MAX32660 Motion Co-processor Firmware Development Challenges

A project log for MAX32660 Motion Co-Processor

Super-small, ultra-low-power 96 MHz Cortex M4F co-processor manages sensors and processes data allowing the MCU host attend to other things.

Greg Tomasch • 11/17/2019 at 02:33 • 3 Comments

Well, it has been a long while since any log entries have been posted for this project... That is not because Kris and I haven't been working on it but because there have been some serious software development challenges to overcome. It is one thing to try out some of the simple I2C loop-back examples in Maxim's software development kit (SDK) or blink an LED but it is a fundamentally harder task to make the MAX32660's peripherals perform to the level required by a high-speed asynchronous sensor fusion engine.

I'm happy to report that the "show-stopper" software infrastructure problems preventing a robust, high-performance motion co-processor implementation on the MAX32660 have been solved. We have demonstrated reliable operation and excellent attitude estimation results from MAX32660-based parts. I plan to open a new project to document general motion co-processor calibration/characterization instrumentation and procedures soon. Results from the MAX32660 motion co-processor will figure prominently into this effort. The remainder of this log entry will be an overview of the firmware development challenges and what it took to overcome them.

When I set out to write motion co-processor firmware for this micro-controller, I already had almost all of the necessary algorithmic and calibration "pieces" in-hand and successfully implemented on other micro-controllers. This should be the hard part, right? Not so fast... There are some other basic pieces of "data plumbing" that are absolutely essential to make a motion co-processor work on any micro-controller:

- **An EEPROM emulator** to store co-processor configuration information and sensor calibration data for retrieval at startup

- **An asynchronous I2C master bus**; sensor data ready (DRDY) interrupts initiate an I2C data read transaction from the appropriate sensor with a completion callback to the main sensor fusion loop
- **An asynchronous I2C slave bus**; the host micro-controller (connected to the MAX32660) must be able to query the co-processor for data and receive a prompt response without significantly delaying the main sensor fusion loop
- **Rising-edge-interrupt-**capable GPIO pins to handle sensor DRDY interrupts.

After an initial assessment of Maxim's SDK, it became clear that only the rising-edge interrupt capability would be straightforward. I addressed the other three items in this order:

1. EEPROM emulator
2. Asynchronous I2C master bus
3. Asynchronous I2C slave bus

I should say that if any one of these capabilities could not be successfully implemented on the MAX32660, completion of the project would not be worthwhile. The final product would simply not be competitive. Before I proceed much further, I want to acknowledge the fact that I was very lucky to have an excellent teacher/mentor along the way. Thomas Roell is truly a master at micro-controller application programming interface (API) development and he shared his expertise with me generously. Without his inputs and help along the way it would have certainly been much, much harder to bring the project along this far.

EEPROM Emulator

An EEPROM capability is crucial for a motion co-processor. The basic proposition here is that we are using economical MEMS sensors as the inputs into the fusion/attitude estimation algorithm. No matter what, there is no algorithm that compensates for bad sensor data... So good sensor calibration is a prerequisite to achieve an accurate attitude estimate. Since MEMS sensors can significantly depart from ideal behavior, effective calibration may require more involved procedures performed under controlled conditions. The calibration procedures generate parametric corrections that are applied at run-time. If the sensor calibration procedure(s) are impractical to do each time at startup, the sensor correction parameters need to be available in non-volatile memory (NVM).

The basic idea behind an EEPROM emulator is that part of the micro-controller's flash memory is set up to mimic a writable ROM without the user having to do any of the hard work of following the flash memory's rules for locking, unlocking, reading, writing and erasure. Furthermore, the EEPROM emulator manages usage of the underlying flash memory to prevent premature wear-out of any individual flash memory bits. It turned out to be fairly easy to develop an EEPROM emulator for the MAX32660... Largely because Thomas Roell had already written a good EEPROM emulator for the STM32L4, another Cortex M4F micro-controller. The basic emulator

algorithm was copied from the STM32L4 and only the MAX32660-specific flash read/write/erase/lock/unlock commands needed to be updated.

Asynchronous I2C Master Bus

I took a long look at the I2C master bus asynchronous API supplied in the MAX32660 API and realized that the way it was written had a serious flaw: It only does a *single* read or write unit transaction per function call. However, an I2C master read is actually *two* unit transactions: 1) *Write* the data register address from the master to the sensor, 2) *Read* data bytes from the sensor (starting at the data register address just written) to the master. The problem here is that the data callback from the first instance of the async I2C transfer function (to write the starting data register byte) has to call the same async transfer function *again* to read the data bytes from the sensor. Why is this a problem? Unless you are very careful:

1. The bus is released in between the sensor data register write and the sensor data read, killing the read transaction
2. Multiple instances of the async I2C transfer function may be left open, causing concurrence issues and/or the stack running out of memory space

After a great deal of User's Guide spelunking and logic analyzer experimentation, I re-wrote the I2C API so that both master read and master write transactions can be done with a single call of the I2C async transfer function and no recursion. This was accomplished by making all cases in the transfer function fully driven by I2C controller hardware interrupts.

Fixing the async I2C master bus API was a big step forward but there was still another huge problem to be solved: concurrent I2C transactions (because there are multiple sensors on the MAX32660's master bus). Fortunately, Thomas Roell already mapped out a good solution for this on other Cortex M4F micro-controllers:

- Write void-type wrapper functions for the I2C data read functions associated with each sensor
- Declare pointer variables for each sensor data read wrapper function
- Declare a Boolean status variable to indicate when the master I2C bus is busy
- Make a read function pointer ring buffer; every time there is a sensor DRDY interrupt, enqueue the appropriate data read function pointer into the ring buffer
- Service the read function pointer ring buffer regularly; if the I2C master bus isn't busy with a transaction, set up the next sensor data read transaction to be executed by the Cortex "PendSV" handler

After working through the actual coding of this method, the results were gratifying. It is possible to asynchronously burst-read the accelerometer, gyroscope, magnetometer and barometer data without causing any significant timing fluctuation of the main co-processor loop. This solution runs robustly at I2C clock speeds up to 1MHz and should be readily expandable to include auxiliary sensors in the future. The current LPS22HB baro sensor doesn't work at 3.4MHz I2C

clock speed... But there seems to be no fundamental reason why the I2C master bus can't run at 3.4MHz so long as all of the sensors on the bus can support the higher clock speed.

Asynchronous I2C Slave Bus

The slave bus turned out to be a larger challenge than anticipated but for very different reasons than the I2C master bus. Again, the async slave bus API included in Maxim's SDK is implemented as a transfer *function* that is called for each unit I2C transaction. This is just like the original master bus API... It turns out that it is a poor approach for an I2C slave bus. Despite this shortcoming I tried to make Maxim's I2C slave bus API work anyway. I was successful but only for 100kHz I2C clock frequency. At higher clock speeds the slave bus would freeze within a few transactions. The logic analyzer showed that during a master read:

- The host successfully writes the data register address and the MAX32660 would acknowledge (ACK) this byte
- However, the MAX32660's I2C *controller hardware* ACKs the data register byte while the *main processor* prepares the data to be read by the master. These are independent parallel processes
- So there is a race condition here; if the MAX32660's processor is late preparing the data to transfer, the data register byte ACK is missed while the master read transfer is being set up... And no data is loaded into the slave bus transmit buffer
- When this happens, the MAX32660 transmits no data to the host micro-controller while the host stretches the I2C clock indefinitely

When the I2C slave bus clock was running at 100kHz, there was enough time to prepare the data and set up the master read transfer before the MAX32660's I2C controller ACK'd the data register byte coming from the master. At 400kHz clock speed, not so much... This was a disappointing outcome because a 100kHz I2C slave bus can't deliver sensor and orientation estimate data to the host micro-controller at the desired 1kHz update frequency; the I2C transfer time overhead is too great.

The solution was a complete re-write of the I2C async slave API:

1. There is only one function call to set up the MAX32660's I2C slave bus peripheral controller
2. All slave bus transactions are driven by peripheral controller hardware interrupts
3. A single interrupt service request (ISR) handler manages all of the peripheral controller interrupts
4. The race condition described above is eliminated by delaying the ACK to the data register byte until the MAX32660 is ready to supply the necessary data for the master read transaction

This approach has yielded far superior results. The slave bus runs robustly at 1MHz clock speeds. The data transfer rates to the host micro-controller are more than sufficient to support

1kHz accelerometer, gyroscope and attitude estimation update frequencies...

Previous Log
Some Power Measurements
08/25/2018 at 01:15 • 5 comments

Next Log
Final Hardware Design
11/19/2019 at 01:01 • 0 comments

DISCUSSIONS

Log In or become a member to leave your comment

Log In/Sign up to comment

Simon Merrett wrote 11/18/2019 at 11:19

Wow, what a generous insight into the challenges, approaches and solutions to "taming" this device for everyone to benefit from. Thank you

Greg Tomasch wrote 11/18/2019 at 14:45

Hi Simon,

Many Thanks for your kind words. It has been quite a journey dragging along this petulant little micro-controller... Kicking and screaming the whole way! :-)

Simon Merrett wrote 11/18/2019 at 14:46

Rather you than me - people would have been waiting a whole lot longer if I'd been at the helm! :-)

Going up?