CrossWorks Platform Library

Version: 3.7

Contents

CrossWorks Platform Library

About the CrossWorks Platform Library

The *CrossWorks Platform Library* presents a standardized API for delivering high-quality example code for a wide range of microcontrollers and evaluation boards. Additional components that integrate with the Platform Library are:

- *CrossWorks Tools Library*: provides add-ons for CTL such as read-write locks and ring buffers.
- *CrossWorks Device Library*: provides drivers for common digital sensors, such as accelerometers, gyroscopes, magnetometers, and so on.
- *CrossWorks Graphics Library*: is a library of simple graphics functions for readily-available LCD controllers.
- *CrossWorks TCP/IP Library*: provides TCP/IP networking for integrated and external network controllers on memory-constrained microcontrollers.
- *CrossWorks Mass Storage Library*: provides a FAT-based file system for mass storage on SD and MMC cards, or any device with a block-based interface.
- *CrossWorks Shield Library*: provides drivers for a range of Arduino-style shields.
- *CrossWorks CoreBASIC Library*: provides a full-featured, network-enabled BASIC interpreter which demonstrates the capabilities of these libraries.

Architecture

The *CrossWorks Platform Library* is one part of the *CrossWorks Target Library*. Many of the low-level functions provided by the target library are built using features of the *CrossWorks Tasking Library* for multi-threaded operation.

Delivery format

The *CrossWorks Platform Library* is delivered in source form.

Feedback

This facility is a work in progress and may undergo rapid change. If you have comments, observations, suggestions, or problems, please feel free to air them on the **CrossWorks Target and Platform API** discussion forum.

License

The following terms apply to the Rowley Associates Platform Library.

Introduction

About the CrossWorks Platform Library

The *CrossWorks Platform Library* is a standard API that runs on a collection of popular microprocessors and evaluation boards. It is a way for Rowley Associates to deliver examples, from simple to complex, for those boards.

In particular, the Platform Library requires the *CrossWorks Tasking Library* for operation. Because the Platform Library, and facilities built on top of it, use interrupts and background processing, we made the decision to use the CrossWorks Tasking Library as a foundation stone for the Platform Library. We have not abstracted the Platform Library to use a generic RTOS as this adds more complexity to the design.

Why use the Platform Library?

Standardizing on the Platform Library provised a certain amount of portability for you applications. Rather than using vendor-supplied libraries that get you running quickly on their silicon, you can invest some time learning the Platform Library and use that knowledge across different architectures. You are, however, committing to use CrossWorks, CTL, and the Platform Library for the long term.

What the Platform Library isn't

The Platform Library it is not a general-purpose API supporting every feature offered by common devices, nor does it cater for all devices within a family. The Platform Library is tested on the microprocessors and evaluation boards that Rowley Associates deliver examples for. Certainly, you can use it with little or no modification on boards that have other processors in the families we support, but you will need to customize the Platform Library implementation yourself.

What the Platform Library runs on

The Platform Library runs on the following microprocessor families:

- LPC1700
- LM3S
- KL05Z
- KL25Z
- STM32F1
- STM32F4

The range of boards and microprocessors that run the Platform Library continues to expand. Please check the CrossWorks web site for the latest information.

Blinking one LED

Ignition on!

Diving straight into code, you can blink a LED on your target board with a few lines of code:

```
// Blink the first platform LED.

#include "libplatform/platform.h"

void
main(void)
{
  // Initialize platform.
  platform_initialize();

  // Blink first LED forever.
  for (;;)
    {
      platform_write_led(0, 1);     // LED on
      platform_spin_delay_ms(500);  // Wait
      platform_write_led(0, 0);     // LED off
      platform_spin_delay_ms(500);  // Wait
    }
}
```

Hopefully, this should be self-explanatory, but here are some noteworthy items:

- All Platform Library functions are prefixed with "`platform`".
- `platform_initialize` sets up the board and processor for the Platform Library. You need to call this before using any other Platform Library function. See **platform_initialize**.
- `platform_write_led(`*x*`, `*y*`)` writes *x* to LED *y*. See **platform_write_led**.
- `platform_spin_delay_ms(`*x*`)` delays execution for *x* milliseconds by busy-waiting in a loop. See **platform_spin_delay_ms**.
- Which LED blinks on your target board depends upon the target board, obviously—consult the documentation for the Platform Library on your target board for details of LED numbering.

By the way...

This does the job, but isn't the kindest way to blink a LED. Because this example uses `platform_spin_delay_ms` to pause between changing the LED state, the processor is active all the time, burning cycles, waiting for the right moment to continue. There is a better way…

You can use `ctl_delay`, rather than `platform_spin_delay_ms`, to delay the user task and let other tasks run. If you do this, you are being much kinder to the tasking system, and in this case the processor is put to sleep whilst waiting.

Blinking all LEDs

More LEDs!

Having mastered a single LED, let's progress to multiple LEDs:

```
// Blink the all platform LEDs in unison.

#include "libplatform/platform.h"

static void
write_all_leds(int state)
{
  int index;

  // Iterate over all platform LEDs.
  for (index = 0; index < PLATFORM_LED_COUNT; ++index)
    platform_write_led(index, state);
}

void
main(void)
{
  // Initialize platform.
  platform_initialize();

  // Blink all LEDs forever.
  for (;;)
    {
      write_all_leds(1);            // All LEDs on
      platform_spin_delay_ms(500);  // Wait
      write_all_leds(0);            // All LEDs off
      platform_spin_delay_ms(500);  // Wait
    }
}
```

This example is only slightly more complex than before. The function `write_all_leds` iterates over all LEDs that the platform provides and sets them all to the same state.

The noteworthy item here is that `PLATFORM_LED_COUNT` is a count of the number of *user-controllable* LEDs that the target platform offers. There may well be more LEDs on the target board, but they usually indicate healthy power supplies or USB status and so on, and are not programmable.

When you run this, all LEDs on the target board flash in unison.

Note

`PLATFORM_LED_COUNT` expands to a numeric constant that enables static allocation of arrays, for example:

```
static float led_duty_cycle[PLATFORM_LED_COUNT];
```

Independence for LEDs

Rather than blink all LEDs in unison, it's visually appealing to pulse them, in turn, quickly:

```
// Chase LEDs around the board.

#include "libplatform/platform.h"

int
main(void)
{
  int i;

  // Initialize platform.
  platform_initialize();

  // All LEDs off.
  for (i = 0; i < PLATFORM_LED_COUNT; ++i)
    platform_write_led(i, 0);

  // Pulse all LEDs, one at a time, forever.
  for (;;)
    {
      for (i = 0; i < PLATFORM_LED_COUNT; ++i)
        {
          platform_write_led(i, 1);
          platform_spin_delay_ms(10);
          platform_write_led(i, 0);
          platform_spin_delay_ms(200);
        }
    }

  // If we ever get out of here...
  return 0;
}
```

There are other things you can do with multiple LEDs, such as a classic KITT or Cylon animation. These things, however, are more impressive when you have dedicated LED hardware to control, rather than a limited number of miniature indicator LEDs on an evaluation board.

<platform.h>

Overview

This is the primary header file for the Platform Library.

For information on the use of this API, see CrossWorks Platform Library.

API Summary

General	
platform_cpu_name	Platform CPU name
platform_initialize	Initialize Platform Library
platform_name	Platform name
Pins	
PLATFORM_PIN_CLAIM_t	Pin claim requirements
PLATFORM_PIN_CONFIGURATION_t	Pin configuration request structure
PLATFORM_PIN_CONNECTION_t	Pin connection
PLATFORM_PIN_FUNCTION_t	Pin function requirements
platform_claim_multi_pin	Claim multiple platform pins
platform_claim_pin	Claim platform pin
platform_claim_pin_configuration	Claim a configuration of platform pins
platform_lock_pin	Lock pin
platform_lock_pin_configuration	Lock a pin configuration
platform_pin_catalog	Pin catalog
platform_pin_catalog_count	Number of entries in pin catalog
platform_pin_connection_name	Get connection name for a pin
platform_pin_function	Registered platform pin function
platform_pin_signal_name	Get signal name for a pin
platform_release_pin	Release pin
I/O	
platform_read_analog_pin	Read analog input
platform_read_digital_pin	Read digital input
platform_write_analog_pin	Write analog output
platform_write_digital_pin	Write digital output
Buttons	

PLATFORM_BUTTON_ATTRIBUTE_t	Button attributes
PLATFORM_BUTTON_CONFIGURATION_t	Button configuration
platform_button_catalog	Get platform button configuration
platform_button_name	Get platform button name
platform_read_button	Read from button
LEDs	
PLATFORM_LED_ATTRIBUTE_t	LED attributes
PLATFORM_LED_CONFIGURATION_t	LED configuration
platform_led_catalog	Get platform LED configuration
platform_led_name	Get platform LED name
platform_write_led	Write to LED
Configuration	
PLATFORM_PIN_DIRECTION_t	Pin I/O direction
PLATFORM_PIN_FEATURE_t	Pin features
PLATFORM_PIN_MODE_t	Pin drive mode requirements
platform_digital_pin_direction	Get I/O direction
platform_digital_pin_drive_strength	Read pin drive strength mode
platform_digital_pin_features	Read digital pin features
platform_digital_pin_mode	Read digital pin mode
platform_digital_pin_speed	Read pin speed
platform_set_digital_pin_direction	Set direction for a single digital I/O
platform_set_digital_pin_drive_strength	Set drive strength for a single digital I/O
platform_set_digital_pin_features	Write features for a single digital I/O
platform_set_digital_pin_mode	Set mode for a single digital I/O
platform_set_digital_pin_speed	Set speed for a single digital I/O
platform_set_multi_digital_pin_drive_strength	Set drive strength for multiple digital I/Os
platform_set_multi_digital_pin_mode	Set mode for multiple digital I/Os
platform_set_multi_digital_pin_speed	Set speed for multiple digital I/Os
Time	
platform_cpu_core_frequency	Get CPU core frequency
platform_cpu_tick	Get CPU tick
platform_cpu_tick_frequency	Get CPU tick frequency
platform_spin_delay_cycles	Delay a number of CPU cycles
platform_spin_delay_ms	Delay a number of milliseconds
platform_spin_delay_us	Delay a number of microseconds

Hooks	
PLATFORM_EDGE_t	Signal edge require to trigger pin hook
PLATFORM_HOOK_t	Platform hook context
platform_hook_background	Hook function to background list
platform_hook_button_press	Hook a button press
platform_hook_pin_edge	Hook function to a pin edge
platform_hook_timer	Hook function to a repetitive timer
platform_unhook_background	Unhook function from background list
platform_unhook_timer	Unhook function from timer
I2C	
platform_configure_i2c_bus	Configure I2C bus
platform_i2c_bus	Get I2C bus interface
platform_i2c_bus_pins	Get pins for I2C bus
SPI	
platform_configure_i2c_bus_ex	Configure I2C bus (extended)
platform_configure_spi_bus	Configure SPI bus
platform_configure_spi_bus_ex	Configure SPI bus (extended)
platform_spi_bus	Get SPI bus interface
platform_spi_bus_pins	Get pins for SPI bus
UART	
platform_configure_uart	Configure UART
platform_uart	Get UART interface
UEXT	
PLATFORM_UEXT_CONFIGURATION_t	UEXT configuration descriptor
platform_uext_configuration	Get UEXT configuration descriptor
Reset	
PLATFORM_RESET_CAUSE_t	Causes of microcontroller reset
platform_reboot	Reboot platform
platform_reset_cause	Read microcontroller reset cause
Watchdog	
platform_watchdog_enable	Enable watchdog
platform_watchdog_remaining	Inquire remaining watchdog time
platform_watchdog_service	Service watchdog
platform_watchdog_set_period	Set watchdog timeout period

PLATFORM_BUTTON_ATTRIBUTE_t

Synopsis

```
typedef enum {
  PLATFORM_BUTTON_STANDARD,
  PLATFORM_BUTTON_UP,
  PLATFORM_BUTTON_DOWN,
  PLATFORM_BUTTON_LEFT,
  PLATFORM_BUTTON_RIGHT,
  PLATFORM_BUTTON_CENTER,
  PLATFORM_BUTTON_MASK_MASK,
  PLATFORM_BUTTON_POSITIVE_LOGIC,
  PLATFORM_BUTTON_NEGATIVE_LOGIC
} PLATFORM_BUTTON_ATTRIBUTE_t;
```

Description

PLATFORM_BUTTON_ATTRIBUTE_t describes the attributes of a push button.

The attributes are a combination of button logic and now the button is sensed.

Buttons that are part of a joystick arrangement have `PLATFORM_BUTTON_JOYSTICK` set along with one of the up, down, left, right, and center attributes.

PLATFORM_BUTTON_STANDARD

The button is a standard momentary push button.

PLATFORM_BUTTON_UP

The button indicates Up direction.

PLATFORM_BUTTON_DOWN

The button indicates Down direction.

PLATFORM_BUTTON_LEFT

The button indicates Left direction.

PLATFORM_BUTTON_RIGHT

The button indicates Right direction.

PLATFORM_BUTTON_CENTER

The button indicates a joystick center-push "select".

PLATFORM_BUTTON_MASK

The mask to isolate the button type above.

PLATFORM_BUTTON_POSITIVE_LOGIC

When set indicates that the button uses positive logic: reading a one from the GPIO indicates the button is pressed, and reading a zero indicates it is released. When clear, indicates the button uses negative logic.

PLATFORM_BUTTON_NEGATIVE_LOGIC

A documentation convenience when constructing button attributes. Indicates that the button uses negative logic: reading a zero from the GPIO indicates the button is pressed, reading a one indicates the button is released.

PLATFORM_BUTTON_CONFIGURATION_t

Synopsis

```
typedef struct {
  unsigned char pin;
  unsigned char attributes;
  unsigned char mode;
  const char *name;
} PLATFORM_BUTTON_CONFIGURATION_t;
```

Description

PLATFORM_BUTTON_CONFIGURATION_t describes the features of a button connected to a GPIO.

`pin`

The digital pin that senses the button. See **PLATFORM_PIN_CONNECTION_t**.

`attributes`

The attributes of the button. See **PLATFORM_BUTTON_ATTRIBUTE_t**.

`mode`

The mode to set the pin, should it be connected by a GPIO. See **PLATFORM_PIN_MODE_t**. Some boards rely on integrated pull-up or pull-down resistors for buttons rather than using an external resistor. You can specify the pull-ups or pull-downs by setting this member appropriately.

`name`

The name of the button. You can set this to the name of the button on the silkscreen or whatever is visible to the user for identification. If **name** is zero, the button's name is derived from the GPIO connection name for **pin** or the pad name for **pin**. See **platform_button_name**.

See Also

platform_button_catalog

PLATFORM_EDGE_t

Synopsis

```
typedef enum {
  PLATFORM_EDGE_FALLING,
  PLATFORM_EDGE_RISING,
  PLATFORM_EDGE_EITHER
} PLATFORM_EDGE_t;
```

Description

PLATFORM_EDGE_t describes the required edge to trigger a pin hook.

Note

`PLATFORM_EDGE_EITHER` is the inclusive-or of `PLATFORM_EDGE_RISING` and `PLATFORM_EDGE_FALLING`.

See Also

platform_hook_pin_edge

PLATFORM_HOOK_t

Synopsis

```
typedef struct {
  void (*fn)(void *);
  void *arg;
  int __internal;
  PLATFORM_HOOK_s *__next;
} PLATFORM_HOOK_t;
```

Description

PLATFORM_HOOK_t describes a *hook* function that is typically executed asynchronously. The Platform Library provides a number of ways for hooks to be run, using high-frequency and low-frequency timers, and on the transitions of platform pins.

`fn`

Method to execute when the hook fires.

`arg`

Argument to pass to `fn` when the hook fires.

`__internal`

Private member for use by the Platform Library.

`__next`

Private member that points to the next hook function an a hook list. Do not assume anything about the list that this member points to.

Note

You only need to initialize `fn` and `arg` in the hook structure when passing the hook to a registration routine—the Platform Library takes care of managing `__internal` and `__next`.

See Also

platform_hook_pin_edge, **platform_hook_background**, **platform_hook_timer**

PLATFORM_LED_ATTRIBUTE_t

Synopsis

```
typedef enum {
  PLATFORM_LED_UNKNOWN,
  PLATFORM_LED_RED,
  PLATFORM_LED_GREEN,
  PLATFORM_LED_BLUE,
  PLATFORM_LED_YELLOW,
  PLATFORM_LED_ORANGE,
  PLATFORM_LED_WHITE,
  PLATFORM_LED_IR,
  PLATFORM_LED_COLOR_MASK,
  PLATFORM_LED_TRICOLOR,
  PLATFORM_LED_POSITIVE_LOGIC,
  PLATFORM_LED_NEGATIVE_LOGIC
} PLATFORM_LED_ATTRIBUTE_t;
```

Description

PLATFORM_LED_ATTRIBUTE_t describes the attributes of a LED.

The attributes are a combination of LED color and how the LED is driven.

LEDs that are part of a tricolor arrangement have `PLATFORM_LED_TRICOLOR` set along with one of the red, green, and blue colors.

`PLATFORM_LED_UNKNOWN`
The LED color is unknown or varies between boards.

`PLATFORM_LED_RED`
The LED is red.

`PLATFORM_LED_GREEN`
The LED is green.

`PLATFORM_LED_BLUE`
The LED is blue.

`PLATFORM_LED_YELLOW`
The LED is yellow.

`PLATFORM_LED_ORANGE`
The LED is orange.

`PLATFORM_LED_WHITE`
The LED is white.

`PLATFORM_LED_IR`
The LED emits infrared light.

`PLATFORM_LED_COLOR_MASK`

The mask to isolate the color component of the LED attributes.

`PLATFORM_LED_TRICOLOR`

When set indicates that the LED is part of a tricolor arrangement.

`PLATFORM_LED_POSITIVE_LOGIC`

When set indicates that the LED is driven using positive logic: writing a one to the GPIO will turn the LED on and writing a zero will turn it off. When clear, indicates that the LED is driven using negative logic.

`PLATFORM_LED_NEGATIVE_LOGIC`

A documentation convenience when constructing LED attributes. Indicates that the LED is driven using negative logic: writing a one to the GPIO will turn the LED off and writing a zero will turn it on.

PLATFORM_LED_CONFIGURATION_t

Synopsis

```
typedef struct {
  unsigned char pin;
  unsigned char attributes;
  const char *name;
} PLATFORM_LED_CONFIGURATION_t;
```

Description

PLATFORM_LED_CONFIGURATION_t describes the features of a LED connected to a GPIO.

`pin`

The digital pin that drives the LED. See PLATFORM_PIN_CONNECTION_t.

`attributes`

The attributes of the LED. See PLATFORM_LED_ATTRIBUTE_t.

name

The name of the LED. You can set this to the name of the LED on the silkscreen or whatever is visible to the user for identification. If **name** is zero, the LED's name is derived from the GPIO connection name for **pin** or the pad name for **pin**. See platform_led_name.

See Also

platform_led_catalog

PLATFORM_PIN_CLAIM_t

Synopsis

```
typedef enum {
  PIN_CLAIM_WEAK,
  PIN_CLAIM_SHARED,
  PIN_CLAIM_EXCLUSIVE,
  PIN_CLAIM_FIXED,
  PIN_CLAIM_LOCKED
} PLATFORM_PIN_CLAIM_t;
```

Description

PLATFORM_PIN_CLAIM_t describes the claim that the client wishes to make on a pin when configuring it using **platform_claim_pin** or **platform_claim_pin_configuration**.

`PIN_CLAIM_WEAK`

The application claims this pin for a function but the pin can be reconfigured for another function without first releasing it.

`PIN_CLAIM_SHARED`

The application claims this pin for a function, and claims of the same pin for the same function will be granted. The pin can be released for reuse by **platform_release_pin**.

`PIN_CLAIM_EXCLUSIVE`

The application claims exclusive use of this pin for a function, and claims of the same pin for the same function will be denied. The pin can be released for reuse by **platform_release_pin**.

`PIN_CLAIM_FIXED`

The application claims exclusive use of this pin for a function, claims of the same pin for the same function will be denied. Fixed pins cannot be released.

`PIN_CLAIM_LOCKED`

As `PIN_CLAIM_FIXED` but used internally by the Platform Library to deny configuration of dedicated or non-existent pins. Locked pins cannot be released. If the underlying microcontroller implements pin locks, the Platform Library may take advantage of this and hardware-lock the pin in addition to locking it in software.

See Also

platform_claim_pin, **platform_claim_pin_configuration**

PLATFORM_PIN_CONFIGURATION_t

Synopsis

```
typedef struct {
  unsigned char pin;
  unsigned char function;
} PLATFORM_PIN_CONFIGURATION_t;
```

PLATFORM_PIN_CONNECTION_t

Synopsis

```
typedef enum {
  PLATFORM_NO_CONNECTION,
  PLATFORM_END_OF_LIST
} PLATFORM_PIN_CONNECTION_t;
```

Description

PLATFORM_PIN_CONNECTION_t is an enumeration that describes a pin connection. A pin connection is platform-specific and encodes a port and a pin within that port using a single integer.

There are two distinguished values that the Platform Library uses when accepting or defining a pin:

- `PLATFORM_NO_CONNECTION` in a pin list indicates that there is no direct connection for this pin. For instance, a button that is not directly connected to a GPIO will specify this for the `pin` member for the button in the button catalog: the button can still be read using `platform_read_button`, but the implementation of the Platform Library will not read directly from the pin for that button. This is useful, for instance, when a joystick or buttons are analog-encoded using resistors to change an analog input to indicate which buttons are pressed.
- `PLATFORM_END_OF_LIST` indicates the end of a list. Any API call that requires a list *must* terminate the list with this value. In addition, any lists returned by an API call (for instance, `platform_led_catalog`) will ensure that the list is correctly terminated by the value.

The pins encoded by the Platform Library must lie in the range `0x00` to `0xF0` inclusive, to allow for future expansion of the API. This allows a pin to be encoded in a structure using an `unsigned char`.

In many API prototypes, the pin connection type is `int` rather than `PLATFORM_PIN_CONNECTION_t` for brevity: any argument with type `int` and name `pin` is understood to mean the argument is of type `PLATFORM_PIN_CONNECTION_t`.

For various platform footprints, a common set of symbols are defined which map the microprocessor pin to a platform connector. For instance, As such, on any platform that supports an Arduino footprint, you can use the symbol `ARDUINO_D0` to indicate "the pin connected to Arduino digital header D0".

Arduino

The following constants are supplied by the Platform Library for evaluation boards that support an Arduino footprint:

`ARDUINO_D0` through `ARDUINO_D12`

These constants define the pins connected to the Arduino digital headers.

`ARDUINO_A0` through `ARDUINO_A5`

These constants define the pins connected to the Arduino analog headers.

In addition, some synonyms are provided by the Platform Library:

`ARDUINO_RX` and `ARDUINO_TX`

Equivalent to `ARDUINO_D0` and `ARDUINO_D1`, for UART communication.

`ARDUINO_MOSI`, `ARDUINO_MISO`, and `ARDUINO_SCK`

Equivalent to `ARDUINO_D11`, `ARDUINO_D12`, and `ARDUINO_D13`, for SPI communication.

`ARDUINO_SDA` and `ARDUINO_SCL`

Equivalent to `ARDUINO_A4` and `ARDUINO_A5`, for I2C communication.

Some Arduino-style platforms, such as the Freedom boards, may also support the R3 format with:

`LEONARDO_SCL` and `LEONARDO_SDA`

For boards that route the A4/A5 signals to SDA/SCL on the digital header, `LEONARDO_SCL` will be set to `ARDUINO_A4` and `LEONARDO_SDA` will be set to `ARDUINO_A5`. For boards with A4/A5 independent from SDA/SCL, `LEONARDO_SCL` and `LEONARDO_SDA` will be defines as the appropriate pin connection.

mbed

`MBED_P4` through `MBED_P30`

These constants define the pins connected to the dual-in-line header pins of an mbed socket.

LaunchPad

`LAUNCHPAD_A2` through `LAUNCHPAD_A7`

These constants define the pins connected to the "A" connector of a LaunchPad socket.

`LAUNCHPAD_B2` through `LAUNCHPAD_B7`

These constants define the pins connected to the "B" connector of a LaunchPad socket.

In addition, some synonyms are provided by the Platform Library:

`LAUNCHPAD_AIN`

Equivalent to `LAUNCHPAD_A2`, an analog input.

`LAUNCHPAD_RX` and `LAUNCHPAD_TX`

Equivalent to `LAUNCHPAD_A3` and `LAUNCHPAD_A4`, for UART communication.

`LAUNCHPAD_SDA` and `LAUNCHPAD_SCL`

Equivalent to `LAUNCHPAD_B6` and `LAUNCHPAD_B7`, for I2C communication.

`LAUNCHPAD_INT_0` and `LAUNCHPAD_INT_1`

Equivalent to `LAUNCHPAD_A5` and `LAUNCHPAD_B3`, pins capable of generating external interrupt requests.

LAUNCHPAD_SPI_A_SCK, LAUNCHPAD_SPI_B_SCK, LAUNCHPAD_MOSI, and LAUNCHPAD_MISO

Equivalent to LAUNCHPAD_A6, LAUNCHPAD_A7, LAUNCHPAD_B6, and LAUNCHPAD_B7, for SPI communication.

LAUNCHPAD_TMR_OUT

Equivalent to LAUNCHPAD_B2, for timer event output.

PLATFORM_PIN_DIRECTION_t

Synopsis

```
typedef enum {
  PIN_DIRECTION_OUTPUT,
  PIN_DIRECTION_INPUT
} PLATFORM_PIN_DIRECTION_t;
```

Description

PLATFORM_PIN_DIRECTION_t describes whether a pin will be configured as a digital input or a digital output using **platform_set_digital_pin_direction**.

`PIN_DIRECTION_OUTPUT`

Pin is configured as a digital output.

`PIN_DIRECTION_INPUT`

Pin is configured as a digital input.

See Also

platform_set_digital_pin_direction

PLATFORM_PIN_FEATURE_t

Synopsis

```
typedef enum {
  PIN_FEATURE_SLOW_SLEW_RATE,
  PIN_FEATURE_FAST_SLEW_RATE,
  PIN_FEATURE_DISABLE_GLITCH_FILTER,
  PIN_FEATURE_ENABLE_GLITCH_FILTER
} PLATFORM_PIN_FEATURE_t;
```

Description

PLATFORM_PIN_FEATURE_t describes the features that a pin may support using **platform_set_digital_pin_feature**.

Note

Not all features are implemented on all platforms, and not all combinations of features are possible on all platforms. Individual platforms may well reject a request to configure a pin for a particular feature if the Platform Library can determine that the request cannot be satisfied.

PIN_FEATURE_SLOW_SLEW_RATE

Configure pin for slow slew rate.

PIN_FEATURE_FAST_SLEW_RATE

Configure pin for fast slew rate.

PIN_FEATURE_DISABLE_GLITCH_FILTER

Disable glitch filter.

PIN_FEATURE_ENABLE_GLITCH_FILTER

Enable glitch filter.

See Also

platform_set_digital_pin_feature

PLATFORM_PIN_FUNCTION_t

Synopsis

```
typedef enum {
  PIN_FUNCTION_FLOATING,
  PIN_FUNCTION_MISO,
  PIN_FUNCTION_MOSI,
  PIN_FUNCTION_SCK,
  PIN_FUNCTION_SDA,
  PIN_FUNCTION_SCL,
  PIN_FUNCTION_DIGITAL_INPUT,
  PIN_FUNCTION_DIGITAL_OUTPUT,
  PIN_FUNCTION_ANALOG_INPUT,
  PIN_FUNCTION_ANALOG_OUTPUT,
  PIN_FUNCTION_TX,
  PIN_FUNCTION_RX,
  PIN_FUNCTION_STATUS,
  PIN_FUNCTION_ETHERNET_INPUT,
  PIN_FUNCTION_ETHERNET_OUTPUT,
  PIN_FUNCTION_SDIO,
  PIN_FUNCTION_MEMORY,
  PIN_FUNCTION_LCD
} PLATFORM_PIN_FUNCTION_t;
```

Description

PLATFORM_PIN_FUNCTION_t describes the functions that the client wishes to assign to the pin.

`PIN_FUNCTION_FLOATING`

Unused during configuration. This indicates that the pin is currently unassigned.

`PIN_FUNCTION_MISO`

Configure for SPI MISO.

`PIN_FUNCTION_MOSI`

Configure for SPI MOSI.

`PIN_FUNCTION_SCK`

Configure for SPI SCK.

`PIN_FUNCTION_SDA`

Configure for I2C SDA.

`PIN_FUNCTION_SCL`

Configure for I2C SCK.

`PIN_FUNCTION_DIGITAL_INPUT`

Configure as a digital input.

`PIN_FUNCTION_DIGITAL_OUTPUT`

Configure as a digital output.

`PIN_FUNCTION_ANALOG_INPUT`

Configure as an analog input. This connects the pin to an ADC.

`PIN_FUNCTION_ANALOG_OUTPUT`

Configure as a digital output. This connects the pin to a DAC function or PWM function, depending upon pin capability.

`PIN_FUNCTION_TX`

Configure as an RS232 Tx signal.

`PIN_FUNCTION_RX`

Configure as an RS232 Rx signal.

`PIN_FUNCTION_STATUS`

Special configuration that is meaningful to the platform.

`PIN_FUNCTION_ETHERNET_INPUT`, `PIN_FUNCTION_ETHERNET_OUTPUT`

Configure for dedicated Ethernet function.

`PIN_FUNCTION_MEMORY`

Special configuration that implements an external memory bus.

See Also

platform_claim_pin, **platform_claim_pin_configuration**

PLATFORM_PIN_MODE_t

Synopsis

```
typedef enum {
  PIN_MODE_STANDARD,
  PIN_MODE_OPEN_DRAIN,
  PIN_MODE_PULL_UP,
  PIN_MODE_PULL_DOWN
} PLATFORM_PIN_MODE_t;
```

Description

PLATFORM_PIN_MODE_t describes the functions that select additional options for an digital pin pin using **platform_set_digital_pin_mode**.

Note

Not all modes are implemented on all platforms, and not all combinations of options are possible on all platforms. Individual platforms may well reject a request to configure a pin in a particular mode if the Platform Library can determine that the request cannot be satisfied.

`PIN_MODE_STANDARD`

Pin is a standard push-pull output or floating input.

`PIN_MODE_OPEN_DRAIN`

Pin is configured in open drain mode.

`PIN_MODE_PULL_UP`

Integrated pull-up resistors are enabled.

`PIN_MODE_PULL_DOWN`

Integrated pull-down resistors are enabled.

See Also

platform_set_digital_pin_mode

PLATFORM_RESET_CAUSE_t

Synopsis

```
typedef enum {
  PLATFORM_RESET_POWER_ON,
  PLATFORM_RESET_EXTERNAL,
  PLATFORM_RESET_SOFTWARE,
  PLATFORM_RESET_WATCHDOG,
  PLATFORM_RESET_BROWNOUT,
  PLATFORM_RESET_OSCILLATOR_FAIL
} PLATFORM_RESET_CAUSE_t;
```

Description

PLATFORM_RESET_CAUSE_t enumerates the causes of a microcontroller reset. Note that some platforms may not be able to support reporting all reset causes.

PLATFORM_RESET_POWER_ON

Power-on reset.

PLATFORM_RESET_EXTERNAL

External reset using reset pin.

PLATFORM_RESET_SOFTWARE

Software reset.

PLATFORM_RESET_WATCHDOG

Reset because watchdog expired.

PLATFORM_RESET_BROWNOUT

Reset after brownout.

PLATFORM_RESET_OSCILLATOR_FAIL

Reset after oscillator fail.

See Also

platform_reset_cause

PLATFORM_UEXT_CONFIGURATION_t

Synopsis

```
typedef struct {
  signed char i2c_bus_index;
  signed char spi_bus_index;
  signed char uart_index;
  unsigned char pin3_txd;
  unsigned char pin4_rxd;
  unsigned char pin5_scl;
  unsigned char pin6_sda;
  unsigned char pin7_miso;
  unsigned char pin8_mosi;
  unsigned char pin9_sck;
  unsigned char pin10_ssel;
} PLATFORM_UEXT_CONFIGURATION_t;
```

Description

PLATFORM_UEXT_CONFIGURATION_t describes the connection topology and bus connection for an Olimex UEXT socket.

`i2c_bus_index`

The index of the platform I2C bus that is routed to the SDA and `SCL` pins on the UEXT socket. If this is negative, the UEXT socket does not support I2C communication.

`spi_bus_index`

The index of the platform SPI bus that is routed to the `MOSI`, `MISO`, and SCK pins on the UEXT socket. If this is negative, the UEXT socket does not support SPI communication.

`uart_index`

The index of the platform UART that is routed to the TXD and RXD pins on the UEXT socket. If this is negative, the UEXT socket does not support UART communication.

`pin3_txd`

The platform pin connected to TXD (pin 3) on the UEXT socket. If this is `PLATFORM_PIN_CONNECTION`, the UEXT pin is unconnected.

`pin4_rxd`

The platform pin connected to RXD (pin 4) on the UEXT socket. If this is `PLATFORM_PIN_CONNECTION`, the UEXT pin is unconnected.

`pin5_scl`

The platform pin connected to `SCL` (pin 5) on the UEXT socket. If this is `PLATFORM_PIN_CONNECTION`, the UEXT pin is unconnected.

`pin6_sda`

The platform pin connected to SDA (pin 6) on the UEXT socket. If this is `PLATFORM_PIN_CONNECTION`, the UEXT pin is unconnected.

`pin7_miso`

The platform pin connected to `MISO` (pin 7) on the UEXT socket. If this is `PLATFORM_PIN_CONNECTION`, the UEXT pin is unconnected.

`pin8_mosi`

The platform pin connected to `MOSI` (pin 8) on the UEXT socket. If this is `PLATFORM_PIN_CONNECTION`, the UEXT pin is unconnected.

`pin9_sck`

The platform pin connected to SCK (pin 9) on the UEXT socket. If this is `PLATFORM_PIN_CONNECTION`, the UEXT pin is unconnected.

`pin10_ssel`

The platform pin connected to `SSEL` (pin 10) on the UEXT socket. If this is `PLATFORM_PIN_CONNECTION`, the UEXT pin is unconnected.

See Also

platform_uext_configuration

platform_button_catalog

Synopsis

```
PLATFORM_BUTTON_CONFIGURATION_t *platform_button_catalog(void);
```

Description

platform_button_catalog returns an array of buttons available on the platform. The end of the array is indicated by the `pin` member set to `PLATFORM_END_OF_LIST`.

See Also

PLATFORM_BUTTON_CONFIGURATION_t

platform_button_name

Synopsis

```
char *platform_button_name(int index);
```

Description

platform_button_name returns the preferred name for the button with index **index**. The returned pointer is guaranteed non-zero. The button name is derived as follows:

- If **index** is not a valid button index, the button name is `INVALID`.
- If the button name is non-zero in the button catalog, the button name is the cataloged name.
- If the cataloged signal name for the button's pin is nonzero, the button name is that signal name.
- If the cataloged connection name for the button's pin is nonzero, the button name is that connection name.
- Otherwise the button name is `ANON`.

See Also

PLATFORM_BUTTON_CONFIGURATION_t

platform_claim_multi_pin

Synopsis

```
CTL_STATUS_t platform_claim_multi_pin(const unsigned char *pins,
                                      int function);
```

Description

platform_claim_multi_pin iterates over the list of pins **pins** and attempts to claim each for `function` using **platform_claim_pin**.

The pin list **pins** must be terminated by `PLATFORM_END_OF_LIST`. If any pin cannot be claimed, **platform_claim_multi_pin** immediately returns the status.

Return Value

platform_claim_multi_pin returns a standard status code.

See Also

platform_claim_pin_configuration, platform_release_pin

platform_claim_pin

Synopsis

```
CTL_STATUS_t platform_claim_pin(int pin,
                                int function);
```

Description

platform_claim_pin attempts to claim the pin **pin** for function **function**. The **function** parameter is the inclusive-or of a PLATFORM_PIN_CLAIM_t constant and a PLATFORM_PIN_FUNCTION_t constant.

Return Value

platform_claim_pin returns a standard status code.

See Also

platform_claim_pin_configuration, **platform_release_pin**

platform_claim_pin_configuration

Synopsis

```
CTL_STATUS_t platform_claim_pin_configuration(const PLATFORM_PIN_CONFIGURATION_t *pins);
```

Description

platform_claim_pin_configuration iterates over the list of pins **pins** and attempts to claim each entry's `pin` using its corresponding `mode` using **platform_claim_pin**.

The pin list **pins** must be terminated by `PLATFORM_END_OF_LIST` in the `pin` member of the final **PLATFORM_PIN_CONFIGURATION_t**. If a pin cannot be configured, **platform_claim_pin_configuration** immediately returns the status.

Return Value

platform_claim_pin_configuration returns a standard status code.

See Also

platform_claim_pin, **platform_release_pin**

platform_configure_i2c_bus

Synopsis

```
CTL_STATUS_t platform_configure_i2c_bus(int index);
```

Description

platform_configure_i2c_bus powers-up and initializes the I2C peripheral and configures the appropriate pins for I2C configuration on I2C bus **index**. The SCL and SDA pins are configured using `PIN_CLAIM_SHARED`.

Return Value

platform_configure_i2c_bus returns a standard status code.

See Also

platform_configure_i2c_bus_ex

platform_configure_i2c_bus_ex

Synopsis

```
CTL_STATUS_t platform_configure_i2c_bus_ex(int index,
                                           PLATFORM_PIN_CLAIM_t claim);
```

Description

platform_configure_i2c_bus_ex processes the parameter **index** as `platform_configure_i2c_bus` but configures the I2C bus pins using the claim mode **claim**.

You can use this to claim the pins of an I2C bus and lock them such that they cannot be reconfigured.

See Also

platform_configure_i2c_bus

platform_configure_spi_bus

Synopsis

```
CTL_STATUS_t platform_configure_spi_bus(int index,
                                        int extended_frames);
```

Description

platform_configure_spi_bus powers-up and initializes the SPI peripheral and configures the appropriate pins for SPI configuration on SPI bus **index**. The MISO, MOSI, and SCK pins are configured for shared use of those functions as returned by **plaform_spi_bus_pins**.

platform_configure_spi_bus guarantees to successfully configure devices for 8-bit frame sizes, or multiples thereof. If you know that all devices connected to an SPI bus require frames that are multiples of a byte, you can use **platform_configure_spi_bus** to configure the bus. In this case, it's likely that the Platform Library will use a hardware SPI controller to drive the bus.

If some device on the bus requires non-byte-multiple frames (for example, some SPI LCDs require 9-bit frames), then **platform_configure_spi_bus** will not, in general, guarantee to support those devices. If you require non-byte-multiple frame sizes, set `extended_frames` to a non-zero value and this will force use of a software SPI controller when the hardware controller only supports 8-bit frame sizes.

Platform notes

KL05Z, KL25Z, STM32F1, and STM32F4 SPI controllers support only 8-bit frames in hardware.

For the Arduino Uno, or compatibles using the ATmega328P such as the Arduino Pro and the Olimexino-328, the SS pin is automatically configured for shared digital output along with the hardware functions for MISO, MOSI, and SCK.

The ATmega328P will switch to SPI slave mode if SS is driven low with SS configured as an input, so the Platform Library silently configures SS as a shared output to avoid inadvertently switching to SPI slave mode.

Return Value

platform_configure_spi_bus returns a standard status code.

See Also

platform_configure_spi_bus_ex

platform_configure_spi_bus_ex

Synopsis

```
CTL_STATUS_t platform_configure_spi_bus_ex(int index,
                                           int extended_frames,
                                           PLATFORM_PIN_CLAIM_t claim);
```

Description

platform_configure_spi_bus_ex processes the parameters **index** and **extended_frames** as `platform_configure_spi_bus` but configures the SPI bus pins using the claim mode **claim**.

You can use this to claim the pins of an SPI bus and lock them such that they cannot be reconfigured.

Return Value

platform_configure_spi_bus_ex returns a standard status code.

See Also

platform_configure_spi_bus

platform_configure_uart

Synopsis

```
CTL_STATUS_t platform_configure_uart(int index);
```

Description

platform_configure_uart powers-up and initializes the platform UART with index **index** and configures the appropriate pins for UART communication. The UART pins are configured using `PIN_CLAIM_SHARED`.

Return Value

platform_configure_uart returns a standard status code.

platform_cpu_core_frequency

Synopsis

```
unsigned long platform_cpu_core_frequency(void);
```

Description

platform_cpu_core_frequency returns the CPU core frequency, in Hertz.

platform_cpu_name

Synopsis

```
char *platform_cpu_name(void);
```

Description

platform_cpu_name returns the presentation (human-readable) name of the microprocessor that the platform runs on.

platform_cpu_tick

Synopsis

```
unsigned long platform_cpu_tick(void);
```

Description

platform_cpu_tick returns the current free-running CPU tick. The CPU tick increments at a rate of **platform_cpu_tick_frequency** ticks per second and wraps around from $2^{32}–1$ to zero.

See Also

platform_cpu_tick_frequency

platform_cpu_tick_frequency

Synopsis

```
unsigned long platform_cpu_tick_frequency(void);
```

Description

platform_cpu_tick_frequency returns the frequency at which the CPU tick increments, in Hertz.

See Also

platform_cpu_tick

platform_digital_pin_direction

Synopsis

```
CTL_STATUS_t platform_digital_pin_direction(int pin);
```

Description

platform_digital_pin_direction returns the digital I/O pin direction for pin **pin** configured as a digital I/O.

This function is fast and does no error checking whatsoever: it is the client's responsibility to use this function correctly.

See Also

PLATFORM_PIN_DIRECTION_t, platform_set_digital_pin_direction

platform_digital_pin_drive_strength

Synopsis

```
CTL_STATUS_t platform_digital_pin_drive_strength(int pin);
```

Description

platform_digital_pin_drive_strength returns the configured drive strength, in milliamps, for the pin **pin**.

Return Value

platform_digital_pin_drive_strength returns an extended status code. If the status indicates an error, **pin** does not drive strength configuration.

Platform notes

Not all platform provide programmable drive strength. See **platform_set_digital_pin_drive_strength** for additional platform information.

See Also

platform_set_digital_pin_drive_strength

platform_digital_pin_features

Synopsis

```
CTL_STATUS_t platform_digital_pin_features(int pin);
```

Description

platform_digital_pin_features returns the features for the pin **pin**.

See Also

platform_set_digital_pin_features

platform_digital_pin_mode

Synopsis

```
CTL_STATUS_t platform_digital_pin_mode(int pin);
```

Description

platform_digital_pin_mode returns the configured pin mode for the pin **pin**.

See Also

platform_set_digital_pin_mode

platform_digital_pin_speed

Synopsis

```
CTL_STATUS_t platform_digital_pin_speed(int pin);
```

Description

platform_digital_pin_speed returns the configured pin speed, in kHz, for the pin **pin**.

Return Value

platform_digital_pin_speed returns an extended status code. If the status indicates an error, **pin** does not support speed configuration.

Platform notes

Not all platform provide programmable pin speed. See **platform_set_digital_pin_speed** for additional platform information.

See Also

platform_set_digital_pin_speed

platform_hook_background

Synopsis

```
void platform_hook_background(PLATFORM_HOOK_t *hook);
```

Description

platform_hook_background hooks **hook** to the list of hooks that run in the background, approximately ten times per second. The platform executes the hook in a task context, not an interrupt context.

Hooking onto the background list is a good way to periodically update environmental sensor readings, flush shadowed LCD contents, and anything else that is not time critical.

Thread Safety

platform_hook_background is thread-safe.

See Also

PLATFORM_HOOK_t, platform_unhook_background

platform_hook_button_press

Synopsis

```
CTL_STATUS_t platform_hook_button_press(int index,
                                        PLATFORM_HOOK_t *hook);
```

Description

platform_hook_button_press hooks a press on the button with index **index** using **hook**. **platform_hook_button_press** will hook the correct edge to detect a button press according to the configuration for button **index**.

Return Value

platform_hook_button_press returns a standard status code.

See Also

PLATFORM_BUTTON_CONFIGURATION_t

platform_hook_pin_edge

Synopsis

```
CTL_STATUS_t platform_hook_pin_edge(int pin,
                                    PLATFORM_EDGE_t edge,
                                    PLATFORM_HOOK_t *hook);
```

Description

platform_hook_pin_edge hooks the function **hook** so that it is activated by an edge on pin **pin**. The **edge** parameter requests that the hook be triggered on a rising edge, a falling edge, or either edge.

The activated hook is executed in an *interrupt context*, not a task context, and without any surrounding calls to **ctl_enter_isr** and **ctl_exit_isr**. If your hook requires CTL facilities, you must ensure that you call **ctl_enter_isr** and **ctl_exit_isr** as you would in a standard CTL interrupt handler.

Return Value

platform_hook_pin_edge returns a standard status code. Hooking an interrupt is inherently platform-dependent and may fail for any of the following reasons:

- A hook is already established for the pin. Some platforms may support more than one hook per pin and chain them, whereas others may support exactly one hook per pin. It is typically not possible to establish distinct hooks for rising and falling edges of a pin, but the platform supports a single hook for both edges.
- The platform is out of resources when associating interrupts with pins. For instance, STM32 devices can hook an interrupt to bits 0 through 31 of any port, but cannot simultaneously establish hooks for the same bit on different ports, i.e. you cannot establish hooks for interrupts on both PORTA[4] and PORTC[4] as they share the same internal resource, but you can establish hooks for PORTA[4] and PORTC[5] as they use different pins.
- The port pin cannot generate interrupts.
- The port pin does not support the requested **edge** trigger.

See Also

PLATFORM_HOOK_t, PLATFORM_EDGE_t

platform_hook_timer

Synopsis

```
CTL_STATUS_t platform_hook_timer(PLATFORM_HOOK_t *p,
                                 unsigned frequency);
```

Description

platform_hook_timer hooks **hook** so that it is activated repetitively **frequency** times per second. The platform executes the hook in an *interrupt context*, not a task context.

Hooking onto a timer is a good way to execute code at a reliable fixed frequency to scan LED matrices or switches, for instance.

Return Value

platform_hook_timer returns a standard status code. Hooking a timer is inherently platform-dependent and may fail for any of the following reasons:

- The desired execution frequency cannot be achieved.
- All timer hooks are already established. The platform API guarantees at least one active timer hook.

Thread Safety

platform_hook_timer is thread-safe.

See Also

PLATFORM_HOOK_t, **platform_unhook_timer**

platform_i2c_bus

Synopsis

```
CTL_I2C_BUS_t *platform_i2c_bus(int index);
```

Description

platform_i2c_bus returns the driver for the I2C bus **index**. If **index** does not correspond to a logical platform bus, **platform_i2c_bus** returns zero.

platform_i2c_bus_pins

Synopsis

```
PLATFORM_PIN_CONFIGURATION_t *platform_i2c_bus_pins(int index);
```

Description

platform_i2c_bus_pins returns the pin list required for I2C communication on I2C bus **index**. If **index** does not correspond to a logical platform bus, **platform_i2c_bus_pins** returns zero. The list of pins is terminated by `PLATFORM_END_OF_LIST` in the `pin` member.

See Also

PLATFORM_PIN_CONFIGURATION_t

platform_initialize

Synopsis

```
void platform_initialize(void);
```

Description

platform_initialize initializes the microprocessor and any hardware on the board for use with Platform Library functions.

Behind the scenes, **platform_initialize** starts up the *CrossWorks Tasking Library* and creates a two-task system with a user task and an idle task. The user task is the thread of execution that continues after **platform_initialize** returns, and the idle task runs when there is nothing else to do, and typically puts the processor to sleep so that it doesn't continue to consume energy.

platform_led_catalog

Synopsis

```
PLATFORM_LED_CONFIGURATION_t *platform_led_catalog(void);
```

Description

platform_led_catalog returns an array of LEDs available on the platform. The end of the array is indicated with the `pin` member set to `PLATFORM_END_OF_LIST`.

See Also

PLATFORM_LED_CONFIGURATION_t

platform_led_name

Synopsis

```
char *platform_led_name(int index);
```

Description

platform_led_name returns the preferred name for the LED with index **index**. The returned pointer is guaranteed non-zero. The LED name is derived as follows:

- If **index** is not a valid LED index, the LED name is `INVALID`.
- If the LED name is non-zero in the LED catalog, the LED name is the cataloged name.
- If the cataloged signal name for the LED's pin is nonzero, the LED name is that signal name.
- If the cataloged connection name for the LED's pin is nonzero, the LED name is that connection name.
- Otherwise the LED name is `ANON`.

See Also

PLATFORM_BUTTON_CONFIGURATION_t

platform_lock_pin

Synopsis

```
CTL_STATUS_t platform_lock_pin(int pin);
```

Description

platform_lock_pin attempts to raise the claim on pin pin to `PIN_CLAIM_LOCKED` If the underlying microcontroller implements pin locks, the Platform Library may take advantage of this and hardware-lock the pin in addition to locking it in software.

Return Value

platform_lock_pin returns a standard status code.

See Also

platform_lock_pin

platform_lock_pin_configuration

Synopsis

```
CTL_STATUS_t platform_lock_pin_configuration(const PLATFORM_PIN_CONFIGURATION_t *config);
```

Description

platform_lock_pin_configuration iterates over the list of pins **pins** and attempts to raise the claim on each entry's `pin` to `PIN_CLAIM_LOCKED` If the underlying microcontroller implements pin locks, the Platform Library may take advantage of this and hardware-lock the pin in addition to locking it in software.

The pin list **pins** must be terminated by `PLATFORM_END_OF_LIST` in the `pin` member of the final **PLATFORM_PIN_CONFIGURATION_t**. If a pin cannot be locked, **platform_lock_pin_configuration** immediately returns the status.

Return Value

platform_lock_pin_configuration returns a standard status code.

See Also

platform_lock_pin

platform_name

Synopsis

```
char *platform_name(void);
```

Description

platform_name returns the presentation (human-readable) name of the platform.

platform_pin_catalog

Synopsis

```
unsigned char *platform_pin_catalog(void);
```

Description

platform_pin_catalog returns an array of pins that the Platform Library exposes to the client. The end of the array is indicated by and entry of `PLATFORM_END_OF_LIST`.

This may well not be the entire range of pins supported by the microprocessor, and is typically only populated with pins that should be modified by a client.

See Also

platform_pin_catalog_count

platform_pin_catalog_count

Synopsis

```
unsigned platform_pin_catalog_count(void);
```

Description

platform_pin_catalog_count returns the number of pins the in pin catalog delivered by `platform_pin_catalog`, excluding the terminating `PLATFORM_END_OF_LIST`.

See Also

platform_pin_catalog

platform_pin_connection_name

Synopsis

```
char *platform_pin_connection_name(int pin);
```

Description

platform_pin_connection_name returns the connection name for pin **pin**. The connection name returned is generally the name of the port and associated pin from the microprocessor's user manual. For instance, it could be `P1[14]` or PB5.

Note

The string may be overwritten by a subsequent call to **platform_pin_connection_name**.

Implementation

Special connection names, such as `PLATFORM_NO_CONNECTION`, are handled by **platform_pin_connection_name** and any platform-specific connections are passed to **platform_private_pin_connection_name** to handle.

See Also

PLATFORM_PIN_CONNECTION_t, **platform_pin_signal_name**

platform_pin_function

Synopsis

```
unsigned char platform_pin_function[];
```

Description

platform_pin_function is an array, indexed by platform pin, contains the claim and function for the pin as set by *platform_claim_pin* or *platform_claim_pin_configuration*.

Note

Clients must not write to this array directly: it is managed by the Platform Library to ensure proper operation.

platform_pin_signal_name

Synopsis

```
char *platform_pin_signal_name(int pin);
```

Description

platform_pin_signal_name returns the signal name for pin **pin**. The signal name returned is generally the name from the schematic or, in the case of buttons and LEDs, the name of the button or LED on the silkscreen.

Note

The string may be overwritten by a subsequent call to **platform_pin_signal_name**.

Implementation

Special signal names, such as `PLATFORM_NO_CONNECTION`, are handled by **platform_pin_signal_name** and any platform-specific pins are passed to **platform_private_pin_signal_name** to handle.

See Also

PLATFORM_PIN_CONNECTION_t, **platform_pin_connection_name**

platform_read_analog_pin

Synopsis

```
float platform_read_analog_pin(int pin);
```

Description

platform_read_analog_pin reads the state of a pin that's configured to be an analog input. The value returned is between 0 and 1 for single-ended analog inputs and -1 and +1 for differential inputs.

This function is fast and does no error checking whatsoever: it is the client's responsibility to use this function correctly.

See Also

platform_write_analog_pin

platform_read_button

Synopsis

```
int platform_read_button(int index);
```

Description

platform_read_button reads the platform button with index **index**, returning `1` when the button is pressed and `0` when released.

platform_read_button takes care of initializing the GPIO pin and handing buttons connected with both positive and negative logic. If the button's GPIO cannot be claimed, the button is not and `0`

Return Value

platform_read_button returns 1 when the button is pressed and `0` when the button is released.

Note

A button doesn't need to be directly attached to a GPIO, but this is the typical configuration.

See Also

platform_hook_button_press

platform_read_digital_pin

Synopsis

```
int platform_read_digital_pin(int pin);
```

Description

platform_read_digital_pin reads the state of a pin that's configured to be a digital input. This function is fast and does no error checking whatsoever: it is the client's responsibility to use this function correctly.

Note

On some platforms, it may be possible to read the state of a pin configured as an output, and doing so may deliver the state of the pad *or* the last-written digital output state. Such functionality is not guaranteed or standardized by this API, and none of the examples written by Rowley Associates make use of this. Some processors, for instance, will correctly read the state of the pad for outputs configured as push-pull, but will not do so for outputs configured as open drain.

See Also

platform_write_digital_pin

platform_reboot

Synopsis

```
void platform_reboot(void);
```

Description

platform_reboot resets the microcontroller and starts a cold boot. Note that a reset using **platform_reboot** may be detectable as a *software reset* using **platform_reset_cause** after the microcontroller resets.

See Also

platform_reset_cause

platform_release_pin

Synopsis

```
void platform_release_pin(int pin);
```

Description

platform_release_pin releases a pin for reuse and reconfiguration. Pins that are successfully claimed with `PIN_CLAIM_FIXED` or `PIN_CLAIM_FIXED` are never released.

This function is be useful when a pin needs to be reconfigured on the fly. For instance, some resistive panels read a touch position on one axis using analog input and require a potential difference applied on the perpendicular axis using digital outputs. When reading the touch position on the perpendicular axis, the roles of digital output and analog inputs are switched, requiring a reconfiguration of each of the pins from analog to digital and vice versa.

See Also

platform_claim_pin, platform_claim_pin_configuration

platform_reset_cause

Synopsis

```
unsigned platform_reset_cause(void);
```

Description

platform_reset_cause reads the reason for a microcontroller reset and clears the reset cause flags. The value returned is the inclusive-or of the individual causes in **PLATFORM_RESET_CAUSE_t**.

See Also

PLATFORM_RESET_CAUSE_t

platform_set_digital_pin_direction

Synopsis

```
void platform_set_digital_pin_direction(int pin,
                                        int direction);
```

Description

platform_set_digital_pin_direction sets the pin direction for pin **pin** to **direction** for a pin configured as a digital I/O.

This function is fast and does no error checking whatsoever: it is the client's responsibility to use this function correctly.

Platform notes

Changing pin direction may well change the drive strength, pin speed, and pull-up configuration of the pin to the defaults for that pin.

See Also

PLATFORM_PIN_DIRECTION_t, platform_digital_pin_direction

platform_set_digital_pin_drive_strength

Synopsis

```
CTL_STATUS_t platform_set_digital_pin_drive_strength(int pin,
                                                     int strength);
```

Description

platform_set_digital_pin_drive_strength sets the pin drive strength for digital I/O **pin** to **strength** milliamps.

If a device cannot support the pin drive strength, **platform_set_digital_pin_drive_strength** returns a configuration error.

The pin drive strength supported by various platforms are:

Processor	**Drive Strengths** (mA)
STM32L1	Not configurable
STM32F1	Not configurable
STM32F4	Not configurable
LM3S	2, 4, 8
LPC1700	Not configurable
KL05Z	Not configurable
KL25Z	5, 18 (assumes Vdd >= 2.7 V)

See Also

platform_set_multi_digital_pin_drive_strength, platform_digital_pin_drive_strength

platform_set_digital_pin_features

Synopsis

```
CTL_STATUS_t platform_set_digital_pin_features(int pin,
                                                int features);
```

Description

platform_set_digital_pin_features sets the pin features for digital I/O.

If a device cannot support the pin features **features**, **platform_set_digital_pin_features** returns a configuration error.

See Also

platform_digital_pin_features

platform_set_digital_pin_mode

Synopsis

```
CTL_STATUS_t platform_set_digital_pin_mode(int pin,
                                           int mode);
```

Description

platform_set_digital_pin_mode sets the pin mode for digital I/O. A digital output is configured in push-pull mode, but can optionally be turned into an open drain output. A digital input is configured without any pull ups, but pull-ups or pull-downs can be requested.

If a device cannot support pin mode **mode**, **platform_set_digital_pin_mode** returns a configuration error.

See Also

platform_set_multi_digital_pin_mode, **platform_digital_pin_mode**

platform_set_digital_pin_speed

Synopsis

```
CTL_STATUS_t platform_set_digital_pin_speed(int pin,
                                             int kHz);
```

Description

platform_set_digital_pin_speed sets the pin speed for digital I/O.

If a device cannot support the pin speed, **platform_set_digital_pin_speed** returns a configuration error.

The pin speeds supported by various platforms are:

Processor	**Speeds** (MHz)
STM32L1	0.4, 2, 10, 40
STM32F1	2, 25, 50
STM32F4	2, 25, 50, 100
LM3S	Not configurable
LPC1700	Not configurable
KL05Z	Not configurable
KL25Z	Not configurable

See Also

platform_set_multi_digital_pin_speed, platform_digital_pin_speed

platform_set_multi_digital_pin_drive_strength

Synopsis

```
CTL_STATUS_t platform_set_multi_digital_pin_drive_strength(const unsigned char *pins,
                                                           int strength);
```

Description

platform_set_multi_digital_pin_drive_strength iterates over the list of pins **pins** and sets each listed pin's drive strength to **strength** milliamps using **platform_set_digital_pin_drive_strength**. The pin list **pins** must be terminated by `PLATFORM_END_OF_LIST`. If any pin cannot be configured, **platform_set_multi_digital_pin_drive_strength** immediately returns the status.

See Also

platform_set_digital_pin_drive_strength, platform_set_multi_digital_pin_drive_strength

platform_set_multi_digital_pin_mode

Synopsis

```
CTL_STATUS_t platform_set_multi_digital_pin_mode(const unsigned char *pins,
                                                 int mode);
```

Description

platform_set_multi_digital_pin_mode iterates over the list of pins **pins** and sets each listed pin's mode to **mode** using **platform_set_digital_pin_mode**. The pin list **pins** must be terminated by `PLATFORM_END_OF_LIST`. If any pin cannot be configured, **platform_set_multi_digital_pin_mode** immediately returns the status.

See Also

platform_set_digital_pin_mode, platform_digital_pin_mode

platform_set_multi_digital_pin_speed

Synopsis

```
CTL_STATUS_t platform_set_multi_digital_pin_speed(const unsigned char *pins,
                                                  int mode);
```

Description

platform_set_multi_digital_pin_speed iterates over the list of pins **pins** and sets each listed pin's speed to **speed** using **platform_set_digital_pin_speed**. The pin list **pins** must be terminated by `PLATFORM_END_OF_LIST`. If any pin cannot be configured, **platform_set_multi_digital_pin_speed** immediately returns the status.

See Also

platform_set_digital_pin_speed, platform_digital_pin_speed

platform_spi_bus

Synopsis

```
CTL_SPI_BUS_t *platform_spi_bus(int index);
```

Description

platform_spi_bus returns the driver for the SPI bus **index**. If **index** does not correspond to a logical platform bus, **platform_spi_bus** returns zero.

platform_spi_bus_pins

Synopsis

```
PLATFORM_PIN_CONFIGURATION_t *platform_spi_bus_pins(int index);
```

Description

platform_spi_bus_pins returns the pin list required for SPI communication on SPI bus **index**. If **index** does not correspond to a logical platform bus, **platform_spi_bus_pins** returns zero. The list of pins is terminated by `PLATFORM_END_OF_LIST` in the `pin` member.

See Also

PLATFORM_PIN_CONFIGURATION_t

platform_spin_delay_cycles

Synopsis

```
void platform_spin_delay_cycles(unsigned long cycles);
```

Description

platform_spin_delay_cycles delays execution by busy-waiting on the CPU timer for *cycles* ticks.

See Also

platform_spin_delay_us, platform_spin_delay_ms

platform_spin_delay_ms

Synopsis

```
void platform_spin_delay_ms(unsigned period);
```

Description

platform_spin_delay_ms delays execution by busy-waiting for at least *period* milliseconds.

See Also

platform_spin_delay_cycles, platform_spin_delay_us

platform_spin_delay_us

Synopsis

```
void platform_spin_delay_us(unsigned period);
```

Description

platform_spin_delay_us delays execution by busy-waiting for at least *period* microseconds.

See Also

platform_spin_delay_cycles, platform_spin_delay_ms

platform_uart

Synopsis

```
CTL_UART_t *platform_uart(int index);
```

Description

platform_uart returns the UART driver for the UART **index**. If **index** does not correspond to a logical platform UART, **platform_uart** returns zero.

platform_uext_configuration

Synopsis

```
PLATFORM_UEXT_CONFIGURATION_t *platform_uext_configuration(int index);
```

Description

platform_uext_configuration returns a configuration descriptor for UEXT socket **index**. This function will return a non-zero result for indexes in the range 0 through `PLATFORM_UEXT_COUNT-1` and a zero results for indexes outside this range.

If the platform does not provide a UEXT socket, `PLATFORM_UEXT_COUNT` is set to zero and **platform_uext_configuration** always returns zero.

See Also

PLATFORM_UEXT_CONFIGURATION_t

platform_unhook_background

Synopsis

```
void platform_unhook_background(PLATFORM_HOOK_t *hook);
```

Description

platform_unhook_background unhooks **hook** from the background hook list such that it no longer runs.

Thread Safety

platform_unhook_background is thread-safe.

See Also

platform_hook_background

platform_unhook_timer

Synopsis

```
void platform_unhook_timer(PLATFORM_HOOK_t *p);
```

Description

platform_unhook_timer unhooks **hook** from the timer list such that it no longer runs.

Thread Safety

platform_unhook_timer is thread-safe.

See Also

platform_hook_timer

platform_watchdog_enable

Synopsis

```
void platform_watchdog_enable(void);
```

Description

platform_watchdog_enable enables the watchdog using the timeout period set by **platform_watchdog_set_period**. The watchdog must be serviced by calling **platform_watchdog_service** within the timeout period to prevent the microcontroller from being reset.

You can detect a reset caused by a watchdog timeout by calling **platform_reset_cause**.

See Also

platform_watchdog_set_period, platform_watchdog_service, platform_reset_cause

platform_watchdog_remaining

Synopsis

```
float platform_watchdog_remaining(void);
```

Description

platform_watchdog_remaining returns the time remaining, in seconds, before the watchdog times out.

platform_watchdog_service

Synopsis

```
void platform_watchdog_service(void);
```

Description

platform_watchdog_service resets the watchdog timeout. The watchdog timeout is reset to the period set by **platform_watchdog_set_period**.

platform_watchdog_set_period

Synopsis

```
CTL_STATUS_t platform_watchdog_set_period(float period);
```

Description

platform_watchdog_set_period sets the timeout period to **period** seconds. If the period is too long for the platform to support, **platform_watchdog_set_period** returns an error status.

Return Value

platform_watchdog_set_period returns a standard status code.

platform_write_analog_pin

Synopsis

```
void platform_write_analog_pin(int pin,
                               float value);
```

Description

platform_write_analog_pin writes the state of a pin that's configured to be an analog output. An analog output can be realized either by a digital-to-analog converter (DAC) or by pulse-width modulation (PWM).

The parameter **value** indicates the desired output level, 0 through 1.

This function is fast and does no error checking whatsoever: it is the client's responsibility to use this function correctly.

See Also

platform_read_analog_pin

platform_write_digital_pin

Synopsis

```
void platform_write_digital_pin(int pin,
                                int value);
```

Description

platform_write_digital_pin writes the state of a pin that's configured to be a digital output. This function is fast and does no error checking whatsoever: it is the client's responsibility to use this function correctly.

Note

On some platforms, writing to a pin configured as a digital input may have undesirable effects, such as turning pull-ups on or off. None of the examples written by Rowley Associates will write to a digital output pin in anything other than digital output mode.

See Also

platform_read_digital_pin

platform_write_led

Synopsis

```
void platform_write_led(int index,
                        int state);
```

Description

platform_write_led sets the platform LED with index **index** on or off according to **state**. If **state** is zero, the LED is turned off and if **state** is non-zero, it is turned on.

platform_write_led takes care of initializing the GPIO pin and handing LEDs connected with both positive and negative logic. If the LED's GPIO cannot be claimed, the LED is not driven—this allows shared use where a LED is connected to a GPIO as an indicator. For instance, the BugBlat Cortino has two LEDs, connected to A4/SDA and A5/SCL so, for instance, a client can request an I2C bus on those two pins and attempting to write to the LEDs using **platform_write_led** will be a no-operation as SCL and SDA are claimed for I2C. If, however, the LEDs are written using **platform_write_led**, the pins are claimed as general purpose outputs and trying to establish an I2C bus on them will fail.

Note

A LED doesn't need to be directly attached to a GPIO, but this is the typical configuration.

<platform_graphics.h>

Overview

This is the primary header file for configuring SD/microSD on a platform.

For information on the use of this API, see **CrossWorks Platform Library**.

API Summary

Graphics	
platform_configure_builtin_graphics	Configure built-in graphics display

platform_configure_builtin_graphics

Synopsis

```
CTL_STATUS_t platform_configure_builtin_graphics(void);
```

Description

platform_configure_builtin_graphics configures the platform's built-in graphics display, if there is one. If there is no built-in display available on the platform, **platform_configure_builtin_graphics** returns `CTL_UNSUPPORTED_OPERATION`.

Return Value

platform_configure_builtin_graphics returns a standard status code.

<platform_network.h>

API Summary

Network	
platform_configure_network	Configure the network interface controller

platform_configure_network

Synopsis

```
CTL_STATUS_t platform_configure_network(CTL_NET_INTERFACE_t *self);
```

Description

platform_configure_network configures the platform's primary network interface controller on the interface **self**. The intention of this is for the network controller to be initialized, ready to run the examples.

Return Value

platform_configure_network returns a standard status code.

<platform_sensors.h>

Overview

This is the primary header file for sensors on a platform.

For information on the use of this API, see **CrossWorks Platform Library**.

The design of this API separates out all sensors into classes and each class of sensor has an individual API entry point. We do this, rather than having a general enumeration function, to conserve code and data space in linked applications. If there is a single API entry point to enumerate all sensors, for example, then the API implementation would need to link in drivers for each sensor offered by the platform irrespective of whether the client requires it or not.

API Summary

Motion	
platform_configure_builtin_accelerometer	Configure built-in accelerometer
platform_configure_builtin_gyroscope	Configure built-in gyroscope
Magnetics	
platform_configure_builtin_magnetometer	Configure built-in magnetometer
Environmental	
platform_configure_builtin_humidity_sensor	Configure built-in humidity sensor
platform_configure_builtin_light_sensor	Configure built-in light sensor
platform_configure_builtin_pressure_sensor	Configure built-in pressure sensor
platform_configure_builtin_temperature_sensor	Configure built-in temperature sensor

platform_configure_builtin_accelerometer

Synopsis

```
CTL_ACCELEROMETER_t *platform_configure_builtin_accelerometer(void);
```

Description

platform_configure_builtin_accelerometer configures the platform's built-in accelerometer, if there is one. If there is no built-in accelerometer available, or the resources (SPI bus, I2C bus etc.) are not available to support the accelerometer, **platform_configure_builtin_accelerometer** returns zero.

platform_configure_builtin_gyroscope

Synopsis

```
CTL_GYROSCOPE_t *platform_configure_builtin_gyroscope(void);
```

Description

platform_configure_builtin_gyroscope configures the platform's built-in gyroscope, if there is one. If there is no built-in gyroscope available, or the resources (SPI bus, I2C bus etc.) are not available to support the gyroscope, **platform_configure_builtin_gyroscope** returns zero.

platform_configure_builtin_humidity_sensor

Synopsis

```
CTL_HUMIDITY_SENSOR_t *platform_configure_builtin_humidity_sensor(void);
```

Description

platform_configure_builtin_humidity_sensor configures the platform's built-in humidity sensor, if there is one. If there is no built-in humidity sensor available, or the resources (SPI bus, I2C bus etc.) are not available to support the humidity sensor, **platform_configure_builtin_humidity_sensor** returns zero.

platform_configure_builtin_light_sensor

Synopsis

```
CTL_LIGHT_SENSOR_t *platform_configure_builtin_light_sensor(void);
```

Description

platform_configure_builtin_light_sensor configures the platform's built-in light sensor, if there is one. If there is no built-in light sensor available, or the resources (SPI bus, I2C bus etc.) are not available to support the light sensor, **platform_configure_builtin_light_sensor** returns zero.

platform_configure_builtin_magnetometer

Synopsis

```
CTL_MAGNETOMETER_t *platform_configure_builtin_magnetometer(void);
```

Description

platform_configure_builtin_magnetometer configures the platform's built-in magnetometer, if there is one. If there is no built-in magnetometer available, or the resources (SPI bus, I2C bus etc.) are not available to support the magnetometer, **platform_configure_builtin_magnetometer** returns zero.

platform_configure_builtin_pressure_sensor

Synopsis

```
CTL_PRESSURE_SENSOR_t *platform_configure_builtin_pressure_sensor(void);
```

Description

platform_configure_builtin_pressure_sensor configures the platform's built-in pressure sensor, if there is one. If there is no built-in pressure sensor available, or the resources (SPI bus, I2C bus etc.) are not available to support the pressure sensor, **platform_configure_builtin_pressure_sensor** returns zero.

platform_configure_builtin_temperature_sensor

Synopsis

```
CTL_TEMPERATURE_SENSOR_t *platform_configure_builtin_temperature_sensor(void);
```

Description

platform_configure_builtin_temperature_sensor configures the platform's built-in temperature sensor, if there is one. If there is no built-in temperature sensor available, or the resources (SPI bus, I2C bus etc.) are not available to support the temperature sensor, **platform_configure_builtin_temperature_sensor** returns zero.

<platform_heaps.h>

Overview

This is the primary header file for platform heaps.

For information on the use of this API, see **CrossWorks Platform Library**.

API Summary

Memory	
platform_network_heap	Network heap
platform_system_heap	System heap
Private	
platform_private_init_heaps	Initialize network and system heaps

platform_network_heap

Synopsis

```
CTL_HEAP_t platform_network_heap;
```

Description

platform_network_heap is a heap that is primarily used by the network library to hold TCP segments for transmission by the MAC. If you need to allocate small control structures, you should use the system heap, **platform_system_heap**. TCP segments in the network heap are fleeting, being created, handed to the MAC for transmission, and freed. With a quiescent network, the network heap will most likely be entirely empty and, therefore, not fragmented.

platform_network_heap is initialized by **platform_configure_nic**.

See Also

platform_configure_nic, **platform_system_heap**

platform_private_init_heaps

Synopsis

```
void platform_private_init_heaps(CTL_NET_MEM_DRIVER_t *self,
                                 void *buf,
                                 size_t byte_count);
```

Description

platform_private_init_heaps initializes the system heap and the network heap using the memory pointed to by **buf** of **byte_count** bytes. The example implementation partitions the memory by allocating 3/4 to the network heap and 1/4 for the system heap.

Once partitioned, the network driver **self** is initialized with methods and data to allocate memory from the network heap.

platform_system_heap

Synopsis

```
CTL_HEAP_t platform_system_heap;
```

Description

platform_system_heap is a general system heap that is primarily used by the network library for maintaining non-data control structures for DNS, ARP, and so on. It is separate from the network heap that is used to hold TCP segments for transmission by the MAC.

platform_system_heap is initialized by **platform_configure_nic**.

See Also

platform_configure_nic, **platform_network_heap**

<platform_private.h>

Overview

Private part of the Platform Library for platform implementation.

These functions are not intended for Platform Library API clients to call directly. These functions are intended to be a framework that simplifies implementing the Platform Library for a new target processor or evaluation board.

API Summary

Platform	
platform_private_idle_task_main	Platform idle task body
platform_private_initialize	Initialize private platform
platform_private_start_tasking	Start CTL and platform tasks
Pins	
platform_private_find_pin_connection	Find pin connection by function
platform_private_lock_pin	Lock pin in hardware
platform_private_pin_connection_name	Get connection name for a pin
platform_private_pin_signal_name	Get signal name for a pin
platform_private_release_pin	Release pin
platform_private_test_pin_claim	Test pin lock
LEDs	
platform_private_configure_leds	Configure advertised GPIO-connected LEDs
platform_private_write_led	Write to GPIO-connected LED
Buttons	
platform_private_read_button	Read GPIO-connected button
Hooks	
platform_private_execute_hooks	Execute functions on a hook list
platform_private_hook_single_timer	Hook a single timer
platform_private_start_single_hook_timer	Start a single hook timer
platform_private_stop_single_hook_timer	Stop the single hook timer
platform_private_timer_hooks	Singleton timer hook
platform_private_unhook_single_timer	Unhook a single timer
I2C	
PLATFORM_PRIVATE_I2C_CONFIGURATION_t	I2C bus configuration

PLATFORM_PRIVATE_I2C_METHODS_t	I2C bus methods
platform_private_i2c_bus_configuration	I2C bus array
platform_private_i2c_bus_instance	Platform I2C bus instances
platform_private_software_i2c_methods	Software I2C methods
platform_private_spi_hardware_claim_pins	Utility methods
SPI	
PLATFORM_PRIVATE_SPI_CONFIGURATION_t	SPI bus configuration
PLATFORM_PRIVATE_SPI_METHODS_t	SPI bus methods
platform_private_i2c_hardware_claim_pins	Utility methods
platform_private_software_spi_methods	Software SPI methods
platform_private_spi_bus_configuration	SPI bus array
platform_private_spi_bus_instance	Platform SPI bus instances

PLATFORM_PRIVATE_I2C_CONFIGURATION_t

Synopsis

```
typedef struct {
  int bus_index;
  const PLATFORM_PIN_CONFIGURATION_t *pins;
  const PLATFORM_PRIVATE_I2C_METHODS_t *methods;
} PLATFORM_PRIVATE_I2C_CONFIGURATION_t;
```

Description

PLATFORM_PRIVATE_I2C_CONFIGURATION_t describes the configuration of a Platform I2C bus.

`bus_index`

The *device* I2C bus index to use for the I2C controller. For instance, platform I2C bus with index 0 may well be implemented using the device I2C bus I2C2, in which case `bus_index` will be 2.

`pins`

The pin connections required by the I2C bus.

`methods`

The methods required to implement the I2C bus. For I2C controllers implemented in software, `methods` should be set to `platform_private_software_i2c_methods`.

See Also

platform_private_i2c_bus_configuration, **platform_private_software_i2c_methods**

PLATFORM_PRIVATE_I2C_METHODS_t

Synopsis

```
typedef struct {
  CTL_STATUS_t (*configure_controller)(int);
  CTL_STATUS_t (*claim_pins)
(int , const PLATFORM_PIN_CONFIGURATION_t *, PLATFORM_PIN_CLAIM_t);
  CTL_I2C_BUS_t *(*controller)(int);
} PLATFORM_PRIVATE_I2C_METHODS_t;
```

Description

PLATFORM_PRIVATE_I2C_METHODS_t contains the methods required to configure an I2C bus. The first parameter of each method is the index of the *device* I2C bus to configure rather than the index of the Platform I2C bus. For instance, platform I2C bus with index 0 may well be implemented using the device I2C bus I2C2, in which case the index will be 2.

`configure_controller`

Method to configure the controller for the I2C bus.

`claim_pins`

Method to claim the pins that the I2C controller will use. For software I2C controllers, the pins are configured for digital I/O.

`controller`

Method to return the I2C bus controller.

PLATFORM_PRIVATE_SPI_CONFIGURATION_t

Synopsis

```
typedef struct {
  int bus_index;
  const PLATFORM_PIN_CONFIGURATION_t *pins;
  const PLATFORM_PRIVATE_SPI_METHODS_t *methods[];
} PLATFORM_PRIVATE_SPI_CONFIGURATION_t;
```

Description

PLATFORM_PRIVATE_SPI_CONFIGURATION_t describes the configuration of a Platform SPI bus.

bus_index

The *device* SPI bus index to use for the SPI controller. For instance, platform SPI bus with index 0 may well be implemented using the device SPI bus SPI2, in which case `bus_index` will be 2.

pins

The pin connections required by the SPI bus.

methods

The methods required to implement the SPI bus for byte frames (index 0) and extended frames (index 1). For SPI buses implemented entirely in software, both entries in `methods` should be set to `platform_private_software_spi_methods`. For SPI buses that are implemented entirely in hardware with the capability of extended frames, both entries should be set to the device-specific methods for that controller. For SPI buses that are can implement byte frame in hardware but require extended frames in software, index 0 should be set to the device-specific methods for that controller, and index 1 should be set to `platform_private_software_spi_methods`.

See Also

platform_private_spi_bus_configuration, platform_private_software_spi_methods

PLATFORM_PRIVATE_SPI_METHODS_t

Synopsis

```
typedef struct {
  CTL_STATUS_t (*configure_controller)(int);
  CTL_STATUS_t (*claim_pins)
(int , const PLATFORM_PIN_CONFIGURATION_t *, PLATFORM_PIN_CLAIM_t);
  CTL_SPI_BUS_t *(*controller)(int);
} PLATFORM_PRIVATE_SPI_METHODS_t;
```

Description

PLATFORM_PRIVATE_SPI_METHODS_t contains the methods required to configure an SPI bus. The first parameter of each method is the index of the *device* SPI bus to configure rather than the index of the Platform SPI bus. For instance, platform SPI bus with index 0 may well be implemented using the device SPI bus SPI2, in which case the index will be 2.

`configure_controller`

Method to configure the controller for the SPI bus.

`claim_pins`

Method to claim the pins that the SPI controller will use. For software SPI controllers, the pins are configured for digital I/O.

`controller`

Method to return the SPI bus controller.

platform_private_configure_leds

Synopsis

```
CTL_STATUS_t platform_private_configure_leds(void);
```

Description

platform_private_configure_leds iterates over all LEDs returned by **platform_led_pins** and configures any GPIO-connected LEDs for use.

Return Value

platform_private_configure_leds returns a standard status code.

platform_private_execute_hooks

Synopsis

```
void platform_private_execute_hooks(PLATFORM_HOOK_t *hook);
```

Description

platform_private_execute_hooks executes all functions on the hook list **hook**. Each function is called and passed the `arg` member of its hook context.

platform_private_find_pin_connection

Synopsis

```
CTL_STATUS_t platform_private_find_pin_connection(const PLATFORM_PIN_CONFIGURATION_t *list,
                                                  int function);
```

Description

platform_private_find_pin_connection searches the list of pins in **list** for a match on the function **function**.

Return Value

If a pin with matching function is found in the list, **platform_private_find_pin_connection** returns the `pin` member of the pin configuration structure. If the pin is not found or the list is empty, **platform_private_find_pin_connection** returns `CTL_UNSUPPORTED_OPERATION`.

platform_private_hook_single_timer

Synopsis

```
CTL_STATUS_t platform_private_hook_single_timer(PLATFORM_HOOK_t *p,
                                                unsigned frequency);
```

Description

platform_private_hook_single_timer is a utility function when implementing `platform_hook_timer` on platforms that offer only one timer. **platform_private_hook_single_timer** calls **platform_private_start_single_hook_timer** passing in **frequency** if this is a valid hook request.

Implementation

For platforms that provide a single timer took, **platform_hook_timer** should call **platform_private_hook_single_timer** and provide implementations of **platform_private_start_single_hook_timer** and **platform_private_stop_single_hook_timer** to control the timer interrupt.

See Also

platform_private_unhook_single_timer

platform_private_i2c_bus_configuration

Synopsis

```
PLATFORM_PRIVATE_I2C_CONFIGURATION_t platform_private_i2c_bus_configuration[];
```

Description

platform_private_i2c_bus_configuration array defines the I2C bus configuration when using the Platform I2C framework.

See Also

PLATFORM_PRIVATE_I2C_CONFIGURATION_t, PLATFORM_PRIVATE_I2C_METHODS_t

platform_private_i2c_bus_instance

Synopsis

```
CTL_I2C_BUS_t *platform_private_i2c_bus_instance[];
```

Description

platform_private_i2c_bus_instance contains the recoded instances of platform I2C buses, indexed using the platform I2C bus index. When an I2C bus is correctly configured by **platform_configure_i2c_bus**, the I2C bus instance is written to **platform_private_i2c_bus_instance** so that **platform_i2c_bus** can access and return an appropriate I2C bus.

See Also

platform_i2c_bus, platform_configure_i2c_bus

platform_private_i2c_hardware_claim_pins

Synopsis

```
CTL_STATUS_t platform_private_i2c_hardware_claim_pins(int index,

  const PLATFORM_PIN_CONFIGURATION_t *pins,
                                                      PLATFORM_PIN_CLAIM_t claim);
```

Description

platform_private_i2c_hardware_claim_pins is a utility method that returns the result of passing **pins** to **platform_claim_pin_configuration**. You can use **platform_private_i2c_hardware_claim_pins** as the `pin_claim` method for an SPI bus using a hardware SPI controller.

See Also

PLATFORM_PRIVATE_SPI_METHODS_t

platform_private_idle_task_main

Synopsis

```
void platform_private_idle_task_main(void *param);
```

Description

platform_private_idle_task_main is the prototype for the microcontroller Platform Library to implement. Typically, the main function will be an infinite loop that puts the processor into low-power mode waiting for an interrupt. However, you can customize this, for instance, to illuminate an LED to show when the processor is active.

The standard implementation of this for the platforms that we distribute is to place the processor into low-power mode awaiting and interrupt.

platform_private_initialize

Synopsis

```
void platform_private_initialize(void);
```

Description

platform_private_initialize initializes the private part of the Platform Library. In particular, for release builds is has a 250 ms delay to allow for power supply stabilization and for external devices to become ready—most LCD controllers require a short delay after reset before responding to commands, for instance.

You can customize this delay for your own applications. If your board doesn't start cleanly after reset but does when debugging with CrossWorks, it's likely that you'll need to adjust the 250 ms delay to suit your hardware.

platform_private_lock_pin

Synopsis

```
void platform_private_lock_pin(int pin);
```

Description

platform_private_lock_pin can hardware-lock the pin connection **pin** if the underlying microcontroller implements pin locks.

See Also

platform_lock_pin

platform_private_pin_connection_name

Synopsis

```
char *platform_private_pin_connection_name(int pin);
```

Description

platform_private_pin_connection_name returns the connection name for pin **pin**. The platform-independent code guarantees to call **platform_private_pin_connection_name** with a correct **pin** parameter.

The connection name returned is generally the name from the schematic or, in the case of buttons and LEDs, the name of the button or LED on the silkscreen.

See Also

PLATFORM_PIN_CONNECTION_t, **platform_pin_connection_name**

platform_private_pin_signal_name

Synopsis

```
char *platform_private_pin_signal_name(int pin);
```

Description

platform_private_pin_signal_name returns the signal name for pin **pin**. The platform-independent code guarantees to call **platform_private_pin_signal_name** with a correct **pin** parameter.

The signal name returned is generally the name from the schematic or, in the case of buttons and LEDs, the name of the button or LED on the silkscreen.

See Also

PLATFORM_PIN_CONNECTION_t, **platform_pin_connection_name**

platform_private_read_button

Synopsis

```
int platform_private_read_button(int index);
```

Description

platform_private_read_button writes **state** directly to the GPIO-controlled LED *index*.

If all platform LEDs are controlled using GPIOs that are accessible using **platform_write_digital_pin**, a platform implementation of **platform_write_led** can call **platform_private_write_led** directly.

platform_private_release_pin

Synopsis

```
void platform_private_release_pin(int pin);
```

Description

platform_private_release_pin releases the pin **pin** by changing it back to its reset state, typically a digital input. The platform-independent code guarantees to call **platform_private_release_pin** with a correct **pin** parameter.

platform_private_software_i2c_methods

Synopsis

```
PLATFORM_PRIVATE_I2C_METHODS_t platform_private_software_i2c_methods;
```

Description

platform_private_software_i2c_methods is a set of methods to drive an I2C bus using software.
If you use the Platform I2C framework, you can set the `methods` member of an I2C bus in the `PLATFORM_PRIVATE_I2C_CONFIGURATION_t` to **platform_private_software_i2c_methods** and the Platform I2C framework will supervise the software I2C bus.

Return Value

platform_private_software_i2c_methods returns a standard status code.

platform_private_software_spi_methods

Synopsis

```
PLATFORM_PRIVATE_SPI_METHODS_t platform_private_software_spi_methods;
```

Description

platform_private_software_spi_methods is a set of methods to drive an SPI bus using software.
If you use the Platform SPI framework, you can set the `methods` member of an SPI bus in the `PLATFORM_PRIVATE_SPI_CONFIGURATION_t` to **platform_private_software_spi_methods** and the Platform SPI framework will supervise the software SPI bus.

Return Value

platform_private_software_spi_methods returns a standard status code.

platform_private_spi_bus_configuration

Synopsis

```
PLATFORM_PRIVATE_SPI_CONFIGURATION_t platform_private_spi_bus_configuration[];
```

Description

platform_private_spi_bus_configuration array defines the SPI bus configuration when using the Platform SPI framework.

See Also

PLATFORM_PRIVATE_SPI_CONFIGURATION_t, PLATFORM_PRIVATE_SPI_METHODS_t

platform_private_spi_bus_instance

Synopsis

```
CTL_SPI_BUS_t *platform_private_spi_bus_instance[];
```

Description

platform_private_spi_bus_instance contains the recoded instances of platform SPI buses, indexed using the platform SPI bus index. When an SPI bus is correctly configured by **platform_configure_spi_bus**, the SPI bus instance is written to **platform_private_spi_bus_instance** so that **platform_spi_bus** can access and return an appropriate SPI bus.

See Also

platform_spi_bus, platform_configure_spi_bus

platform_private_spi_hardware_claim_pins

Synopsis

```
CTL_STATUS_t platform_private_spi_hardware_claim_pins(int index,

  const PLATFORM_PIN_CONFIGURATION_t *pins,
                                                     PLATFORM_PIN_CLAIM_t claim);
```

Description

platform_private_spi_hardware_claim_pins is a utility method that returns the result of passing **pins** to **platform_claim_pin_configuration**. You can use **platform_private_spi_hardware_claim_pins** as the `pin_claim` method for an I2C bus using a hardware I2C controller.

See Also

PLATFORM_PRIVATE_I2C_METHODS_t

platform_private_start_single_hook_timer

Synopsis

```
void platform_private_start_single_hook_timer(unsigned frequency);
```

Description

platform_private_start_single_hook_timer is a utility function when implementing `platform_hook_single_timer` on platforms that offer only one timer. **platform_private_start_single_hook_timer** starts the single instance of a hook timer which fires **frequency** times per second.

Implementation

The hook timer, once active, should call **platform_private_execute_hooks** passing in `platform_private_timer_hooks`.

See Also

platform_private_hook_single_timer

platform_private_start_tasking

Synopsis

```
void platform_private_start_tasking(void);
```

Description

platform_private_start_tasking starts the CTL timer to provide CTL time and services, and starts the idle task which has the body function **platform_private_idle_task_main**.

platform_private_stop_single_hook_timer

Synopsis

```
void platform_private_stop_single_hook_timer(void);
```

Description

platform_private_stop_single_hook_timer is a utility function when implementing `platform_unhook_single_timer` on platforms that offer only one timer. **platform_private_stop_single_hook_timer** stops a the previously-started single instance of a hook timer.

See Also

platform_private_unhook_single_timer

platform_private_test_pin_claim

Synopsis

```
CTL_STATUS_t platform_private_test_pin_claim(int pin,
                                             int function);
```

Description

platform_private_test_pin_claim tests whether pin **pin** can be claimed for function **function**. The **function** parameter is the inclusive-or of a `PLATFORM_PIN_CLAIM_t` constant and a `PLATFORM_PIN_FUNCTION_t` constant.

Return Value

platform_private_test_pin_claim returns a standard status code.

See Also

platform_claim_pin

platform_private_timer_hooks

Synopsis

```
PLATFORM_HOOK_t *platform_private_timer_hooks;
```

Description

platform_private_timer_hooks is the list of timer hooks set up by **platform_private_hook_single_timer**. If no hook has been set, **platform_private_timer_hooks** is null.

See Also

platform_private_hook_single_timer

platform_private_unhook_single_timer

Synopsis

```
void platform_private_unhook_single_timer(PLATFORM_HOOK_t *p);
```

Description

platform_private_unhook_single_timer is a utility function when implementing `platform_unhook_timer` on platforms that offer only one timer.

Implementation

For platforms that provide a single timer took, **platform_unhook_timer** should call **platform_private_unhook_single_timer** and provide implementations of **platform_private_start_single_hook_timer** and **platform_private_stop_single_hook_timer** to control the timer interrupt.

See Also

platform_private_unhook_single_timer

platform_private_write_led

Synopsis

```
void platform_private_write_led(int index,
                                int state);
```

Description

platform_private_write_led writes **state** to the GPIO-controlled LED with index *index*. **platform_private_write_led** takes care of inverting **state** for negative-logic LEDs.

If all platform LEDs are controlled using GPIOs that are accessible using **platform_write_digital_pin**, a platform implementation of **platform_write_led** can call **platform_private_write_led** directly.

<platform_stm32f1.h>

Overview

The STM32F1 platform implements the Platform Library private API for a subset of STM32F1 processors. The STM32F1 platform implementation uses the following resources:

- Timer 2, to provide the CPU tick as part of **platform_cpu_tick**.
- Port interrupt handlers for each port, to enable hooks on pins with **platform_hook_pin_edge**.

SPI communication is DMA-driven.

API Summary

Platform	
stm32_platform_initialize	Initialize STM32 platform
Pins	
STM32_PAD	Construct a pin connection
STM32_PIN	Extract pin within port from pin connection
STM32_PORT	Extract port from pin connection
STM32_PORT_BASE	Get CMSIS GPIO structure
STM32_PORT_t	STM32 ports
stm32_release_pin	Release configured pin connection
stm32_set_multi_pin_alternate_function	Configure pin connection list for alternate function
stm32_set_pin_alternate_function	Configure pin connection for alternate function

STM32_PAD

Synopsis

```
#define STM32_PAD(PORT, PIN)  (((PORT)<<4) | (PIN))
```

Description

STM32_PAD creates a PLATFORM_PIN_CONNECTION_t by combining STM32 port, **PORT**, and a pin within that port, **PIN**.

The port and pin are extracted from the connection by **STM32_PORT** and **STM32_PIN**:

- STM32_PORT(STM32_PAD(x, y)) == x
- STM32_PIN(STM32_PAD(x, y)) == y.

See Also

STM32_PORT, STM32_PIN

STM32_PIN

Synopsis

```
#define STM32_PIN(PIN)   ((PIN) & 15)
```

Description

STM32_PIN extracts the STM32 pin within a port from an encoded `PLATFORM_PIN_CONNECTION_t` value.

In other words, `STM32_PIN(STM32_PAD(x, y)) == y`.

See Also

STM32_PAD, STM32_PORT

STM32_PORT

Synopsis

```
#define STM32_PORT(PIN)  ((PIN) >> 4)
```

Description

STM32_PORT extracts the STM32 port (see STM32_PORT_t) from an encoded PLATFORM_PIN_CONNECTION_t value.

In other words, STM32_PORT(STM32_PAD(x, y)) == x.

See Also

STM32_PAD, STM32_PIN

STM32_PORT_BASE

Synopsis

```
#define STM32_PORT_BASE(X) ((GPIO_TypeDef *) (GPIOA_BASE + 0x400 * (X)))
```

Description

STM32_PORT_BASE returns a pointer to the STM32 CMSIS GPIO type for the port **X**.

STM32_PORT_t

Synopsis

```
typedef enum {
  STM32_PORT_A,
  STM32_PORT_B,
  STM32_PORT_C,
  STM32_PORT_D,
  STM32_PORT_E,
  STM32_PORT_F,
  STM32_PORT_G,
  STM32_PORT_H,
  STM32_PORT_I
} STM32_PORT_t;
```

Description

STM32_PORT_t enumerates the STM32 ports for the STM32F1 implementation, by name.

stm32_platform_initialize

Synopsis

```
void stm32_platform_initialize(void);
```

Description

stm32_platform_initialize initializes the base STM32 platform by powering-up GPIO ports A through I and configuring timer 2 to provide a CPU tick counter.

stm32_release_pin

Synopsis

```
void stm32_release_pin(unsigned pin);
```

Description

stm32_release_pin releases the pin connection **pin**. If the pin is configured for PWM output, the PWM channel is freed for reuse.

stm32_set_multi_pin_alternate_function

Synopsis

```
void stm32_set_multi_pin_alternate_function(const unsigned char *pins,
                                            unsigned function);
```

Description

stm32_set_multi_pin_alternate_function configures the list of pin connections **pins** to use the alternative function **function**. The list must be terminated by `PLATFORM_END_OF_LIST`.

stm32_set_pin_alternate_function

Synopsis

```
void stm32_set_pin_alternate_function(unsigned pin,
                                      unsigned function);
```

Description

stm32_set_pin_alternate_function configures the platform connection **pin** to use alternative function **function**.

<platform_stm32f4.h>

Overview

The STM32F4 platform implements the Platform Library private API for a subset of STM32F4 processors. The STM32F4 platform implementation uses the following resources:

- Timer 2, to provide the CPU tick as part of **platform_cpu_tick**.
- Port interrupt handlers for each port, to enable hooks on pins with **platform_hook_pin_edge**.

SPI communication is DMA-driven.

API Summary

Platform	
stm32_platform_initialize	Initialize STM32 platform
Pins	
STM32_PAD	Construct a pin connection
STM32_PIN	Extract pin within port from pin connection
STM32_PORT	Extract port from pin connection
STM32_PORT_BASE	Get CMSIS GPIO structure
STM32_PORT_t	STM32 ports
stm32_set_multi_pin_alternate_function	Configure pin connection list for alternate function
stm32_set_pin_alternate_function	Configure pin connection for alternate function

<platform_lpc1700.h>

Overview

The LPC1700 platform implements the Platform Library private API for a subset of LPC1700 processors. The LPC1700 platform implementation uses the following resources:

- Timer 0, to provide the CPU tick as part of **platform_cpu_tick**.

API Summary

Platform	
lpc1700_platform_initialize	Initialize LPC1700 platform
Pins	
LPC1700_PAD	Construct a pin connection
LPC1700_PIN	Extract pin within port from pin connection
LPC1700_PORT	Extract port from pin connection
LPC1700_PORT_t	LPC1700 ports
Clocking	
LPC1700_PCLK_SOURCE_t	Peripheral clock selection (LPC1700)

LPC1700_PAD

Synopsis

```
#define LPC1700_PAD(PORT, PIN)  (((PORT)<<5) | (PIN))
```

Description

LPC1700_PAD creates a `PLATFORM_PIN_CONNECTION_t` by combining LPC1700 port, **PORT**, and a pin within that port, **PIN**.

The port and pin are extracted from the connection by **LPC1700_PORT** and **LPC1700_PIN**:

- `LPC1700_PORT(LPC1700_PAD(x, y)) == x`
- `LPC1700_PIN(LPC1700_PAD(x, y)) == y`.

See Also

LPC1700_PORT, LPC1700_PIN

LPC1700_PCLK_SOURCE_t

Synopsis

```
typedef enum {
  LPC1700_PCLK_WDT,
  LPC1700_PCLK_TIMER0,
  LPC1700_PCLK_TIMER1,
  LPC1700_PCLK_UART0,
  LPC1700_PCLK_UART1,
  LPC1700_PCLK_RESERVED_0,
  LPC1700_PCLK_PWM1,
  LPC1700_PCLK_I2C0,
  LPC1700_PCLK_SPI,
  LPC1700_PCLK_RESERVED_1,
  LPC1700_PCLK_SSP1,
  LPC1700_PCLK_DAC,
  LPC1700_PCLK_ADC,
  LPC1700_PCLK_CAN1,
  LPC1700_PCLK_CAN2,
  LPC1700_PCLK_ACF,
  LPC1700_PCLK_QEI,
  LPC1700_PCLK_GPIOINT,
  LPC1700_PCLK_PCB,
  LPC1700_PCLK_I2C1,
  LPC1700_PCLK_RESERVED_2,
  LPC1700_PCLK_SSP0,
  LPC1700_PCLK_TIMER2,
  LPC1700_PCLK_TIMER3,
  LPC1700_PCLK_UART2,
  LPC1700_PCLK_UART3,
  LPC1700_PCLK_I2C2,
  LPC1700_PCLK_I2S,
  LPC1700_PCLK_RESERVED_3,
  LPC1700_PCLK_RIT,
  LPC1700_PCLK_SYSCON,
  LPC1700_PCLK_MC
} LPC1700_PCLK_SOURCE_t;
```

LPC1700_PIN

Synopsis

```
#define LPC1700_PIN(X)   ((X) & 31)
```

Description

LPC1700_PIN extracts the LPC1700 pin within a port from an encoded `PLATFORM_PIN_CONNECTION_t` value.

In other words, `LPC1700_PIN(LPC1700_PAD(x, y)) == y`.

See Also

LPC1700_PAD, LPC1700_PORT

LPC1700_PORT

Synopsis

```
#define LPC1700_PORT(X)  ((X) >> 5)
```

Description

LPC1700_PORT extracts the LPC1700 port (see LPC1700_PORT_t) from an encoded `PLATFORM_PIN_CONNECTION_t` value.

In other words, `LPC1700_PORT(LPC1700_PAD(x, y)) == x`.

See Also

LPC1700_PAD, LPC1700_PIN

LPC1700_PORT_t

Synopsis

```
typedef enum {
  LPC1700_PORT_0,
  LPC1700_PORT_1,
  LPC1700_PORT_2,
  LPC1700_PORT_3,
  LPC1700_PORT_4
} LPC1700_PORT_t;
```

Description

LPC1700_PORT_t enumerates the LPC1700 ports for the platform implementation, by name.

lpc1700_platform_initialize

Synopsis

```
void lpc1700_platform_initialize(void);
```

Description

lpc1700_platform_initialize initializes the base LPC1700 platform and configures timer 0 to provide a CPU tick counter.

SolderCore

SolderCore Platform

This is the Platform Library implementation for the SolderCore.

http://www.soldercore.com/

Mass Storage

Examples use the built-in microSD socket.

Networking

Examples use the built-in Ethernet port.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` and selecting an appropriate graphics adapter.

I2C

The platform I2C bus routing is:

- *bus #0*: Arduino header.
- *bus #1*: Secondary I2C connector.

SPI

The platform SPI bus routing is:

- *bus #0*: Arduino header.
- *bus #1*: Internal bus to the microSD socket and the two SPI memory sites on the underside of the PCB.

API

```
/* Copyright (c) 2004-2013 Rowley Associates Limited.
*/

#ifndef __SOLDERCORE_PLATFORM_H
#define __SOLDERCORE_PLATFORM_H

#include "libplatform/platform.h"
#include "libplatform/platform_lm3s_gpio.h"


// ===========================================
// Arduino footprint to port pin mapping
// ===========================================

#define ARDUINO_D0  LM3S_PORT_PIN(LM3S_PORT_D, 0)  // can be PWM0
  U1RX          Analog.15
```

```
#define ARDUINO_D1  LM3S_PORT_PIN(LM3S_PORT_E, 4)  //
            Analog.3
#define ARDUINO_D2  LM3S_PORT_PIN(LM3S_PORT_E, 5)  //
            Analog.2
#define ARDUINO_D3  LM3S_PORT_PIN(LM3S_PORT_E, 6)  // can be PWM4           Arduino PWM
            Analog.1
#define ARDUINO_D4  LM3S_PORT_PIN(LM3S_PORT_G, 1)  // can be PWM5* or PWM1
#define ARDUINO_D5  LM3S_PORT_PIN(LM3S_PORT_G, 0)  // can be PWM0* or PWM4  Arduino PWM
#define ARDUINO_D6  LM3S_PORT_PIN(LM3S_PORT_D, 1)  // can be PWM1           Arduino PWM
  U1TX          Analog.14
#define ARDUINO_D7  LM3S_PORT_PIN(LM3S_PORT_F, 1)  // can be PWM1

#define ARDUINO_D8  LM3S_PORT_PIN(LM3S_PORT_C, 7)  //
  U1TX
#define ARDUINO_D9  LM3S_PORT_PIN(LM3S_PORT_C, 4)  // can be PWM6
#define ARDUINO_D10 LM3S_PORT_PIN(LM3S_PORT_D, 2)  // can be PWM2           Arduino PWM
  U1RX          Analog.13
#define ARDUINO_D11 LM3S_PORT_PIN(LM3S_PORT_A, 5)  // can be PWM7           Arduino PWM
     MOSI
#define ARDUINO_D12 LM3S_PORT_PIN(LM3S_PORT_A, 4)  // can be PWM6
     MISO
#define ARDUINO_D13 LM3S_PORT_PIN(LM3S_PORT_A, 2)  // can be PWM4
     SCK

#define ARDUINO_A0  LM3S_PORT_PIN(LM3S_PORT_D, 7)  //
            Analog.4
#define ARDUINO_A1  LM3S_PORT_PIN(LM3S_PORT_D, 6)  //
            Analog.5
#define ARDUINO_A2  LM3S_PORT_PIN(LM3S_PORT_D, 5)  //
            Analog.6
#define ARDUINO_A3  LM3S_PORT_PIN(LM3S_PORT_D, 4)  //
            Analog.7

// Mapping of A4 and A5 as digital, typically I2C
#define ARDUINO_A4  LM3S_PORT_PIN(LM3S_PORT_B, 3)   // Configured by solder jumper PB3
#define ARDUINO_A5  LM3S_PORT_PIN(LM3S_PORT_B, 2)   // Configured by solder jumper PB2

// Mapping of A4 and A5 as analog
#define ARDUINO_A4_ANALOG          LM3S_PORT_PIN(LM3S_PORT_E, 3)
  // Configured by solder jumper PB3
#define ARDUINO_A5_ANALOG          LM3S_PORT_PIN(LM3S_PORT_E, 2)
  // Configured by solder jumper PB2

// SD connections
#define SOLDERCORE_SD_SCK          LM3S_PORT_PIN(LM3S_PORT_H, 4)
#define SOLDERCORE_SD_MISO         LM3S_PORT_PIN(LM3S_PORT_F, 4)
#define SOLDERCORE_SD_MOSI         LM3S_PORT_PIN(LM3S_PORT_F, 5)

// I2C header
#define SOLDERCORE_I2C1_SCL        LM3S_PORT_PIN(LM3S_PORT_J, 0)
#define SOLDERCORE_I2C1_SDA        LM3S_PORT_PIN(LM3S_PORT_J, 1)


// ===========================================
// Internal device to port pin mapping
// ===========================================

// LEDs.
#define SOLDERCORE_USER_LED        LM3S_PORT_PIN(LM3S_PORT_C, 5)
#define SOLDERCORE_MICROSD_LED     LM3S_PORT_PIN(LM3S_PORT_J, 4)
#define SOLDERCORE_RUN_LED         LM3S_PORT_PIN(LM3S_PORT_E, 7)
```

```
// LEDs controlled by the PHY.
#define SOLDERCORE_ETH_LED0        LM3S_PORT_PIN(LM3S_PORT_F, 3)
#define SOLDERCORE_ETH_LED1        LM3S_PORT_PIN(LM3S_PORT_F, 2)

// SPI memory site selects.
#define SOLDERCORE_MEM1_SELECT     LM3S_PORT_PIN(LM3S_PORT_J, 3)
#define SOLDERCORE_MEM2_SELECT     LM3S_PORT_PIN(LM3S_PORT_J, 5)

// microSD socket select.
#define SOLDERCORE_MICROSD_SELECT  LM3S_PORT_PIN(LM3S_PORT_G, 7)

// Platform API LED indexes in LED catalog.
#define SOLDERCORE_USER_LED_INDEX     0
#define SOLDERCORE_RUN_LED_INDEX      1
#define SOLDERCORE_MICROSD_LED_INDEX  2


// ===========================================
// Platform limits
// ===========================================

#define PLATFORM_PIN_COUNT      (9*8)  // 9 ports, 8 bits/port.
#define PLATFORM_LED_COUNT      3
#define PLATFORM_BUTTON_COUNT   0
#define PLATFORM_UART_COUNT     2
#define PLATFORM_SPI_BUS_COUNT  2
#define PLATFORM_I2C_BUS_COUNT  2
#define PLATFORM_UEXT_COUNT     0


#endif
```

Cortino3RE

Cortino3RE Platform

This is the Platform Library implementation for the BugBlat Cortino3RE.

http://www.bugblat.com/products/cor.html

Power

We have found that debugging is unreliable when powering the Cortino3RE from the barrel connector. Always power the board from the USB connector.

Mass Storage

Examples require a SparkFun microSD shield.

Networking

Examples require a NuElectronics ENC28J60 shield.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` and selecting an appropriate graphics adapter.

I2C

The platform I2C bus routing is:

- *bus #0*: Arduino headers.

SPI

The platform SPI bus routing is:

- *bus #0*: Arduino headers.

API

```
***../../samples/BugBlat_Cortino3RE/platform_config.h not found ***
```

FRDM-KL25Z

FRDM-KL25Z Platform

This is the Platform Library implementation for the Freescale FRDM-KL25Z.

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=FRDM-KL25Z

Sensors

The FRDM-KL25Z has a built-in MMA8541Q accelerometer.

Mass Storage

This platform does not have enough RAM to support mass storage.

Networking

This platform does not have enough RAM to support networking.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` and selecting an appropriate graphics adapter.

I2C

The platform I2C bus routing is:

- *bus #0*: Arduino headers.
- *bus #1*: Internal bus to the built-in MMA8451Q accelerometer on the PCB.

SPI

The platform SPI bus routing is:

- *bus #0*: Arduino headers.

API

```
***../../samples/Kinetis/FRDM_KL25Z/platform_config.h not found ***
```

FRDM-KL26Z

FRDM-KL26Z Platform

This is the Platform Library implementation for the Freescale FRDM-KL26Z.

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=FRDM-KL26Z

Sensors

The FRDM-KL26Z has a built-in FXOS8700CQ accelerometer and magnetometer.

Mass Storage

This platform does not have enough RAM to support mass storage.

Networking

This platform does not have enough RAM to support networking.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` and selecting an appropriate graphics adapter.

I2C

The platform I2C bus routing is:

- *bus #0*: Arduino headers.
- *bus #1*: Internal bus to the built-in FXOS8700CQ accelerometer on the PCB.

SPI

The platform SPI bus routing is:

- *bus #0*: Arduino headers.

API

```
***../../samples/Kinetis/FRDM_KL26Z/platform_config.h not found ***
```

FRDM-KL46Z

FRDM-KL46Z Platform

This is the Platform Library implementation for the Freescale FRDM-KL46Z.

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=FRDM-KL46Z

Sensors

The FRDM-KL46Z has a built-in MMA8541Q accelerometer and MAG3110 magnetometer.

Mass Storage

Examples require a SparkFun microSD shield.

Networking

Examples require a NuElectronics ENC28J60 shield.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` and selecting an appropriate graphics adapter.

I2C

The platform I2C bus routing is:

- *bus #0*: Arduino headers.
- *bus #1*: Internal bus to the built-in MMA8451Q accelerometer on the PCB.

SPI

The platform SPI bus routing is:

- *bus #0*: Arduino headers.

API

```
***../../samples/Kinetis/FRDM_KL46Z/platform_config.h not found ***
```

MCBSTM32C

MCBSTM32C Platform

This is the Platform Library implementation for the Keil MCBSTM32C.

http://www.keil.com/mcbstm32c/

Mass Storage

Examples use the built-in microSD socket.

Networking

Examples use the built-in Ethernet port.

Graphics

Examples use the built-in QVGA display. The display uses an Ampire AM320240LDTNQW module and ORISE SPFD5408B LCD driver.

I2C

The platform I2C bus routing is:

- *bus #0*: Internal to accelerometer, touch screen controller, codec, and EEPROM. (Codec and EEPROM are not supported by any high-level platform code).

SPI

The platform SPI bus routing is:

- *bus #0*: microSD socket.

API

```
***../../samples/Keil_MCBSTM32C/platform_config.h not found ***
```

Nucleo-F103RB

Nucleo-F103RB Platform

This is the Platform Library implementation for the STMicroelectronics Nucleo-F103RB.

www.st.com/nucleoF103RB-pr

Mass Storage

Examples require a SparkFun microSD shield.

Networking

Examples require a NuElectronics ENC28J60 shield.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` and selecting an appropriate graphics adapter.

API

```
***../../samples/ST_Nucleo_F103RB/platform_config.h not found ***
```

Nucleo-F401RE

Nucleo-F401RE Platform

This is the Platform Library implementation for the STMicroelectronics Nucleo-F401RE.

www.st.com/nucleoF401RE-pr

Mass Storage

Examples require a SparkFun microSD shield.

Networking

Examples require a NuElectronics ENC28J60 shield.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` and selecting an appropriate graphics adapter.

API

```
***../../samples/ST_Nucleo_F401RE/platform_config.h not found ***
```

Arch Pro

Arch Pro Platform

This is the Platform Library implementation for the Seeed Studio Arch Pro.

http://www.seeedstudio.com/depot/Arch-Pro-p-1677.html

Mass Storage

Examples require a SparkFun microSD shield.

Networking

Examples use the built-in Ethernet port.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` to select an appropriate graphics adapter.

I2C

The platform I2C bus routing is:

- *bus #0*: Arduino headers on A4/A5.
- *bus #1*: Arduino R3 headers on SCL/SDA.
- *bus #2*: Grove I2C socket.

SPI

The platform SPI bus routing is:

- *bus #0*: Arduino headers.
- *bus #1*: 6-pin SPI programming header.

API

```
/* Copyright (c) 2004-2013 Rowley Associates Limited.
*/

#ifndef __SEEED_STUDIO_ARCH_PRO_PLATFORM_H
#define __SEEED_STUDIO_ARCH_PRO_PLATFORM_H

#include "libplatform/platform_lpc1700.h"

// ==============================================
// Arduino footprint pin mapping
// ==============================================

// Digital headers
#define ARDUINO_D0            LPC1700_PAD(LPC1700_PORT_4, 29)
#define ARDUINO_D1            LPC1700_PAD(LPC1700_PORT_4, 28)
#define ARDUINO_D2            LPC1700_PAD(LPC1700_PORT_0, 4)
```

```
#define ARDUINO_D3              LPC1700_PAD(LPC1700_PORT_0, 5)
#define ARDUINO_D4              LPC1700_PAD(LPC1700_PORT_2, 2)
#define ARDUINO_D5              LPC1700_PAD(LPC1700_PORT_2, 3)
#define ARDUINO_D6              LPC1700_PAD(LPC1700_PORT_2, 4)
#define ARDUINO_D7              LPC1700_PAD(LPC1700_PORT_2, 5)
#define ARDUINO_D8              LPC1700_PAD(LPC1700_PORT_0, 0)
#define ARDUINO_D9              LPC1700_PAD(LPC1700_PORT_0, 1)
#define ARDUINO_D10             LPC1700_PAD(LPC1700_PORT_0, 6)
#define ARDUINO_D11             LPC1700_PAD(LPC1700_PORT_0, 9)
#define ARDUINO_D12             LPC1700_PAD(LPC1700_PORT_0, 8)
#define ARDUINO_D13             LPC1700_PAD(LPC1700_PORT_0, 7)

// Analog header
#define ARDUINO_A0              LPC1700_PAD(LPC1700_PORT_0, 23)
#define ARDUINO_A1              LPC1700_PAD(LPC1700_PORT_0, 24)
#define ARDUINO_A2              LPC1700_PAD(LPC1700_PORT_0, 25)
#define ARDUINO_A3              LPC1700_PAD(LPC1700_PORT_0, 26)
#define ARDUINO_A4              LPC1700_PAD(LPC1700_PORT_1, 30)
#define ARDUINO_A5              LPC1700_PAD(LPC1700_PORT_1, 31)

// On digital header
#define ARDUINO_AREF            LPC1700_PAD(LPC1700_PORT_2, 13)

// Additional Uno pins
#define UNO_SCL                 LPC1700_PAD(LPC1700_PORT_0, 28)
#define UNO_SDA                 LPC1700_PAD(LPC1700_PORT_0, 27)
#define UNO_NC                  LPC1700_PAD(LPC1700_PORT_2, 12)
  // on power header; Uno has N/C...

// Arduino SPI programming header
#define ARDUINO_SPI_MOSI        LPC1700_PAD(LPC1700_PORT_0, 18)
#define ARDUINO_SPI_MISO        LPC1700_PAD(LPC1700_PORT_0, 17)
#define ARDUINO_SPI_SCK         LPC1700_PAD(LPC1700_PORT_0, 15)
#define ARDUINO_SPI_SSEL        LPC1700_PAD(LPC1700_PORT_0, 16)

// Grove I2C socket
#define GROVE_I2C_SDA           LPC1700_PAD(LPC1700_PORT_0, 10)
#define GROVE_I2C_SCL           LPC1700_PAD(LPC1700_PORT_0, 11)

// Grove UART socket
#define GROVE_UART_TX           LPC1700_PAD(LPC1700_PORT_2, 0)
#define GROVE_UART_RX           LPC1700_PAD(LPC1700_PORT_2, 1)

// LEDs
#define ARCH_PRO_LED1           LPC1700_PAD(LPC1700_PORT_1, 18)  // green
#define ARCH_PRO_LED2           LPC1700_PAD(LPC1700_PORT_1, 20)  // red
#define ARCH_PRO_LED3           LPC1700_PAD(LPC1700_PORT_1, 21)  // blue
#define ARCH_PRO_LED4           LPC1700_PAD(LPC1700_PORT_1, 23)  // yellow

// LAN
#define ARCH_PRO_LAN_RST          LPC1700_PAD(LPC1700_PORT_1, 28)
#define ARCH_PRO_LAN_OSC_EN       LPC1700_PAD(LPC1700_PORT_1, 27)
#define ARCH_PRO_LAN_LED_SPEED    LPC1700_PAD(LPC1700_PORT_1, 26)
#define ARCH_PRO_LAN_LED_LINK     LPC1700_PAD(LPC1700_PORT_1, 25)
#define ARCH_PRO_LAN_50_MHZ       LPC1700_PAD(LPC1700_PORT_1, 15)
#define ARCH_PRO_LAN_TXD0         LPC1700_PAD(LPC1700_PORT_1, 0)
#define ARCH_PRO_LAN_TXD1         LPC1700_PAD(LPC1700_PORT_1, 1)
#define ARCH_PRO_LAN_TXEN         LPC1700_PAD(LPC1700_PORT_1, 4)
#define ARCH_PRO_LAN_CRS          LPC1700_PAD(LPC1700_PORT_1, 8)
#define ARCH_PRO_LAN_RXD0         LPC1700_PAD(LPC1700_PORT_1, 9)
#define ARCH_PRO_LAN_RXD1         LPC1700_PAD(LPC1700_PORT_1, 10)
#define ARCH_PRO_LAN_RXER         LPC1700_PAD(LPC1700_PORT_1, 14)
```

```
#define ARCH_PRO_LAN_REFCLK     LPC1700_PAD(LPC1700_PORT_1, 15)
#define ARCH_PRO_LAN_MDC        LPC1700_PAD(LPC1700_PORT_1, 16)
#define ARCH_PRO_LAN_MDIO       LPC1700_PAD(LPC1700_PORT_1, 17)


// ==============================================
// Platform limits
// ==============================================

#define PLATFORM_PIN_COUNT      (5*32)
#define PLATFORM_SPI_BUS_COUNT  2
#define PLATFORM_I2C_BUS_COUNT  3
#define PLATFORM_UART_COUNT     0
#define PLATFORM_LED_COUNT      4
#define PLATFORM_BUTTON_COUNT   0
#define PLATFORM_UEXT_COUNT     0


#endif
```

Olimexino-STM32

Olimexino-STM32 Platform

This is the Platform Library implementation for the Olimex Olimexino-STM32.

http://www.olimex.com/Products/Duino/STM32/OLIMEXINO-STM32/

Mass Storage

Examples require an Olimex MOD-SDMMC attached to the UEXT socket.

Note that there is not enough RAM available, as delivered, to run the generic Card Benchmark example.

Networking

Examples require an Olimex MOD-ENC28J60 attached to the UEXT socket.

Note that there is not enough RAM available, as delivered, to run the generic FTP Server and HTTP Server examples.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` and selecting an appropriate graphics adapter.

I2C

The platform I2C bus routing is:

- *bus #0*: Arduino header.
- *bus #1*: UEXT header.

SPI

The platform SPI bus routing is:

- *bus #0*: Arduino headers and UEXT socket.
- *bus #1*: microSD socket on the main PCB.

API

```
***../../samples/Olimex_Olimexino_STM32/platform_config.h not found ***
```

STM32-103STK

STM32-103STK Platform

This is the Platform Library implementation for the Olimex STM32-103STK.

http://www.olimex.com/Products/ARM/ST/STM32-103STK/

Noteworthy

The *Crazyflie Remote* example uses the built-in 3310 LCD and the nRF24L01, together with a Nintendo classic controller and MOD-WII plugged into the UEXT socket, to control a Bitcraze Crazyflie.

Mass Storage

Examples use the built-in SD/MMC socket.

Note that there is not enough RAM available, as delivered, to run the generic Card Benchmark example.

Networking

Examples require an Olimex MOD-ENC28J60 attached to the UEXT socket.

Note that there is not enough RAM available, as delivered, to run the generic FTP Server and HTTP Server examples.

Graphics

Examples use the built-in Nokia 3310 LCD display.

I2C

The platform I2C bus routing is:

- *bus #0*: UEXT socket.
- *bus #1*: LIS3LV02DL on the main PCB.

SPI

The platform SPI bus routing is:

- *bus #0*: UEXT socket.
- *bus #1*: nRF24L01 and 3310 LCD on the main PCB.

API

```
***../../samples/Olimex_STM32_103STK/platform_config.h not found ***
```

STM32-405STK

STM32-405STK Platform

This is the Platform Library implementation for the Olimex STM32-405STK.

http://www.olimex.com/Products/ARM/ST/STM32-405STK/

Noteworthy

The *Crazyflie Remote* example uses the built-in 3310 LCD and the nRF24L01, together with a Nintendo classic controller and MOD-WII plugged into the UEXT socket, to control a Bitcraze Crazyflie.

Mass Storage

Examples use the built-in SD/MMC socket.

Networking

Examples require an Olimex MOD-ENC28J60 attached to the UEXT socket.

Graphics

Examples use the built-in Nokia 3310 LCD display.

I2C

The platform I2C bus routing is:

- *bus #0*: UEXT socket.
- *bus #1*: BMA250 on the main PCB.

SPI

The platform SPI bus routing is:

- *bus #0*: UEXT socket.
- *bus #1*: nRF24L01 and 3310 LCD on the main PCB.

API

```
***../../samples/Olimex_STM32_P405/platform_config.h not found ***
```

STM32-E407

STM32-E407 Platform

This is the Platform Library implementation for the Olimex STM32-E407.

http://www.olimex.com/Products/ARM/ST/STM32-E407/

Mass Storage

Examples use the built-in SD/MMC socket.

Networking

Examples use the built-in Ethernet port.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` and selecting an appropriate graphics adapter.

I2C

The platform I2C bus routing is:

- *bus #0*: Arduino headers.
- *bus #1*: UEXT socket.

SPI

The platform SPI bus routing is:

- *bus #0*: Arduino headers.
- *bus #1*: UEXT socket.

API

```
***../../samples/Olimex_STM32_E407/platform_config.h not found ***
```

STM32-LCD

STM32-LCD Platform

This is the Platform Library implementation for the Olimex STM32-LCD.

http://www.olimex.com/Products/ARM/ST/STM32-LCD/

Mass Storage

Examples require an Olimex MOD-SDMMC attached to the UEXT#1 socket.

Networking

Examples require an Olimex MOD-ENC28J60 attached to the UEXT#2 socket.

Graphics

Examples use the built-in QVGA display.

I2C

The platform I2C bus routing is:

- *bus #0*: UEXT#1 socket.
- *bus #1*: UEXT#2 socket.
- *bus #2*: LIS3LV02DL on the main PCB.

SPI

The platform SPI bus routing is:

- *bus #0*: UEXT#1 socket.
- *bus #1*: UEXT#2 socket.

API

```
***../../samples/Olimex_STM32_LCD/platform_config.h not found ***
```

STM32-P107

STM32-P107 Platform

This is the Platform Library implementation for the Olimex STM32-P107.

https://www.olimex.com/Products/ARM/ST/STM32-P107/

Mass Storage

Examples use the built-in microSD socket.

Networking

Examples use the built-in Ethernet port.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` and selecting an appropriate graphics adapter (for instance, a MOD-LCD3310 or a MOD-LCD6610 attached to a UEXT socket).

I2C

The platform I2C bus routing is:

- *bus #0*: UEXT header.

SPI

The platform SPI bus routing is:

- *bus #0*: UEXT header and microSD socket.

API

```
***../../samples/Olimex_STM32_P107/platform_config.h not found ***
```

STM32-P405

STM32-P405 Platform

This is the Platform Library implementation for the Olimex STM32-P405.

http://www.olimex.com/Products/ARM/ST/STM32-P405/

Mass Storage

Examples use the built-in microSD socket.

Networking

Examples require an Olimex MOD-ENC28J60 attached to the UEXT socket.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` and selecting an appropriate graphics adapter.

I2C

The platform I2C bus routing is:

- *bus #0*: UEXT header.
- *bus #1*: microSD socket.

SPI

The platform SPI bus routing is:

- *bus #0*: UEXT header.
- *bus #1*: microSD socket on the main PCB.

API

```
***../../samples/Olimex_STM32_P405/platform_config.h not found ***
```

STM32-P407

STM32-P407 Platform

This is the Platform Library implementation for the Olimex STM32-P407.

http://www.olimex.com/Products/ARM/ST/STM32-P407/

Notes

The LCD uses the TRST signal, so to debug and program applications that use the LCD, please ensure that you use SWD mode rather than JTAG mode.

Mass Storage

Examples use the built-in SD/MMC socket.

Networking

Examples use the built-in Ethernet port.

Graphics

Examples use the built-in color LCD. *Please see notes above*.

The LCD requires a 9-bit SPI protocol that the STM32F4 does not provide in hardware. Therefore, SPI is emulated in software using the methods in `<software_spi.h>` as part of the CrossWorks Platform API.

I2C

The platform I2C bus routing is:

- *bus #0*: UEXT socket.

SPI

The platform SPI bus routing is:

- *bus #0*: UEXT and microSD sockets.
- *bus #1*: LCD.

API

```
***../../samples/Olimex_STM32_P407/platform_config.h not found ***
```

STM3240G-EVAL

STM3240G-EVAL Platform

This is the Platform Library implementation for the STMicroelectronics STM3240G-EVAL.

http://www.st.com/stm3240g-eval

Sensors

The platform provides a built-in LIS302DL accelerometer.

Mass Storage

Examples use the built-in SD/MMC socket.

Networking

Examples use the built-in Ethernet port.

Graphics

Examples use the built-in color LCD.

I2C

The platform I2C bus routing is:

- *bus #0*: is routed to on-board I2C devices.

SPI

- *bus #0*: is routed MISO/PA6, MOSI/PB5, SCK/PA5.

API

```
***../../samples/STM32/ST_STM3240G_EVAL/platform_config.h not found ***
```

STM32F429II-EXP

STM32F429II-EXP Platform

This is the Platform Library implementation for the IAR STM32F429II-EXP board which comes with the Game Controller Kit:

http://old.iar.com/website1/1.0.1.0/3084/1/?item=prod_prod-s1%2F622

or the Magnetometer Kit:

http://old.iar.com/website1/1.0.1.0/3084/1/?item=prod_prod-s1%2F625

Mass Storage

Examples require an Olimex MOD-SDMMC attached to the UXT#1 socket.

Networking

Examples require an Olimex MOD-ENC28J60 attached to the UXT#2 socket.

Graphics

There are no built-in graphics. You can enable graphics by editing `example_plugin_graphics.c` and selecting an appropriate graphics adapter.

I2C

The platform I2C bus routing is:

- *bus #0*: UXT#1 socket.
- *bus #1*: UXT#2 socket.
- *bus #3*: UXT#3 socket.

SPI

The platform SPI bus routing is:

- *bus #0*: UXT#1 socket.
- *bus #1*: UXT#2 socket.
- *bus #3*: UXT#3 socket.

API

```
***../../samples/IAR_STM32F429II_EXP/platform_config.h not found ***
```

STM32F4-DISCOVERY

STM32F4-DISCOVERY Platform

This is the Platform Library implementation for the STMicroelectronics STM32F4-DISCOVERY.

http://www.st.com/internet/evalboard/product/252419.jsp

This platform provides both mass storage and Ethernet support using the Farnell STM32F4DIS-BB base board:

http://www.element14.com/community/docs/DOC-51084

Mass Storage

Examples use the SD/MMC socket of the STM32F4DIS-BB.

Networking

Examples use the Ethernet port of the STM32F4DIS-BB.

Graphics

Examples use an STM32F4DIS-LCD attached to an STM32F4DIS-BB. Note that the signals PD13, PD14, and PD15 have shared functions: they are routed to the Orange, Red, and Blue LEDS as well as being used as the LCD backlight and data bus. The Green LED is independent of the LCD. Therefore, if you intend to use graphics, make sure you initialize the built-in graphics first, which allocates those signals for the LCD and prevents them from being used for LEDs.

Accelerometer

The STM32F4DISCOVERY is fitted with either an LIS302DL or LIS3DSH accelerometer, depending upon the revision of board you have. Revision A and B boards have the LIS302DL accelerometer and Revision C board have the LIS3DSH accelerometer. The Platform API will sense the type of accelerometer fitted to the board and initialize the correct driver for it.

Note that the accelerometer I2C/SPI interface is selected by PE3 which conflicts with the LCD where PE3 is mapped to the D/C signal. As such, it is impossible to use the accelerometer and the LCD in FSMC mode at the same time.

You can use the accelerometer and LCD sequentially, with restrictions, by configuring the LCD in GPIO mode rather than FSMC mode. In GPIO mode, PE3 can be multiplexed between LCD and accelerometer as long as both are not used from different CTL tasks. This mode also requires that the accelerometer is *the only device on the SPI bus* as it is selected onto the SPI bus when a command is issued to the LCD with D/C=0.

See the function `platform_configure_builtin_graphics` in `stm32f4discover.c` to select between fast FSMC LCD mode without accelerometer and GPIO mode with accelerometer.

API

```
***../../samples/STM32/ST_STM32F4DISCOVERY/platform_config.h not found ***
```

Defender

About Defender

For execution on a SolderCore and a SolderCore Arcade Shield or SolderCore LCD Shield. I've even run this code on a Windows PC using Qt to do GUI heavy lifting.

Background

This code replicates, as accurately as I can make it, a Williams Defender unit. Defender was one of those games that was pretty awesome for its time.

Please don't complain about the coding style, don't ask how it works, just do not bother me. I send this code out into the world to fend for itself and for you to unravel any puzzles you find. You have a SolderCore, you have an Arcade Shield, you have CrossWorks, you have a debugger, so all is not lost.

Core hardware

This code is primarily intended to run on a SolderCore and a SolderCore Arcade Shield or a SolderCore LCD Shield. Best gameplay comes from using the Arcade Shield because the display is bigger and it is considerably faster.

You can also run this code, using a SolderCore Arcade Shield or SolderCore LCD Shield, on:

- an Olimex STM32-E407.
- an Olimex STM32-H407.
- a BugBlat Cortino.
- a Netduino Plus 2.
- an mbed-LPC1768 with an ELMICRO TestBed. (This platform is a bit of a challenge as the LPC1768 RAM is split into several regions, none big enough accommodate a complete frame buffer.)

And you can run this code using the integrated LCD of:

- an Olimex STM32-LCD.

...and perhaps this code is included in CrossStudio as an Easter Egg? :-)

Human interface

The code can either use a Defender Playboard with a standard joystick and arcade buttons or a Nintendo Wii Classic Controller.

You can attach a Classic Controller using a SolderCore SenseCore and WiiChuk adapter. I happen to lay everything out using a 2x2 "flat four" base.

I've also coded up an interface using the Nintendo Wii Nunchuk Controller for the STM32-LCD in case you purchased one of those from Olimex. In this case, use the analog joystick for ship control and C to let off a smart bomb and Z to fire.

I laser-cut my Defender Playboard from 5mm acrylic and fitted some proper arcade buttons and a nice joystick. A good place to purchase these in the UK is **Gremlin Solutions**. You will find that 3mm acrylic is a better fit for the arcade buttons from Gremlin because they have a spring-latch underneath that will not lock on 5mm acrylic—I rebated the cuts for the buttons so mine would.

A warning: I purchased some arcade buttons from SparkFun but these are very deep and have a seriously naff feel. Don't use these arcade buttons, they are truly awful.

What's different

I took a little artistic license with the gameplay:

- The two-player game pits you against an AI-controlled Defender that is on screen, and playing, when you play. Neither Defender can collide with the other Defender, and neither Defender can shoot down or smart bomb the other Defender.
- The game doesn't stop and restart the wave when you die. This is a consequence of two-player mode. Although it would be possible to restart, I quite like it this way.

What's not implemented

Some things have not been implemented yet, and I may well get round to implementing them when I feel the need. Things left out for the moment are:

- Sound effects. I started putting in the hooks for sound effects, but I am no sound designer and haven't found a satisfactory way to get sound effects integrated into a SolderCore setup. Ideas run along the lines of the GinSing, the Fluxamasynth (MIDI), or custom VS1053 firmware (don't want to do this…), and then dry up.
- Baiter hurry-ups.
- Exploding landscape and hyperspace on loss of last human.
- Warping when pressing the hyperspace button.
- High scores. Who needs 'em? :-)
- The AI could be much better. Have a go!

Minimal FTP Server

Minimal FTP Server README

This note is a description of the FTP server example.

Overview

The FTP server example is a minimal implementation of an FTP server. It will serve simultaneous client connections to the server if you configure `MAXIMUM_FTP_CLIENTS` in `example_ftp_server.c`.

Limitations

The server is *minimal* and therefore has certain limitations. If all you wish to do is store and retrieve files from an SD card managed by the Mass Storage Library, this will do that for you. It will not, however, rename files or act as a full FTP server: that is not its purpose.

This server has no compile-time configuration options. If you wish to remove PUT or GET capability, do this by editing the code. You can extend the capabilities of the server, and customize it for your needs, as it is delivered in source form.

Minimal HTTP Server

Minimal HTTP Server README

This note is a description of the HTTP server example.

Overview

The HTTP server example is a minimal implementation of an HTTP server that serves pages from a mounted disk. It will serve simultaneous client connections to the server if you configure `MAXIMUM_HTTP_CLIENTS` in `ctl_http_server.c`.

Limitations

The server is *minimal* and therefore has certain limitations. If all you wish to do is serve files from an SD card managed by the Mass Storage Library, this will do that for you. It will not, however, provide capabilities such as POST, CGI, and so on.

This server has no compile-time configuration options. If you wish to remove capabilities, do this by editing the code. You can extend the capabilities of the server, and customize it for your needs, as it is delivered in source form.

Weather Station LCD1x9

About the example

The application searches an I2C bus to find a light sensor, a pressure sensor, a humidity sensor, and a temperature sensor. After enumerating the available sensors, it will show a carousel of measurements on the LCD.

MOD-LCD1x9 Setup

For boards with a UEXT socket:

- Plug a MOD-LCD1x9 into the first UEXT socket.

On everything else:

- Wire the MOD-LCD1x9 SDA/SCL signals to the primary platform I2C bus.

Arduino-format setup

You can use a SenseCore with a CorePressure module and CoreLight module, for instance. Or you can use a Jee Labs plug shield with some sensors that they offer. Or you can plug both of them in at once, if you really want to.

Adafruit TFT Touch Shield

Adafruit TFT Touch Shield README

This note covers use of the Adafruit TFT Touch Shield on platforms that prove problematic.

Using the Touch Shield on an Arduino (or compatible) and Olimexino-5510

The Adafruit TFT Touch Shield routes the system reset signal from the Arduino header direct to the LCD reset input. This is a **serious problem** if you intend to debug using an Arduino Uno, Olimexino-328P or Olimexino-5510 because the debugger communicates using the RESET signal: on the AVR, it's used for debugWIRE, and on the MSP430 it's used for Spy-Bi-Wire. So, you're out of luck debugging the LCD because it is continually reset when you single step! What you can do is build and program your target board, then simply reset it with the debug cable unplugged.

Using the Touch Shield on the SolderCore or Freedom Board

There are no issues using the shield with a SolderCore or Freedom Board because the debug connection does not use the reset signal.

Alternative products

The ITead Studio ITDB02 range of LCDs has LCD reset mapped to a general purpose I/O pin rather than directly to system reset, so debugging using and Arduino or Olimexino-5510 is possible.

Alternatively, consider a less pin-hungry shield, such as the SolderCore LCD Shield which runs using SPI at up to 40 MHz.