Low-Level I/O

TEP:	117
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x
Author:	Phil Buonadonna, Jonathan Hui

Note

This memo documents a part of TinyOS for the TinyOS Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited. This memo is in full compliance with TEP 1.

Abstract

The memo documents the TinyOS 2.x interfaces used for controlling digital IO functionality and digital interfaces.

1. Introduction

The canonical TinyOS device is likely to have a variety of digital interfaces. These interfaces may be divided into two broad categories. The first are general purpose digital I/O lines (pins) for individual digital signals at physical pins on a chip or platform. The second are digital I/O interfaces that have predefined communication protocol formats. The three buses covered in this document are the Serial Peripheral Interface (SPI), the Inter-Integrated Circuit (I2C) or Two-Wire interface, and the Universal Asynchronous Receiver/Transmitter (UART) interface. While there are likely other bus formats, we presume SPI, I2C, and UART to have the largest coverage.

This memo documents the interfaces used for pins and the three buses.

2. Pins

General Purpose I/O (GPIO) pins are single, versatile digital I/O signals individually controllable on a particular chip or platform. Each GPIO can be placed into either an input mode or an output mode. On some platforms a third 'tri-state' mode may exist, but this functionality is platform specific and will not be covered in this document.

On many platforms, a physical pin may function as either a digital GPIO or another special function I/O such. Examples include ADC I/O or a bus I/O. Interfaces to configure the specific function of a pin are platform specific.

The objective of the interfaces described here is not to attempt to cover all possibilities of GPIO functionality and features, but to distill down to a basis that may be expected on most platforms.

In input mode, we assume the following capabilities:

- The ability to arbitrarily sample the pin
- The ability to generate an interrupt/event from either a rising edge or falling edge digital signal.

In output mode, we assume the following capabilities:

- An I/O may be individually cleared (low) or set (hi)

Platform that provide GPIO capabilities MUST provide the following HIL interfaces:

- GeneralIO
- GpioInterrupt

Platforms MAY provide the following capture interface.

- GpioCapture

2.1 GeneralIO

The GeneralIO HIL interface is the fundamental mechanism for controlling a GPIO pin. The interface provides a mechanism for setting the pin mode and reading/setting the pin value. The toggle function switches the output state to the opposite of what it currently is.

Platforms with GPIO functionality MUST provide this interface. It SHOULD be provided in a component named GeneralIOC, but MAY be provided in other components as needed.

```
interface GeneralIO
{
  async command void set();
  async command void clr();
  async command void toggle();
  async command bool get();
  async command void makeInput();
  async command bool isInput();
  async command void makeOutput();
  async command bool isOutput();
}
```

2.2 GpioInterrupt

The GPIO Interrupt HIL interface provides baseline event control for a GPIO pin. It provides a mechanism to detect a rising edge OR a falling edge. Note that calls to enableRisingEdge and enableFallingEdge are NOT cumulative and only one edge may be detected at a time. There may be other edge events supported by the platform which MAY be exported through a platform specific HAL interface.

```
interface GpioInterrupt {

  async command error_t enableRisingEdge();
  async command bool isRisingEdgeEnabled();
  async command error_t enableFallingEdge();
  async command bool isFallingEdgeEnabled();
  async command error_t disable();
  async event void fired();

}
```

2.3 GpioCapture

The GpioCapture interface provides a means of associating a timestamp with a GPIO event. Platforms MAY provide this interface.

Some platforms may have hardware support for such a feature. Other platforms may emulate this capability using the SoftCaptureC component. The interface makes not declaration of the precision or accuracy of the timestamp with respect to the associated GPIO event.

```
interface GpioCapture {

  async command error_t captureRisingEdge();
  async command bool isRisingEdgeEnabled();
  async command error_t captureFallingEdge();
  async command bool isFallingEdgeEnabled();
  async event void captured(uint16_t time);
  async command void disable();

}
```

3. Buses

Bus operations may be divided into two categories: data and control. The control operations of a particular bus controller are platform specific and not covered here. Instead, we focus on the data interfaces at the HIL level that are expected to be provided.

3.1 Serial Peripheral Interface

The Serial Peripheral Interface (SPI) is part of a larger class of Synchronous Serial Protocols. The term SPI typically refers to the Motorola SPI protocols. Other protocols include the National Semiconductor Microwire, the TI Synchronous Serial Protocol and the Programmable Serial Protocol. The dataside interfaces here were developed for the Motorola SPI format, but may work for others.

Platforms supporting SPI MUST provide these interfaces.

Of note, the interfaces DO NOT define the behavior of any chip select or framing signals. These SHOULD determined by platform specific HAL interfaces and implementations.

The interface is split into a synchronous byte level and an asynchronous packet level interface. The byte level interface is intended for short transactions (3-4 bytes) on the SPI bus.

```
interface SpiByte {
  async command uint8_t write( uint8_t tx );
}
```

The packet level interface is for larger bus transactions. The pointer/length interface permits use of hardware assist such as DMA.

```
interface SpiPacket {
  async command error_t send( uint8_t* txBuf, uint8_t* rxBuf, uint16_t len
  async event void sendDone( uint8_t* txBuf, uint8_t* rxBuf, uint16_t len,
                             error_t error );
}
```

3.2 I2C

The Inter-Integrated Circuit (I2C) interface is another type of digital bus that is often used for chip-to-chip communication. It is also known as a two-wire interface.

The I2CPacket interface provides for asynchronous Master mode communication on an I2C with application framed packets. Individual I2C START-STOP events are controllable which allows the using component to do multiple calls within a single I2C transaction and permits multiple START sequences

Platforms providing I2C capability MUST provide this interface.

```
interface I2CPacket<addr_size> {
  async command error_t read(i2c_flags_t flags, uint16_t addr, uint8_t leng
  async event void readDone(error_t error, uint16_t addr, uint8_t length, u
  async command error_t write(i2c_flags_t flags, uint16_t addr, uint8_t len
  async event void writeDone(error_t error, uint16_t addr, uint8_t length,
}
```

The interface is typed according to the addressing space the underlying implementation supports. Valid type values are below.

```
TI2CExtdAddr - Interfaces uses the extended (10-bit) addressing mode.
TI2CBasicAddr - Interfaces uses the basic (7-bit) addressing mode.
```

The i2c_flags_t values are defined below. The flags define the behavior of the operation for the call being made. These values may be ORed together.

```
I2C_START    - Transmit an I2C STOP at the beginning of the operation.
I2C_STOP     - Transmit an I2C STOP at the end of the operation. Cannot be
               with the I2C_ACK_END flag.
I2C_ACK_END  - ACK the last byte sent from the buffer. This flags is only v
               a write operation. Cannot be used with the I2C_STOP flag.
```

3.3 UART

The Universal Asynchronous Receiver/Transmitter (UART) interface is a type of serial interconnect. The interface is "asynchronous" since it recovers timing from the data stream itself, rather than a separate control stream. The interface is split into an asynchronous multi-byte level interface and a synchronous single-byte level interface.

The multi-byte level interface, UartStream, provides a split-phase interface for sending and receiving one or more bytes at a time. When receiving bytes, a byte-level interrupt can be enabled or an interrupt can be generated after receiving one or more bytes. The latter is intended to support use cases where the number of bytes to receive is already known. If the byte-level receive interrupt is enabled, the receive command MUST return FAIL. If a multi-byte receive interrupt is enabled, the enableReceiveInterrupt command MUST return FAIL.

```
interface UartStream {
  async command error_t send( uint8_t* buf, uint16_t len );
  async event void sendDone( uint8_t* buf, uint16_t len, error_t error );

  async command error_t enableReceiveInterrupt();
  async command error_t disableReceiveInterrupt();
  async event void receivedByte( uint8_t byte );

  async command error_t receive( uint8_t* buf, uint8_t len );
  async event void receiveDone( uint8_t* buf, uint16_t len, error_t error )
}
```

The single-byte level interface, UartByte, provides a synchronous interface for sending and receiving a single byte. This interface is intended to support use cases with short transactions. Because UART is asynchronous, the receive command takes a timeout which represents units in byte-times, after which the command returns with an error. Note that use of this interface is discouraged if the UART baud rate is low.

```
interface UartByte {
  async command error_t send( uint8_t byte );
  async command error_t receive( uint8_t* byte, uint8_t timeout );
}
```

4. Implementation

Example implementations of the pin interfaces can be found in tos/chips/msp430/pins, tos/chips/atm128/pins, and tos/chips/pxa27x/gpio.

Example implementations of the SPI interfaces can be found in tos/chips/msp430/usart, tos/chips/atm128/spi, and tos/chips/pxa27x/ssp.

Example implementations of the I2C interfaces can be found in tos/chips/msp430/usart, tos/chips/atm128/i2c, and tos/chips/pxa27x/i2c.

Example implementations of the UART interfaces can be found in tos/chips/msp430/usart, tos/chips/atm128/uart/ and tos/chips/pxa27x/uart.

5. Author's Address

Phil Buonadonna

Arch Rock Corporation
657 Mission St. Ste 600
San Francisco, CA 94105-4120

phone - +1 415 692-0828 x2833

Jonathan Hui
Arch Rock Corporation
657 Mission St. Ste 600
San Francisco, CA 94105-4120

phone - +1 415 692-0828 x2835

6. Citations