Angelo Maldonado-Liu
ECE 372
8/12/2022

Design Log part 1a

go to BBB mux and find the correct pin

control module base address is 0x44E1_0000

uart1_rtsn is I2C2_SCL offset 97C h
we write 0x2B at 0x44E1_097C

uart1_ctsn is I2C2_SDA offset 978 h
we write 0x2B at 0x44E1_0978

44h cm_per_i2c2 base address is 0x44E0_0000

can turn on the clock by writing 0x2 into 0x44E0_0000 + 44 h

base address of I2C2 registers is 0x4819_C000
for SCLH ignore the - sign
write 10 at 0x4819_C0B8

I2C_PSC offset B0h
want divide by 4, so we subtract 1 and get 3. so we write 0x3

for the SCLL, we want tLOW to be the period of 400kbps
write 8 at 0x4819_C0B4

we do not care what the beagle bone's address is

write this to I2C_CON 0x8600 which is address 0x4819_C0A4

find slave address of PC8
we need at least 3 bytes to send address, register address, and then data.
this is wrong, we send 2 bytes because of I2C_SA

poll busy bus, which is address 0x4819_C024
we mask all bits except bit 12, so we and the read value with 0x1000 then check if the value is equal to 0, if it is, then we can continue and send info, if not, it is busy.

to start transfer we write 0x8603 to 0x4819_C0A4
because the dcount in I2C_cnt automatically sends a stop condition once the dcount reaches 0.
also write to I2C_SA which is 0x4819_C0AC, and write the address which we need to find
address is 0x100_0000

then we check the I2C_IRQSTATUS_RAW register at bit 4, so we have to and the read value with 0x8, then check if it is equal to 0x8. if it is, then we write data to I2C_data

I2C_data address is 0x4819_C09C, write whatever data needs to go in here.

do not need to use draining feature

when checking I2C_IRQSTATUS_RAW, we have to mask the bits other than bit 12 to check BB

High level algorithm

1. write 0x2B to set I2C2_SCL
2. write 0x2B to set I2C2_SDA
3. write 0x2 to CM_PER_I2C2_CLKCTRL to turn on clock
4. write A to I2C_SCLH to set SCLH
5. write 8 at I2C_SCLLto set SCLL
6. write 0x8600 to I2C_CON
7. read I2C_IRQSTATUS_RAW
8. mask bits by using or 0x1000
9. compare with 0x1000
10. if equal then it is busy
11. if not equal, then continue
12. write 0x3 to I2C_CNT to set the dcount to 2
13. write 0x8603 to I2C_CON to start the transfer
14. check I2C_IRQSTATUS_RAW bit 4 with 1
15. if equal write data to I2C_DATA
16. repeat steps 7 - 15 2 times to send all data

low level algorithm

1. write 0x2B at 0x44E1_097C to set I2C2_SCL
2. write 0x2B at 0x44E1_0978 to set I2C2_SDA
3. write 0x2 into 0x44E0_0044 to turn on clock
4. write A into 0x4819_C0B8 to set SCLH
5. write 8 at 0x4819_C0B4 to set SCLL
6. write 0x8600 to 0x4819_C0A4 which sets the settings for I2C_CON
7. poll busy bus, which is address 0x4819_C024
    a. read at address 0x4819_C024
    b. mask bits by using or 0x1000
    c. compare with 0x1000
        i. if equal then it is busy
        ii. if not equal, then continue
8. write 0x2 to 0x4819_C098 to set the dcount to 2
9. write 0x8603 to 0x4819_C0A4 to start the transfer
10. read 0x4819_C024 and check I2C_IRQSTATUS_RAW
    a. mask bits by using 0x10;
    b. compare with 0x10;
        i. if equal write data to 0x4819_C09C
11. repeat steps 7 - 10 2 times to send all data

Design Log part 1b

write to PCA 0x00000000 and write the value of nothing because default is already correct, so dont actually write or write 0x11

write at address FE the value of 0x79 to set clock speed to 50hz

next write to PCA 0x00 and write the value 0x81

write to 0x01 value of 0x00100
hex is 0x04

write to 0x42 write value 0x0
write to 0x43 write value 0x0
this says always on from the beginning

wait 50 loops

write to 0x00000044 write value 0x0
write to 0x00000045 write value 0x4

High level algorithm

1. write 0x2B to set I2C2_SCL
2. write 0x2B to set I2C2_SDA
3. write 0x2 to CM_PER_I2C2_CLKCTRL to turn on clock
4. write A to I2C_SCLH to set SCLH
5. write 8 at I2C_SCLLto set SCLL
6. write 0x8600 to I2C_CON
7. next send data
    a. read I2C_IRQSTATUS_RAW
    b. mask bits by using or 0x1000
    c. compare with 0x1000
    d. if equal then it is busy
    e. if not equal, then continue
    f. write 0x3 to I2C_CNT to set the dcount to 2
    g. write 0x8603 to I2C_CON to start the transfer
    h. check I2C_IRQSTATUS_RAW bit 4 with 1
    i. if equal write data to I2C_DATA
8. repeat step 7 2 more times to send 1 set of data
    a. sending in 0x00, 0x11
9. repeat step 7 3 times to send
    a. address
    b. 0xFE
    c. 0x79
10. repeat step 7 3 times to send
    a. address
    b. 0x00
    c. 0x81
11. repeat step 7 3 times to send
    a. address
    b. 0x06
    c. 0x00
12. repeat step 7 3 times to send
    a. address
    b. 0x07
    c. 0x10

13. wait for a bit
14. repeat step 7 3 times to send
    a. address
    b. 0x8
    c. 0x0
15. repeat step 7 3 times to send
    a. address
    b. 0x09
    c. 0x14

low level algorithm

1. write 0x2B at 0x44E1_097C to set I2C2_SCL
2. write 0x2B at 0x44E1_0978 to set I2C2_SDA
3. write 0x2 into 0x44E0_0044 to turn on clock
4. write A into 0x4819_C0B8 to set SCLH
5. write 8 at 0x4819_C0B4 to set SCLL
6. write 0x8600 to 0x4819_C0A4 which sets the settings for I2C_CON
7. send data
    a. poll busy bus, which is address 0x4819_C024
        i. read at address 0x4819_C024
        ii. mask bits by using or 0x1000
        iii. compare with 0x1000
            1. if equal then it is busy
            2. if not equal, then continue
    b. write 0x2 to 0x4819_C098 to set the dcount to 2
    c. write 0x8603 to 0x4819_C0A4 to start the transfer
    d. read 0x4819_C024 and check I2C_IRQSTATUS_RAW
        i. mask bits by using 0x10;
        ii. compare with 0x10;
            1. if equal write data to 0x4819_C09C
8. repeat step 7 2 times to send all data
    a. sending in 0x00, 0x11 at address 0x4819_C024
9. repeat step 7 3 times to send at address 0x4819_C024
    a. address
    b. 0xFE
    c. 0x79
10. repeat step 7 3 times to send at address 0x4819_C024
    a. address
    b. 0x00
    c. 0x81

11. repeat step 7 3 times to send at address 0x4819_C024
    a. address
    b. 0x06
    c. 0x00
12. repeat step 7 3 times to send at address 0x4819_C024
    a. address
    b. 0x07
    c. 0x10
13. wait for a bit
14. repeat step 7 3 times to send at address 0x4819_C024
    a. address
    b. 0x8
    c. 0x0
15. repeat step 7 3 times to send at address 0x4819_C024
    a. address
    b. 0x09
    c. 0x14

Design Log part 2

in order to turn the 20 ms clock to 1.5 ms, then of the 4096 chunks, i need 307 chunks, which in hex is 0x133

in order to turn the 20 ms clock to 1.75 ms, then of the 4096 chunks, i need 358 chunks, which in hex is 0x166

in order to turn the 20 ms clock to 1.25 ms, then of the 4096 chunks, i need 256 chunks, which in hex is 0x100

in order to turn the 20 ms clock to 2 ms, then of the 4096 chunks, i need 410 chunks, which in hex is 0x19A

in order to turn the 20 ms clock to 1.00 ms, then of the 4096 chunks, i need 205 chunks, which in hex is 0xCD

calculate the timer of each of the timers. the first is 1s, which comes out to be 0xFFFFC003
timer for 0.5 seconds is FFFFBFFF

the address of the PCA is 0x40 if unsoldered.

dont forget to clear interrupts

dont forget to send the setup for the PCA, that will make it not work most of the time. I forgot that I had commented out the setup data for some reason.

part 2 high level algorithm

1. Set register 13 = to USR_STACK variables
2. Put 0x100 into R13 for our stack
3. Set initial GPIO values
4. set output enable
5. wakeup timer 5
6. set clock speed
7. software reset
8. clear irqs
9. enable overflow IRQ
10. reset INTC
11. unmast INTC_TINT5
12. write 0x2B to set I2C2_SCL
13. write 0x2B to set I2C2_SDA
14. write 0x2 to CM_PER_I2C2_CLKCTRL to turn on clock
15. write A to I2C_SCLH to set SCLH
16. write 8 at I2C_SCLLto set SCLL
17. write 0x8600 to I2C_CON
18. write 0x40 to put in secondary address
19. send data 0x11 to 0x00
20. send data 0xfe to 0x79
21. send 0x81 to 0x00
22. go to forever loop and wait for interrupt

int handler

1. check 0x20000000 to see if it is our interrupt
2. clear timer5 interrupts, write 0x7
3. clear NEWIRQ bit in INTC, write 0x1

4. check which setting we are in, and then choose the next in the order.

Send data functions

5. next send data
    a. read I2C_IRQSTATUS_RAW
    b. mask bits by using or 0x1000
    c. compare with 0x1000
    d. if equal then it is busy
    e. if not equal, then continue
    f. write 0x3 to I2C_CNT to set the dcount to 2
    g. write 0x8603 to I2C_CON to start the transfer
    h. check I2C_IRQSTATUS_RAW bit 4 with 1
    i. if equal write data to I2C_DATA
6. change the data you send depending on what setting we are on in the FSM
7. wait for interrupt

part 2 low level algorithm

1. Set register 13 = to USR_STACK variables
2. Put 0x100 into R13 for our stack
3. Set initial GPIO1 initialization by writing 0x2 to 0x44E000AC
4. set initial GPIO values by writing 0x01E00000 to 0x4804C190
5. set output enable by writing 0xFE1FFFFF to 0x4804C134
6. wakeup timer 5 by writing 0x2 to 0x44E000EC
7. set clock speed by writing 0x2 to 0x44E00518
8. software reset by writing 0x1 to 0x48046010
9. clear irqs by writing 0x7 to 0x48046028
10. enable overflow IRQ by writing 0x2 to 0x4804602C
11. reset INTC by writing 0x2 to 0x48200010
12. unmast INTC_TINT5 by writing 0x20000000 to 0x482000C8
13. write 0x2B to 0x44E00044
14. write 0x2B to 0x44E1097C
15. write 0x2 to 0x44E10978
16. write A to 0x4819C0B8 to set SCLH
17. write 8 at 0x4819C0B4 to set SCLL
18. write 0x3 to 0x4819C0B0

19. write 0x8600 to 0x4819C0A4
20. write 0x40 to 0x4819C0AC
21. send data 0x11 to 0x00
22. send data 0xfe to 0x79
23. send 0x81 to 0x00
24. set timer by writing 0xFFFFFFFA to 0x4804603C
25. start timer by writing 0x1 to 0x48046038
26. go to forever loop and wait for interrupt

int handler

1. check 0x20000000 with the value from 0x482000D8 to see if it is our interrupt
2. clear timer5 interrupts, write 0x7 to 0x48046028
3. clear NEWIRQ bit in INTC, write 0x1 to 0x48200048
4. check which setting we are in, and then choose the next in the order.

Send data functions

1. next send data
    a. poll busy bus, which is address 0x4819_C024
        i. read at address 0x4819_C024
        ii. mask bits by using or 0x1000
        iii. compare with 0x1000
            1. if equal then it is busy
            2. if not equal, then continue
    b. write 0x2 to 0x4819_C098 to set the dcount to 2
    c. write 0x8603 to 0x4819_C0A4 to start the transfer
    d. read 0x4819_C024 and check I2C_IRQSTATUS_RAW
        i. mask bits by using 0x10;
        ii. compare with 0x10;
            1. if equal write data to 0x4819_C09C
2. change the data you send depending on what setting we are on in the FSM
    a. for setting 0
        i. turn on LED by writing 0x00200000 to 0x4804C194
        ii. 0x00 to 0x26
        iii. 0x00 to 0x27
        iv. 0x33 to 0x28
        v. 0x1 to 0x29
    b. for setting 1 (45)
        i. turn on LED by writing 0x00400000 to 0x4804C194
        ii. 0x00 to 0x26
        iii. 0x00 to 0x27
        iv. 0x66 to 0x28
        v. 0x1 to 0x29
    c. for setting 2 (-45)
        i. turn on LED by writing 0x00800000 to 0x4804C194

        ii. 0x00 to 0x26
        iii. 0x00 to 0x27
        iv. 0x00 to 0x28
        v. 0x1 to 0x29
    d. for setting 3 (90)
        i. turn on LED by writing 0x01000000 to 0x4804C194
        ii. 0x00 to 0x26
        iii. 0x00 to 0x27
        iv. 0x9A to 0x28
        v. 0x1 to 0x29
    e. for setting 4 (-90)
        i. turn on LED by writing 0x01E00000 to 0x4804C190
        ii. 0x00 to 0x26
        iii. 0x00 to 0x27
        iv. 0xCD to 0x28
        v. 0x00 to 0x29
3. wait for interrupt

Design Log Extra Credit

When determining the amount of threads that can be created using the beagle bone, the beagle bone has 512 MB of DRAM, and each thread has a stack size of a minimum of 2 MB, so this means we have a theoretical maximum thread count of 256.

in order to get threading to work, you need to include the pthread.h library. Also it is possible to download the libraries and run the code from the linux side of the beaglebone

like with C and C++, in order to create the threads, you need to use
pthread_t thread;
pthread_create(&thread, NULL, function, function_args);

this creates a thread with the given pointer.

if we want to control multiple motors and send data, we need to use some form of locking in order to get it to work. From what I have learned, it would be easiest to implement the mutex

lock, as I already know how to use that one. It will prevent any of the other threads from accessing the sending data part of the code, and will allow the code to send data while in the while loop of the code.

in order to use the mutex lock, we need a
pthread_mutex_t variable

and we need to use the commands:

pthread_mutex_lock(variable_name);
some code
pthread_mutex_unlock(variable_name);

inside the locks is where we would put the send data code, and we only need to lock individual send data functions, because we need to send 2 bytes of data along with the SA in order to send the whole command, and so between sending the whole command, we can alternate between the threads, so we can sort of set up multiple motors at the same time.

in order to do this, we need to set up the 4 timers that exist in the beaglebone, which means we can only really use 4 threads. we would modify the code of the current functions, and when we do the interrupts, we check to see which of the 4 timers flagged the interrupt, this would then send us down the path to which thread needs to activate and send data down to the motor that the thread dictates. Unfortunately based on what I understand, if a

the highlighted doesn't seem like it would work as the threads most likely would be overwritten by a new interrupt from the clocks of the other threads and would just stop each other.

What would make more sense with using this with my current capabilities would be to use the standard timer, and send the data of all the threads at the same time, using the locking to prevent them overwriting when they are sending their 2 bytes. By doing this, if there is any dead time between the commands, it might speed up the process of sending the data. and we could have each of the threads tied to a different motor of the PCA, or we could use multiple PCAs and have each thread communicate with a different one. The advantage of doing it this way is that we are no longer limited to just 4 threads, we can use up to a theoretical 256, which means we would have 256 different devices we are communicating to (though the efficiency of that considering the beagle bone only has 2 processors, means that beyond 2, the efficiency is up in the air.)