I2C Slave on STM32

Roy Ratcliffe

Abstract

Embedded developers can easily mock up I^{2}C peripherals using a spare STM32's I^{2}C channel in slave mode. This article introduces two new software components: an I^{2}C slave and an I^{2}C sequencer.

Keywords: C, FreeRTOS, STM32, I2C

I^{2}C is a very ubiquitous bus interface. Embedded systems often include multiple devices wired to the bus. Breadboarding a multi-peripheral system is not always easy or convenient using breakout boards. Would it not be very handy if the developer could easily simulate all I^{2}C peripherals in software using the microprocessor's own I^{2}C channel acting as a *slave* answering in slave mode to *multiple* addresses?

Sounds tricky but turns out relatively straightforward. The goal is to give the embedded software developer a working hardware environment that usefully helps to iron on the firmware *before* the target board has completed design or delivery—call it real-world rapid embedded stubbing.

Code at GitHub along with its supporting functions.

1. Embedded Scenario

Take the STM32 family of microprocessors as an example. The STM32 microprocessor equips a highly-capable set of I^{2}C peripherals. Using a development board we will mock a not-yet-available board with four temperature sensor TMP10x devices by Texas Instruments. Our experiment will mock the TMP10x devices using the processor's own hardware operating in slave mode.

1.1. TMP10x by Texas Instruments

The TMP10x is a 4-register file peripheral that measures printed-circuit board temperature. It carries core registers:

- a 16-bit read-only temperature register (TMP)
- an 8-bit read-write configuration register (CFG)
- a 16-bit read-write temperature low register (TLO)
- a 16-bit read-write temperature high register (THI)
- a 2-bit indexing register (PTR)

For the sake of experimentation, we can simplify the device model to comprise four 16-bit registers addressed by the first I^{2}C data frame's least significant two bits thereby indexing the four-by-sixteen-bit register file. This simplification feels natural regardless. The bus interfaces with the internal register set. Internal logic dictates how the registers interact with the peripheral's external pins.

For a teaser, the final code looks like this snippet. Note the functional approach. Please forgive the size of the extract; it represents an almost fully-functional TMP10x emulation that responds to write and read transfers between

*Corresponding author
Email address: `roy@ratcliffe.me` (Roy Ratcliffe)

"By hacking code *for itself, don't you see. Deriving pleasure from the gift of pure coding."—Wise Martian*

four 16-bit registers and an I^{2}C bus. It does *not* of course measure a temperature but any master-mode transfer will never know that.

```c
/*!
 * \brief Set up device at address.
 *
 * For the sake of simplicity, all errors assert. Assume the happy path.
 */
static portTASK_FUNCTION(prvTMP10xTask, pvParameters) {
  struct TMP10x *xTMP10x = pvParameters;

  /*
   * 16-bit register file.
   */
  uint16_t usFile[] = {0x1234U, 0xff00U, 0x1111U, 0xffffU};

  void prvAddr(I2CSeqHandle_t xI2CSeq) {
    switch (ucI2CSeqTransferDirection(xI2CSeq)) {
      HAL_StatusTypeDef xStatus;
    case I2C_DIRECTION_TRANSMIT:
      /*
       * Receive the first and next frames.
       */
      vI2CSeqBufferLengthBytes(xI2CSeq, 3U);
      xStatus = xI2CSeqFirstFrame(xI2CSeq);
      configASSERT(xStatus == HAL_OK);
      break;
    case I2C_DIRECTION_RECEIVE:
      /*
       * Transmit the last frames.
       */
      switch (xI2CSeqXferBytes(xI2CSeq)) {
        uint8_t ucPtr;
        uint16_t usReg;
        uint8_t ucBig[2];
      case 1U:
        vI2CSeqCopyTo(xI2CSeq, &ucPtr);
        /*
         * Send big-endian word.
         */
        usReg = usFile[ucPtr & 0x03U];
        ucBig[0] = usReg >> 8U;
        ucBig[1] = usReg;
        vI2CSeqCopyFrom(xI2CSeq, ucBig);
        xStatus = xI2CSeqLastFrame(xI2CSeq);
        configASSERT(xStatus == HAL_OK);
      }
    }
  }

  /*
```

```
     * The slave buffer receive completes.
     */
    void prvSlaveRxCplt(I2CSeqHandle_t xI2CSeq) {
      switch (ucI2CSeqTransferDirection(xI2CSeq)) {
      case I2C_DIRECTION_TRANSMIT:
        switch (xI2CSeqXferBytes(xI2CSeq)) {
          uint8_t *pcBuffer;
        case 3U:
          pcBuffer = pvI2CSeqBuffer(xI2CSeq);
          usFile[pcBuffer[0U] & 0x03U] = (pcBuffer[1U] << 8U) | pcBuffer[2U];
        }
      }
    }

    vI2CSlaveDeviceAddr(xTMP10x->xI2CSlave, xTMP10x->ucAddr, prvAddr);
    vI2CSlaveDeviceSlaveRxCplt(xTMP10x->xI2CSlave, xTMP10x->ucAddr, prvSlaveRxCplt);
    uint32_t ulNotified;
    xTaskNotifyWait(0UL, tmp10xSTOP_NOTIFIED, &ulNotified, portMAX_DELAY);
    vI2CSlaveDeviceAddr(xTMP10x->xI2CSlave, xTMP10x->ucAddr, NULL);
    vI2CSlaveDeviceSlaveRxCplt(xTMP10x->xI2CSlave, xTMP10x->ucAddr, NULL);
    vTaskDelete(NULL);
  }
```

This code extract runs in a FreeRTOS task. Note the functional closures and their bindings to I^{2}C slave register file, `usFile`. The next sections describe the dependent components: `xI2CSeq`, an abstract transmit-receive transfer sequencer; and `xI2CSlave`, an I^{2}C slave service.

2. I^{2}C Sequencer

See Figure 1 for a class-style 'unified model.'

2.1. *No-Option Frames*

The sequencer supports option frames and no-option frames. The HAL software by ST has two frame-related I^{2}C interfaces: option frames and no-option frames. The former option frames trigger with options: first, first and next, first and last, and last frame. These options determine the injection of automatic- or software-end mode during transfer configuration. Option-based framing suits repeated starts.

The HAL does *not* offer a no-option version of the optional frame interfaces. The options do not offer an `I2C_NO_OPTION_FRAME` externally. The option exists internally *only*. The no-option framing interface *is* the non `Seq` interface. In other words,

- `HAL_I2C_Master_Transmit_IT` *is* the no-option master transmit transfer;
- `HAL_I2C_Master_Receive_IT` *is* the no-option master receive;
- same for `HAL_I2C_Slave_Transmit_IT` and `HAL_I2C_Slave_Receive_IT`.

Inspect the internal implementations. They all set up `I2C_NO_OPTION_FRAME`. The caller cannot invoke the `Seq` versions of the interface with this null transfer option[1].

[1]Privately defined as $FFFF0000_{16}$.

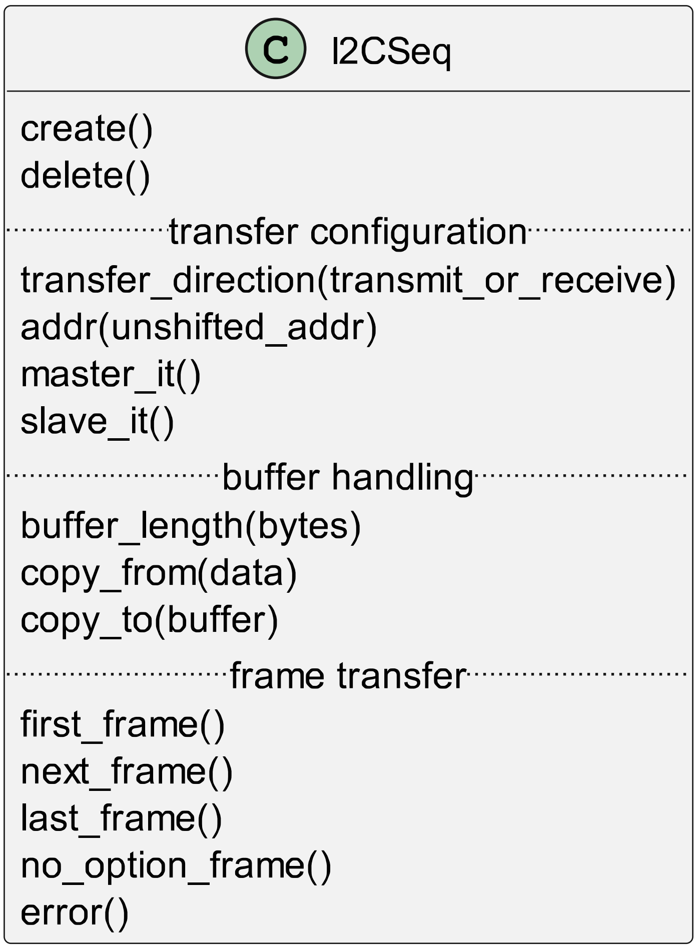

Figure 1: I^{2}C sequencer abstraction

3. I^{2}C Slave

The slave encapsulates 0 through 127 sub-devices for 7-bit I^{2}C addressing; 10-bit addressing is not yet supported. See UML model, see Figure 2.

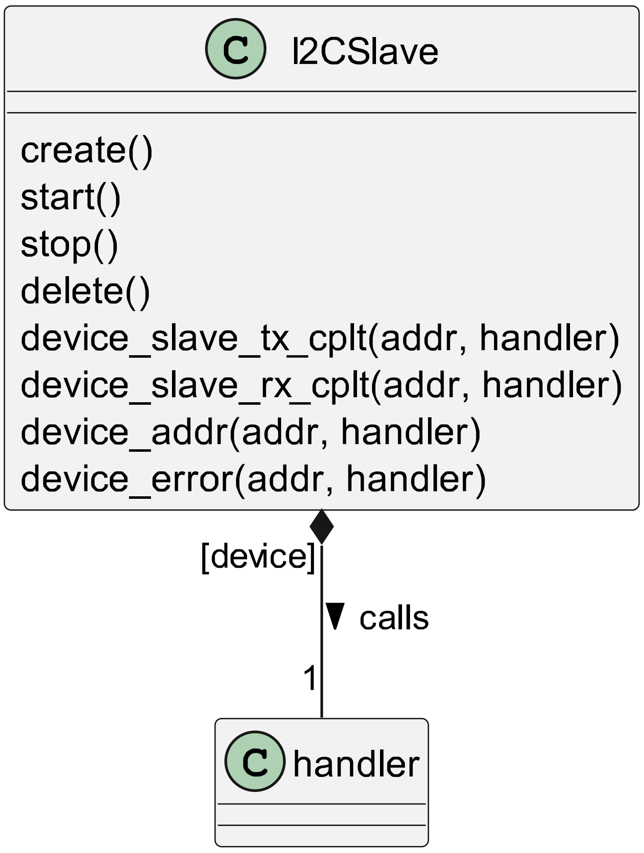
I2CSlave
create()
start()
stop()
delete()
device_slave_tx_cplt(addr, handler)
device_slave_rx_cplt(addr, handler)
device_addr(addr, handler)
device_error(addr, handler)
[device]
▼ calls
1
handler

Figure 2: I^{2}C slave abstraction

A slave is a service: create, start, stop and delete. Once created, wire up device handlers: functions that respond to one or more of:

1. slave transmission completed,
2. slave reception completed,
3. address matched, and
4. error detected.

The function names “tx cplt” and so forth mirror the underlying hardware abstraction layer names for the raw slave events. When the slave matches an address, either the primary or by mask-matching a secondary address, it raises an interrupt and invokes the device handler based on the address.

Secondary address matching requires a mask and allows the slave service to catch multiple addresses.

4. Conclusions

The result makes extensive use of the GNU compiler’s nested functions. This style allows for ‘functional capture’ but requires an executable stack and additional stack space for the compiler’s clever ‘trampoline’ work.

The slave service captures interrupt-driven events and bounces them to its private task using task notification. This is an important feature because it enables all ordinary task-level operations for all the associated device handlers, including services such as memory allocation. This carries a small cost. The interrupt handlers register notifications and then a task switch must occur. The switching adds a small amount of latency but not significantly provided that the slave task fits into the real-time range of task priorities, allowing it to preempt any more compute-bound workers.

The sequencer and slave combination works well. It resolves some important but subtle considerations. First, the dynamic sequencer buffer addresses the requirement to persist the transfer data during asynchronous activity while additionally registering the initial transfer size. Since the internal implementation of ST's abstraction layer down-counts the size, the sequencer can conveniently compute the difference in order to know the final number of bytes transferred if less than expected. Actual transfer size usefully helps to decide what the slave device should do with the data frames, typically along with the initial byte.

Secondly, the sequencer accounts for the transfer direction's master orientation. 'Transmit' means that the *master* wants to transmit: the slave must therefore *receive*. The master's transmission is the slave's reception. Sequencing a frame or frames consequently inverts depending on the transfer direction. The sequencer implementation correctly maps the behaviour accordingly.

5. Future Work

Future development could add a pseudo-dynamic temperature reading. In its current state, the simulation lets the I^{2}C master *write* to the temperature register. The real device prohibits such writes.