf`.

We shall now examine *stdio_demonstration.ino*:

```
 1 #include "CowPi.h"
 2
 3 void setup() {
 4     cowpi_stdio_setup(9600);
 5     cowpi_setup(0);
 6
 7     printf("Unlike normal Arduino-based systems, CowPi lets you use `prin
 8     printf("\tbut only if you call `cowpi_stdio_setup` first.\n");
 9
10     Serial.println(F("If you plan to use Arduino's `F()` macro to save RA
11     Serial.println(F("\tand/or `Serial.println`."));
12
13     char s[83];
14     printf("%s", strcpy_P(s, PSTR("But using the avr-libc `PSTR()` macro
15     printf("%s", strcpy_P(s, PSTR("\t`strcpy_P` function is a way to put
16     printf("%s", strcpy_P(s, PSTR("\tand use them with `printf`.\n")));
17
18     printf("%s", strcpy_P(s, PSTR("Floating point conversion won't work w
19     printf("%s", strcpy_P(s, PSTR("\ttwice about using floats on a microc
20 }
21
22 void loop() {
23     char s[83];
24     char c;
25     int i;
26     printf("%s", strcpy_P(s, PSTR("Enter an integer from the host compute
27     scanf("%d", &i);
28     scanf("%c", &c);    // consume the linefeed character
29     printf("\nYou entered: %d\n", i);
30     printf("%s", strcpy_P(s, PSTR("Enter a character from the host comput
31     scanf("%c", &c);
32     printf("\nYou entered: %c\n", c);
33     printf("%s", strcpy_P(s, PSTR("Enter a string from the host computer'
34     scanf("%80s", s);
35     printf("\nYou entered: %s\n", s);
36 }
```

The call to `cowpi_stdio_setup()` initializes the USART connection to the serial terminal and then links the CowPi-implemented functions to avr-libc's `stdio` stubs. The call to `cowpi_setup()` is provided no non-zero arguments because this particular example makes no use of peripherals. (A typical application program would, of course, provide useful arguments.)

The rest of the `setup()` function demonstrates the use of `printf()`. The simplest way to use `printf()` is to use it like you would on a desktop system or server (lines 7–8).

Due to the limited amount of SRAM, you may wish to store string constants in flash memory instead of SRAM. While avr-gcc will place string constants in flash memory by default microcontrollers that map instruction memory into the data memory address space, such as the ATmega4809, it does not do so for microcontrollers that have fully-distinct instruction and data address spaces, such as the ATmega328P. The Arduino toolchain includes the `F()` macro that places string constants in flash memory and generates pointers that can be used by `Serial.printXX()` functions (lines 10–11). The `F()` macro does not work with `printf()`, but a work-around is shown in lines 13–19:

- Create a buffer large enough to hold the largest string constant (here, `char s[83]`)
- Use the avr-libc `PSTR()` macro to place string constants in flash memory
- Use the avr-libc `strcpy_P()` function to copy string constants from flash memory into the buffer as needed
  - ***NOTE:*** *`strcpy_P()`, like `strcpy()`, returns a pointer to the destination buffer, which is in data memory*
- Use the format specifier `%s` in the `printf()` format string to print the string at the address returned by `strcpy_P()`

The `loop()` function demonstrates the use of `scanf()`, which has no nuances other than those that are part of `scanf()`'s specification.

3.2 Simple I/O Demonstration

This example program makes use of all the Cow Pi's peripherals except its display module. While this program's principal use is to test whether a mark 1 Cow Pi or a mark 2 Cow Pi has been assembled correctly (and so is named *simple_io_test.ino*), it also demonstrates the use of the functions described in Sections 2.2–2.4.

We shall now examine *simple_io_test.ino*:

```
#include "CowPi.h"

void setup() {
    cowpi_stdio_setup(9600);
    char s[83];
    printf("CowPi library version %s\n", COWPI_VERSION);
    printf("%s", strcpy_P(s, PSTR("This demonstration makes no assumption
    printf("%s", strcpy_P(s, PSTR("\tso your display module may display g
    printf("%s", strcpy_P(s, PSTR("The `cowpi_setup` function will be cal
    printf("%s", strcpy_P(s, PSTR("\tthe CowPi library knows where your s
    printf("%s", strcpy_P(s, PSTR("\tYour instructor should have told you

    cowpi_setup(SPI);
//    cowpi_setup(I2C);

```

```
16      printf("%s", strcpy_P(s, PSTR("The simple I/O test will print the sta
17      printf("%s", strcpy_P(s, PSTR("\tbutton, switch, and LED every half-s
18      printf("%s", strcpy_P(s, PSTR("Press the Enter key on your host compu
19      scanf("%c", s);
20  }
21
22  void loop() {
23      bool left_led, right_led;
24      char c;
25      printf("\n");
26      printf("Keypad:      %c        Column pins:  %d%d%d%d    Keypad NAND:
27             digitalRead(14), digitalRead(15), digitalRead(16), digitalRead
28      printf("Left switch: %s    Right switch: %s\n",
29             cowpi_left_switch_is_in_left_position() ? "LEFT " : "RIGHT",
30             cowpi_right_switch_is_in_left_position() ? "LEFT " : "RIGHT");
31      printf("Left button: %s    Right button: %s",
32             cowpi_left_button_is_pressed() ? "DOWN " : "UP   ",
33             cowpi_right_button_is_pressed() ? "DOWN " : "UP   ");
34  #if defined ARDUINO_AVR_UNO || defined ARDUINO_AVR_NANO || defined ARDUIN
35      printf("   Button NAND: %d", digitalRead(2));
36  #endif
37      printf("\n");
38      if (cowpi_left_button_is_pressed() && cowpi_left_switch_is_in_right_p
39          left_led = true;
40          cowpi_illuminate_left_led();
41      } else {
42          left_led = false;
43          cowpi_deluminate_left_led();
44      }
45      if (cowpi_right_button_is_pressed() && cowpi_right_switch_is_in_right
46          right_led = true;
47          cowpi_illuminate_right_led();
48      } else {
49          right_led = false;
50          cowpi_deluminate_right_led();
51      }
52      printf("Left LED:    %s      Right LED:    %s\n", left_led ? "ON " :
53      delay(500);
54  }
```

Because this program does not use the display module, the call to `cowpi_setup()` does not specify a display module. We do, however, need to specify which serial communication protocol will be used with the display module (lines 13–14) because that affects which pins the toggleable switches are connected to for Arduino-based Cow Pi development boards. For the Cow Pi mk1d, specify the I^{2}C protocol.

The function calls on lines 27 and 35 are to `digitalRead()` from the Arduino toolchain. If you're using the CowPi library, using `digitalRead()` (and `digitalWrite()`) is generally frowned-upon. Much more efficient code is possible by using I/O registers (see Section 4); however, in this case we wanted the example to be portable across many microcontrollers. Directly reading individual keypad column pins or the NAND pins isn't a likely scenario in application code, so the CowPi library doesn't have functions to support it. (Reading the keypad column pins would be part of code that scans the keypad, and the NAND pins would be used to trigger external interrupts (see Section 5)).

The call to `cowpi_get_keypress()` on line 26 (truncated here but fully-visible in the copy available through the Arduino IDE's *Examples* menu) demonstrates that the function behaves as specified: it returns a `NUL` character when no key is pressed, and a printable character when a key is pressed. We see on lines 29–30, 38, and 45, that the left and right switches can be independently checked for being in the left or right position. Similarly, on lines 32–33, 38, and 45, we see that the left and right buttons can be independently checked to determine whether one is pressed. Finally, on lines 40, 43, 47, and 50, we see that the left and right LEDs can be independently illuminated and deluminated.

3.3 Hello World for the LCD1602 Display Module

Let us now consider a simple program that uses the display module.

```
#include "CowPi.h"

void setup() {
    cowpi_stdio_setup(9600);

    unsigned int protocol;

    /* Select either the SPI protocol or the I2C protocol */
//    protocol = SPI;
    protocol = I2C;

    /* The COWPI_DEFAULT mapping from the serial adapter to the LCD1602 m
     * explicitly set, but if a different mapping needs to be used, then
//    cowpi_set_display_dialect(ADAFRUIT);

    /* If I2C, then the I2C address and possibly dialect (I2C-to-LCD1602
     * need to be set before calling `cowpi_setup`. */
    if (protocol == I2C) {
        cowpi_set_display_i2c_address(0x27);
    }

    cowpi_setup(LCD1602 | protocol);

    printf("You should see the backlight blinking.\n");
```

```
25      printf("Adjust the contrast until you can see the \"Hello, world!\" m
26
27      if (protocol == I2C) {
28          uint8_t i2c_address = cowpi_get_display_i2c_address();
29          cowpi_lcd1602_place_cursor(0x4F);
30          cowpi_lcd1602_send_command(LCDENTRY_CURSORMOVESLEFT);
31          cowpi_lcd1602_send_character((i2c_address & 0xF) + '0');
32          cowpi_lcd1602_send_character(((i2c_address >> 4) & 0xF) + '0');
33          const char backwards_message[] = "x0=sserdda c2i";
34          const char *c = backwards_message;
35          while (*c != '\0') {
36              cowpi_lcd1602_send_character(*c++);
37          }
38          cowpi_lcd1602_return_home();
39          cowpi_lcd1602_send_command(LCDENTRY_CURSORMOVESRIGHT);
40      }
41      const char hello_world[] = "Hello, world!";
42      const char *c = hello_world;
43      while (*c != '\0') {
44          cowpi_lcd1602_send_character(*c++);
45      }
46  }
47
48  void loop() {
49      static bool backlight_is_on = false;
50      backlight_is_on = !backlight_is_on;
51      cowpi_lcd1602_set_backlight(backlight_is_on);
52      delay(500);
53  }
```

In general, there is no need to setup `stdio` (line 4); however in this case we do so that we can provide information to someone who is assembling a Cow Pi development board (lines 24–25).

Unlike the simple I/O demonstration, since this example uses a display module, we do need to specify which display module we're using on line 22 so that the display module can be configured properly. We also specify whether to use SPI or I^2C. If using I^2C, we need to call `cowpi_set_display_i2c_address()` (line 19) before calling `cowpi_setup()`. For LCD1602 serial adapters, specifying `0x27` as the address is a very good choice, and `0x3F` is a good choice if `0x27` does not work for an unmodified serial adapter, unless you know that your adapter has been modified to use a different address. If `0x27` and `0x3F` don't work when assembling a Cow Pi development board, double-check the wiring; if the wiring is good, then run a program to scan the I^2C for peripherals and report their addresses.[2]

Some of the functionality demonstrated in this program is:

[2] Such a program is available on the Arduino website: https://playground.arduino.cc/Main/I2cScanner/.

- Using `cowpi_lcd1602_place_cursor()` (line 29) to position the cursor on the bottom row, column 15
- Using `cowpi_lcd1602_return_home()` (line 38) to position the cursor on the top row, column 0
- Using `cowpi_lcd1602_send_command()` (lines 30 & 39) to change the direction that the cursor moves after placing a character
  - Note that since we didn't specify whether the text should shift after placing a character, the display module defaults to the text remaining stationary on the display
- Using `cowpi_lcd1602_send_character()` (lines 31, 32, 36 & 44)
  - On lines 31–32 we are obtaining the character representation of the address's hexadecimal address by adding the integer value of the character '0' to each of the address's halfbyte[3]
  - Notice that on lines 31–37 we are printing a "backwards message" from right to left, so that it appears correct when read from left to right
  - Note the idiom used in lines 35–37 and 43–45 to iterate over a string; alternatively:
    ```
    for (const char *c = the_message; c != '\0'; c++) {
        cowpi_lcd1602_send_character(c);
    }
    ```
    (the principal advantage of the form that appears in the example program is that it would immediately stop on an empty string)

The `loop()` function simply turns the display module's backlight on and off every half-second so that someone assembling a Cow Pi development board knows that they have successfully connected to the display module, even if they don't see any text.

3.4 Adding Custom Characters to the LCD1602 Display Module

Up to eight (8) custom characters can be programmed into the LCD1602 display module, corresponding to characters 0–7. In this program, we create three characters to produce a crude animation of a stick figure running:

```
#include "CowPi.h"

const uint8_t runner[][8] = {
        {0x06, 0x06, 0x0c, 0x16, 0x04, 0x06, 0x09, 0x01},
        {0x06,0x06,0x0c,0x0f,0x04,0x0c,0x12,0x02},
        {0x06,0x06,0x04,0x0e,0x05,0x04,0x06,0x08}
};

```

[3]These lines have a bug in that they will not correctly display hex digits `A`–`F`.

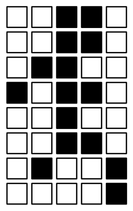
```
const int8_t number_of_frames = 3;

void setup() {
    cowpi_stdio_setup(9600);

    unsigned int protocol;

    /* Select either the SPI protocol or the I2C protocol */
//    protocol = SPI;
    protocol = I2C;

    /* The COWPI_DEFAULT mapping from the serial adapter to the LCD1602 m
     * explicitly set, but if a different mapping needs to be used, then
//    cowpi_set_display_dialect(ADAFRUIT);

    /* If I2C, then the I2C address and possibly dialect (I2C-to-LCD1602
     * need to be set before calling `cowpi_setup`. */
    if (protocol == I2C) {
        cowpi_set_display_i2c_address(0x27);
    }

    cowpi_setup(LCD1602 | protocol);

    for (int i = 0; i < number_of_frames; i++) {
        cowpi_lcd1602_create_character(i, runner[i]);
    }
    cowpi_lcd1602_set_backlight(true);
    cowpi_lcd1602_place_character(0, 0);
}

void loop() {
    static uint8_t frame = 0;
    static uint8_t position = 0;
    delay(250);
    cowpi_lcd1602_place_character(position, ' ');
    if (++position == 0x10) {
        position = 0x40;
    } else if (position == 0x50) {
        position = 0;
    }
    if (++frame == number_of_frames) {
        frame = 0;
    }
    cowpi_lcd1602_place_character(position, frame);
}
```

We'll examine creating custom characters after looking at some other aspects of the program.

- This program uses `cowpi_lcd1602_place_character()` (lines 36, 43, & 52) instead of `cowpi_lcd1602_send_character()`; the difference is that `cowpi_lcd1602_send_character()` places a character at the cursor's location and then moves the cursor, whereas `cowpi_lcd1602_place_` places a character at a location specified in its arguments and does not change the cursor's location
- On lines 44 & 46, we check whether the next position would be located off of the 16-column screen (column 16 is a valid off-screen column but we don't want to place our stick figure someplace that we can't see it)
- The custom characters are programmed into the display module with `cowpi_lcd1602_create_charac` (line 33)

The second argument to `cowpi_lcd1602_create_character()` is an 8-element array of bitvectors. Each bitvector corresponds to one of the eight rows in a 5×8 character matrix. Even though a bitvector is stored as a `uint8_t`, only the five least-significant bits are used. Of these five bits, a 0 indicates that the corresponding pixel is "off," and a indicates that the corresponding pixel is 'on'. For example, consider the first element in the `runner` matrix:
`{0x06, 0x06, 0x0c, 0x16, 0x04, 0x06, 0x09, 0x01}`
Viewed in binary (and expressing only the five least-significant bits in each bitvector), that's
`{0b00110, 0b00110, 0b01100, 0b10110, 0b00100, 0b00110, 0b01001, 0b00001}`

We can then see one of the frames for our running stick figure by drawing an empty square wherever there's a 0 and a filled square wherever there's a 1.



0*b*00110
0*b*00110
0*b*01100
0*b*10110
0*b*00100
0*b*00110
0*b*01001
0*b*00001

4 Input/Output Register Descriptions

The ATmega328P microcontroller has the AVR instruction set's original port-mapped input/output registers; however, it also has a much more extensive set of memory-mapped input/output registers. We recommend using the memory-mapped I/O registers and describe them here.

The tables in Section 4 include each register's address offset, in bytes, from the base address of the I/O region of the ATmega328P's memory address space. The CowPi library provides a constant for that base address, `COWPI_IO_BASE`.

Name	Offset	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTD	0x0B	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
DDRD	0x0A	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
PIND	0x09	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
PORTC	0x08	—	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
DDRC	0x07	—	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
PINC	0x06	—	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
PORTB	0x05	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
DDRB	0x04	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
PINB	0x03	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0

Table 1: ATmega328P I/O port registers. Original data from ATmega382P datasheet, §30.[?]

4.1 External Pins Input/Output

The ATmega328P microcontroller has three input/output ports accessible by external pins. Each port has three registers, the PIN input register, the PORT output register, and the DDR data direction register used to set each pin as input or output. Each pin is individually controlled by a particular bit in the port registers. Table 1 shows these nine registers and their corresponding address offsets from the I/O base address. *You do not need to configure the pins' directions for input or output; the `cowpi_setup()` function takes care of all necessary configuration.*

Figure 2 shows which bit in which port corresponds to each Arduino Nano pin. For example, pin `D10` is labeled "PB2" indicating that it is part of port B and uses bit 2 in each of port B's registers. If `D10` were an input pin, then we could determine the pin's logic level by using a bitmask to examine `PINB`'s bit 2. On the other hand, if `D10` were an output pin, then we could set the pin's logic level to high or low by assigning a 1 or 0, respectively, to `PORTB`'s bit 2, using the read/modify/write pattern.

4.1.1 Structure for Memory-Mapped Input/Output

The CowPi library provides data structures to access the memory-mapped I/O registers in a more readable form. Specifically, the `cowpi_ioport_t` structure eliminates the need to remember which I/O port registers are used for output to peripherals and which are used for input from peripherals.

```

/**
 * @brief Structure for the general-purpose I/O pins.
 *
 * An array of these structures can be indexed using named constants
 * (COWPI_PB, etc).
 */
typedef struct {
    uint8_t input;                  //!< Read inputs from field (PINx)
    uint8_t direction;              //!< Set the pin's direction using this fie
    uint8_t output;                 //!< Write outputs to this field, and set/
```

The ATmega328P's three I/O ports are placed contiguously in the memory address space, which will allow us to create a pointer to the lowest-addressed port (0x03 bytes above `COWPI_IO_BASE`, per Table 1) and then treat that pointer as an array of I/O ports. Some named constants that we can use to index that array further eliminate the need to remember which port corresponds to each Arduino Nano pin.

```

#define COWPI_PB   0                    //!< Index for arrays to access PINB/DDRB/PORT
#define D8_D13     0                    //!< Alias of COWPI_PB corresponding to pins D
#define COWPI_PC   1                    //!< Index for arrays to access PINC/DDRC/PORT
#define A0_A5      1                    //!< Alias of COWPI_PC for corresponding to pi
#define D14_D19    1                    //!< Alias of COWPI_PC for corresponding to pi
#define A0_A7      1                    //!< Alias of COWPI_PC for corresponding to pi
#define D14_D21    1                    //!< Alias of COWPI_PC for corresponding to pi
#define COWPI_PD   2                    //!< Index for arrays to access PIND/DDRD/PORT
```

We recommend using `D0_D7`, `D8_D13`, and `D14_D19`. Using our earlier hypotheticals:

If `D10` were an input pin, then we could determine the pin's logic level with C code similar to this:

```
    volatile cowpi_ioport_t *ioports =
        (cowpi_ioport_t *)(COWPI_IO_BASE + 0x3); // an array of I/O ports
    uint8_t logic_level = ioports[D8_D13].input & (1 << (10-8));
```

In the first two lines, we created our array of `cowpi_ioport_t` structures and assigned the array's base address to three bytes above the I/O base address. Most likely, you would only need to do this once per program. In the final line, we indexed the array using a named constant. The convenient mapping of the Arduino Nano's pins to the ATMega328P's I/O registers allows us to use named constants whose names help us remember which constant is appropriate for the pin we're using. After indexing the array, we select the `input` field because in this hypothetical, pin `D10` is an input pin. We use a bitmask so that we only capture the logic level of the pin we're interested in. Both `0x04` and `0b00000100` would be entirely suitable literal masks, and a mask created from a bitshift (*i.e.*, `(1 << 2)`) is also appropriate. Here we used `(1 << (10-8))` because the convenient mapping of pins to registers allows us to create a mask from a bitshift without having to think about how many positions to shift – we simply subtracted the pin number (10) from the lowest-number pin in this bank (8).

Of course, in this example, `logic_level` would take on either a zero or non-zero value, which is fine for most applications. If `logic_level` must take on either zero or one, then you could either shift the bits:

```
    uint8_t logic_level = (ioports[D8_D13].input & (1 << (10-8)))
                                                   >> (1 >> (10-8));
```

or double-negate:

```
    uint8_t logic_level = !!(ioports[D8_D13].input & (1 << (10-8)));
```

On the other hand, if `D10` were an output pin, then we could set the pin's logic level with C code similar to this:

```
    volatile cowpi_ioport_t *ioports =
        (cowpi_ioport_t *)(COWPI_IO_BASE + 0x3); // an array of I/O ports
    // to clear pin 10 to a 0:
    ioports[D8_D13].output &= ~(1 << (10-8));
    // to set pin 10 to a 1:
    ioports[D8_D13].output |= 1 << (10-8);
```

This code uses the read/modify/write pattern: Obtain the existing output values for the relevant bank of pins, then create a bit vector that can be used to set 0 or 1 in the specific bit while preserving all of the other pins' output values, and then finally assign the resulting bit vector to the bank's output register. If the new logic level is in a variable and you don't know whether you're assigning a 0 or a 1, a good choice would be to clear the relevant bit to 0 and then use a bitwise OR to assign the appropriate value to the specific bit:

```
    uint8_t logic_level = ... // assume logic_level is strictly 0 or 1
    ioports[D8_D13].output = (ioports[D8_D13].output & ~(1 << (10-8)))
                                                  | (logic_level << (10-8));
```

4.1.2 Mapping Input/Output Devices to I/O Port Array

Section 4.1.1 describes the I/O ports, a structure definition, and named constants that can be used to access the inputs and outputs attached to the Arduino Nano's pins. Figure 2 in Section 1.1 shows which input/output devices are attached to the various Arduino Nano pins. Combining this information, we arrive at the mapping in Table 2.

Array Element	Field	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ioports[D0_D7]	.input	—	—	—	—	Keypad NAND	Button NAND	—	—
	.output	Keypad Row *	Keypad Row 7	Keypad Row 4	Keypad Row 1	—	—	—	—
ioports[D8_D13]	.input	—	—	—	—	Left Switch	Right Switch	Right Button	Left Button
	.output	—	—	Left LED	Right LED	—	—	—	—
ioports[D14_D19]	.input	—	—	controlled by I^{2}C for display module		Keypad Col A	Keypad Col 3	Keypad Col 2	Keypad Col 1
	.output	—	—			—	—	—	—

Table 2: A mapping of input/output devices to fields in the I/O ports array elements.

4.2 Inter-Integrated Circuit Protocol

The ATmega328P uses six registers for I^{2}C. In this datasheet, we use the conventional terms "Inter-Integrated Circuit," or I^{2}C; however, the ATmega328P datasheet[?] uses the terms "Two Wire Interface," or TWI, to describe the I^{2}C protocol. We mention this because the six registers, and the bits contained therein, have names derived from "TWI."

The six registers are:

`TWBR` The TWI Bit Rate Register, along with the prescaler bits in TWSI, is used to set the transmission bit rate. *You do not need to configure the bit rate; the **`cowpi_setup()`** function takes care of all necessary configuration to set the bit rate to 100kHz.*

`TWCR` The TWI Control Register controls the operation of the I^{2}C hardware. The particular bits are described in Section 4.2.2.

`TWSR` The TWI Status Register is principally used to reflect the status of the I^{2}C hardware and the I^{2}C serial bus. Bits 1..0 are define the prescaler that, along with TWBR, set the transmission bit rate. The remaining bits are described in Section 4.2.2.

`TWDR` The TWI Data Register contains either the next byte to transmit or the last byte received, depending on the current mode of operation. The particular bits are described in Section 4.2.2.

`TWAR` The TWI Address Register sets the microcontroller's address when the I^{2}C hardware is configured to act as a peripheral. *Under normal Cow Pi operation, TWAR is unused.*

`TWAMR` The TWI Address Mask Register instructs the I^{2}C hardware, when configured to act as a peripheral, to ignore particular bits when determining whether this microcontroller is being addressed. *Under normal Cow Pi operation, TWAMR is unused.*

The I^{2}C hardware has four modes of operation: controller transmitter, controller receiver, peripheral transmitter, and peripheral receiver.[4] In the Cow Pi's typical usage, the controller transmitter mode will ber used to drive the display module. For this reason, the discussion in this datasheet will focus on the controller transmitter mode.

The nature of I^{2}C allows for uses other than the display module without compromising the ability to work with the display module. If you choose to expand the Cow Pi in such a manner that other I^{2}C modes are necessary, see Section 21.7 of the ATmega328P datasheet[**?**] for details.

4.2.1 Structure for Memory-Mapped Input/Output

The CowPi library provides data structures to access the memory-mapped I/O registers in a more readable form. Specifically, the `cowpi_i2c_t` structure provides meaningfully-named fields in place of the 4–5-letter register names.

[4]The ATmega328P datasheet uses the older terms: master transmitter, master receiver, slave transmitter, and slave receiver. In the Cow Pi datasheet, we will use the preferred terminology recommended by the Open Source Hardware Association.

```

/**
 * @brief Structure for the TWI (aka I2C, IIC) hardware.
 */
typedef struct {
    uint8_t bit_rate;                   //!< TWI bit rate register, works in conce
    uint8_t status;                     //!< TWI status register (TWSR)
    uint8_t address;                    //!< TWI peripheral address register (TWAR
    uint8_t data;                       //!< TWI data register (TWBB)
    uint8_t control;                    //!< TWI control register(TWCR)
    uint8_t peripheral_address_mask;    //!< TWI peripheral address mask register (
```

Unlike the I/O registers for the external pins, you will not have an array of `cowpi_i2c_t` structures; you'll have just the one. Create a pointer to a `cowpi_i2c_t` structure that points to the lowest-addressed register (TWBR, 0x98 bytes above `COWPI_IO_BASE`, per Table 3). For example, if we wanted to determine if a status had been set and then set the TWI Enable bit (TWEN), then we could do so with C code similar to this:

```
    volatile cowpi_i2c_t *i2c = (cowpi_i2c_t *)(COWPI_IO_BASE + 0x98);
    uint8_t status = i2c->status & 0xF8; // mask-off the irrelevant bits
    i2c->control = 0x4; // Set the Enable bit
```

You may have noticed that this code does not use the read/modify/write pattern. Because of the particular uses of the control bits, you may find it easier to explicitly assign each control bit value afresh, rather than modify the pre-existing values.

4.2.2 Control and Data Bits

Table 3 identifies the particular bits in each of the I^{2}C registers.

In this section we shall describe only the control and data bits. If you need information about the setting the bit rate, or configuring the peripheral address and address mask, see Section 21.9 of the ATmega328P datasheet[?] for the bit descriptions, and Chapter 21 generally for the bits' uses.

Name	Offset	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Peripheral Address Mask Register									
TWAMR	0x9D	TWAM6	TWAM5	TWAM4	TWAM3	TWAM2	TWAM1	TWAM0	—
Control Register									
TWCR	0x9C	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	—	TWIE
Data Register									
TWDR	0x9B	TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0
Peripheral Address Register									
TWAR	0x9A	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE
Status Register									
TWSR	0x99	TWS7	TWS6	TWS5	TWS4	TWS3	—	TWPS1	TWPS0
Bit Rate Register									
TWBR	0x98	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0

Table 3: ATmega328P "Two Wire Interface" registers. Original data from ATmega382P datasheet, §21.9.[?]

Data Bits The eight data bits are straight-forward. When in controller transmitter or peripheral transmitter mode, place the byte that needs to be transmitted into the TWI Data Register (or the `data` field of a `cowpi_i2c_t` variable); there is generally no need to use the distinct bits. Similarly, when in controller receiver or peripheral receiver mode, the last byte sent by the transmitter can be found in the TWI Data Register.

Control Bits There are seven bits that either allow a program to control the I^{2}C hardware or to learn when it is safe to control the hardware.

Bit 7, TWI Interrupt Flag The I^{2}C hardware sets this bit to a 1 when it has finished with its last operation and the program can safely write to the data and status registers. Perhaps counterintuitively, the program clears the flag by writing a 1 to this bit; this causes the bit to become 0. Once the bit is 0, the program should not write to the status or data registers until it is 1 again. You can create a busy-wait loop that blocks the program while the bit is 0.

Alternatively, if TWI interrupts are enabled then you can create an interrupt handler that updates the data and status registers only when it is safe to do so (and then clears this bit). The `cowpi_setup()` function initially disables TWI interrupts; you must explicitly enable TWI interrupts if you intend to use them.

Bit 6, TWI Enable Acknowledge Bit When the program has set this bit to 1, it instructs the I^{2}C hardware to generate an ACK signal at the appropriate times when in controller receiver, peripheral transmitter, or peripheral receiver modes.

Bit 5, TWI Start Condition Bit As part of the I^{2}C protocol, the controller must send a "Start Bit" when it needs to control the I^{2}C bus. A program can instruct the ^{2}C hardware to claim control of the bus (or wait until it can do so) by writing a 1 to this bit and also to bit 7 (the TWI Interrupt Flag bit). When the controller has control of the bus, the TWI Interrupt Flag (bit 7) will become 1.

Bit 4, TWI Stop Condition Bit As part of the I^{2}C protocol, the controller must send a "Stop Bit" when it no longer needs to control the I^{2}C bus. A program can instruct the ^{2}C hardware to release control of the bus by writing a 1 to this bit and also to bit 7 (the TWI Interrupt Flag bit).

Bit 3, TWI Write Collision Flag If a program writes to the data register while the TWI Interrupt Flag (bit 7) is 0, a data collision will occur, and this bit will become 1.

Bit 2, TWI Enable Bit This bit must be a 1 at all times for the I^{2}C hardware to work, and the `cowpi_setup()` function initially sets it to 1; If, as recommended in Section 4.2.1, you explicitly assign each bit value instead of using the read/modify/write pattern, then be sure that you always assign a 1 to this bit.

Bit 0, TWI Interrupt Enable If this bit is 1, then TWI interrupt requests will be activated whenever appropriate. If this bit is 0, then TWI interrupt requests will not be activated. The `cowpi_setup()` function initially disables TWI interrupts.

4.2.3 Controller Transmitter Sequence

Generally speaking, the I^{2}C controller transmitter sequence consists of a start bit, the desired peripheral's address (plus a mode bit), one or more data bytes, and a stop bit. After each transmission, the program should busy-wait until the TWI Interrupt Flag has been set (bit 7 of TWCR or the `control` field of a `cowpi_i2c_t` variable). After the busy-wait terminates, the I^{2}C status should be checked (bits 7..3 of TWSR or the `status` field of a `cowpi_i2c_t` variable) to determine whether there were any errors.

The pseudocode for this sequence is:

```
(* assume variable i2c is a reference to a cowpi_i2c_t structure *)

        (* every assignment to i2c≻control needs to:
           write a 1 to bit 7 to clear the interrupt flag, and
           write a 1 to bit 2 to keep I2C enabled *)
control_bits ← bitwise_or(shift_left(1,7), shift_left(1,2))

        (* send the start bit by writing a 1 to bit 5 of i2c≻control *)
i2c≻control ← bitwise_or(control_bits, shift_left(1,5))

        (* wait until operation finishes *)
busy_wait_while(bit 7 of i2c≻control = 0)

        (* when sending the peripheral's address, it should be in the data
           register's bits 7..1 — bit 0 should be 0 for controller-transmitter *)
i2c≻data ← shift_left(peripheral_address, 1)

        (* send contents of data register *)
i2c≻control ← control_bits
busy_wait_while(bit 7 of i2c≻control = 0)

        (* send the data that the peripheral needs *)
for each byte of data:
    i2c≻data ← data
    i2c≻control ← control_bits
    busy_wait_while(bit 7 of i2c≻control = 0)

        (* send the stop bit by writing a 1 to bit 4 of i2c.control *)
i2c≻control ← bitwise_or(control_bits, shift_left(1,4))
```

4.2.4 I^{2}C with LCD1602 Display Module

The `cowpi_setup()` function configures the display module so that it can be controlled with 8 bits in parallel. One of the tradeoffs is that each character or command byte must be transmitted as two halfbytes. The CowPi library's `cowpi_lcd1602_send_command()` and `cowpi_lcd1602_send_character()` functions take care of dividing the full byte into two halfbytes, and passing each halfbyte to `cowpi_lcd1602_send_halfbyte()` in the appropriate order. ***When a halfbyte is passed to*** `cowpi_lcd1602_send_halfbyte()`***, it will be in the lower 4 bits of the*** `halfbyte` ***argument***, regardless of which of the two halfbytes it is.

Data Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
LCD1602 Bit	D7	D6	D5	D4	BT	EN	RW	RS
Bit source	`shift_left(halfbyte,4)`				backlight on/off	latch data	read/write	`!is_command`

Table 4: The `COWPI_DEFAULT` mapping of I^{2}C data bits to LCD1602 bits.

The I^{2}C adapter[?] converts the serial data coming from the microcontroller into the parallel data that the display module requires. For this to be effective, the `cowpi_lcd1602_send_halfbyte()` function must pack the bits in the order that the I^{2}C adapter expects.

Data Byte for LCD1602 Display Module The `COWPI_DEFAULT` bit order is described in Table 4. When constructing a byte to place in the I^{2}C Data Register:

Bits 7..4 The upper four bits are the `halfbyte` argument passed to `cowpi_lcd1602_send_halfbyte()`, left-shifted four places.

Bit 3 Bit 3 is a 1 if you want the LCD1602's backlight to illuminate, or 0 if you want it deluminated.[5]

Bit 2 As described below, bit 2 is used to send a pulse to the LCD1602 that instructs the display module that it should latch-in the halfbyte that it has received.

Bit 1 Bit 1 informs the LCD1602 whether data is being sent to it, or if a data request is being made of it; while it is possible to query the display module's memory, the Cow Pi library does not support this feature, and bit 1 should always be 0.

Bit 0 Bit 0 informs the LCD1602 whether the halfbyte that it receives is part of a command or is part of a character; if the `is_command` argument passed to `cowpi_lcd1602_send_halfbyte()` is true, then bit 0 should be 0; otherwise, bit 0 should be 1.

Data Byte Sequence When the `cowpi_lcd1602_send_halfbyte()` function executes the controller-transmitter sequence (see the pseudocode in Section 4.2.3), it will have three (3) data bytes to transmit.

1. First, the halfbyte needs to be sent *without* yet instructing the display module to latch-in the halfbyte:
```
bitwise_or(
    shift_left(halfbyte,4),
    shiftleft((1 if backlight_on else 0),3),
    shiftleft(0,2), (*not yet latching halfbyte *)
    shiftleft(0,1),
    shiftleft((0 if is_command else 1),0)
)
```

[5]While the `cowpi_lcd1602_set_backlight()` function can be used to turn the backlight on and off, bit 3 needs to preserve the appropriate setting.

2. Second, the start of the "latch pulse" needs to be sent:
    ```
    bitwise_or(
       shift_left(halfbyte,4),
       shiftleft((1 if backlight_on else 0),3),
       shiftleft(1,2), (*latch the halfbyte *)
       shiftleft(0,1),
       shiftleft((0 if is_command else 1),0)
    )
    ```
    - **The pulse needs to stay active for at least 0.5μs.** This could be managed by introducing a delay based on the microcontroller's clock cycle, achieved with the AVR-libc `_delay_us()` function,[?] to be able to introduce a delay of nearly exactly 0.5μs, or this could be managed by introducing a 1μs delay using the Arduino core library's `delayMicroseconds()` function,[?] which is portable across all devices using the Arduino toolchain.
3. Third, the end of the "latch pulse" needs to be sent:
    ```
    bitwise_or(
       shift_left(halfbyte,4),
       shiftleft((1 if backlight_on else 0),3),
       shiftleft(0,2), (*complete the latch *)
       shiftleft(0,1),
       shiftleft((0 if is_command else 1),0)
    )
    ```

5 Interrupts

Most interrupts on the ATmega328P are handled either by creating an interrupt service routine (ISR) using AVR-libc's `ISR()` macro. External interrupts, however, can be handled either by creating an ISR or by using the Arduino Core's `attachInterrupt()` function to register an interrupt handler.

5.1 Sharing data with ISRs and Interrupt Handlers

Regardless of whether you create an ISR using the `ISR()` macro or register an interrupt handler using the `attachInterrupt()` function, data cannot be passed to the interrupt-handling code through parameters, and the interrupt-handling code cannot return data through a return value. This necessitates the use of global variables to provide data to, and obtain data from, the interrupt-handling code.

Because the compiler cannot detect any definition-use pairs for these global variables – they are updated in one function and read in another, and no call chain exists between the two functions – the compiler will optimize-away these variables and the code that accesses them in the interest of reducing the program's memory footprint. The way to prevent this mis-optimization is to use the `volatile` keyword.

*Any global variables that interrupt-handling code reads from and/or writes to **must** have the `volatile` modifier.*

5.2 Registering Interrupt Service Routines using the `ISR()` Macro

To create an interrupt service routine, write this code that looks like a function, outside of any other function:

```
ISR(vector) {
    ...
}
```

where *vector* is one of the vectors listed in the AVR-libc's interrupts documentation[?] (look in the table rows that have "ATmega328P" in the "Applicable for Device" cell). Replace "..." with the code that should execute whenever the timer interrupt occurs. You want to keep your ISR short, no more than a few lines of code. If anything more elaborate needs to happen, code in your `loop()` function (or a function called by `loop()`) can do that based on changes made from within your ISR.

Any necessary configuration to establish the conditions under which the ISR will be invoked, typically through the use of memory-mapped I/O registers, will need to occur either in the `setup()` function or in a helper function called by `setup()`.

5.3 Registering External Interrupt Handlers using `attachInterrupt()`

The ATmega328P has two types of interrupts that are based on changes detected at the pins.

Pin Change Interrupts are the less flexible of the two but can be triggered by changes on any of the digital pins. See the ATmega328P datasheet[?] for details on configuring pin change interrupts. Pin change interrupts must be handled through an ISR, as described above.

External Interrupts can also be manually configured[?] and handled through an ISR; however, the Arduino Core has functions that abstract-away all of the configuration details.[?] While external interrupts on the ATmega328P are limited to only two pins (digital pins D2 & D3), they can be triggered for many conditions, and their interrupt handlers can be easily registered. *For this reason, a NAND of the pushbuttons is input to pin D2, and a NAND of the keypad's columns is input to pin D3.*

To handle an interrupt, first write a function, such as `handle_buttonpress()` or `handle_keypress()`. This function must not have any parameters, and its return type must be `void`. Then, in the `setup()` function (or in one of its helper functions), register the interrupt handler with this code:

```
attachInterrupt(digitalPinToInterrupt(pin_number), interrupt_handler_name, mode);
```

This will configure all of the necessary registers to call the function `interrupt_handler_name()` whenever the input value on the pin *pin_number* satisfies the *mode*. The *mode* is one of:

LOW to trigger the interrupt whenever the pin is low

RISING to trigger the interrupt whenever the pin goes from low to high

FALLING to trigger the interrupt whenever the pin goes from high to low

CHANGE to trigger the interrupt whenever the pin rises or falls

As with ISRs registered with the `ISR()` macro, you want to keep your interrupt handler short.

6 Timers

The ATmega328P used by the Arduino Nano in the Cow Pi has three timers. As with the other I/O registers, the registers used by these timers are mapped into the data memory address space.

6.1 Structures for Memory-Mapped Timer Registers

The CowPi library provides data structures for both 8-bit and 16-bit timers, allowing access to the memory-mapped timer registers in a more readable form.

```

/**
 * @brief Structure for 8-bit timer/counter (TIMER0 or TIMER1).
 *
 * The timer/counter interrupt mask register (TIMSKx) and the timer/counter
 * interrupt flag register (TIFRx) are not part of this structure. Neither
 * are the asynchronous status register (ASSR) nor the general timer/counter
 * control register (GTCCR).
 */
typedef struct {
    uint16_t control;                //!< Timer/counter control registers A & B,
    uint8_t counter;                 //!< Timer/counter register (TCNTx)
    uint8_t compareA;                //!< Output compare register A (OCRxA)
    uint8_t compareB;                //!< Output compare register B (OCRxB)
} cowpi_timer8bit_t;

/**
 * @brief Structure for 16-bit timer/counter (TIMER1).
 *
 * The timer/counter interrupt mask register (TIMSKx) and the timer/counter
 * interrupt flag register (TIFRx) are not part of this structure. Does not
 * include the general timer/counter control register (GTCCR).
 */
typedef struct {
    uint32_t control;               //!< Timer/counter control registers A, B, &
    uint16_t counter;               //!< Timer/counter register (TCNTxH TCNTxL, a
    uint16_t capture;               //!< Input capture register (ICRxH ICRxL, aka
    uint16_t compareA;              //!< Output compare register A (OCRxAH OCRxAL
    uint16_t compareB;                    //!< Output compare register B (OCRxBH OCRxB
```

The ATmega328P's TIMER0 and TIMER2 are 8-bit timers, and TIMER1 is a 16-bit timer.

NOTE: Even though TIMER0 and TIMER2 are both 8-bit timers and make use of the same `cowpi_timer8bit_t` structure definition, the interpretation of the bits in their fields is subtly different. Be sure to use the correct tables when configuring the timers.

NOTE: The timers have uses which may not be immediately obvious. The most critical of these is that TIMER0 is used for the pseudo-clock that allows the Arduino Core's `millis()` function to report the number of milliseconds since power-up. While you can safely configure and handle comparison interrupts that do not reset the timer's counter, *you should not change TIMER0's period, reset TIMER0's counter, nor register an ISR for* ***`TIMER0_OVF0_vect`*** unless you are willing to accept the adverse impact on `millis()` and any code that depends on `millis()`.

The five fields in the structures are:

`control` The concatenation of the timer's two (TIMER0, TIMER2) or three (TIMER1) control registers

`counter` The 8-bit (TIMER0, TIMER2) or 16-bit (TIMER1) value that increments by one in each timer period

`capture` Stores the counter's value at the exact moment that an input capture event occurs (TIMER1 only)

`compareA` The counter value at which a `TIMER`*`n`*`_COMPA_vect` interrupt should be triggered, where n is the timer number

`compareB` The counter value at which a `TIMER`*`n`*`_COMPB_vect` interrupt should be triggered, where n is the timer number

6.2 Configuring the Timer Period

In a timer's "Normal" mode, it will increment its counter to the maximum value representable by its counter, and then overflow to zero, possibly generating a timer overflow interrupt in the process. You can adjust the rate at which its counter increments using a prescaler that is applied to the system clock's period. Because the number of increments is fixed, a timer has only 5 or 7 possible periods without introducing an external timer source. In other timer modes, you have a much wider range of possible intervals between interrupts available to you, which you can obtain by setting both a prescaler and a comparison value.

Once you know the desired timer period between timer interrupts, you need to determine the comparison value and the timer prescaler that will realize this timer period. You can then determine the parameters for the timer using this equation:

$$16,000,000\frac{\text{cycles}}{\text{second}} = comparison\_value\frac{\text{beats}}{\text{interrupt}} \times prescaler\frac{\text{cycles}}{\text{beat}} \times interrupt\_frequency\frac{\text{interrupts}}{\text{second}}$$

or, equivalently:

$$comparison\_value\frac{\text{beats}}{\text{interrupt}} \times prescaler\frac{\text{cycles}}{\text{beat}} = 16,000,000\frac{\text{cycles}}{\text{second}} \times interrupt\_period\frac{\text{seconds}}{\text{interrupt}}$$

where:

16,000,000 Hz is the system clock frequency (the inverse of the clock period).

comparison_value is a number you will place in one of the timer's `compare` registers for a comparison-based timer interrupt. This can be any possible value of an unsigned 16-bit integer for Timer1, or any possible value of an unsigned 8-bit integer for Timer2. (If you plan to use an overflow interrupt, then *comparison_value* is 256 or 65,536, depending on whether you're using an 8-bit or a 16-bit timer.)

prescaler is a multiplier applied to the clock period to adjust the time between counter increments ("beats"). Possible values are 1, 8, 64, 256, and 1024 for Timer0 and Timer1, or 1, 8, 32, 64, 128, 256, and 1024 for Timer2.

interrupt_frequency is how often you want a timer interrupt (the inverse of the interrupt period).

interrupt_period is the time between timer interrupts (the inverse of the interrupt frequency).

You may have to iterate on your design until you arrive at one that works with the constraints of that equation's terms for whichever timer you choose to use.

If you cannot generate a timer period great enough for your needs, then follow the example used by the Arduino pseudo-clock: Find a comparison value and a prescaler that yield a timer period that is a whole-number factor of your actual desired period. Then introduce a counter variable in your ISR that increments each time the ISR is invoked, and when it reaches a particular value (the value at which $timer\_period \times counter\_value = desired\_period$) then take the appropriate action.

To understand the available waveform generation modes, see the ATmega328P datasheet;[?] specifically, see Section 14.7 (TIMER0), Section 15.9 (TIMER1), or Section 17.7 (TIMER2). A good choice for many applications is Clear Timer on Compare (*CTC*) with the "TOP" value set by an output compare register (`OCRxx`, aka one of the *compareX* fields in the data structure). But, again, check the ATmega328P datasheet to be sure. Use the tables below to select the appropriate WGM bits.

Based on the prescaler you chose for the above equation, use the tables below to select the appropriate CS bits.

6.3 Configuring TIMER0

CAUTION: TIMER0 is used by the Arduino Core to track time since power-up. The Arduino Core configures TIMER0 for "Normal" mode with a prescaler of 64, generating a timer overflow interrupt every 1,024μs. While it is perfectly safe to set up comparison interrupts using TIMER0, **DO NOT change TIMER0's Waveform Generation Mode**

or Clock Select bits, nor register an ISR for `TIMER`*n*`_OVF_vect`, if your code relies upon the Arduino Core to track time.

Table 5 shows the mapping of TIMER0's registers to the `cowpi_timer8bit_t` struct. Creating a pointer to TIMER0's memory-mapped registers is as simple as

```
    volatile cowpi_timer8bit_t *timer
            = (cowpi_timer8bit_t *)(COWPI_IO_BASE + 0x24);
```

Having creating that pointer, you can access the registers using the `cowpi_timer8bit_t`'s fields.

Name	Offset								
`compareB`		**Bit 7**			. . .				**Bit 0**
OCR0B	0x28	Comparison Value "B"							
`compareA`		**Bit 7**			. . .				**Bit 0**
OCR0A	0x27	Comparison Value "A"							
`counter`		**Bit 7**			. . .				**Bit 0**
TCNT0	0x26	Timer Counter Value							
`control`		**Bit 15**	**Bit 14**	**Bit 13**	**Bit 12**	**Bit 11**	**Bit 10**	**Bit 9**	**Bit 8**
TCCR0B		`FOC0A`	`FOC0B`	—	—	`WGM02`	`CS02`	`CS01`	`CS00`
		Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
TCCR0A	0x24	`COM0A1`	`COM0A0`	`COM0BA1`	`COM0B0`	—	—	`WGM01`	`WGM00`

Table 5: Timer0's registers. Adapted from ATmega328P Data Sheet, §14.9.[?]

The two comparison values, which you can set, are continuously compared to the timer's counter value. A comparison match can be used to generate an output compare interrupt (`TIMER0_COMPA_vect` or `TIMER0_COMPB_vect`). The timer's counter value can be read from or written to by your program. Polling the counter value is a notional use case, but configuring an interrupt would be more appropriate. Assigning a value, such as 0, to the counter would be a mechanism to reset its counter to a known value.

Among the bits in the `control` field (the `TCCR0A` & `TCCR0B` registers), most can be left as 0. If you believe that you need to set custom "Force Output Compare" "Compare Output Mode" bits, then consult the ATmega328P datasheet, Section 14.9.[?] Under typical Cow Pi usage, you should only need to set the "Waveform Generation Mode" and "Clock Select" bits.

Using the prescaler that you determined above, you should assign the `CS00`, `CS01`, and `CS02` bits using Table 6.

The Waveform Generation Bits are used to set the Timer/Counter mode of operation. There are two modes most useful for typical Cow Pi usage. The first is "Normal" mode, in which the counter increases monotonically until it reaches the greatest possible representable value and then overflows to 0. The other mode is "Clear Timer on Compare" (*CTC*) with the "TOP" value set by output compare register "A," in which the counter increases monotonically until it reaches the value in the comparison register and then resets to 0. The `WGM` bits for these two modes are shown in Table 7. For the Pulse Width Modulation modes, consult Section 14.7 and Table 14-8 of the ATmega328P datasheet.[?].

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$\frac{clk}{1}$ (no prescaling)
0	1	0	$\frac{clk}{8}$ (from prescaler)
0	1	1	$\frac{clk}{64}$ (from prescaler)
1	0	0	$\frac{clk}{256}$ (from prescaler)
1	0	1	$\frac{clk}{1024}$ (from prescaler)

Table 6: Timer0's Clock Select Bit Description. Abridged from ATmega328P Data Sheet, Table 14-9.[?] See the original table for the clock select bits when using an external clock source.

WGM02	WGM01	WGM00	Timer/Counter Mode of Operation	TOP
0	0	0	Normal	0xFF
0	1	0	CTC	OCR2A

Table 7: Timer0's Waveform Generation Mode Bit Description. Abridged from ATmega328P Data Sheet, Table 14-8.[?] See the original table for the WGM bits when using a PWM mode, and for the "OCRx [sic] Update" and "TOV [sic] Flag Set" columns.

After configuring the timer, enable the relevant interrupt(s) as described in Section 6.6, and register any necessary ISRs as described in Section 5.1.

6.4 Configuring TIMER1

Table 8 shows the mapping of TIMER1's registers to the `cowpi_timer16bit_t` struct. Creating a pointer to TIMER1's memory-mapped registers is as simple as

```
    volatile cowpi_timer16bit_t *timer
            = (cowpi_timer16bit_t *)(COWPI_IO_BASE + 0x60);
```

Having creating that pointer, you can access the registers using the `cowpi_timer16bit_t`'s fields.

The two comparison values, which you can set, are continuously compared to the timer's counter value. A comparison match can be used to generate an output compare interrupt (`TIMER1_COMPA_vect` or `TIMER1_COMPB_vect`). Input capture is beyond the scope of typical Cow Pi usage; see the ATmega328P datasheet, Section 15.6[?] for discussion of the input capture unit. The timer's counter value can be read from or written to by your program. Polling the counter value is a notional use case, but configuring an interrupt would be more appropriate. Assigning a value, such as 0, to the counter would be a mechanism to reset its counter to a known value.

Among the bits in the `control` field (the `TCCR1A`, `TCCR1B`, & `TCCR1C` registers), most can be left as 0. If you believe that you need to set custom "Force Output Compare," "Input Capture," or "Compare Output Mode" bits, then consult the ATmega328P datasheet, Section 15.11.[?] Under typical Cow Pi usage, you should only need to set the "Waveform Generation Mode" and "Clock Select" bits.

Using the prescaler that you determined above, you should assign the `CS10`, `CS11`, and `CS12` bits using Table 9.

The Waveform Generation Bits are used to set the Timer/Counter mode of operation.

Name	Offset								
`compareB`		**Bit 15**			. . .				**Bit 0**
OCR1B	0x6A				Comparison Value "B"				
`compareA`		**Bit 15**			. . .				**Bit 0**
OCR1A	0x68				Comparison Value "A"				
`capture`		**Bit 15**			. . .				**Bit 0**
ICR1	0x66				Input Capture's Counter Value				
`counter`		**Bit 15**			. . .				**Bit 0**
TCNT1	0x64				Timer Counter Value				
`control`		**Bit 23**	**Bit 22**	**Bit 21**	**Bit 20**	**Bit 19**	**Bit 18**	**Bit 17**	**Bit 16**
TCCR1C		`FOC1A`	`FOC1B`	—	—	—	—	—	—
		Bit 15	**Bit 14**	**Bit 13**	**Bit 12**	**Bit 11**	**Bit 10**	**Bit 9**	**Bit 8**
TCCR1B		`ICNC1`	`ICES1`	—	`WGM13`	`WGM12`	`CS12`	`CS11`	`CS10`
		Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
TCCR1A	0x60	`COM1A1`	`COM1A0`	`COM1BA1`	`COM1B0`	—	—	`WGM11`	`WGM10`

Table 8: Timer1's registers. Adapted from ATmega328P Data Sheet, §15.11.[?]

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$\frac{clk}{1}$ (no prescaling)
0	1	0	$\frac{clk}{8}$ (from prescaler)
0	1	1	$\frac{clk}{64}$ (from prescaler)
1	0	0	$\frac{clk}{256}$ (from prescaler)
1	0	1	$\frac{clk}{1024}$ (from prescaler)

Table 9: Timer1's Clock Select Bit Description. Abridged from ATmega328P Data Sheet, Table 15-6.[?] See the original table for the clock select bits when using an external clock source.

There are two modes most useful for typical Cow Pi usage. The first is "Normal" mode, in which the counter increases monotonically until it reaches the greatest possible representable value and then overflows to 0. The other mode is "Clear Timer on Compare" (*CTC*) with the "TOP" value set by output compare register "A," in which the counter increases monotonically until it reaches the value in the comparison register and then resets to 0. The `WGM` bits for these two modes are shown in Table 10. For the Pulse Width Modulation modes, consult Section 15.9 and Table 15-5 of the ATmega328P datasheet.[?].

WGM13	WGM12	WGM11	WGM10	Timer/Counter Mode of Operation	TOP
0	0	0	0	Normal	0xFFFF
0	1	0	0	CTC	OCR1A

Table 10: Timer1's Waveform Generation Mode Bit Description. Abridged from ATmega328P Data Sheet, Table 15-5.[?] See the original table for the WGM bits when using a PWM mode, and for the "OCR1x Update" and "TOV1 Flag Set" columns.

After configuring the timer, enable the relevant interrupt(s) as described in Section 6.6, and register any necessary ISRs as described in Section 5.1.

6.5 Configuring TIMER2

Table 11 shows the mapping of TIMER2's registers to the `cowpi_timer8bit_t` struct. Creating a pointer to TIMER2's memory-mapped registers is as simple as

```
    volatile cowpi_timer8bit_t *timer
            = (cowpi_timer8bit_t *)(COWPI_IO_BASE + 0x90);
```

Having creating that pointer, you can access the registers using the `cowpi_timer8bit_t`'s fields.

Name	Offset								
`compareB`		**Bit 7**			…				**Bit 0**
OCR2B	0x94				Comparison Value “B”				
`compareA`		**Bit 7**			…				**Bit 0**
OCR2A	0x93				Comparison Value “A”				
`counter`		**Bit 7**			…				**Bit 0**
TCNT2	0x92				Timer Counter Value				
`control`		**Bit 15**	**Bit 14**	**Bit 13**	**Bit 12**	**Bit 11**	**Bit 10**	**Bit 9**	**Bit 8**
TCCR2B		`FOC2A`	`FOC2B`	—	—	`WGM22`	`CS22`	`CS21`	`CS20`
		Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
TCCR2A	0x90	`COM2A1`	`COM2A0`	`COM2BA1`	`COM2B0`	—	—	`WGM21`	`WGM20`

Table 11: Timer2's registers. Adapted from ATmega328P Data Sheet, §17.11.[?]

The two comparison values, which you can set, are continuously compared to the timer's counter value. A comparison match can be used to generate an output compare interrupt (`TIMER2_COMPA_vect` or `TIMER2_COMPB_vect`). The timer's counter value can be read from or written to by your program. Polling the counter value is a notional use case, but configuring an interrupt would be more appropriate. Assigning a value, such as 0, to the counter would be a mechanism to reset its counter to a known value.

Among the bits in the `control` field (the `TCCR2A` & `TCCR2B` registers), most can be left as 0. If you believe that you need to set custom “Force Output Compare” “Compare Output Mode” bits, then consult the ATmega328P datasheet, Section 17.11.[?] Under typical Cow Pi usage, you should only need to set the “Waveform Generation Mode” and “Clock Select” bits.

Using the prescaler that you determined above, you should assign the `CS20`, `CS21`, and `CS22` bits using Table 12.

The Waveform Generation Bits are used to set the Timer/Counter mode of operation. There are two modes most useful for typical Cow Pi usage. The first is “Normal” mode, in which the counter increases monotonically until it reaches the greatest possible representable value and then overflows to 0. The other mode is “Clear Timer on Compare” (*CTC*) with the “TOP” value set by output compare register “A,” in which the counter increases monotonically until it reaches the value in the comparison register and then resets to 0. The `WGM` bits for these two modes are shown in Table 13. For the Pulse Width Modulation modes, consult Section 17.7 and Table 17-8 of the ATmega328P datasheet.[?].

CS22	CS21	CS20	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$\frac{clk}{1}$ (no prescaling)
0	1	0	$\frac{clk}{8}$ (from prescaler)
0	1	1	$\frac{clk}{32}$ (from prescaler)
1	0	0	$\frac{clk}{64}$ (from prescaler)
1	0	1	$\frac{clk}{128}$ (from prescaler)
1	1	0	$\frac{clk}{256}$ (from prescaler)
1	1	1	$\frac{clk}{1024}$ (from prescaler)

Table 12: Timer2's Clock Select Bit Description. Copied from ATmega328P Data Sheet, Table 17-9.[?]

WGM22	WGM21	WGM20	Timer/Counter Mode of Operation	TOP
0	0	0	Normal	0xFF
0	1	0	CTC	OCR2A

Table 13: Timer1's Waveform Generation Mode Bit Description. Abridged from ATmega328P Data Sheet, Table 17-8.[?] See the original table for the WGM bits when using a PWM mode, and for the "OCRx [sic] Update" and "TOV [sic] Flag Set" columns.

After configuring the timer, enable the relevant interrupt(s) as described in Section 6.6, and register any necessary ISRs as described in Section 5.1.

6.6 Timer Interrupts

Table 14 shows the bits of the Timer Interrupt Mask registers. To access these as memory-mapped registers, create a `uint8_t` pointer, and assign that pointer to the lowest address of these registers:

```
    volatile uint8_t *timer_interrupt_masks = COWPI_IO_BASE + 0x4E;
```

This pointer can then be used as a 3-element array, indexed by the timer number. For example, `timer_interrupt_masks[0]` can be used to enable any of the TIMER0 interrupts.

Name	Offset	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TIMSK2	0x50	—	—	—	—	—	OCIE2B	OCIE2A	TOIE2
TIMSK1	0x4F	—	—	ICIE1	—	—	OCIE1B	OCIE1A	TOIE1
TIMSK0	0x4E	—	—	—	—	—	OCIE0B	OCIE0A	TOIE0

Table 14: Timer Interrupt Mask registers. Original data from ATmega328P datasheet, §30.[?]

For Timer Overflow interrupts, to enable `TIMER`*n*`_OVF_vect`, set the `TOIE`*n* bit to 1. For Timer Comparison interrupts, to enable `TIMER`*n*`_COMPA_vect`, set the `OCIE`*n*`A` bit to 1; to enable `TIMER`*n*`_COMPB_vect`, set the `OCIE`*n*`B` bit to 1. For Input Capture interrupts, to enable `TIMER1_CAPT_vect`, set the `ICIE1` bit.

After enabling timer interrupts, be sure to register any necessary ISRs as described in Section 5.1.

7 Expansion Options

This section reserved for future use.