chipKIT MPIDE Board Variant Mechanism

Gene Apperson, Digilent Inc.
Revision A: November 4, 2013

Introduction:

The chipKIT MPIDE system has been designed to make it relatively easy to adapt the operation of the system to new hardware as boards are designed. This document describes the mechanisms used in the system to allow a board developer to describe the features of a new board to the system and allow the system to use the new board without the need to modify any of the core hardware abstraction layer code or standard libraries.

The board variant mechanism makes use of a standardized folder structure and set of definition files that can be installed into the MPIDE system that adapt the system to work with new boards. In some cases, these files are distributed with the MPIDE, but board variant files can be easily added into an installation of MPIDE after it has been installed on the client computer.

In addition to the board variant mechanism described here, a developer designing a new board will also need to produce a boot loader for the board. This document does not describe the process required to create a new custom boot loader from the boot loader project. This process is described in a separate document.

Basic Operation of the Board Variant Mechanism

The user interfaces with the board variant mechanism through the `Tools->Board` menu in the MPIDE. The MPIDE uses the information obtained for a set of files collectively called `boards.txt` to draw the `Tools->Board` menu. The `boards.txt` files contain a number of items of information about each board, but from the perspective of the board variant mechanism at run time, the primary piece of information is the variant name.

The selection from the `Tools->Boards` menu specifies the name of the board variant used by the selected board. This name is used to generate the path to the folder containing the variant files for the board. The variant files are a source file and a header file that contain the necessary definitions to allow the core hardware abstraction layer and the standard libraries to adapt to the selected board.

Folder Organization and Key Files

The following folder locations are significant to the board variant mechanism. All of these locations are relative to the root directory where MPIDE is installed on the user computer:

`./hardware/pic32`

This is the root folder of the PIC32/chipKIT platform within the MPIDE system. The default `boards.txt` file, `platforms.txt`, file and `bootloaders.txt` file is located here.

`./hardware/pic32/cores/pic32`

This folder contains the core hardware abstraction layer source files, header files, and the default linker scripts.

`./hardware/pic32/libraries`

This folder is the root for where the standard libraries are stored. Within this folder is a separate folder for each standard library.

`./hardware/pic32/variants`

This folder is the root for where the board variant files are stored. Within this folder is a separate folder for each board variant. The subfolders at this location are named for each board variant.

There are a number of files that make up the board variant mechanism or that define symbols, macros, and data tables used by the board variant mechanism.

The following data files are significant to the board variant mechanism:

`boards.txt` The `boards.txt` file contains information used by the MPIDE system to determine basic things about the board, such as which compiler tool chain is used, what the processor on the board is, compiler options to use when building the sketch, and so on. The `boards.txt` entries are used to populate the list of known boards. There is a `boards.txt` file in the folder: `<mpide_install_dir>/hardware/pic32`. There can also be a `boards.txt` file in the board variant folder. MPIDE will scan all folders contained within the `./hardware/pic32/variants` folder looking for files named `boards.txt`. Each one found is read and its contents added to the list of known boards.

linker scripts Linker scripts are input files used to describe to the linker things such as the layout of memory. There is a separate linker script needed for each specific PIC32 microcontroller. In most cases, the specific PIC32 device on a board has already been used on some other board and there will already be a linker script for it. If there isn't already a linker script for the specific microcontroller, or if the board has some particular need for a different memory layout, then a linker script will have to be added for the board. The default linker scripts are located in the `cores` folder. If a board variant requires a custom linker script, it can be placed in the board variant folder.

`avrdude.conf` AVRDUDE is the program used by the MPIDE to communicate with the boot loader on the board and to actually download a sketch to the board. AVRDUDE needs certain information about the microcontroller on the board to function.

`avrdude.conf` is a data file that AVRDUDE uses to configure itself with the specific information it needs about the microcontroller on the board. If the particular PIC32 device on the board is one used by some other chipKIT board, there will already be an entry in `avrdude.conf` for that microcontroller. If there isn't already an entry in `avrdude.conf` for the specific microcontroller, then one will have to be added.

The following source files are significant to the board variant mechanism:

`p32_defs.h` This file contains a number of definitions for symbols describing the PIC32 hardware resources. It defines structure types for accessing the special function registers in the PIC32 hardware, bit definitions for control, and status bits in the SFRs. These symbol and type definitions are used throughout the system as well as being part of the board definition mechanism. This file also defines a number of symbols that are at the core of the peripheral pin select mechanism defined as part of the board variant facility.

`pins_arduino.h` This file is the primary "entry point" into the board variant mechanism. This file was inherited from the original Arduino system, but the way that it is used by the system has been completely redefined from its role in Arduino. This file is implicitly included in all sketches, as it is included by `WProgram.h` which is automatically included in the compilation of every sketch. This file defines a number of symbols and macros that are used by the board variant files and then includes the board variant header file: `Board_Defs.h`.

`pins_arduino.c` The primary purpose of this file is to cause the `Board_Data.c` file for the selected board variant to be included into the set of files being compiled. It defines some generic tables that are part of the board variant mechanism and then includes `Board_Data.c`.

`Board_Defs.h` This is a board-specific header file that contains declarations for common symbols and macros that describe the specific details of a given board to the system. There is one of these files for each defined board variant.

`Board_Data.c` This is a board-specific file that contains the definitions for a number of board-specific data tables and functions that make up the implementation for the board variant support for a given board. There is one of these files for each defined board variant.

The files: `p32_defs.h`, `pins_arduino.h`, and `pins_arduino.c` are defined in the `cores` folder and are part of the core hardware abstraction layer.

There is a separate copy of `Board_Defs.h` and `Board_Data.c` for each supported board variant. These exist in the variants folder under a separate sub-folder named for each supported board.

Boards.Txt Entries

The boards.txt file contains an entry for each board known to the system. These entries are used to populate the list of boards available on the Tools->Boards menu. Each board entry is made up of a number of lines setting a number of configuration parameters used by the system. In some cases, these configuration parameters apply to the AVR microcontrollers originally used on Arduino boards and are not relevant to chipKIT boards. Many of the entries will also have the same value for all, or most, chipKIT boards, and so do not need to be given unique values.

The best procedure is for a board variant developer to copy an existing boards.txt entry and then edit it to modify the entries that need to be given unique values for the board variant. Each parameter contains the board name as part of the parameter name. This part will have to be edited on each line of the new entry. In the description below the value xxx is a place holder for the board name. The following boards.txt configuration values may need to be changed:

xxx.name	This provides the text that will show up in the Tools->Boards menu item.
xxx.group	This provides the name for the submenu under which the board should appear on the Tools->Boards menu.
xxx.platform	This tells the MPIDE which platform is used by this board. This sets which compiler tool chain and run-time implementation is used by the board. All chipKIT boards should use pic32.
xxx.board	This defines a symbol for the compilation that board-specific sketch code can use to identify when the sketch is being compiled for a particular target board.
xxx.ldscript	This identifies the linker script to be used when building a sketch for the board.
xxx.upload.maximum_size	This specifies the amount of useable program memory on the microcontroller.
xxx.build.mcu	This identifies the particular microcontroller used on the board.
xxx.build.f_cpu	This sets a symbol available at compile-time to give the operating speed of the microcontroller on the board.
xxx.build.variant	This identifies the board variant to be used when building for this board. This causes the specified variant folder for the board to be on the include list so that the correct board variant files will be available when building a sketch for the board.

The following example is taken from the default boards.txt file and is the entry for the chipKIT uC32 board:

```
##############################################################
chipkit_uc32.name=chipKIT uC32
```

```
chipkit_uc32.group=chipKIT

# new items
chipkit_uc32.platform=pic32
chipkit_uc32.board=_BOARD_UC32_
chipkit_uc32.board.define=
chipkit_uc32.ccflags=ffff
chipkit_uc32.ldscript=chipKIT-application-32MX340F512.ld
# end of new items

# Use a high -Gnum for devices that have less than 64K of data memory
# For -G1024, objects 1024 bytes or smaller will be accessed by
# gp-relative addressing
chipkit_uc32.compiler.c.flags=-O2::-c::-mno-smart-io::-w::-ffunction-sections::-fdata-
sections::-G1024::-g::-mdebugger::-Wcast-align::-fno-short-double
chipkit_uc32.compiler.cpp.flags=-O2::-c::-mno-smart-io::-w::-fno-exceptions::-ffunction-
sections::-fdata-sections::-G1024::-g::-mdebugger::-Wcast-align::-fno-short-double

chipkit_uc32.upload.protocol=stk500v2
chipkit_uc32.upload.maximum_size=520192
chipkit_uc32.upload.speed=115200

chipkit_uc32.bootloader.low_fuses=0xff
chipkit_uc32.bootloader.high_fuses=0xdd
chipkit_uc32.bootloader.extended_fuses=0x00
chipkit_uc32.bootloader.path=not-supported
chipkit_uc32.bootloader.file=not-supported
chipkit_uc32.bootloader.unlock_bits=0x3F
chipkit_uc32.bootloader.lock_bits=0x0F

chipkit_uc32.build.mcu=32MX340F512H
chipkit_uc32.build.f_cpu=80000000L
chipkit_uc32.build.core=pic32
chipkit_uc32.build.variant=uC32
#chipkit_uc32.upload.using=
```

General Symbol Definitions

There are a number of symbols defined by the `Board_Defs.h` file for the board variant. In some cases, these symbols are used to describe the available resources on the board. In other cases these symbols are intended to be used by a user's sketch to provide a portable mechanism to access resources across different boards.

Resource Availability Symbols

The following symbols are generally used internally by the system, but they are also available for the user sketch to use to determine the availability of resources on the board.

`NUM_DIGITAL_PINS` — Nominally, this symbol gives the number of digital I/O pins defined by the board variant that exist natively in the microcontroller on the board. This isn't strictly the actual number of pins. There may be holes in the range and pins within this range that are not valid. `NUM_DIGITAL_PINS-1` is the highest numbered digital pin that is accessed directly by the microcontroller.

NUM_DIGITAL_PINS_EXTENDED This symbol gives the number of digital pins that can be accessed on the board including digital pins that are implemented external to the microcontroller via some kind of I/O extender. There may be holes in the total range of pin numbers. NUM_DIGITAL_PINS_EXTENDED-1 is the highest numbered digital pin implemented on the board including any implemented externally to the microcontroller. In most cases, the value of this symbol is the same as NUM_DIGITAL_PINS, and a default definition of the symbol will be created in pins_arduino.h with that value if the Board_Defs.h file for the board variant doesn't define it otherwise.

There is no requirement that all of the digital I/O pins appear on a connector. There may, and often are, pins that control internal functions on a board that don't go out to a connector. This is often the case, for example, with LEDs where the corresponding pin only drives the LED and doesn't go out to a connector.

If a board implements digital I/O pins that are external to the microcontroller, such as would be the case if the board contains an I/O expansion chip of some kind, the digital pins implemented in the microcontroller should be the lower numbered pins (i.e., starting with pin 0) and the external pins should then follow the internal pins (i.e., externally defined pins start with digital pin number NUM_DIGITAL_PINS and go up from there to NUM_DIGITAL_PINS_EXTENDED-1).

NUM_ANALOG_PINS This symbol gives the number of analog inputs defined by the board variant that are implemented using internal A/D converters in the microcontroller. Numbers in the range 0 through NUM_ANALOG_PINS-1 are valid analog inputs number that can be passed to analogRead().

NUM_ANALOG_PINS_EXTENDED Similarly to the NUM_DIGITAL_PINS_EXTENDED symbol, this symbol gives the number of analog inputs that are implemented on the board including any implemented externally from A/D converters internal to the microcontroller. As with digital pins, in most cases, the value of this symbol will be the same as NUM_ANALOG_PINS and a default definition is created by pins_arduino.h if the Board_Defs.h file for the board variant doesn't define it.

As with digital pins, this doesn't imply that all of these analog inputs are accessible from connectors. There may be analog inputs internal to a board that measure levels on a board and that don't go out to a connector. There is also no implication that these "analog inputs" are implemented via an A/D converter. A board could, for example, have a built-in temperature sensor, or a built-in accelerometer. These could be represented by the board variant as analog inputs that are internal to the board, and the code to access the "analog values" associated with these channels (e.g., read the acceleration from the accelerometer) could be provided as part of the definition of the board variant. See the section below on Board Extension Functions for the mechanism used to accomplish this.

If a board provided analog inputs implemented externally to the microcontroller, their analog input numbers should follow the analog input numbers for the analog inputs implemented internally to the microcontroller. This is analogous to the case described above for externally implemented digital pins.

`NUM_OC_PINS`	This gives the number of timer output compare channels that are meaningfully usable on the board.
`NUM_IC_PINS`	This gives the number of timer input capture channels that are meaningfully usable on the board.
`NUM_TCK_PINS`	This gives the number of timer external clock input channels that are meaningfully usable on the board.
`NUM_INT_PINS`	This gives the number of external interrupt inputs that are meaningfully usable on the board.
`NUM_SERIAL_PORTS`	This gives the number of UARTs that are meaningfully usable on the board.
`NUM_SPI_PORTS`	This gives the number of SPI ports that are usable via the SPI standard library. The SPI standard library only supports a single SPI port, so this symbol is normally only ever defined to be 1. A board that had no hardware SPI ports available would define it to be 0.
`NUM_I2C_PORTS`	This gives the number of I2C ports that are usable via the wire standard library. The wire library only supports a single I2C port, so this symbol is normally only ever defined to be 1. A board that has no hardware I2C ports available would define it to be 0.
`NUM_DSPI_PORTS`	This gives the number of hardware SPI ports that are accessible using the DSPI standard library.
`NUM_DTWI_PORTS`	This gives the number of hardware I2C ports that are accessible using the DTWI standard library. (NOTE: The DTWI library is not currently implemented and is planned for future expansion).
`NUM_LED`	This gives the number of LEDs on the board that are accessible using `digitalWrite()`.
`NUM_BTN`	This gives the number of momentary contact push buttons on the board that are accessible using `digitalRead()`.
`NUM_SWT`	This gives the number of switches on the board. This nominally represents slide switches or toggle switches, but these could also be “press to make, press to break” type button switches or jumpers. This is intended to be used for non-monentary-contact type switches.

NUM_SERVO	This gives the number of connectors on the board suitable for direct connection of an RC hobby-type servo.

Digital Resource Access Symbols

The following sets of symbols are defined to provide a mechanism for user sketches to have portable access to digital-pin-based resources on different boards (x represents an integer value). They give the pin numbers to use to access various system resources available on the board.

PIN_LEDx	These symbols give the pin numbers used to access the user accessible LEDs available on the board.
PIN_BTNx	These symbols give the pin numbers used to access the push-button inputs available on the board.
PIN_SWTx	These symbols give the pin numbers used to access the switch inputs available on the board.
PIN_SERVOx	These symbols give the pin numbers used to access the servo connectors available on the board.
PIN_OCx	These symbols give the pin numbers where the useable timer output compare channels are located.
PIN_ICx	These symbols give the pin numbers where the usable timer input capture channels are located.
PIN_TCKx	These symbols give the pin numbers where the usable timer external clock inputs are located.
PIN_INTx	These symbols give the pin numbers where the usable external interrupt inputs are located.
PIN_CNx	These symbols give the pin numbers where the usable change notice input pins are located.

Analog Resource Access Symbols

The analog resource access symbols are of the form Ax, where x ranges from 0 to the highest numbered analog input(e.g., A0, A1, etc.). These symbols are intended to be used by sketches as the parameter to `analogRead()` to specify the analog input to be read.

The Arduino system allows two different types of values to be used to specify the analog input number as the parameter to `analogRead()`. Either the analog input number can be specified, or the digital pin number corresponding to the analog input can be specified. Unfortunately, this has the potential to create an ambiguity that can't be resolved. The general presumption is that a number in the range 0 to `NUM_ANALOG_PINS-1` is the analog input number and a number outside of this range is the digital pin number of the pin that shares that analog input. However, there are three actual cases that could occur:

For the following discussion, assume that Ax refers to analog input x, N refers to the number of analog inputs, and D refers to the digital pin number of the first digital pin sharing an analog input. Analog inputs are always in the range Ax >= 0 and Ax <= N-1.

Case 1: D > N: This is the usual case and the condition which the original Arduino design was intended to accommodate. There is no overlap between the range of analog input numbers, Ax, and the digital pin numbers, and there is no ambiguity.

Case 2: D < N, but Ax = Dx: In this case, there is an overlap between the range of analog input numbers and the digital pin numbers, but it happens to be that for the pins that overlap, the analog input number and the digital pin number are the same. This will occur when the analog inputs start on digital pin 0 and are on a continuous range of digital pin numbers. This happens to be the case on the Digilent Cerebot MX4cK board. In this case, there is ambiguity between whether the given value is intended to be an analog input number or a digital pin number, but it doesn't matter because they both have the same value.

Case 3: D < N, but Ax != Dx: In this case, there is an ambiguity that can't be resolved. Some of the analog inputs are on digital pins where the pin number is less than N, but the analog input number and the digital pin number of a given input are different numbers. In this case, the Arduino rule that either the analog input number or the digital pin number can be passed to analogRead() breaks down.

If a board falls into the third case above, the board variant designer has to make a decision as to whether to give precedence to the analog input number or the digital pin number. The definition of the analog input number symbols, Ax, and the analog input mapping macros and mapping table described below have to be set up appropriately to give the desired precedence.

If the preference is to give precedence to the digital pin number, then define the Ax symbols to have the values corresponding to the digital pin numbers and code the digitalPinToAnalog() macro to make the transformation appropriately. If the preference is to give precedence to the analog input number, define the Ax symbols to have the analog input number values (i.e., 0 to N-1) and code the digitalPinToAnalog() macro appropriately for this choice.

SPI Port Pin Declarations

The following symbols are defined for compatibility with the original Arduino SPI library: SS, MOSI, MISO, and SCK. Typically, this looks like the following, taken from the Board_Defs.h file for the Uno32:

```
const static uint8_t SS   = 10;          // this is RD4 (JP4 in RD4 pos)
const static uint8_t MOSI = 11;          // PIC32 SDO2
const static uint8_t MISO = 12;          // PIC32 SDI2
const static uint8_t SCK  = 13;          // PIC32 SCK2
```

For a board that is compatible with the Uno32 shield footprint, these will be the correct pin numbers.

For a board that is compatible with the Max32 shield footprint, these will be the corresponding pin definitions as taken from the Board_Defs.h file for the Max32:

```
const static uint8_t SS   = 53;          // PIC32 SS2A
const static uint8_t MOSI = 51;          // PIC32 SDO2A
const static uint8_t MISO = 50;          // PIC32 SDI2A
const static uint8_t SCK  = 52;          // PIC32 SCK2A
```

For a board that is not shield footprint compatible, use whatever pin numbers are appropriate for the location of the primary SPI port for the board.

For use with the DSPI library, symbols are defined to give the location of the SS pin for each SPI port usable with DSPI. These symbols are of the form PIN_DSPIx_SS.

This example is taken from the Board_Defs.h file for the Uno32:

```
#define       PIN_DSPI0_SS  10
#define       PIN_DSPI1_SS  14
```

This is another example taken from the Board_Defs.h file for the Max32:

```
#define       PIN_DSPI0_SS  53
#define       PIN_DSPI1_SS  74
#define       PIN_DSPI2_SS  19
#define       PIN_DSPI3_SS  15
```

Pin Mapping

Pin mapping is the process of translating from the logical digital pin numbers and analog input numbers to the physical port and bit for digital I/O and the analog input channel number for analog input. The mechanism for performing the mapping involves a number of macros and data lookup tables.

In addition to the macros and data tables used for the digital pin mapping, there is also a set of macros and a data table used to map timer resources. These provide a mechanism for mapping from a digital pin number to the timer resources, if any, that appear on the pin.

Pin Mapping Mechanism

All of the code in the core files is written to perform the translation from logical digital pin number or logical analog input number to the corresponding physical values (e.g., port address and bit position) using macros defined in pins_arduino.h. The default definition of these macros is provided in pins_arduino.h, but a given board variant can, and often will, override these default definitions by providing board specific definitions of some of the macros.

In many cases, these macros will make use of look-up tables stored in program flash to perform the translation.

The default definition for the mapping macros are given in pins_arduino.h. The board variant will override the default definitions, if necessary, in Board_Defs.h by #undef'ing the macro and then using #define to give a new definition for the macro.

There are some generic pin mapping tables defined in `pins_arduino.c`, but the majority of the mapping tables will be board specific and are defined in the board variant file `Board_Data.c`. In many cases, the default definition for a macro will reference a table that the `Board_Data.c` file is expected to provide.

Digital Pin Mapping

Digital pin mapping is the mechanism used to convert from a logical pin number as used by the digital I/O functions (e.g., `pinMode()`, `digitalRead()`, `digitalWrite()`) to the physical port and bit used by that logical pin as defined by the design of the board. In general, the mapping from digital pin number to port and bit is accomplished via three tables. The pin number is used as an index into the "pin to port" table. This is used to find the logical port number. There is then a "logical port number to physical port address" table that is used to convert the logical port number to the physical address of the port. The pin number is also used as an index into a "pin number to bit mask" table that provides a bit mask giving the bit within the port.

Once the physical base address of the port is known, a pointer variable is initialized with this address, and then a port data structure defined in `p32_defs.h` is used to access the appropriate physical register within the set of registers that make up the I/O port. The bit mask value is used to manipulate the appropriate bit within the register.

Port Data Structures

There are two structures defined in `p32_defs.h` used by the digital I/O mechanism: `p32_regset` and `p32_ioport`.

In the PIC32, most special function registers have a regular structure made up of four physical registers: the actual register itself; a clear register; a set register; and an invert register. The `p32_regset` structure describes this layout.

```
/* This structure describes the register layout of the primary
** register, clear, set, and invert registers associated with
** most special function registers.
*/
typedef struct {
	volatile uint32_t	reg;
	volatile uint32_t	clr;
	volatile uint32_t	set;
	volatile uint32_t	inv;
} p32_regset;
```

In the PIC32, an I/O port is made up of a collection of these special function registers. The actual I/O port on PIC32 devices has two variants: The first variant is for devices in the PIC32MX3xx/4xx/5xx/6xx/7xx families; the second variant is for devices in the PIC32MX1xx/2xx families. The PIC32MX1xx/2xx devices expanded the definition of the I/O port and rearranged the order of the registers.

```
/* This structure describes the register layout of an I/O port.
*/
#if defined(__PIC32MX1XX__) || defined(__PIC32MX2XX__)
typedef struct {
	volatile p32_regset ansel;
```

```
        volatile p32_regset  tris;
        volatile p32_regset  port;
        volatile p32_regset  lat;
        volatile p32_regset  odc;
        volatile p32_regset cnpu;
        volatile p32_regset cnpd;
        volatile p32_regset cncon;
        volatile p32_regset cnen;
        volatile p32_regset cnstat;
} p32_ioport;
#else
typedef struct {
        volatile p32_regset  tris;
        volatile p32_regset  port;
        volatile p32_regset  lat;
        volatile p32_regset  odc;
} p32_ioport;
#endif
```

As can be seen by these definitions, the `p32_ioport` is made up of a number of members, each of which is a `p32_regset`.

Although the following glosses over many of the details, this shows the general mapping technique and using the mapped value to set an output bit high:

```
//* Get the port number for this pin.
prt = digitalPinToPort(pin);

//* Obtain pointer to the registers for this io port.
iop = portRegisters(prt);

//* Obtain bit mask for the specific bit for this pin.
bit = digitalPinToBitMask(pin);

//* Manipulate the bit in the latch register associated with the port.
iop->lat.set = bit;
```

The calls to `digitalPinToPort()`, `portRegisters()`, and `digitalPinToBitMask()` are actually expansions of macros defined in `pins_arduino.h` and possibly redefined in `Board_Defs.h`. These three particular macros are generally not redefined, however, and the default definitions are coded to use mapping tables presumed to be supplied by the `Board_Data.c` file.

Digital Pin Mapping Macros

The digital pin mapping macros fall into three general categories: the macros used by the digital input/output mechanism for doing pin I/O; the macros used to map from pin number to other peripheral functions; and the macros used by the peripheral pin-select mechanism. The peripheral pin select related macros will be described in the section on peripheral pin select below.

Digital Input/Output Macros

In general, these macros have completely generic definitions and do not need to be redefined for any board variant:

Macro 1: digitalPinToPort(P)

```
#define digitalPinToPort(P) ( digital_pin_to_port_PGM[P]  )
```

The `digitalPinToPort()` macro is used to map from the logical pin number to the logical port number for the pin. This simply uses the pin number as an index to look up the logical port number from the table.

Macro 2: digitalPinToBitMask(P)

```
#define digitalPinToBitMask(P) ( digital_pin_to_bit_mask_PGM[P]  )
```

The `digitalPinToBitMask()` macro is used to map from the logical pin number to the bit for the pin. This uses the pin number as an index into the table and returns a bit vector with the appropriate bit set for the bit used by the pin.

Macro 3: portRegisters(P)

```
#if defined(__PIC32MX1XX__) || defined(__PIC32MX2XX__)
#define portRegisters(P) ((p32_ioport *)(port_to_tris_PGM[P] - 0x0010))
#else
#define portRegisters(P) ((p32_ioport *)(port_to_tris_PGM[P]))
#endif
```

The `portRegisters()` macro has two definitions: one for the PIC32MX1xx/2xx series devices, and another for all other PIC32 devices. This macro maps from a logical port number to the physical base address of the port. For historical reasons, the mapping is done via a "port_to_tris" table which contains the physical address of the `TRIS` register for the port. On non-MX1xx/2xx devices the `TRIS` register is the first register in the port, and so the address of the `TRIS` register corresponds to the base address of the port. On the PIC32MX1xx/2xx devices, the IOPORT was redefined and the `TRIS` register is not the first register in the port. The PIC32MX1xx/2xx definition of the macro translates from the address of the `TRIS` register to the physical base address of the port.

Timer Resource Mapping Macros

The timer resource mapping macros work in conjunction with a data table to provide a mechanism to identify which timer resources, if any, appear on a given pin.

On a PIC32 microcontroller, the timer resources that are potentially available are an output compare (`OCx`), an input capture (`ICx`) and an external clock input (`TCKx`). The following macros are available:

```
#define digitalPinToTimerOC(P)  ( (digital_pin_to_timer_PGM[P] & _MSK_TIMER_OC)  )
#define digitalPinToTimerIC(P)  ( (digital_pin_to_timer_PGM[P] & _MSK_TIMER_IC)  )
#define digitalPinToTimerTCK(P) ( (digital_pin_to_timer_PGM[P] & _MSK_TIMER_TCK) )
#define digitalPinToTimer(P) digitalPinToTimerOC(P)
```

The `digitalPinToTimer()` macro is a legacy item for compatibility with previous code. It returns the output compare number associated with the pin and is synonymous with `digitalPinToTimerOC()`. Similarly, for legacy compatibility reasons, the field that contains the output compare number is in the low-order bits of the table entries in the mapping table.

Each of these macros takes a pin number as the argument and returns a value indicating which timer resources are available on that pin. These macros make use of the `digital_pin_to_timer_PGM` table described below. The table entries and the corresponding values returned from these macros are detailed in the description for this table.

Digital Pin and Timer Resource Mapping Tables

There are three tables that are used for translating a digital pin number to port and bit. These tables are defined in the `Board_Data.c` file for each board variant and the correct definition of these tables is required for each board variant to work correctly.

Table 1: `const uint32_t port_to_tris_PGM[]`

This table is used to map from the logical port number to the actual physical base address of the port. For historical reasons, it maps the logical port number to the address of the `TRIS` register for the port. The `portRegisters()` macro accesses this table and handles translation from the address of the `TRIS` register to the actual base address of the I/O port structure depending on the processor family that the board is declared to be based on. The definition of this table is the same for all boards, and arguably it should have been placed in `pins_arduino.c` rather than having a separate definition for each board.

A typical entry in this table looks like this:

```
#if defined(_PORTA)
       (uint32_t)&TRISA,
#else
       NOT_A_PORT,
#endif
```

The symbols `_PORTA` and `TRISA` are defined in the processor specific header file that is part of the standard header files provided as part of the compiler tool chain (e.g., `p32mx460f512l.h`). The `_PORTx` and `TRISx` symbols will be defined in the processor-specific header file for each port that is physically implemented in the processor. The symbol `NOT_A_PORT` is defined in `pins_arduino.h`.

Table 2: `const uint8_t digital_pin_to_port_PGM[]`

This table is used to map from the logical pin number to the logical port number for the pin. The logical pin number is used as an index into this table. This table is accessed by the macro: `digitalPinToPort()`. The default definition for this macro is in `pins_arduino.h` and this macro does not normally need to be redefined by the board variant. The default definition looks like this:

```
#define digitalPinToPort(P) ( digital_pin_to_port_PGM[P]  )
```

A typical entry in this table might look like this:

```
_IOPORT_PF,          //     0 RF2  SDA1A/SDI1A/U1ARX/RF2
```

This particular entry is the first one in the table for the chipKIT Max32 board and thus the index (pin number) used to select this entry is 0. This entry indicates that digital pin 0 is located in logical port

_IOPORT_PF. This value is then used as an index into the port_to_tris_PGM table to find the physical address of the port. The comment indicates that this pin is on Port F, bit 2. The symbol _IOPORT_PF is defined in pins_arduino.h, and is a standard symbol used by the board variant mechanism. There are _IOPORT_Px symbols defined for all possible ports on all extant PIC32 devices.

Another possible entry for this table could look like this:

```
NOT_A_PIN,  //  0
```

This would indicate that the corresponding digital pin (as selected by the pin number used as the index) is not a valid digital pin number on this board. The symbol NOT_A_PIN is defined in pins_arduino.h.

Table 3: const uint16_t digital_pin_to_bit_mask_PGM[]

This table is used to map from the logical pin number to a bit mask giving the particular bit in the port for this pin. The logical pin number is used as an index into this table.

A typical entry in this table would look like this:

```
_BV( 2 ),           //     0 RF2  SDA1A/SDI1A/U1ARX/RF2
```

This particular entry is the first one in the table for the chipKIT Max32 board and corresponds to the example given for Table 2 above. The index (pin number) used to select this entry is 0. This entry indicates that this particular digital pin is on bit 2 of the port.

The _BV() macro is defined in pins_arduino.h and has the following definition:

```
#define _BV(bit) (1ul << (bit))
```

Table 4: const uint16_t digital_pin_to_timer_PGM[]

This table is used to provide information for the mapping between timer resources and the digital pin numbers on which those timer resources appear. There are three resources associated with each timer: an output compare (OC); an input capture (IC); and an external clock input (TCK). In most cases, these resources appear on different pins. There are a few PIC32 devices where multiple resources share the same pin.

This table is an array of unsigned 16-bit values. Each table entry is divided into four, 4-bit fields. There is a field for the OC, a field for the IC, a field for the TCK and a spare field that is currently unused. Each field in each entry of the table will indicate which timer resource appears on the corresponding pin, or 0 if no corresponding timer resource exists on that pin. Access to this table is normally accomplished by using the timer resource mapping macros described above.

An example entry in the table would look like this:

```
_TIMER_OC1,             //     3 RD0  SDO1/OC1/INT0/RD0
```

This example is the fourth entry in the table for the chipKIT Max32 board. The fourth entry corresponds to digital pin 3, and this indicates that the output compare for timer 1 (OC1) exists on this pin.

Another example showing a case where two timer resources appear on the same pin:

```
_TIMER_OC5|_TIMER_IC5,      //     10 RD4        PMWR/OC5/IC5/CN13/RD4
```

This example is the eleventh entry in the table for the Uno32 board. The particular PIC32 microcontroller on that board has both the output compare and the input capture for timer 5 (OC5 and IC5) on the same pin, which happens to be Port D, bit 4, and that pin is digital pin 10 on the Uno32.

The symbols _TIMER_OCx, _TIMER_ICx, and _TIMER_TCKx are defined in pins_arduino.h and are generic to all boards and all PIC32 microcontrollers. These symbols are defined in the following way:

```
#define       _TIMER_OC1          (1 << _BN_TIMER_OC)

#define       _TIMER_IC1          (1 << _BN_TIMER_IC)

#define       _TIMER_TCK1         (1 << _BN_TIMER_TCK)
```

The _TIMER_OCx, _TIMER_ICx, and _TIMER_TCKx values are defined to be the corresponding ordinal numbers (1, 2, 3, etc.) for the timer resource shifted to the correct bit position for the field corresponding to the resource type.

The shift count and mask values are defined in pins_arduino.h and are defined as follows:

```
#define       _BN_TIMER_OC        0
#define       _BN_TIMER_IC        4
#define       _BN_TIMER_TCK       8

#define       _MSK_TIMER_OC       0x000F
#define       _MSK_TIMER_IC       0x00F0
#define       _MSK_TIMER_TCK      0x0F00
```

Analog Pin Mapping

In order for the analogRead() function to work, it must be able to translate from the analog source specified by the function parameter into the correct analog input channel to select on the input mux of the analog-to-digital converter. This process is complicated by the fact that the Arduino system allows the programmer to specify the digital pin number or the logical analog input number as the parameter to analogRead().

The mapping process makes use of two macros and, optionally, one or two data tables depending on the needs of the specific board. There is more variability in the implementation of the analog pin mapping system than in the digital pin mapping system.

Analog Pin Mapping Macros

The analog pin mapping process makes use of two macros.

The first macro translates from the input parameter into a logical analog input number. The logical analog input number is a number in the range 0 to N-1, where N is the number of analog inputs supported by the board. This macro must have the property that if the input value is already a logical analog input number it leaves the value unchanged and that if the input value is a digital pin number it translates it to the corresponding logical analog input number.

The second macro translates the logical analog input number into the physical input channel number for the mux channel on the A/D converter.

Currently, there are no PIC32 microcontrollers that have more than one A/D converter. If that ever changes, this mechanism will have to be modified so that it additionally specifies which A/D converter as well as which channel.

The macro used to perform the initial translation to logical analog input number is defined in pins_arduino.h as follows:

```
#define digitalPinToAnalog(P) ( (((P) > 15) && ((P) < 32)) ? (P)-16 : NOT_ANALOG_PIN )
```

This macro is responsible for taking an input value that is either already an analog input number or a digital pin number and returning an analog input number. The default definition for digitalPinToAnalog() as given in pins_arduino.h is incorrect, as it does not correctly handle an input value that is already an analog input number. It presumes that the input value is a digital pin number in the range 16 to 31 and converts it to the corresponding analog input number by subtracting 16. This is not correct for any board and this macro is always redefined by the Board_Defs.h file in the board variant.

There are essentially two cases that need to be considered for a correct definition for the digitalPinToAnalog() macro:

The first case is the simpler to handle. In this case, the board has the property that all of the analog inputs are on a contiguous range of digital pins starting at some base pin number. In this case the definition of digitalPinToAnalog() can simply involve computation based on the pin numbers specifying the range of digital pins.

The Uno32 board fits into this case. The analog inputs on the Uno32 are on digital pins 14 to 25. The Board_Defs.h file for the Uno32 contains the following two lines:

```
#undef digitalPinToAnalog
#define digitalPinToAnalog(P) ( (P) < 12 ? (P) : ((P) >= 14) && ((P) < 26) ? (P)-14 :
NOT_ANALOG_PIN )
```

This removes the default definition and provides a new definition. This definition first checks for the input value being less than 12. If the input value is less than 12, it is already the analog input number and is returned unchanged. If the input value is not less than 12, it is tested to see if it is in the range 14 to 25. If so, it is a digital pin number corresponding to an analog input and the analog input number is computed by subtracting 14. That value is then returned. If the input value isn't in this range, then it is

not a valid analog input and the value NOT_ANALOG_PIN is returned. For many boards, this form of the definition can be used and only the specific pin numbers need to be changed. The chipKIT Max32 uses a similar definition:

```
#undef digitalPinToAnalog
#define digitalPinToAnalog(P) ( (P) < 16 ? (P) : ((P) >= 54) && ((P) < 70) ? (P)-54 :
NOT_ANALOG_PIN )
```

The Max32 has sixteen analog input pins on digital pin 54 to 69.

The second case to consider for `digitalPinToAnalog()` is for any board that doesn't have the property that all analog inputs are on adjacent digital pins. In this case, a mapping table is used to make the translation. If a mapping table is used, it is called `digital_pin_to_analog_PGM[]` and the definition is placed in the `Board_Data.c` file for the board variant. The Digilent Cerebot MX3cK is a board that falls into this category. The following lines taken from the MX3cK `Board_Defs.h` and show the macro definition for this board.

```
#undef digitalPinToAnalog
#define digitalPinToAnalog(P) ( ((P) < NUM_ANALOG_PINS) ? (P) : digital_pin_to_analog_PGM[P] )
```

This definition checks to see if the input parameter is less than `NUM_ANALOG_PINS` and, if so, returns the value unchanged. This is the case if the input value is already the analog input number. Otherwise, the input value is used as an index into the `digital_pin_to_analog_PGM[]` mapping table and the corresponding value returned. See the section below on Analog Pin Mapping Tables for a discussion of how the `digital_pin_to_analog_PGM[]` table is defined.

The second analog pin mapping macro, used to translate from logical analog input number to physical analog mux channel, is defined in `pins_arduino.h` as follows:

```
#define analogInPinToChannel(P) ( P )
```

The default definition for `analogInPinToChannel()` simply returns the parameter value. This is appropriate for boards where the physical analog input channels correspond to the logical analog input numbers (i.e., analog 0 maps to channel 0, analog 1 maps to channel 1, and so on). Many boards have this property, and so the default definition is suitable for use by many boards. The chipKIT Max32 board is one where this property holds, and so the chipKIT Max32 board variant files do not redefine this macro and uses the default definition.

In the cases where the logical analog input numbers do not correspond to the physical mux channel numbers, a lookup table is used and a different definition for the macro is required. The chipKIT Uno32 is a board where this is true and the following shows the definition used by the Uno32.

```
#undef analogInPinToChannel
#define analogInPinToChannel(P) ( analog_pin_to_channel_PGM[P]  )
```

This definition uses the analog input number as an index into a mapping table, `analog_pin_to_channel_PGM[]` and returns the value at the corresponding location. This alternate definition of the macro, and a suitable definition of the `analog_pin_to_channel_PGM[]` table is sufficient

to describe all other boards. See the section below on Analog Pin Mapping Tables for a discussion of how the `analog_pin_to_channel_PGM[]` table is defined.

Analog Pin Mapping Tables

There are two tables involved in mapping analog pins that may need to be defined for any given board. Either or both of these tables may be optional depending on the design of the board.

Table 1: `const uint8_t digital_pin_to_analog_PGM[]`

This table is used to map from a digital pin number to the corresponding analog input number. It is indexed by the digital pin number and contains analog input numbers or the value `NOT_ANALOG_PIN`.

The following shows part of the definition for this table for the Digilent Cerebot MX3cK as an example:

```
const uint8_t digital_pin_to_analog_PGM[] = {
        // Connector JA
        NOT_ANALOG_PIN,             //     0  RE0         PMD0/RE0
        NOT_ANALOG_PIN,             //     1  RE1         PMD1/RE1
        NOT_ANALOG_PIN,             //     2  RE2         PMD2/RE2
        NOT_ANALOG_PIN,             //     3  RE3         PMD3/RE3
        NOT_ANALOG_PIN,             //     4  RE4         PMD4/RE4
        NOT_ANALOG_PIN,             //     5  RE5         PMD5/RE5
        NOT_ANALOG_PIN,             //     6  RE6         PMD6/RE6
        NOT_ANALOG_PIN,             //     7  RE7         PMD7/RE7

        // Connector JB
        NOT_ANALOG_PIN,             //     8  RD9         IC2/U1CTS/INT2/RD9
        NOT_ANALOG_PIN,             //     9  RF3         U1TX/SDO1/RF3
        NOT_ANALOG_PIN,             //     10 RF2         U1RX/SDI1/RF2
        NOT_ANALOG_PIN,             //     11 RF6         U1RTS/SCK1/INT0/RF6
        NOT_ANALOG_PIN,             //     12 RD6         CN15/RD6
        NOT_ANALOG_PIN,             //     13 RD5         PMRD/CN14/RD5
        NOT_ANALOG_PIN,             //     14 RD4         PMWR/OC5/IC5/CN13/RD4
        NOT_ANALOG_PIN,             //     15 RD7         CN16/RD7

        // Connector JC
        _BOARD_AN0,                 //     16 RB8         U2CTS/C1OUT/AN8/RB8
        NOT_ANALOG_PIN,             //     17 RF5         U2TX/PMA8/SCL2/CN18/RF5
        NOT_ANALOG_PIN,             //     18 RF4         U2RX/PMA9/SDA2/CN17/RF4
        _BOARD_AN1,                 //     19 RB14        PMALH/PMA1/U2RTS/AN14/RB14
        _BOARD_AN2,                 //     20 RB0         PGED1/PMA6/AN0/VREF+/CVREF+/CN2/RB0
        _BOARD_AN3,                 //     21 RB1         AN1/VREF-/CVREF-/CN3/RB1
        NOT_ANALOG_PIN,             //     22 RD0         OC1/RD0
        NOT_ANALOG_PIN,             //     23 RD1         OC2/RD1
```

This shows that the first sixteen digital pins are not analog inputs. Analog input 0 (`_BOARD_AN0`) is on pin sixteen; the next two pins are not analog inputs; analog input 1 (`_BOARD_AN1`) is on pin eighteen, and so on. It is not required that the analog input numbers be monotonically increasing throughout the table, but this is the normal convention. The mapping of digital pin number to analog input is completely arbitrary and the board designer can make the assignments in whatever way seems best for the needs of the board.

The symbols _BOARD_ANx are defined in pins_arduino.h. The following is taken from pins_arduino.h and illustrates the form of how these symbols are defined:

```
#define        _BOARD_AN0            0
#define        _BOARD_AN1            1
#define        _BOARD_AN2            2
#define        _BOARD_AN3            3
```

These symbol definitions simply define _BOARD_ANx to be the number x. The constants 0, 1, 2, and so on, could be used in the table, but good programming practices would favor using these symbols.

Table 2: const uint8_t analog_pin_to_channel_PGM[] =

This table is used to map from the analog input number to the physical mux channel used for that analog input. It is indexed by the logical analog input number (e.g., _BOARD_AN0 and _BOARD_AN1) and contains the corresponding mux channel to be used. The chipKIT Uno32 is a board that makes use of this table. The following is the definition of this table from the Uno32 board variant files:

```
const uint8_t analog_pin_to_channel_PGM[] =
{
                        //*    chipKIT Pin           PIC32 Analog channel
        2,              //*    A0                    AN2
        4,              //*    A1                    AN4
        8,              //*    A2                    AN8
        10,             //*    A3                    AN10
        12,             //*    A4                    AN12
        14,             //*    A5                    AN14
        3,              //*    A6                    AN3
        5,              //*    A7                    AN5
        9,              //*    A8                    AN9
        11,             //*    A9                    AN11
        13,             //*    A10                   AN13
        15,             //*    A11                   AN15

};
```

This shows that the first analog input, _BOARD_AN0 (input 0), is mapped to physical analog mux channel 2; the second analog input, _BOARD_AN1 (input 1), is mapped to physical analog mux channel 4, and so on.

Mapping Logical Peripherals to Physical Peripherals

There are various standard libraries that are part of the MPIDE system that are used to access various hardware peripherals in the microcontroller. These libraries provide access to “logical” I/O ports. It is necessary to configure the system to associate physical peripheral devices to be used for the logical I/O ports provided by the libraries.

The HardwareSerial library isn’t a library, per se, because it actually comes from the cores files, but its behavior is the same as a library that is automatically included. The HardwareSerial library creates an object instance to work with each usable UART on the board, as well as, optionally, an object instance to

configure the internal USB controller as a USB serial device. The association of HardwareSerial objects with the physical UART is accomplished via the mechanism described below.

There are two libraries for accessing SPI ports. The original Arduino SPI library was written in such a way that it wasn't really feasible to extend it to work with multiple SPI ports, and so a second library, DSPI, was created for chipKIT to allow access to more than one SPI port. The SPI library accesses the "primary" SPI port. The board variant designer decides which SPI port is the primary one. If the board supports the shield interface footprint, this would be the SPI port connected to digital pins 10 to 13 and the 2x3 SPI connector on the right side of the shield connectors. If the board isn't shield compatible, then the board designer decides which SPI port should be considered the primary one.

The DSPI library provides access to all usable SPI ports on the board. The library creates a subclass of the DSPI class and an instance of each of these subclasses for each SPI port. The DSPI0 subclass and object typically should work with the "primary" SPI port, i.e., the one that the SPI library works with.

The assignment of which physical SPI ports the SPI library object and each DSPI library object uses is accomplished through the mechanism described below.

HardwareSerial Library

The HardwareSerial object class is written to work with any PIC32 UART, provided it is given the necessary configuration information. This information is passed as parameters into the constructor when the object instance is created and stored in member data for later use by the object methods. There are six symbols that need to be defined for each usable UART. The definition for these symbols goes in the `Board_Defs.h` file for the board variant and is typically placed in the "Core Configuration Declarations" section. The following symbols need to be defined:

`_SERx_BASE`	The physical base address of the set of special function registers for the UART.
`_SERx_IRQ`	The interrupt request number used by the UART.
`_SERx_VECTOR`	The interrupt vector number of the receive interrupt used by the UART.
`_SERx_IPL_ISR`	The interrupt privilege level for the interrupt service routine to the receive interrupt.
`_SERx_IPL`	The interrupt privilege level to be programmed into the privilege level register. This value must agree with the value given by `_SERx_IPL_ISR`. There are two different symbols as the form needed for the ISR is different than the form needed for programming the IPL register.
`_SERx_SPL`	The sub-privilege level to be programmed into the interrupt privilege level register.

The hardware UARTs available on the microcontroller are described by corresponding symbols for the UARTs. Some of these symbols are defined in the processor header file that is distributed as part of the compiler tool chain, and some of these symbols are defined in `System_Defs.h` in the `cores` folder.

The following symbols are used:

`_UARTx_BASE_ADDRESS` This symbol is defined in the processor header file and gives the physical base address of the special function registers for the UART.

`_UARTx_ERR_IRQ` This symbol is defined in the processor header file and gives the base interrupt request number used by the UART.

`_UART_x_VECTOR` This symbol is defined in the processor header file and gives the interrupt vector number for the interrupt vector used by the UART.

`_UARTx_IPL_ISR` This symbol is defined in `System_Defs.h` and gives the interrupt privilege level to be used by the interrupt service routine.

`_UARTx_IPL_IPC` This symbol is defined in `System_Defs.h` and gives the interrupt privilege level to be programmed into the interrupt privilege register.

`_UARTx_SPL_IPC` This symbol is defined in System_Defs.h and gives the sub-privilege level to be programmed into the interrupt privilege level register.

The following example is taken from the `Board_Defs.h` file for the chipKIT Max32 board and shows that logical serial port 0 (i.e., the Serial object) is implemented using UART1 on this board:

```
#define       _SER0_BASE          _UART1_BASE_ADDRESS
#define       _SER0_IRQ           _UART1_ERR_IRQ
#define       _SER0_VECTOR        _UART_1_VECTOR
#define       _SER0_IPL_ISR       _UART1_IPL_ISR
#define       _SER0_IPL           _UART1_IPL_IPC
#define       _SER0_SPL           _UART1_SPL_IPC
```

The constructor for the Serial object uses these symbols as parameters that are then used to initialize member data for the object to set it up to work with UART1 as shown in the following example. This creates the object instance for the Serial object and passes the parameters specified in the board variant file to configure the object to work with the correct UART.

```
HardwareSerial Serial((p32_uart *)_SER0_BASE, _SER0_IRQ, _SER0_VECTOR, _SER0_IPL,
_SER0_SPL, IntSer0Handler);
```

Refer to the code in `HardwareSerial.cpp` in the `cores` folder to see how the code in the HardwareSerial object class uses these values to allow it to operate with the UART specified by the parameters to the constructor.

This next example, also taken from the Max32, shows that logical serial port 3 (i.e., the Serial3 object) is implemented using UART5 on this board:

```
#define       _SER3_BASE          _UART5_BASE_ADDRESS
#define       _SER3_IRQ           _UART5_ERR_IRQ
#define       _SER3_VECTOR        _UART_5_VECTOR
#define       _SER3_IPL_ISR       _UART5_IPL_ISR
#define       _SER3_IPL           _UART5_IPL_IPC
#define       _SER3_SPL           _UART5_SPL_IPC

HardwareSerial Serial3((p32_uart *)_SER3_BASE, _SER3_IRQ, _SER3_VECTOR, _SER3_IPL,
_SER3_SPL, IntSer3Handler);
```

The instantiation of HardwareSerial object is based on the definition of the `_SERx_BASE` symbols. The code in `HardwareSerial.cpp` will instantiate up to eight serial objects, Serial through Serial7, based on the presence or absence of these symbols.

```
#if defined(_SER3_BASE)
HardwareSerial Serial3((p32_uart *)_SER3_BASE, _SER3_IRQ, _SER3_VECTOR, _SER3_IPL,
_SER3_SPL, IntSer3Handler);
#endif
```

USB Serial Port

In addition to serial ports implemented using UARTs in the microcontroller, the MPIDE provides the option to use the USB controller in PIC32 devices that support it for the primary serial interface. This is done using a subclass of the HardwareSerial object class.

To use the USB Serial port facility, the symbol `_USE_USB_FOR_SERIAL_` must be defined. This is typically done using a define command in the `boards.txt` entry for the board. This is shown in the following example taken from the `boards.txt` entry for the chipKIT DP32 board:

```
chipkit_DP32.board.define=-D_USE_USB_FOR_SERIAL_
```

When `_USE_USB_FOR_SERIAL_` is defined, an object of the class USBSerial is instantiated as the Serial object, and the UART specified for `_SER0_BASE` will be defined as Serial0. This is shown in the following simplified version of the code taken from `HardwareSerial.cpp`:

```
#if defined(_USB) && defined(_USE_USB_FOR_SERIAL_)
USBSerial     Serial(&rx_bufferUSB);
#if defined(_SER0_BASE)
HardwareSerial Serial0((p32_uart *)_SER0_BASE, _SER0_IRQ, _SER0_VECTOR, _SER0_IPL,
_SER0_SPL, IntSer0Handler);
#endif
#endif
```

SPI Library

The SPIClass object class is written to work with any PIC32 SPI interface, provided it is given the necessary configuration information. This information is passed as parameters into the constructor when the object instance is created and stored in member data for later use by the object methods. There are eight symbols that need to be defined to configure the library to work with the SPI port. The

definition for these symbols goes in the Board_Defs.h file for the board variant and is typically placed in the “Core Configuration Declarations” section. The following symbols need to be defined:

_SPI_BASE	The physical base address of the set of special function registers for the SPI port.
_SPI_ERR_IRQ	The interrupt request number used by the SPI error interrupt.
_SPI_RX_IRQ	The interrupt request number used by the SPI receive interrupt.
_SPI_TX_IRQ	The interrupt request number used by the SPI transmit interrupt.
_SPI_VECTOR	The vector number of the interrupt vector used by the SPI port.
_SPI_IPL_ISR	The interrupt privilege level for the interrupt service routine to the receive interrupt.
_SPI_IPL	The interrupt privilege level to be programmed into the privilege level register. This value must agree with the value given by _SPI_IPL_ISR. There are two different symbols as the form needed for the ISR is different than the form needed for programming the IPL register.
_SPI_SPL	The sub-privilege level to be programmed into the interrupt privilege level register.

The hardware SPI ports available on the microcontroller are described by corresponding symbols for the actual hardware ports available. Some of these symbols are defined in the processor header file that is distributed as part of the compiler tool chain, and some of these symbols are defined in System_Defs.h in the cores folder.

The following symbols are used:

_SPIx_BASE_ADDRESS	This symbol is defined in the processor header file and gives the physical base address of the special function registers for the SPI port.
_SPIx_ERR_IRQ	This symbol is defined in the processor header file and gives the error interrupt request number used by the SPI port.
_SPIx_RX_IRQ	This symbol is defined in the processor header file and gives the receive interrupt request number used by the SPI port.
_SPIx_TX_IRQ	This symbol is defined in the processor header file and gives the transmit interrupt request number used by the SPI port.
_SPI_x_VECTOR	This symbol is defined in the processor header file and gives the interrupt vector number for the interrupt vector used by the SPI port.

_SPIx_IPL_ISR	This symbol is defined in System_Defs.h and gives the interrupt privilege level to be used by the interrupt service routines.
_SPIx_IPL_IPC	This symbol is defined in System_Defs.h and gives the interrupt privilege level to be programmed into the interrupt privilege register.
_SPIx_SPL_IPC	This symbol is defined in System_Defs.h and gives the sub-privilege level to be programmed into the interrupt privilege level register.

The following example, taken from the Board_Defs.h file for the Uno32, illustrates assigning hardware SPI port 2 to be used by the SPI library.

```
#define       _SPI_BASE           _SPI2_BASE_ADDRESS
#define       _SPI_ERR_IRQ        _SPI2_ERR_IRQ
#define       _SPI_RX_IRQ         _SPI2_RX_IRQ
#define       _SPI_TX_IRQ         _SPI2_TX_IRQ
#define       _SPI_VECTOR         _SPI_2_VECTOR
#define       _SPI_IPL_ISR        _SPI2_IPL_ISR
#define       _SPI_IPL            _SPI2_IPL_IPC
#define       _SPI_SPL            _SPI2_SPL_IPC
```

DSPI Library

The DSPI library works very closely to the way that the SPI library works. The DSPI library defines a subclass of the DSPI base class for each SPI port accessible on the board. These are named DSPI0, DSPI1, etc. The DSPI0 object should refer to the "primary" hardware SPI port and refer to the same SPI hardware port that is used by the SPI standard library.

The base object class for the DSPI library is written to operate with any available SPI port using configuration parameters passed into the object constructor. These configuration values are supplied using configuration symbols defined in the Board_Defs.h file for the board variant. The following symbols are defined:

_DSPIx_BASE	The physical base address of the set of special function registers for the SPI port.
_DSPIx_ERR_IRQ	The interrupt request number used by the SPI error interrupt.
_DSPIx_RX_IRQ	The interrupt request number used by the SPI receive interrupt.
_DSPIx_TX_IRQ	The interrupt request number used by the SPI transmit interrupt.
_DSPIx_VECTOR	The vector number of the interrupt vector used by the SPI port.
_DSPIx_IPL_ISR	The interrupt privilege level for the interrupt service routine to the receive interrupt.
_DSPIx_IPL	The interrupt privilege level to be programmed into the privilege level register. This value must agree with the value given by _SPI_IPL_ISR. There are two

different symbols as the form needed for the ISR is different than the form needed for programming the IPL register.

`_DSPIx_SPL` The sub-privilege level to be programmed into the interrupt privilege level register.

The physical hardware SPI ports are described using exactly the same symbols as described above for the SPI library.

The following example, taken from the `Board_Defs.h` file for the Uno32, board illustrates assigning hardware SPI2 to be used by the DSPI0 object:

```
#define     _DSPI0_BASE             _SPI2_BASE_ADDRESS
#define     _DSPI0_ERR_IRQ          _SPI2_ERR_IRQ
#define     _DSPI0_RX_IRQ           _SPI2_RX_IRQ
#define     _DSPI0_TX_IRQ           _SPI2_TX_IRQ
#define     _DSPI0_VECTOR           _SPI_2_VECTOR
#define     _DSPI0_IPL_ISR          _SPI2_IPL_ISR
#define     _DSPI0_IPL              _SPI2_IPL_IPC
#define     _DSPI0_SPL              _SPI2_SPL_IPC
```

Wire Library

The wire library is used to access I2C. As with the SPI library, the Arduino wire library wasn't written is such a way as to easily allow it to be extended to work with multiple I2C ports. The wire library works with the "primary" I2C port. The TwoWire object class in the wire library is written to work with any hardware I2C controller on the microcontroller. The wire object is configured by configuration values passed in as parameters to the constructor. There are eight symbols that are defined to be passed to the wire constructor:

`_TWI_BASE` The physical base address of the set of special function registers for the I2C controller.

`_TWI_BUS_IRQ` The interrupt request number used by the I2C bus interrupt.

`_TWI_SLV_IRQ` The interrupt request number used by the I2C slave interrupt.

`_TWI_MST_IRQ` The interrupt request number used by the I2C master interrupt.

`_TWI_VECTOR` The vector number of the interrupt vector used by the I2C controller.

`_TWI_IPL_ISR` The interrupt privilege level for the interrupt service routine to the I2C interrupt

`_TWI_IPL` The interrupt privilege level to be programmed into the privilege level register. This value must agree with the value given by `_TWI_IPL_ISR`. There are two different symbols as the form needed for the ISR is different than the form needed for programming the IPL register.

_TWI_SPL	The sub-privilege level to be programmed into the interrupt privilege level register.

The hardware SPI ports available on the microcontroller are described by corresponding symbols for the actual hardware ports available. Some of these symbols are defined in the processor header file that is distributed as part of the compiler tool chain, and some of these symbols are defined in System_Defs.h in the cores folder.

The following symbols are used:

_I2Cx_BASE_ADDRESS	This symbol is defined in the processor header file and gives the physical base address of the special function registers for the I2C controller.
_I2Cx_BUS_IRQ	This symbol is defined in the processor header file and gives the bus interrupt request number used by the I2C controller.
_I2Cx_SLAVE_IRQ	This symbol is defined in the processor header file and gives the slave interrupt request number used by the I2C controller.
_I2Cx_MASTER_IRQ	This symbol is defined in the processor header file and gives the master interrupt request number used by the I2C controller.
_I2C_x_VECTOR	This symbol is defined in the processor header file and gives the interrupt vector number for the interrupt vector used by the I2C controller.
_I2Cx_IPL_ISR	This symbol is defined in System_Defs.h and gives the interrupt privilege level to be used by the interrupt service routines.
_I2Cx_IPL_IPC	This symbol is defined in System_Defs.h and gives the interrupt privilege level to be programmed into the interrupt privilege register.
_I2Cx_SPL_IPC	This symbol is defined in System_Defs.h and gives the sub-privilege level to be programmed into the interrupt privilege level register.

The following example, taken from the Board_Defs.h file for the Uno32, board illustrates assigning hardware I2C1 to be used by the wire object:

```
#define       _TWI_BASE           _I2C1_BASE_ADDRESS
#define       _TWI_BUS_IRQ        _I2C1_BUS_IRQ
#define       _TWI_SLV_IRQ        _I2C1_SLAVE_IRQ
#define       _TWI_MST_IRQ        _I2C1_MASTER_IRQ
#define       _TWI_VECTOR         _I2C_1_VECTOR
#define       _TWI_IPL_ISR        _I2C1_IPL_ISR
#define       _TWI_IPL            _I2C1_IPL_IPC
#define       _TWI_SPL            _I2C1_SPL_IPC
```

DTWI Library

The DTWI library is intended to be analogous to the DSPI library and provide access to additional I2C controllers available on the microcontroller. At the time that this document was written, the DTWI library hasn't been written. To allow the library to function properly when it is written, the board variant files should define the correct interface values for use by the interface objects. The configuration of the DTWI library objects will be exactly analogous to the way that DSPI objects are configured, using configuration symbols named similarly to the TWI symbols in the same way that the DSPI symbols are similar to the SPI library symbols.

The following example, taken from the `Board_Defs.h` file for the chipKIT Uno32, illustrates assigning hardware I2C controller I2C1 to be used by the DTWI0 object.

```
#define       _DTWI0_BASE          _I2C1_BASE_ADDRESS
#define       _DTWI0_BUS_IRQ       _I2C1_BUS_IRQ
#define       _DTWI0_SLV_IRQ       _I2C1_SLAVE_IRQ
#define       _DTWI0_MST_IRQ       _I2C1_MASTER_IRQ
#define       _DTWI0_VECTOR        _I2C_1_VECTOR
#define       _DTWI0_IPL_ISR       _I2C1_IPL_ISR
#define       _DTWI0_IPL           _I2C1_IPL_IPC
#define       _DTWI0_SPL           _I2C1_SPL_IPC
```

Board Extension Functions

The board extension functions are available for use when a board requires special processing that doesn't fit into the normal model of how the core code for a given function behaves. This allows extending the behavior of core functions beyond the basic behavior for a given board.

As an example of this kind of extension, consider a board that uses an I/O expander chip to provide additional digital I/O pins beyond the ones available on the microcontroller. The `pinMode()`, `digitalRead()`, and `digitalWrite()` functions in the `core` files have no idea how to do digital I/O through an arbitrary external I/O expander chip. The code to handle reading or writing to these additional I/O pins can be placed in the board extension functions.

Another example of the board extension facility: The Arduino system uses pulse width modulation to support a pseudo-analog output facility accessed using the `analogWrite()` function. A board that was intended to have true analog output capability could be designed to have D/A converters for analog output. A board extension function would be used to allow `analogWrite()` to be used to produce true analog output via these converters.

As another example, the Max32 board requires some custom initialization code to disable the secondary oscillator so that the pins can be used as general purpose I/O. This code is placed in the board initialization extension function.

The board extension functions are divided into logical function groups. Compilation of the code for a given function group is enabled by defining the appropriate enabling symbol in the Board_Defs.h file for the board variant.

The following board extension function groups are defined:

- OPT_BOARD_INIT
- OPT_BOARD_DIGITAL_IO
- OPT_BOARD_ANALOG_READ
- OPT_BOARD_ANALOG_WRITE

The definition for these symbols is made in Board_Defs.h in the section following the "Internal Declarations" comment. When one of these symbols is defined to have a non-zero value, it causes the implementation of the functions in Board_Data.c to be compiled.

The following code is taken from the digitalWrite() function in wiring_digital.c in the cores files. It illustrates how these board extension functions are used:

```
#if (OPT_BOARD_DIGITAL_IO != 0)
        /* Peform any board specific processing.
        */
int     _board_digitalWrite(uint8_t pin, uint8_t val);

        if (_board_digitalWrite(pin, val) != 0)
        {
                return;
        }
#endif          // OPT_BOARD_DIGITAL_IO
```

The board extension code in digitalWrite() is enabled by setting the symbol OPT_BOARD_DIGITAL_IO to a non-zero value in the Board_Defs.h file for the board. The function: _board_digitalWrite() is declared and then a call to it is made. If the _board_digitalWrite() function returns 0, then execution continues on to the normal code in digitalWrite(). This allows the board extension function to pre-fix the normal digitalWrite() code with some board specific code, but still execute the normal digitalWrite() code. When _board_digitalWrite() returns a non-zero value, this indicates that the board extension function handled the operation completely and the normal digitalWrite() code is not executed.

The following functions are included in the various board extension function groups. These are listed under the name of the symbols that is defined to enable them to be compiled into the code:

OPT_BOARD_INIT

```
void _board_init(void)
```

This function is called once during initialization of the system shortly before control transfers to the user sketch. This occurs after basic hardware initialization has been done but before the task manager has

been initialized. (Arguably, task manager initialization should be moved to be before `_board_init()` is called). This is called from the cores function `init()` in `wiring.c`.

OPT_BOARD_DIGITAL_IO

The following functions allow for board specific override/extension of the core digital I/O functions defined in `wiring_digital.c`.

```
int _board_pinMode(uint8_t pin, unit8_t mode)
```

This function is called at the beginning of the `pinMode()` function. It is responsible for performing any board-specific setup required for the `pinMode()` function, or alternatively, to perform the entire `pinMode()` operation if none of the standard code is appropriate. It returns 0 if execution is to continue with the normal `pinMode()` code and non-zero if the normal `pinMode()` code is to be skipped.

```
int _board_getPinMode(uint_t pin, uint8_t * mode)
```

This function is called at the beginning of the `getPinMode()` function. It is responsible for performing any board-specific setup required for the `getPinMode()` function, or alternatively, to perform the entire `getPinMode()` operation if the standard code isn't appropriate. The returned "mode" value is returned through the pointer to mode (`*mode`). The function return value is 0 if execution is to continue with the normal `getPinMode()` code and non-zero if the normal `getPinMode()` code is to be skipped.

```
int _board_digitalWrite(uint8_t pin, uint8_t val)
```

This function is called at the beginning of the `digitalWrite()` function. It is responsible for performing any board specific setup required for the `digitalWrite()` function, or alternatively, to perform the entire digitalWrite operation if the standard code isn't appropriate. This function returns 0 if execution is to continue with the normal `digitalWrite()` code and non-zero if the normal `digitalWrite()` code is to be skipped.

```
int _board_digitalRead(uint8_t pin, uint8_t * val)
```

This function is called at the beginning of the `digitalRead()` function. It is responsible for performing any board specific setup required for the `digitalRead()` function, or alternatively, to perform the entire `digitalRead()` operation if the standard code isn't appropriate. This function returns 0 if execution is to continue with the normal `digitalRead()` code and non-zero if the normal `digitalRead()` code is to be skipped.

OPT_BOARD_ANALOG_READ

```
int _board_analogRead(uint8_t pin, int * val)
```

This function is called at the beginning of the `analogRead()` function. It is responsible for performing any board specific setup required for the `analogRead()` function, or alternatively, to perform the entire analogRead operation if the standard code isn't appropriate. This returned analog value read is returned via the pointer to value provided (`*val`). The function return value is 0 if execution is to continue with the normal `analogRead()` code or non-zero if the normal `analogRead()` code is to be skipped.

```
int _board_analogReference(uint8_t mode)
```

This function is called at the beginning of the `analogReference()` function. It is responsible for performing any board specific processing required for the `analogReference()` opearation. This function returns 0 if execution is to continue with the normal `analogReference()` code, or non-zero if the normal `analogReference()` code is to be skipped.

OPT_BOARD_ANALOG_WRITE

```
int _board_analogWrite(uint8_t pin, int val)
```

This function is called at the beginning of the `analogWrite()` function. It is responsible for performing any board specific setup required by the `analogWrite()` function, or alternatively, to perform the entire `analogWrite()` operation. This function returns 0 if execution is to continue with the normal `analogWrite()` code or non-zero if the normal `analogWrite()` code is to be skipped.

Peripheral Pin Select

Peripheral Pin Select, PPS, is a technology featured in certain chip families within the PIC32 line. PPS is currently implemented in the PIC32MX1xx and PIC32MX2xx families. The PPS technology allows peripheral input and output functions to be mapped to one of several pins rather than the fixed connection from peripheral function to pin that exists in the other PIC32 microcontroller families. This allows the board designer more flexibility in coming up with a usable set of peripherals as it is easier to avoid conflicting uses of the pins, but adds a great deal of complexity to the design of the board variant mechanism. Because of the way PPS is implemented, it is not possible to avoid using it on the chips that support it. Therefore, any board designed using a PIC32 microcontroller that supports PPS will have to deal with the peripheral pin select support in the MPIDE runtime. Refer to section 11.3, Peripheral Pin Select, in the Microchip PIC32MX1XX/2XX Family Data Sheet for more detail about the PPS mechanism while reading the following discussion about PPS.

An important point to note about the PPS implementation in the board variant mechanism is that it provides a facility for the board designer to assign a default mapping of peripheral functions to pins. There are, additionally, functions available in the runtime to allow the sketch programmer to dynamically change the pin mapping, but the PPS mechanism described here allows the board designer to define the “standard” or default mapping of pins for the board.

On the microcontrollers where PPS is implemented, only some of the pins support PPS. Some peripheral functions are not able to be remapped at all, and within peripherals that are able to be remapped, not all of the signals associated with the peripheral can be remapped. For example, analog functions are not remappable using the PPS mechanism. All analog inputs, both A/D converter inputs, and analog comparator inputs have fixed pin assignments. The I2C bus has special drive requirements so that the I2C pins are not able to be remapped. On the PIC32MX1xx and 2xx devices, the SPI clock inputs are not reassignable. There are also limits on the total number of pins that can support PPS, and so on larger pin-count packages, some digital pins are not part of the PPS mechanism.

The implementation of PPS involves the use of multiplexers. For output functions, e.g., a UART TX signal, there are mux'es at the pins, allowing a selection to be made for one of several output signals to be assigned to that pin. For input functions, e.g., a UART RX signal, there are mux'es at the peripheral inputs allowing a selection of one of several pins to be the source of that signal into the peripheral. For each of these mux'es, there is a register into which a selection value is loaded to specify the output function to be assigned to the pin or the pin to be assigned to the input function. There is a selection register for each mux. For outputs, there is a selection register associated with each pin that has PPS output capability. For inputs, there is a selection register for each peripheral input function that has PPS input capability. On all PIC32 microcontrollers, there are pins and peripheral functions that are not PPS capable, and therefore not able to be reassigned.

The PPS capable pins are divided up into four disjoint sets. For input, peripheral input functions are partitioned between these four pin sets. So, for example, external interrupt input `INT4` can be assigned to take its input from one of eight pins. Similarly, external interrupt input `INT3` can be assigned to take its input from one of eight pins, but they are a different set of pins than the ones for `INT4`. The PIC32MX1XX/2XX data sheets shows that there are an additional eight selection values that are reserved for future use in each pin set, so presumably, some future PIC32 device will divide the PPS capable pins into four sets of sixteen rather than four sets of eight.

For peripheral output, the output functions are divided into four sets for association with the pin sets, but the peripheral output sets are not disjoint. In some cases, a given peripheral output function appears in more than one set, allowing some outputs to be routed to one of sixteen pins rather than just one of eight pins. As with the inputs, the data sheet shows eight additional reserved values for each set, so some future PIC32 device will, presumably, allow mapping up to sixteen different peripheral outputs to each PPS output pin.

As an example of PPS input mapping, consider external interrupt `INT4`. There is an input selection register associated with this peripheral input called `INT4R`. To map `INT4` to take its input from the pin associated with PORT B, bit 3, the value `0b0001` is loaded into `INT4R`. Similarly, for mapping an output, again consider PORT B, bit 3. There is an output selection register associated with this pin called `RPB3R`. To map the output signal `OC1` (output compare 1) to this pin, the value `0b0101` is loaded into `RPB3R`. Obviously, one wouldn't do both of these at the same time as this would produce a pin conflict. To configure a pin for general purpose output, the value `0b0000` is loaded into the output selection register for the pin. This value is described as "No Connect" in Table 11-2, Output Pin Selection, in the Microchip data sheet.

It is important to keep in mind that for peripheral output functions, the mux is at the pin and the selection register is associated with the pin; for peripheral input functions, the mux is at the peripheral input and the selection register is associated with the peripheral input function.

The PIC32 special function registers associated with PPS input selection are a set of 32-bit registers that appear in memory starting at the address of `INT1R` (virtual address `0xBF80FA04`). This value is defined in the processor-specific header file for each particular processor, although all current processors have all

of these registers at the same addresses. It is important to note that this set of SFRs is not continuous. There are holes in the range of memory addresses where no SFR is implemented. The details for these registers can be found in Table 4-22, Peripheral Pin Select Input Register Map, in the data sheet.

Similarly to the PPS input selection registers, the PPS output selection registers are a set of 32-bit registers appearing in memory starting at the address for `RPA0R` (the PORT A, bit 0, output mapping register) that is located at address `0XBF80FB00`. Similarly to the input mapping registers, the symbols for all of the output mapping registers are defined in the processor-specific header file for the processor. Also, similarly to the input mapping registers, the SFRs for the output selection mapping registers are not a continuous set. There are holes in the address range where no register is implemented. The details for these registers can be found in Table 4-23, Peripheral Pin Select Output Register Map, in the data sheet.

PPS Support Implementation

The PPS portion of the board variant mechanism makes use of a large number of symbols defined in `p32_defs.h`, helper macros defined in `pins_arduino.h`, and data tables defined in `pins_arduino.c` and the `Board_Data.c` file for each board variant. The additional data tables in `Board_Data.c` are, of course, only defined for boards that use PIC32 devices that support peripheral pin select. The following sections describe in detail these various supporting symbols, macros, and data tables.

Function: `boolean mapPps(uint8_t pin, ppsFunctionType func)`

The primary run-time interface between the sketch or libraries and the PPS mechanism is the function `mapPps()`. This function takes the pin number and the function to be mapped. If the mapping is valid, it performs the mapping and returns `true`. If the mapping is invalid and can't be performed, it returns `false`.

The `mapPps()` function is defined in `wiring.c` in the `cores` folder. Most of the code is involved in error checking and makes use of the PPS macros for its operation.

The error checking is performed in the following way:

```
// Check for valid PPS pin number and valid function number (input or output)
   if (
    !isPpsPin(pin)
    ||
    ((ppsInputFromFunc(func) > NUM_PPS_IN) && ppsFuncIsInput(func))
    ||
    ((ppsOutputFromFunc(func) > NUM_PPS_OUT) && ppsFuncIsOutput(func))
)
   {
          return false;
   }

   /* Check if the requested peripheral input can be mapped to
   ** the requested pin.
   */
   if ((ppsSetFromPin(pin) & ppsSetFromFunc(func)) == 0)
   {
```

```
            return false;
        }
```

This checks that the pin is capable of PPS operation, that for input operations, the function number is in the valid range and that for output functions the function number is in the valid range. It then checks that the pin and the requested function are in the same set.

Assuming that the above error checks succeed, the mapping is accomplished in the following way:

```
    if (ppsFuncIsInput(func))
    {
       /* An input is mapped from the pin to the peripheral input
       ** function by storing the select value into the register associated
       ** with the peripheral function.
       */
       pps = ppsInputRegister(func);
       *pps = ppsInputSelect(pin);
       }
    else
       {

       /* An output is mapped by storing the select value for the output function
       ** being mapped into the mapping register associated with the pin.
       */
       pps = ppsOutputRegister(pin);
       *pps = ppsOutputSelect(func);
   }
```

If the requested function is an input, the address of the mapping register is obtained using the `ppsInputRegister(func)` macro. The register address is dereferenced and the value returned by the `ppsInputSelect(pin)` macro is stored there.

Similarly, if the requested function is an output, the address of the mapping register is obtained using the `ppsOutputRegister(pin)` macro and the address dereferenced to store the value obtained by using the `ppsOutputSelect(func)` macro.

The details of how this mapping occurs is contained within the definitions of these macros and the data tables that they use. The following sections describe the symbols, macros, and data tables used.

PPS Symbol Definitions

The vast majority of the definitions for symbols associated with the PPS mechanism are defined in `p32_defs.h` in the `cores` folder. There are also a small number of symbols defined in `pins_arduino.h`. Refer to the Peripheral Pin Select Declarations section of `p32_defs.h` for the following discussion:

PPS Functions Definition Symbols

The `enum ppsFunctionType`, defines all possible PPS input and output functions. These symbols are used as a parameter to the `mapPps()` function to specify the PPS function to be mapped. The declaration of this enum is as follows (taken from `p32_defs.h`):

```
typedef enum {
/* The following symbols define the output functions that are mappable with
```

```
** PPS. These give the select values used to map a peripheral function to a PPS
** output pin combined with their set membership. The PPS output select values
** are divided into four sets. Some peripheral functions are duplicated in more
** than one set. In this case they have the same select value in each set.
*/
    PPS_OUT_GPIO     = (0 + (_PPS_SET_A|_PPS_SET_B|_PPS_SET_C|_PPS_SET_D)),

    PPS_OUT_U1TX     = (1 + _PPS_SET_A),
    PPS_OUT_U2RTS    = (2 + _PPS_SET_A),
    PPS_OUT_SS1      = (3 + _PPS_SET_A),
    PPS_OUT_OC1      = (5 + _PPS_SET_A),
    PPS_OUT_C2OUT    = (7 + _PPS_SET_A),

    PPS_OUT_SDO1     = (3 + (_PPS_SET_B | _PPS_SET_C)),
    PPS_OUT_SDO2     = (4 + (_PPS_SET_B | _PPS_SET_C)),
    PPS_OUT_OC2      = (5 + _PPS_SET_B),

    PPS_OUT_OC4      = (5 + _PPS_SET_C),
    PPS_OUT_OC5      = (6 + _PPS_SET_C),
    PPS_OUT_REFCLKO  = (7 + _PPS_SET_C),

    PPS_OUT_U1RTS    = (1 + _PPS_SET_D),
    PPS_OUT_U2TX     = (2 + _PPS_SET_D),
    PPS_OUT_SS2      = (4 + _PPS_SET_D),
    PPS_OUT_OC3      = (5 + _PPS_SET_D),
    PPS_OUT_C1OUT    = (7 + _PPS_SET_D),

/* The following symbols define the input functions that are mappable
** using PPS. These are used as an index to the input selection mapping
** register for that peripheral function. In the current (as of 07/12/2012)
** PIC32MX1xx/2xx devices, this is a direct mapping. If future devices add
** new input functions, or change the order of the input select registers,
** this mapping will have to be changed to be done through a table.
*/

    PPS_IN_INT1            = (0  + _PPS_SET_D + _PPS_INPUT_BIT),
    PPS_IN_INT2            = (1  + _PPS_SET_C + _PPS_INPUT_BIT),
    PPS_IN_INT3            = (2  + _PPS_SET_B + _PPS_INPUT_BIT),
    PPS_IN_INT4            = (3  + _PPS_SET_A + _PPS_INPUT_BIT),
    PPS_IN_T2CK            = (5  + _PPS_SET_A + _PPS_INPUT_BIT),
    PPS_IN_T3CK            = (6  + _PPS_SET_B + _PPS_INPUT_BIT),
    PPS_IN_T4CK            = (7  + _PPS_SET_C + _PPS_INPUT_BIT),
    PPS_IN_T5CK            = (8  + _PPS_SET_D + _PPS_INPUT_BIT),
    PPS_IN_IC1             = (9  + _PPS_SET_C + _PPS_INPUT_BIT),
    PPS_IN_IC2             = (10 + _PPS_SET_D + _PPS_INPUT_BIT),
    PPS_IN_IC3             = (11 + _PPS_SET_B + _PPS_INPUT_BIT),
    PPS_IN_IC4             = (12 + _PPS_SET_A + _PPS_INPUT_BIT),
    PPS_IN_IC5             = (13 + _PPS_SET_C + _PPS_INPUT_BIT),
    PPS_IN_OCFA            = (17 + _PPS_SET_D + _PPS_INPUT_BIT),
    PPS_IN_OCFB            = (18 + _PPS_SET_C + _PPS_INPUT_BIT),
    PPS_IN_U1RX            = (19 + _PPS_SET_C + _PPS_INPUT_BIT),
    PPS_IN_U1CTS           = (20 + _PPS_SET_B + _PPS_INPUT_BIT),
    PPS_IN_U2RX            = (21 + _PPS_SET_B + _PPS_INPUT_BIT),
    PPS_IN_U2CTS           = (22 + _PPS_SET_C + _PPS_INPUT_BIT),
    PPS_IN_SDI1            = (32 + _PPS_SET_B + _PPS_INPUT_BIT),
    PPS_IN_SS1             = (33 + _PPS_SET_A + _PPS_INPUT_BIT),
    PPS_IN_SDI2            = (35 + _PPS_SET_C + _PPS_INPUT_BIT),
    PPS_IN_SS2             = (36 + _PPS_SET_D + _PPS_INPUT_BIT),
```

```
    PPS_IN_REFCLKI          = (45 + _PPS_SET_A + _PPS_INPUT_BIT),
} ppsFunctionType;
```

The members of this `enum` are made up of bit fields. The low eight bits are the selection value or index value associated with the function. The low-order bits of the high byte are bit flags indicating function set membership. The high-order bit is an input/output flag. Setting the bit indicates that the function is an input function. The function membership bits are used by the `mapPps()` function for error checking.

The `PPS_OUT_xxx` values are made up of a selection value for the function itself and the function sets of which that function is a member. Some output functions are members of multiple sets. The GPIO output function is a member of all sets. The two SPI data output signals (`SDO1` and `SDO2`) are members of sets B and C. The selection value for each `PPS_OUT_` function is the value loaded into the PPS mapping register associated with the pin to specify the function mapped to that pin.

The `PPS_IN_xxx` values are made up of an index value for the function and the set membership. The input functions additionally have the `_PPS_INPUT_BIT` set to indicate that they are input functions. As with the `PPS_OUT_xxx` symbols, the set membership bits are used by the `mapPps()` function for error checking. The `_PPS_INPUT_BIT` is used by the `mapPps()` function to determine whether the function being mapped is an input or an output. The process of mapping an input is different than that of mapping an output.

The index field contained within the `PPS_IN_xxx` values gives the index within the "array" of special function registers. The PIC32 special function registers are all memory mapped, and the range of memory addresses making up the PPS input registers can be viewed as an array in memory (see Table 4-22 in the PIC32MX1XX/2XX Family Data Sheet). The index value is used to compute the memory address of the corresponding PPS input mapping register. Note that there are holes in the range of addresses. The PPS input mapping registers are not a contiguous sequence of registers.

The set membership bits and the input bit are defined as follows:

```
#define       _PPS_SET_A          0x0100
#define       _PPS_SET_B          0x0200
#define       _PPS_SET_C          0x0400
#define       _PPS_SET_D          0x0800

#define       _PPS_INPUT_BIT      (1 << 15)
```

PPS Pin Definition Symbols

There are two kinds of symbols defined that specify a pin. One set of symbols give the selection values to load into a PPS input mapping register to map that input to the specific pin. The other set of symbols give the index to the PPS output mapping register within the array of memory addresses that correspond to the PPS output mapping registers. The first set of symbols is used for mapping inputs. The second set of symbols is used when mapping outputs.

The following symbols define the inputs. These are the values loaded into the input mapping register to assign the pin to that input function:

```
#define       _PPS_RPA0           (0 + _PPS_SET_A)
#define       _PPS_RPB3           (1 + _PPS_SET_A)
#define       _PPS_RPB4           (2 + _PPS_SET_A)
#define       _PPS_RPB15          (3 + _PPS_SET_A)
#define       _PPS_RPB7           (4 + _PPS_SET_A)
#define       _PPS_RPC7           (5 + _PPS_SET_A)
#define       _PPS_RPC0           (6 + _PPS_SET_A)
#define       _PPS_RPC5           (7 + _PPS_SET_A)

#define       _PPS_RPA1           (0 + _PPS_SET_B)
#define       _PPS_RPB5           (1 + _PPS_SET_B)
#define       _PPS_RPB1           (2 + _PPS_SET_B)
#define       _PPS_RPB11          (3 + _PPS_SET_B)
#define       _PPS_RPB8           (4 + _PPS_SET_B)
#define       _PPS_RPA8           (5 + _PPS_SET_B)
#define       _PPS_RPC8           (6 + _PPS_SET_B)
#define       _PPS_RPA9           (7 + _PPS_SET_B)

#define       _PPS_RPA2           (0 + _PPS_SET_C)
#define       _PPS_RPB6           (1 + _PPS_SET_C)
#define       _PPS_RPA4           (2 + _PPS_SET_C)
#define       _PPS_RPB13          (3 + _PPS_SET_C)
#define       _PPS_RPB2           (4 + _PPS_SET_C)
#define       _PPS_RPC6           (5 + _PPS_SET_C)
#define       _PPS_RPC1           (6 + _PPS_SET_C)
#define       _PPS_RPC3           (7 + _PPS_SET_C)

#define       _PPS_RPA3           (0 + _PPS_SET_D)
#define       _PPS_RPB14          (1 + _PPS_SET_D)
#define       _PPS_RPB0           (2 + _PPS_SET_D)
#define       _PPS_RPB10          (3 + _PPS_SET_D)
#define       _PPS_RPB9           (4 + _PPS_SET_D)
#define       _PPS_RPC9           (5 + _PPS_SET_D)
#define       _PPS_RPC2           (6 + _PPS_SET_D)
#define       _PPS_RPC4           (7 + _PPS_SET_D)
```

These symbols declare set membership for the pins so that `mapPps()` can check that a function being mapped belongs to the same set as the pin to which it is being mapped. Note that the actual selection values in these symbols repeat for each set.

The following symbols are used for mapping outputs. These symbols give the index to the PPS output selection register associated with the pin:

```
#define       _PPS_RPA0R          0
#define       _PPS_RPA1R          1
#define       _PPS_RPA2R          2
#define       _PPS_RPA3R          3
#define       _PPS_RPA4R          4
#define       _PPS_RPA8R          8
#define       _PPS_RPA9R          9

#define       _PPS_RPB0R          11
```

```
#define       _PPS_RPB1R          12
#define       _PPS_RPB2R          13
#define       _PPS_RPB3R          14
#define       _PPS_RPB4R          15
#define       _PPS_RPB5R          16
#if defined(__PIC32MX1XX__)
#define       _PPS_RPB6R          17
#else
#define       _PPS_RPB6R          NOT_PPS_PIN
#endif
#define       _PPS_RPB7R          18
#define       _PPS_RPB8R          19
#define       _PPS_RPB9R          20
#define       _PPS_RPB10R         21
#define       _PPS_RPB11R         22
#define       _PPS_RPB13R         24
#define       _PPS_RPB14R         25
#define       _PPS_RPB15R         26

#define       _PPS_RPC0R          27
#define       _PPS_RPC1R          28
#define       _PPS_RPC2R          29
#define       _PPS_RPC3R          30
#define       _PPS_RPC4R          31
#define       _PPS_RPC5R          32
#define       _PPS_RPC6R          33
#define       _PPS_RPC7R          34
#define       _PPS_RPC8R          35
#define       _PPS_RPC9R          36
```

Each PPS capable pin has an output selection register. The set of these special function registers can be viewed as an array of memory locations. These symbols give the index within this array for the register associated with a specific pin. Note that there are holes in the range of indices for these registers.

Note that the set of input selection symbols and output index symbols look very similar. The symbols of the form: `_PPS_RPxx` are for input selection and specify the selection value to load into an input mapping register to select the pin. The symbols of the form: `_PPS_RPxxR` are for output selection and specify the index of the output select register associated with the pin.

PPS Macro Definitions

The macros associated with the PPS system are defined in `pins_arduino.h`. These macros fall into two groups: macros used to create the supporting data tables; and macros used to access the data tables and to translate the parameter values to the `mapPps()` function.

The following macros are used in the definition of the PPS mapping tables:

```
#define       _PPS_OUT(P) (P)
#define       _PPS_IN(P) (uint8_t)(((P) & 0x0F) | ((P) >> 4))
```

The `_PPS_OUT` macro is used in the definition of the data table used to map from the digital pin number to the PPS select register associated with the pin. Currently, this macro just returns its parameter value,

but the macro should be used in case future definitions that do actual processing are required. The values used as parameters to this macro are the symbols documented above of the form _PPS_RPxxR.

The _PPS_IN macro is used in the definition of the data table used to map from the digital pin number to the selection value used to specify that pin for an input function. In this case, the macro is used to compress the parameter value into an 8-bit value for storage in the table. The parameters to this macro are the symbols _PPS_RPxx described above. To allow for future expansion, these symbols are defined in a format that occupies 16-bits. For all current PIC32 devices, this can be compressed without loss into 8-bits to allow for more compact storage in memory. If some future PIC32 device support more than 4 function/pin sets, or allows for more than 16 selection options in a selection register, this macro will need to be redefined to store the values in 16-bits for chips in that family.

The remaining macros are used to access the PPS mapping tables or to extract fields from the pin or function symbols.

The following macros are used:

```
#define isPpsPin(P) ((digital_pin_to_pps_out_PGM[P] == NOT_PPS_PIN) ? 0 : 1)
```

This macro is used to determine if a pin has PPS capability. The parameter is the chipKIT digital pin number, and the digital_pin_to_pps_out_PGM[] table is used.

```
#define ppsInputSelect(P) (digital_pin_to_pps_in_PGM[P] & 0x000F)
```

This macro returns the selector value from an element in the digital_pin_to_pps_in_PGM[] table. The selector value for an input mapping is the value to load into the input function select register to map the input to the pin. The parameter is the chipKIT digital pin number. This macro should use the symbol PPS_IN_MASK defined in p32_defs.h rather than the magic number 0x000F.

```
#define ppsOutputSelect(F) ((F) & PPS_OUT_MASK)
```

This macro returns the function selector value to be loaded into a PPS output mapping register. The parameter to this macro is the PPS function value. These are the symbols defined as output function values in the enum ppsFunctionType described above, e.g., PPS_OUT_U1TX.

```
#define ppsSetFromPin(P) ((digital_pin_to_pps_in_PGM[P] >> 4) & 0x000F)
```

This macro returns the PPS function/pin set membership associated with the chipKIT digital pin number. The parameter to this macro is the digital pin number. This macro will have to be redefined for any future PPS capable PIC32 devices that support more than 16 selection options for input or output mapping.

```
#define ppsSetFromFunc(F) (((F) >> 8) & 0x000F)
```

This macro returns the PPS function/pin set membership associated with the specified PPS peripheral function.

```
#define ppsInputFromFunc(F) ((F) & PPS_IN_MASK)
```

This macro returns the selection value associated with the specified PPS input function.

```
#define ppsOutputFromFunc(F) ((F) & PPS_OUT_MASK)
```

This macro returns the selection value associated with the specified PPS output function.

```
#define ppsFuncIsInput(F)   ((F) & _PPS_INPUT_BIT)
#define ppsFuncIsOutput(F)  (!ppsFuncIsInput(F))
```

These macros check the `_PPS_INPUT_BIT` bit to determine if the specified function is an input function or an output function.

```
#define ppsOutputRegister(P) (volatile uint32_t *)((uint32_t)(&_RPOBASE) +
4*digital_pin_to_pps_out_PGM[P])
```

This macro returns the address of the PPS output selection register associated with the specified chipKIT digital pin number. The `digital_pin_to_pps_out_PGM[]` table contains the indices within the array of PPS output mapping special function registers indexed by the chipKIT digital pin number. The PPS output mapping registers are treated as an array of 32-bit values, and so the index value is multiplied by four and then added to `_RPOBASE`. The symbol `_RPOBASE` is defined in `pins_arduino.h` and is defined to be the address of the first output mapping register.

```
#define ppsInputRegister(F) ((uint32_t *)(4*(ppsInputFromFunc(F)) + (uint32_t)&_RPIBASE))
```

This macro returns the address of the PPS input selection register associated with the specified PPS input function. The parameter to this macro is an input function value as defined in the enum `ppsFunctionType`. The `ppsInputFromFunc()` macro is used to extract the selection value. The selection value is the index into the array of PPS input mapping special function registers. These registers are an array of 32-bit values, so the index is multiplied by four and added to `_RPIBASE`. The symbol `_RPIBASE` is defined in `pins_arduino.h` and is defined to be the address of the first PPS input mapping register.

PPS Mapping Table Definitions

There are several data tables that the board-variant designer needs to provide to configure the system to the board. These tables define the default pin assignments for the board.

Table 1: `const uint8_t digital_pin_to_pps_out_PGM[]`

This table is used to determine the PPS output selection register associated with the chipKIT digital pin number. The index into this table is the digital pin number. The table entries contain the index of the PPS output mapping register to use to map output for that digital pin number.

A typical entry for this table would look like the following:

```
    _PPS_OUT(_PPS_RPB5R),   //  0   J4-1    RB5     TMS/RPB5/USBID/RB5
```

This is the first entry in the table for the chipKIT DP32 board, and is the mapping associated with digital pin 0. On this board, digital pin 0 is connected to the pin associated with Port B, bit 5. Note that the

_PPS_OUT macro is used to define the table entry, and the value being defined is the value that specifies the index of the output mapping register (i.e., _PPS_RPB5R).

Some pins are not PPS capable, and in the case where the pin isn't capable of PPS operation, or the pin is undefined for some other reason, the following table entry is used:

```
NOT_PPS_PIN,
```

Table 2: const uint8_t digital_pin_to_pps_in_PGM[]

This table is used to specify the value to be loaded into a PPS input select register to select the physical pin associated with a chipKIT digital pin. The index into this table is the chipKIT digital pin number. The value in the table contains both the selection value to load into the input mapping register and the set membership of the pin for runtime error checking by the mapPps() function.

A typical entry in this table looks like this:

```
_PPS_IN(_PPS_RPB5),     //  0   J4-1    RB5     TMS/RPB5/USBID/RB5
```

This is the first entry in the table for the chipKIT DP32 board. This is the mapping associated with digital pin 0, and specifies that this digital pin is on the physical pin associated with Port B, bit 5. The _PPS_IN macro is used to define table entries in this table and the value assigned specifies the input select value to use for this pin (i.e., _PPS_RPB5).

If the pin is undefined, or not a PPS capable pin, use the following value for the table entry:

```
NOT_PPS_PIN,
```

Table 3: const uint8_t output_compare_to_digital_pin_PGM[]

This table maps from the output compare number as stored in the digital_pin_to_timer_PGM[] table. The definition of this table will typically look like this:

```
const uint8_t output_compare_to_digital_pin_PGM[] = {
       PIN_OC1,
       PIN_OC2,
       PIN_OC3,
       PIN_OC4,
       PIN_OC5,
};
```

The symbols PIN_OCx are defined in the Board_Defs.h file for the board variant.

Table 4: const uint8_t external_int_to_digital_pin_PGM

This table maps from the external interrupt number to the digital pin associated with the external interrupt. The definition of this table will typically look like this:

```
const uint8_t external_int_to_digital_pin_PGM[] = {
       NOT_PPS_PIN,         // INT0 is not mappable;    RB7
       PIN_INT1,
       PIN_INT2,
       PIN_INT3,
```

```
        PIN_INT4
};
```

External interrupt `INT0` is not mappable through PPS on current PIC32MX1xx/2xx devices. The symbols `PIN_INTx` are defined in the `Board_Defs.h` file for the board variant.

Revision History

Revision 0 – October 3, 2013
Internal release for review

Revision A – November 4, 2013
Initial public release