Exposing the PRU as a SPI,I2C Master controller

A Project under BeagleBoard.org Foundation for Google Summer of Code-2016

By-Vaibhav Choudhary(Chanakya_vc)

Introduction

This report is an effort to document the various stages of project development and what I have learned while doing the project. It also attempts to document the improvements that can be done to the present code base and tries to analyze the present limitations of the code. The report also the documents my personal experiences with the linux kernel and tries to present the difficulties that I faced as a beginner in Linux kernel Development. At the outset, I would like to thank all my mentors, who gave me this wonderful opportunity. This project not only exposed me to the field of Kernel Development but also taught me a lot about the processes involved in real world software development. I hope to continue being associated with BeagleBoard.org and keep contributing to open source software development.

Experience With Kernel Development And The Challenges That I Encountered

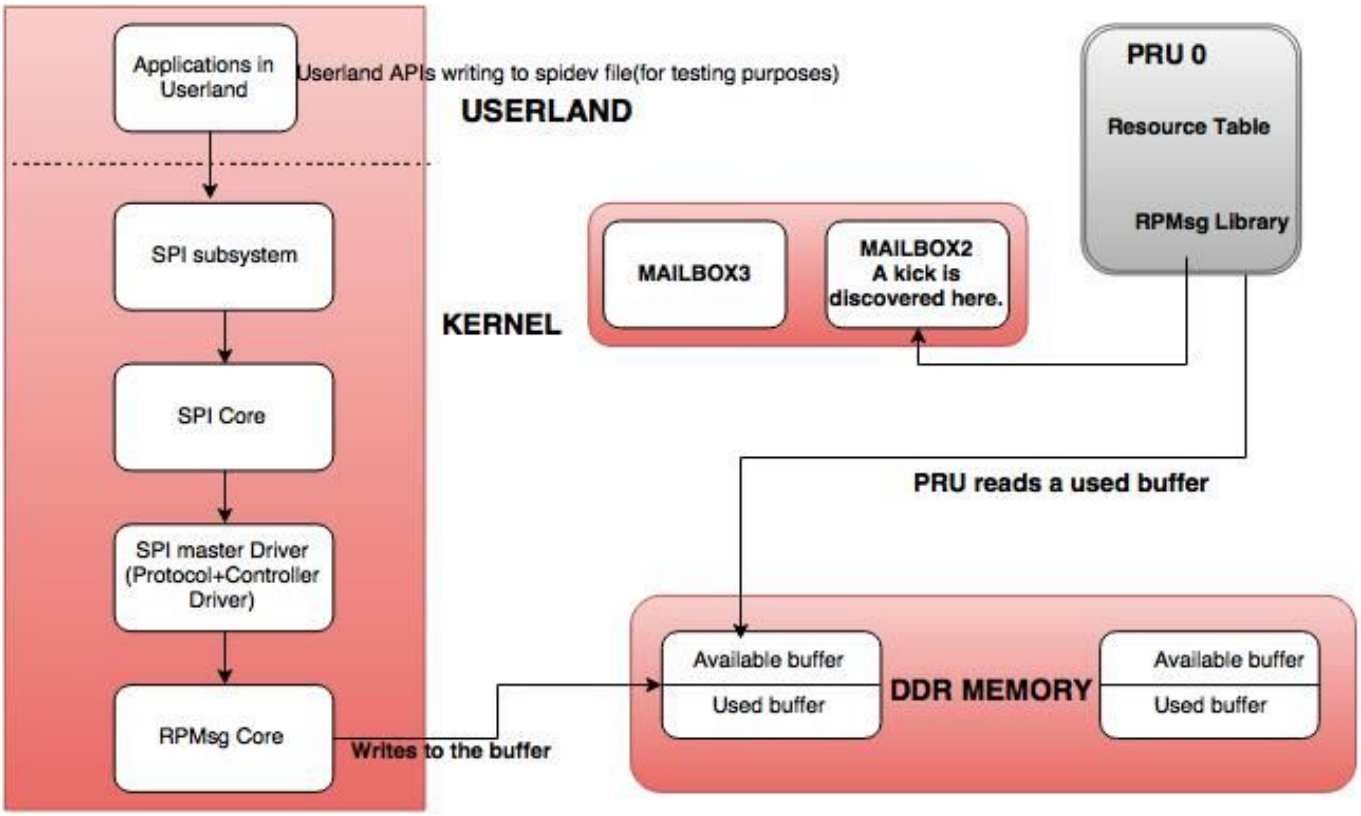
I had almost no experience with kernel development (or Linux for the matter) before i got selected for GSOC. it has been a humbling and fascinating journey from there to being able to write device drivers for SPI and I2C. I have come to appreciate linux better and also gained a new perspective about Open Source Software development in general.

Open Source software is just not about writing free software that is available to all, it also emphasizes that almost anyone, with or without any experience can contribute to the community and learn in the process of doing so.

I have learnt a lot in these three months. Some things from my mentors ,some from my fellow students in the program. Some of the major things that I have learnt/gained experience about are as follows:

1. Git and version control system: Though I had some little idea about how to use github before the program started, I really did not have much experience with it. I must really thank all my mentors for introducing me to the various aspects of git and version control. I would like to thank Wormo(don't know her real name) for introducing me to the basic commands of git. I would like to also thank my mentors Andrew sir,Michael sir and Alex sir who taught me the basic conventions of git and corrected me whenever I made mistakes.
2. I now have a very clear idea as to how write a SPI and I2C Master driver. I have learnt about all the important functions and structures that are required in a SPI Master and I2C Controller Driver. Thanks to my mentors, Matt sir and Hunyue sir, I have a very clear picture as to how the drivers are paired or matched with the device and how is their probe function is called. I am also clear on the role of the platform bus.
3. I have also learnt a lot about PRU and compiling firmware for it. I had to read the PRU-ICSS guide for finding the local addresses of the shared memory. In the process, i gained a very clear idea of the the configuration registers, __R30, __R31 registers and also gained an understanding of the architecture of the PRU.
    a. I have gained some idea about the working of the pru_rproc and rpmsg. As I also explained in my proposal, I am completely clear about the working of the RPMsg especially about how the Vring Data structures are allocated and the message box in the memory.
    b. I am also more clear on the working of the pru_rproc. I knew that pru_rproc use the API published by the core remoteproc driver to load the firmware to the PRU's and also provide the additional resources as requested by the firmware in the resource table. Over the period of three months, I have achieved clarity on how this process actually happens.

   i. Pru_rproc checks for PRU binaries in /lib/firmware/am335x-pru0-fw and then uploads it to the PRU after pru_rproc is insmod. This goes into the Instruction Ram of the PRU
   ii. It also copies the resource table into the Data RAM of the PRU.
4. I also wrote a driver implementing ioctls and learnt how ioctls actually works in the period just before GSOC. I also wrote a char driver and familiarized myself with all the struct and functions to write a functional driver that can implement ioctls.
5. I also learnt about sshfs and it proved very helpful when I wanted to transfer cross compiled binaries. It seems to a be a much better option than scp. This was taught to me by one of my fellow students, Zubeen Tolani. He has been a very good friend and has taught me a lot of useful things during the course of GSOC.
6. I also learnt HUGO, a static website generator written in the Go language. I used it to code my website, www.vaibhavchoudhary.com
7. Apart from things related to the project, I also got a lot to learn a lot about linux from my mentors. Andrew sir taught me a bit about init systems. I took it further and learnt the difference between upstart and systemd. Another example would be that I learnt a lot about the difference between SSDs and HDD's and the meaning class of an SD card when I was ordering one for burning the latest kernel to my BBB. I also learnt a lot about Single Level Cell and Multi Level Cell in case of SSD's.
8. Planning and executing a project. This was one of the most important things that I learnt. I learnt that one should be realistic about one's project timeline and only commit to what one could realistically achieve in the given time. This became apparent when I had a shortage of time (what my mentors had warned during the proposal submission period) towards the end. Luckily, i had listened to my mentors and put UART as a reach goal in my proposal.



There were many challenges that I faced during the course of this project.

1. I do not have a oscilloscope in my house. And the saleae logic analyzer that I ordered was unable to correctly capture the waveforms that the firmware was generating. So I had to travel to a lab in my college which had a MSO to check the output waveforms. My college is an hour's ride by bus from my home and it became pretty difficult to ride back and forth especially at night. I think that I could have avoided this situation had I made previous arrangements in hostels in my college. I had been of the opinion that Saleae would be sufficient to debug my code. I think not being to test my code when I was home(during night time when I was home and when most of the mentors were online) was probably the biggest challenge. It became very difficult to write code, without getting an active feedback from mentors on the results of the changes that I was making(as I was not able to run the code myself and test it).
2. The next big challenge that I encountered was when I had to cross compile the kernel source for cross-compiling the driver that I had written. It took me nearly a

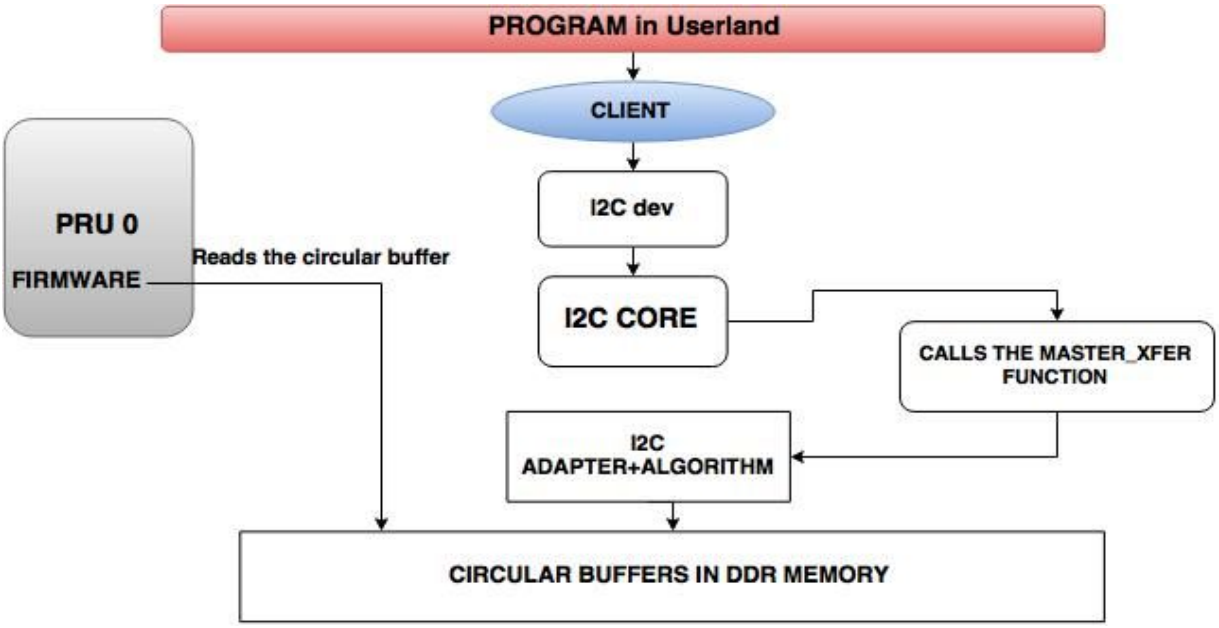
week and a half to get the correct kernel source with the correct vermagic string and revision number.

   a. I was using the config file that is there in the BBB gunzip it and place in the kernel source folder. This was allowing the driver to be compiled but not allowing it to insmod as the vermagic strings were different.
   b. I then downloaded the exact the kernel source as that was there on my BBB and tried again. Now the kernel wasn't compile again. Michael sir came to my rescue and then told me that there were carriage returns in my config file and that was probably causing the error in compilation. I tried again after using dos2unix command, the kernel source compiled successfully. However, the same problem arose again. The compiled driver did not insmod successfully on the target.
   c. After a lot of search, I finally realized that the kernel had to be compiled with option of an extraversion(for specific revision numbers, in my case it was ti-25). I again compiled the kernel and this time the driver insmod perfectly.

3. Another place where I was stuck was when I was testing out the firmware for multibyte transfer in SPI. I had looped back MISO to MOSI. However, I did not ever get MOSI equal to MISO. Then my mentors confirmed that although a simple loop -back should work in theory, it doesn't actually work perfectly. I actually learnt one very important lesson here, not discounting anything before properly research. I was most probably going to delete my code and think of something new instead of releasing that a loop-back would not be the most appropriate test in my case.
4. Another issue which I felt caused delay in my project was my limited background with Operating Systems and Computer Architecture. I have not been taught these subjects so far in my curriculum. And without a solid background in these subjects, it became very difficult for me to appreciate many things in my project.(like functions provided by ioread/iowrite). Although now at the end of GSOC, I feel I have learnt a lot about these subjects, I feel things would have a lot smoother had I got an understanding about these before GSOC. I also think that my lack of exposure to SPI and I2C devices and using them also made things difficult to appreciate and understand .On a theoretical level, I had understood the objective of the project that it was giving us extra serial interfaces. However,

actually feeling the need of having extra serial interfaces and the facing the problems that arose without them, would have probably gone a long way in making things much clearer.

Basic Working of my code:



In this section, I will briefly explain the working of my code. I will do so using I2C. However the same basic structure applies for the driver for SPI as well:

1. In the diagram above you can see the I2C core calling the master xfer function which is a part of the driver that I have written.
2. The master xfer function receives the data from the userland (and in my case) the xfer_one_message function. This function firsts write whether the master wants to read or write data and the slave address to a certain memory location in the shared memory location in the PRU-ICSS. This is read by the firmware which is blocked by a while loop until this event takes place.
3. After this the firmware writes to the certain memory location polled by the driver that it has successfully received the parameters and has bitbanged the address.

4. After this depending on whether the master wants to write or read data, the driver either reads the data from another memory location or writes to it so that the firmware can then bitbang it.

Project Timeline

- During the initial few weeks, I tried to first write a simple firmware that only consisted of MOSI and clock. My aim was to observe the frequency at which this firmware worked and also to familiarize myself with the registers and the architecture of the PRU.
    - Initially I was stuck with understanding how to actually write expressions in my code that would allow me to toggle bits on the __R30 and__R31 register. I also spent some time studying the ARM TRM. I also followed PRU Hands on Lab for examples of firmware for the PRU.
    - The next challenge was to compile the firmware and write a makefile for the same. I adopted a makefile from the examples given in the PRU Code Generation tools and modified it to compile the firmware that I had written for the project.
- After successfully compiling the firmware, I started working on writing a simple char driver in order to send data from the userland and have the firmware successfully bitbang it.
    - I studied about ioremap function and understood iowrite8() and ioread8() functions. I also researched on virtual and real memory.
    - I spent some time researching about the exact memory addresses of the shared memory local to the PRU-ICSS.
- The next step was to write a driver using the SPI subsystem. I initially wrote the driver for a single byte transfer per transaction and later added multibyte transfer. The driver wrote the MOSI data along with parameters into memory locations which was picked up by the firmware. Also, flags were set up in the memory which were polled by the firmware and driver to let them know that data transfer had occurred.
    - Another problem was to figure out a way to call the probe function of the driver. The ideal way was to make the firmware virtio compliant and let

virtio subsystem do the job. However, my mentors suggested that due to the time constraints of GSOC, I should hack the rproc and call the probe function from there. Another option was to write a simple device tree file and let it call the probe function. I chose the latter and with the help of my mentors, wrote a device tree file.

- After completing the SPI , I started coding for I2C. The I2C driver implements most of the features that I mentioned in my proposal. Since, I had figured out the basics while coding for SPI, coding for I2C was mostly about getting the logic and the transfer rate right.
    - 7 bit addressing mode: Since the slave address is a buffer of 16 bits, I had to write code that only read bits 0-6 and ignored the rest.
    - I2C delay: The minimum time that the clock should hold so that the changes are propagated and correctly detected by the slave is 4.7 microseconds. I took a base time of 5 microseconds and accordingly added the delay loop.

SERIAL PERIPHERAL INTERFACE(SPI)

In my proposal, I had outlined the various features that I would be implementing in case of SPI. The features that I was successful in implementing along with those I was not able to achieve/need improvement are as follows:

Features:

1. All the four clock phases and polarity.
2. Option of MSB first transfer or LSB first transfer
3. Multibyte transfer in one transaction
4. Option of CS active high or active low
5. Transfer speeds up to 7 Mhz(limited by bitbanging)

Things that need improvement/that I did not do :

1. I had proposed that I would implement multi slave support by providing more than one CS pins. This required me to design a dtc file and provide multiple CS pins via the num cs parameter. I was not able to write such a dtc file(except for the very basic one that just causes the spi driver to probe and gives it a spidev registration) and then write the necessary code in the firmware due to time constraints of GSOC. The future goal is to have the driver read nodes from a dtc file.
2. I did not use RPMsg in order to transfer messages. I had researched a lot on RPMsg before the coding period started. As I had explained in my proposal as well, I think that in RPMsg, the data buffers are not assigned any fixed memory addresses. Hence the only way to make it work would be to code additional logic into the firmware that would tell that the first set of values it receives are the parameters while all subsequent buffers would contain data to be bitbanged. I felt that the time required and complexity of a such a code would be a lot more than simply using ioread/iowrite to read/write data and flags in the shared memory in the PRU-ICSS. Moreover since PRU had direct access to these addresses, it would only be required to dereference a pointer in the firmware to access the data at these locations. Hence I went ahead with coding my own buffers and set flag locations and did not use RPMsg.
3. A future goal would be to shift to the entire data transfer mechanism to DMA. Currently it is based on ioread/iowrite which pose serious limitations to the amount of data that can transferred at a time from the driver to the firmware.This is so as at one time , the maximum data that I can write to the firmware is 64 bits(with iowrite64()). I had to code additional logic in the driver so as to parse the input buffer and then write to the firmware. After that my driver continuously polls and after getting confirmation then parses the input buffer again to write to the firmware. I believe this is not a very efficient system and takes a lot of processor resources. Moreover as my mentors have repeatedly told me, it is best to avoid polling in the kernel. DMA on the other hand with its own clock, would probably be more efficient and would not consume so many processor resources. With DMA, I would probably pass the entire buffer to the firmware and

let it parse the buffer and bitbang it to MOSI. I did not get the time to research on DMA otherwise I would have implemented DMA in the first place itself.

4. A future goal would be to optimize the algorithm and try to reach speeds of up to 10 Mhz. I believe that this can be done by not checking for MISO in between clock transitions and write separate function for them. Anything in between the clock transition will slow the clock down. Obviously, the clock speed is constrained by the execution speed of the PRU. Also another addition to the project could be to have configurable clock speeds. This could be done by adding appropriate delay in clock transitions depending on parameters.
5. Currently the driver and firmware poll flags set up in the shared memory in the PRU-ICSS to know when they should read the data buffer i.e when data transfer is complete. A future goal would be to add interrupts so that continuous polling in not required.
6. The multibyte transfer is still subject to some tests and improvements that I have to do to the firmware. I have to still have to do some work on the chip selects in order to make the multibyte transfer fully functional.

INTER INTEGRATED CIRCUIT (I2C)

I have accomplished almost all the features that I had proposed for I2C . They are listed below along with future goals to this project:

Features:

1. Implemented master mode. An external NPN transistor would have to employed to pull down SDA and SCL.
2. Implemented Clock stretching so as to allow the slaves to hold down SCL in case they are unable to sample at the speed of the master.
3. Implemented 7 bit addressing mode.
4. Multiple Bytes can be transferred. The firmware implements repeated start condition.

Things that need improvement/that have not been done:

1. I would try and implement 10 bit addressing mode in the future.
2. As with SPI, a future goal would be to shift the entire transfer mechanism to DMA.
3. A future goal would be to have multi-master support.
4. The code for I2C has not been tested on a slave device so far.

Apart from these , I had also mentioned UART as a reach goal in my proposal. I did not get the time to focus on UART during the three months for GSOC. i will definitely try to achieve it as a future goal post GSOC.

Conclusion

I would like to once again thank all my mentors for giving me this opportunity. It's been a pleasure learning from you all and being associated with the BeagleBoard Community. I would also like to thank all my fellow students for helping me out. I think I am very lucky to have found friends and seniors like you(I think I am the youngest amongst all the students) to guide me.

This aim of this project was to provide extra serial interfaces to the BeagleBone without the need to buy extra hardware controllers. This project would make it possible to connect a variety of sensors/shift registers and EEPROM to the BeagleBone. I believe it meets the most of the original aim and implements most of the features that I had proposed. I hope that the project will be helpful to the community at large and cut down on the cost of interfacing various sensors with BeagleBone Black. I hope to continue to work on my project after GSOC ends and improve upon the code that I have written.