DigiKey

Enter keyword or part #

Additional Content In: Japanese German Hebrew Korean Chinese Spanish

Using the STM32Cube HAL I2C Driver in Master Mode

development-boards-kits-prog… , design-tools-and-resources, stmicroelectronics, embedded

Matt_Mielke 1 July 1, 2021, 1:33pm

Background

For each of their MCU series, STMicroelectronics offers an embedded firmware package which includes, among other things, a Hardware Abstraction Layer (HAL) driver. This driver provides a high-level set of APIs designed to abstract away the complexity of the MCU and its peripherals in a highly portable fashion. The I^{2}C interface is one such peripheral. For a beginner's guide to getting started with the HAL I^{2}C driver, please see this tutorial by Shawn Hymel. For a quick overview of the communication functions available to an I^{2}C master device, read on.

The HAL driver supports three programming models for its data processing functions: polling, interrupt, and DMA. Polling functions operate in **blocking mode**, meaning these functions will not return until the operation is complete. To prevent the application from hanging, the user must provide a suitable timeout value. Interrupt and DMA functions operate in **non-blocking mode**, which means these functions will return after the operation is initiated, allowing the application to continue execution while the operation continues in the background. However, a callback function must be configured and enabled to handle the signal raised once the operation completes.

When acting as an I^{2}C master in blocking mode, there are four API functions available for communicating with a slave device:

- `HAL_I2C_Master_Transmit()`
- `HAL_I2C_Master_Receive()`
- `HAL_I2C_Mem_Write()`
- `HAL_I2C_Mem_Read()`

For non-blocking functionality, the interrupt and DMA modes have equivalent functions. However, since the polling functions are the more straightforward of the three, they will be used for this reference guide.

Transmitting Data

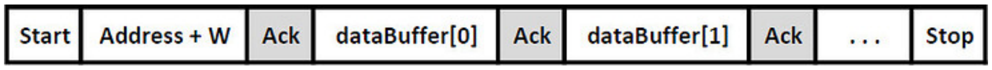
Sending data from the master device to a slave device is for the most part uncomplicated. Either the `HAL_I2C_Master_Transmit()` or `HAL_I2C_Mem_Write()` functions may be used. Which one to choose depends on the message structure or simply on personal preference.

HAL_I2C_Master_Transmit()

The function prototype for this API function is shown below. The first parameter is simply a configuration structure, the creation of which is detailed in the getting started tutorial. The second parameter is the address of the slave device (which must be shifted to the left by one). The third and fourth parameters are a pointer to the data buffer and the amount of data from the buffer that should be sent to the slave device, respectively. The last parameter is the timeout duration in milliseconds. Note that the user can supply `HAL_MAX_DELAY` as an argument to disable the timeout and infinitely block until the function returns.

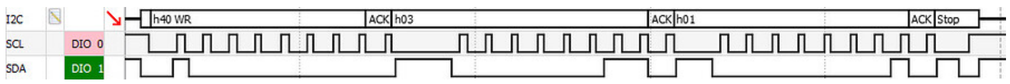
```
HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c,
                                          uint16_t DevAddress,
                                          uint8_t *pData,
                                          uint16_t Size,
                                          uint32_t Timeout);
```

The I^{2}C sequence resulting from calling this function is shown below. The shaded regions indicate where the signal is driven by the slave device. In this case, the slave only acknowledges its own address (plus the write bit) and any data bytes that follow. Recall that the number of data bytes sent is determined by the `Size` argument provided to the function call.



As an example of this function in use, consider the code below where the contents of the buffer are sent to a slave device with the address 0x40. The I^{2}C transmission was captured with a logic analyzer and the resulting waveforms are shown below as well. Note that the start condition *is* present, but there isn't room for the label in the decoded protocol graphic.

```
uint8_t dataBuffer[10] = {0x03, 0x01};
HAL_I2C_Master_Transmit(&hi2c1, (0x40 << 1), dataBuffer, 2, HAL_MAX_DELAY)
```

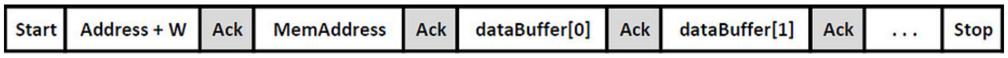


HAL_I2C_Mem_Write()

This function is intended for the common scenario in which the master device wants to write to a specific memory location on a slave device. Most I^{2}C sensors, for instance, contain configuration and command registers used to change settings and initiate measurements. The prototype for this function is as follows.

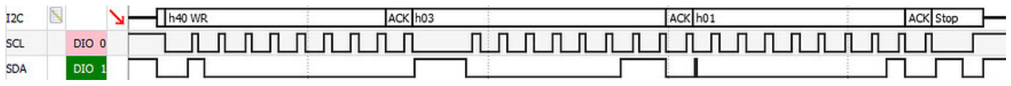
```
HAL_StatusTypeDef HAL_I2C_Mem_Write(I2C_HandleTypeDef *hi2c,
                                    uint16_t DevAddress,
                                    uint16_t MemAddress,
                                    uint16_t MemAddSize,
                                    uint8_t *pData,
                                    uint16_t Size,
                                          uint32_t Timeout);
```

Clearly, it contains all the same parameters as the `HAL_I2C_Master_Transmit()` function along with two additional ones. The first, `MemAddress`, is the starting memory address in the slave device that the buffer contents will be written to. The second, `MemAddSize`, is simply the size of the internal memory address (either `I2C_MEMADD_SIZE_8BIT` or `I2C_MEMADD_SIZE_16BIT`). The I^{2}C sequence will now appear as shown below.



It is the same as the sequence generated by `HAL_I2C_Master_Transmit()`, except the `MemAddress` argument is sent after the slave address and before the first byte from data buffer. The following example uses the `HAL_I2C_Mem_Write()` function to write the value 0x01 to a register located at memory address 0x03 on the slave device. Notice that the I^{2}C operation captured by the logic analyzer is exactly the same as that shown in the example for the `HAL_I2C_Master_Transmit()` function above.

```
uint8_t dataBuffer[10] = {0x01};
HAL_I2C_Mem_Write(&hi2c1, (0x40 << 1), 0x03, I2C_MEMADD_SIZE_8BIT, dataBuf
```



Receiving Data

Unlike sending data from the master to the slave, the two functions available for receiving data from the slave to the master are not interchangeable. One will only receive data while the other will first specify an address from where to receive data.

HAL_I2C_Master_Receive()

This API function is used to simply request data from the slave device. Notice in the below prototype that its parameters are identical to those of `HAL_I2C_Master_Transmit()`. In this case, however, the data buffer is used to store the incoming data and the `Size` parameter specifies how many bytes of data to receive before sending a Nack.

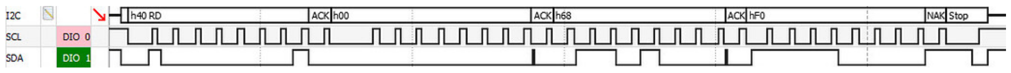
```
HAL_StatusTypeDef HAL_I2C_Master_Receive(I2C_HandleTypeDef *hi2c,
                                         uint16_t DevAddress,
                                         uint8_t *pData,
                                         uint16_t Size,
                                         uint32_t Timeout);
```

The sequence diagram describing this function's operation is provided below. Notice that it is fundamentally different from either of the data transmission functions discussed above. First, the direction bit in the address byte is set to read rather than write. Second, after the slave acknowledges its own address, it begins sending data bytes to the master (recall that the shaded fields indicate the slave is driving the signal). It is now the master's responsibility to acknowledge each byte it receives until it no longer wishes to receive them. It tells the slave to stop sending data by sending a Nack followed by a Stop condition.

Start	Address + R	Ack	Data	Ack	Data	Ack	. . .	Data	Nack	Stop

The code example below shows the master requesting three bytes of data from the slave device with address 0x40. We can see by observing the logic analyzer capture, that after the operation is complete, the data buffer will contain the values {0x00, 0x68, 0xF0}.

```
HAL_I2C_Master_Receive(&hi2c1, (0x40 << 1), dataBuffer, 3, HAL_MAX_DELAY);
```



HAL_I2C_Mem_Read()

This API function is used to request data from a slave device *from a specific memory address*. Again, consider an I^{2}C sensor in which measurements are stored in a specific register on the slave and the master must read data from that register. As shown below, the function prototype contains the same arguments as the `HAL_I2C_Mem_Write()` function. However, as was the case above, the data buffer will be used to store the incoming data rather than source it.

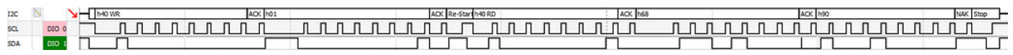
```
HAL_StatusTypeDef HAL_I2C_Mem_Read(I2C_HandleTypeDef *hi2c,
                                   uint16_t DevAddress,
                                   uint16_t MemAddress,
                                   uint16_t MemAddSize,
                                   uint8_t *pData,
                                   uint16_t Size,
                                   uint32_t Timeout);
```

What makes this function unique is that it first performs an I^{2}C transmit operation to tell the slave device what memory address to source the data from. This is followed by a repeated start condition to begin the receive operation. The complete I^{2}C sequence is shown below. Note that **`HAL_I2C_Mem_Read()` is the only function capable of generating a repeated start condition in blocking mode.** If a repeated start is required, it is not sufficient to call `HAL_I2C_Master_Transmit()` immediately followed by a call to `HAL_I2C_Master_Receive()`.

Start	Address + W	Ack	MemAddress	Ack	Restart	Address + R	Ack	Data	Ack	Data	Ack	. . .	Data	Nack	Stop

The following code example receives two bytes of data from the slave device with address 0x40 starting at the memory address 0x01. Notice from the logic analyzer capture that the buffer will contain {0x68, 0x90} when the operation completes.

```
HAL_I2C_Mem_Read(&hi2c1, (0x40 << 1), 0x01, I2C_MEMADD_SIZE_8BIT, dataBuff
/* This is NOT guaranteed to work as a substitute for the above! */
// dataBuffer[0] = 0x01;
// HAL_I2C_Master_Transmit(&hi2c1, (0x40 << 1), dataBuffer, 1, HAL_MAX_DEL
// HAL_I2C_Master_Receive(&hi2c1, (0x40 << 1), dataBuffer, 2, HAL_MAX_DELA
```



Summary

The HAL I^{2}C driver provided by STMicroelectronics allows the master device to communicate with slave devices in either blocking mode or non-blocking mode (blocking mode being the simpler of the two). To send data to a slave device in blocking mode, either the `HAL_I2C_Master_Transmit()` function or the `HAL_I2C_Mem_Write()` function may be used. The two are interchangeable. The only difference is that a memory address on the slave is explicitly specified as an argument to `HAL_I2C_Mem_Write()`. The user should decide which is most appropriate for their application. To receive data from a slave device, either the `HAL_I2C_Master_Receive()` function or the `HAL_I2C_Mem_Read()` function may be used. However, these two functions are **not** interchangeable. While `HAL_I2C_Master_Receive()` simply reads data from the slave device, `HAL_I2C_Mem_Read()` first sends a memory address to the slave and then issues a repeated start condition followed by a read operation to get the data located at that memory address. The choice of which function to use depends on the I^{2}C sequence expected by the slave device.

2 Likes

STM32Cube HAL I2Cドライバをマスターモードで使用する