I²C Emulator
With HAL for Sensoray Model 826

by Jim Lamberson ~ April 4, 2016

Sensoray's I²C emulator is open source software that emulates a full-featured I²C bus master via bit-banged general-purpose digital I/Os (GPIOs). The emulator communicates with the GPIOs through a hardware abstraction layer (HAL), allowing it to be easily ported to different hardware. An example HAL is provided for Sensoray's model 826 multi-function I/O board.

Bus Arbitration

The emulator implements bus arbitration, allowing it to coexist with other masters on the I²C bus. Upon detecting loss of arbitration it will abort the current transaction and notify the application program. The application is responsible for taking appropriate corrective action (e.g., retry the transaction).

Clock Stretching

In theory, there is no upper limit to how long a slave can stretch the clock. In practice, however, the master must limit the duration of clock stretches so that the application won't hang if a slave fails to release the clock line. The emulator does this by timing each clock stretch and, upon detecting an abnormally long stretch, it will abort the current transaction and return I2CERR_SCL. This indicates that the slave (or some other current sink) is holding SCL low and that no further transactions are possible until the condition is cleared.

If you are debugging a slave and using breakpoints that could potentially interrupt an I²C transaction, set ENABLE_FAULT_DETECTION to 0 to disable clock stretch timing. This allows the slave to break at any point during a transaction without causing an I2CERR_SCL error.

Hang Recovery

I²C bus-hang is a deadlock condition in which a slave cannot release SDA until it receives a clock, while at the same time no master can begin a transaction (and thus issue a clock) until SDA is released. It can occur during normal I²C bus operation, when a master is reset while reading from a slave.

The emulator checks for bus-hang when it opens. If bus-hang is detected, the emulator will attempt to force the I²C network into a usable state; an error code will be returned if the attempt fails.

Portability

The code is separated into two functional categories: Emulator and HAL. The emulator is hardware-independent whereas the HAL is designed to work with particular hardware. For example, the HAL distributed with this project is designed to interact with a Sensoray model 826 board.

This functional separation simplifies the task of porting the emulator to different hardware. To port the emulator, it is only necessary to modify the HAL; the emulator can be ported without modification.

The emulator can be ported to any hardware having the following resources:

- 2 GPIOs, used to bit-bang SDA and SCL. These must have open-drain (or open-collector) outputs, or have tri-state outputs that can simulate open-drain outputs. The 826 hardware performs atomic GPIO bit set/clear operations; if your hardware doesn't do this then your HAL may need to to treat bit operations as critical sections.
- Precision clock, used to avoid “dirty” software timing loops. The HAL exposes this as a 32-bit value with 1 µs resolution. This function is provided by the timestamp generator on Sensoray's 826; when porting, this can be implemented with a dedicated counter/timer or by using a precision system clock.

Protocol Support

Some I²C slave devices (e.g., EEPROMs) employ a protocol in which the master sends an internal device address to the slave prior to each data transfer. The emulator directly implements this protocol: all I²C transaction functions will optionally send an internal device address to the slave before data is exchanged.

If you don't want to send an address, set the function's address length argument (`addrsize`) to zero. If you do want to send an address, set `addrsize` to the address length (in bytes). Alternatively, you can "manually" send an address by setting `addrsize` to zero and prepending the address to the data to be written to the slave; if you do this, be sure to pay attention to address endianness.

Example

This program snippet shows how to write two data bytes, starting at internal device address 3, to the slave mapped to I²C address 0x50. The hypothetical slave has a 256-byte internal address space, so `addrsize` is set to 1. Five data bytes are then read from the slave starting at internal device address 123. Error checking has been omitted for clarity.

```
#include "i2c_emulator.h"

unsigned char txdata[] = {100, 200};  // Data to be sent to I2C slave.
unsigned char rxdata[5];              // Data received from I2C slave.

 i2c_open();           // Open the emulator -- DO THIS BEFORE CALLING OTHER EMULATOR FUNCTIONS.

 i2c_writeBytes(0x50,  // Write data to slave at I2C address 0x50:
     3,                //   Precede data with internal slave address 0x03,
     1,                //     which has address size == 1 byte.
     sizeof(txdata),   //   Number of data bytes to send.
     &txdata);         //   Data to be sent.

 i2c_readBytes(0x50,   // Read data from slave at I2C address 0x50:
     123,              //   Before reading data, send internal slave address 123,
     1,                //     which has address size == 1 byte.
     5,                //   Number of data bytes to read.
     &rxdata);         //   Receive data into this buffer.

                       // ... do some other stuff

 i2c_close();          // Close the emulator -- DO THIS BEFORE TERMINATING THE PROGRAM.
```

Source Code

The project consists of these source code files:

`i2c_emulator.c`	Generic I²C master emulator. This does not require any particular hardware or operating system.
`i2c_emulator.h`	Generic emulator API. Include this in your application program.
`i2c_hal.c`	Hardware abstraction layer for the emulator. This version works with Sensoray's model 826 board.
`i2c_hal.h`	Generic HAL/emulator interface.
`826api.h`	API for Sensoray model 826 board (included in HAL).

Download the Source Code