ECEN 5823-001
Internet of Things Embedded Firmware

Lecture #13

09 October 2018

Be Boulder.

University of Colorado Boulder

Agenda

- Class Announcements
- Reading assignment
- Mid-term structure
- Firmware best practices
- Bluetooth Low Energy / Smart

Class Announcements

- Quiz #7 is due at 11:59 on Sunday, October 7th, 2018
- ~~Homework #4: BLE Server + MITM + LCD due on Sunday, October 7th, at 11:59pm~~
- Homework #5: Client flowchart due on Wednesday, October 10th, at 11:59pm
- Homework #6: Client-Server due on Wednesday, October 17th, at 11:59pm
- Mid-term in class on Thursday the 18th
  - Attendance is required for on-campus students

Reading Assignment

ECEN5823-001, -001B – Reading List
Internet of Things Embedded Firmware
Week 7

Note: The material in this reading as well as all lectures and assignments may be on this week's quiz, quiz 7.

- "How to be a Star Engineer," by Robert E. Kelley
- Bluetooth blog: Bluetooth Mesh Networking: Friendship by Kei Ren
  - http://blog.bluetooth.com/bluetooth-mesh-networking-series-friendship
- Bluetooth blog: Management of Devices in a Bluetooth Network by Martin Wooley
  - https://blog.bluetooth.com/management-of-devices-bluetooth-mesh-network

careers

by Robert E. Kelley,
Carnegie Mellon University

Engineers from the best companies helped researchers to dispel the myths about star performers and uncover the surprising secrets of stellar achievement

HOW TO BE A STAR ENGINEER

ILLUSTRATIONS: NOAH WOODS

Mid-term structure

- Attendance is a must for on-campus students – Thursday the 18th
  - A zero will be recorded for on-campus students that do not take it in class
  - Be on-time!
- CU Honor Code is in affect
- A single sheet of 8x11 paper, both sides, of notes is allowed
  - These notes must be of your own
  - They will be handed in at the end of the mid-term
  - If the notes are copied or determined not individual work, a zero will be recorded for the mid-term and a CU Honor code violation will be initiated resulting in a “F” in the course
- Distant students will have until 11:59pm on Friday the 19th to complete the exam

Mid-term structure

- Mid-term will comprise of 60 questions taken via Canvas
  - 35 questions from the Quiz 1 thru 7 question banks
  - 25 questions from a mid-term question bank
- If there is a canvas question issue such as no way to input an answer, please submit the issue to me after the exam via slack.
  - I will make a manual adjustment
- You can ask for clarification from the mid-term proctor

Quiz 6 review

In Bluetooth Smart, privacy is the ability to prevent others from _____ (single word answer) by the devices that you are carrying.

Quiz 6 review

The parameter which defines the number of consecutive connection events which the Bluetooth Smart slave is not required to listen to the master and thus turn off its radio and possible go to sleep.

- connEvent
- SlaveLatency
- connInterval
- connLatency

Quiz 6 review

List the sequence in pairing two BLE devices

Announce their input/output capabilities	[Choose]
Agree upon a Temporary Key	[Choose]
Short Term Key is obtained by both devices	[Choose]
the 128-bit Long-Term Key (LTK); the Connection Security Resolving Key (CSRK), the Identity Resolving Key (IRK).	[Choose]

Quiz 6 review

When two BLE devices are reconnecting, either device can initiate encryption. Each and every data packet that is transmitted from then will incorporate the following?

- MIC
- Packet size
- Header
- CRC

Quiz 6 review

Select the Bluetooth Smart authentication method based on the following:

device 1: Keyboard only

device 2: No input and no output

- Passkey Entry
- Just Works

Quiz 6 review

Every Bluetooth connection event starts with a transmission of a package by the master.

- ○ True
- ○ False

Quiz 6 review

For a Bluetooth Smart application where somewhat a real-time response is required, less than 500ms, which settings would you select? (select all that apply)

- [ ] High connInterval
- [ ] Low connInterval
- [ ] Low SlaveLatency
- [ ] High SlaveLatency

Quiz 6 review

The initiator coordinates the medium access by providing information on the Time Division Multiple Access (TDMA) scheme and provides the slave with the frequency hopping algorithm during which transmission?

- Connection event
- Connection request
- Advertising event

Quiz 6 review

The maximum number of slaves that a BLE master can connect to is 7.

- True
- False

Quiz 6 review

Which algorithms are used to protect against the man-in-the-middle attack during BLE pairing to obtain the Temporary Key?

- [ ] Randomization
- [ ] Out of Band
- [ ] Passkey Entry
- [ ] Just Works

Quiz 6 review

Maximum BLE L2CAP payload size?

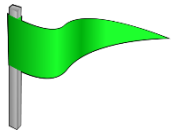
- 23
- 8
- 47
- 32

Quiz 6 review

Which modulation scheme does Bluetooth BLE support?

- Binary Phase Shift Keying
- Frequency Shift Keying
- Amplitude Shift Keying
- Gaussian Frequency Shift Keying
- Phase Shift Keying

Firmware best practices



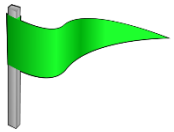
- How to make an operation such as a read-modify-write atomic?
  - Interrupts must be disabled before the global variable access
  - And, then interrupts must be enabled after the global variable access

  - What are the downsides of disabling interrupts while an atomic operation is in progress?
    - Increases the system latency (the time to respond to an interrupt)
    - Can have a negative affect in Real-Time Systems

Firmware best practices

- State Machines can keep your system organized while you have more than one task
- A state machine will have a defined behavior based on the following:
  - "context" (internal state)
  - And, the environment, input variables

- A visual representation of a state machine is a flow chart
  - A next state should be defined for every possible combination of input variables or events, even if it is assumed not to be possible
  - The "not possible" event could specify to remain in the same state or possibly generate a system fault

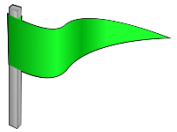


Firmware best practices

- State Machines can have global variables that define the current state of the State Machine
  - For example, in the Lifting Accelerometer, there is a state machine that sets the sleep mode of the Leopard Gecko
    - The state machine global variables are controlled by two routines
      - BlockSleepMode()
      - And,
      - UnBlockSleepMode()
    - If the global variables are not set or cleared correctly, the next state of the sleep() routine could put the Leopard Gecko in an Energy Mode that a required peripheral cannot support

Firmware best practices

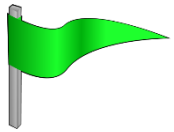
- The solution to insuring that the state bits do not change with uncertainty, is to make the change of these state bits atomic



```
Void BlockSleepMode(int EMlevel) {
        INT_Disable();
        SleepModeBlockLeve (EMlevel) ++;
        INT_Enable();
}
```

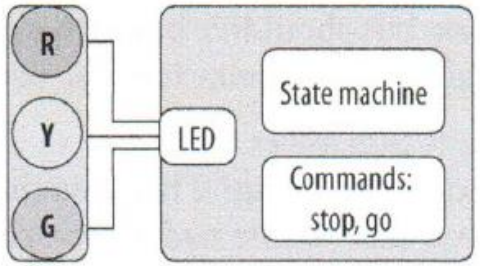
Firmware best practices

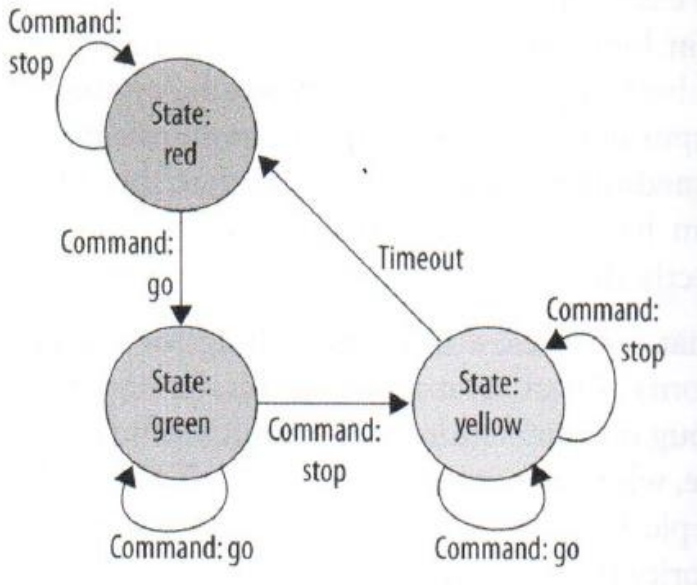
- State Machines make code testing and verification easier
  - Each state has a fixed possible combinations based on the events to the current context, internal state
  - Having a manageable possible combination of next states, enables code verification at the design, testing, and verification stages
    - Significantly reducing possible errors
  - During the design phase, check whether the state changes based on events match the flow chart of the state machine

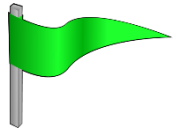


Firmware best practices

- Flow diagram example:







Firmware best practices

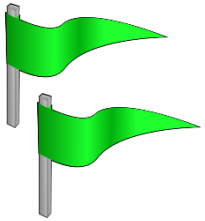
Current State	Event (1 = Go, 0 = Stop)	Time Out	Next State
00 (Red)	0 (Stop)	0 (False)	00 (Red)
00 (Red)	0 (Stop)	1 (True)	00 (Red)
00 (Red)	1 (Go)	0 (False)	00 (Go)
00 (Red)	1 (Go)	1 (True)	00 (Go)
01 (Green)	0 (Stop)	0 (False)	00 (Yellow)
01 (Green)	0 (Stop)	1 (True)	00 (Yellow)
01 (Green)	1 (Go)	0 (False)	01 (Green)
01 (Green)	1 (Go)	1 (True)	01 (Green)
11 (Yellow)	0 (Stop)	0 (False)	11 (Yellow))
11 (Yellow)	0 (Stop)	1 (True)	00 (Red)
11 (Yellow)	1 (Go)	0 (False)	11 (Yellow)
11 (Yellow)	1 (Go)	1 (True)	00 (Red)

Firmware best practices

- A **watchdog timer** (sometimes called a *computer operating properly* or *COP* timer, or simply a *watchdog*) is an electronic timer that is used to detect and recover from computer malfunctions.
  - During normal operation, the computer regularly resets the watchdog timer to prevent it from elapsing, or "timing out".
  - If, due to a hardware fault or program error, the computer fails to reset the watchdog, the timer will elapse and generate a timeout signal.
  - The timeout signal is used to initiate corrective action or actions.
  - The corrective actions typically include placing the computer system in a safe state and restoring normal system operation. (source: Wikipedia)

Firmware best practices

- The watchdog timer's goal is to fail in a save manner if it fails
  - Goal is to have the watchdog service routing in a part of the code that is demonstrating that all systems are running as expected
  - Generally, this is the main loop
  - For board bring up, watchdog timer can cause issues in the debugger at or after bring up
  - Its good practice to have a watchdog timer to fail in a safe manner
  - And, its good practice to disable the watchdog timer during board bring up



Firmware best practices

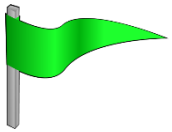
- Data Handling
  - Data-driven systems
    - Get data
    - Process it
    - Do something with the results
    - Repeat
  - Implementing data-driven system is straight forward because the process of the data is repetitive
    - Example
      - A producer of data such as an ambient light sensor
      - The consumer, MCU, processes the data
      - Sending the data to the greater system
      - Repeat
  - Most system have elements of both an event-driven and data-driven system

Firmware best practices

- In a data-driven system, if data is produced at one data-rate while it is consumed at a different-rate, what data structure or technique is commonly used?
- Circular Buffers is a key implementation among data-driven systems
  - They are First In First Out (Buffers)
  - A producer puts data in the circular buffer at some rate
  - And, the consumer take data out of the circular buffer at a rate equivalent or faster than the producers put it in
- The Circular Buffer needs to keep track of the next elements
  - Where to put, write, the next buffer
    - Circular_Write_Ptr
  - Where to take, read, the current buffer
    - Circular_Read_Ptr

Firmware best practices

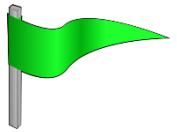
- Both the producers and consumers are accessing the circular buffer, the buffer is a global variable
  - To prevent data race contentions to the buffers, the read and write pointers are used and must not point to the same buffer location
- Also, both Circular_Write_Ptr and Circular_Read_Ptr are shared between multiple tasks, thus, the pointers are global variables



  - With the pointers being global variables, the update of these variables must be atomic
- The pointers being atomic, insures that the data circular buffers are atomic as well

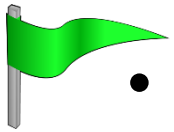
Firmware best practices

- Circular buffers
  - The buffer is empty when the read pointer is equal to the write pointer
  - But, when the buffer is full, the read pointer is equal to the write pointer as well due to being a circular buffer
    - A common workaround is to call the circular buffer full when the write buffer is one away from the read pointer

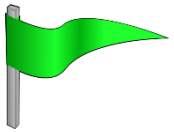


  - The number of elements in the buffer for ease of use should be the power of two
    - Enabling a simple mask to allow the pointers to rollover
      - Example: 4 elements in the circular buffer
      - Pseudo code:
        - Write_Ptr = (Write_Ptr++) & 0x03;
        - Will result in pointer values of 00, 01, 10, 11, 00, 01, 10, 11 ……

Firmware best practices

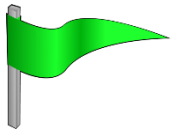


- Circular buffers with their "atomic" pointers are a very good way of sharing streams of data between two or more tasks



  - Try to size the elements in the buffer to a reasonable size that will minimize the chances of possible buffer overflow conditions

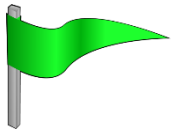
Firmware Best Practices (Circular Buffer ex.)



```
void Atomic_Print(char *outputstring){
    INT_Disable();

    TX_Buffer_Write_Ptr = (TX_Buffer_Write_Ptr + 1) % NUM_TX_Buffers;
    strncpy(BluetoothTransmitBuffer[TX_Buffer_Write_Ptr],"",BluetoothMaxTXBufferSize);

strncpy(BluetoothTransmitBuffer[TX_Buffer_Write_Ptr],outputstring,BluetoothMaxTXBufferSize);
    if (!TransmittingData) Print_Bluetooth_Out();
```



```
    INT_Enable();
}
```

Firmware Best Practices (Circular Buffer ex.)

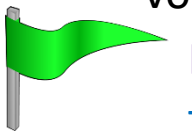
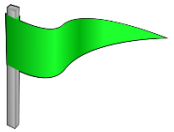
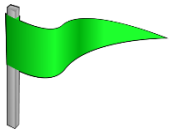
```
void LEUART0_TX_dmaTransferDone(unsigned int channel, bool primary, void *user) {

INT_Disable();

    DMA->IFC = LEUART0_TX_DMA_Channel;
    LEUART0->CTRL &= ~LEUART_CTRL_TXDMAWU;
    strncpy(BluetoothTransmitBuffer[TX_Buffer_Transmit_Ptr],"",BluetoothMaxTXBufferSize);  //
after transmit, initialize to ""
    TransmittingData=false;
    if (TX_Buffer_Transmit_Ptr != TX_Buffer_Write_Ptr) Print_Bluetooth_Out();

    INT_Enable();
}
```

Firmware Best Practices (Circular Buffer ex.)

```
void Print_Bluetooth_Out(void){
    INT_Disable();
    TransmittingData = true;
    TX_Buffer_Transmit_Ptr = (TX_Buffer_Transmit_Ptr + 1) % NUM_TX_Buffers;
    str_length = strlen(BluetoothTransmitBuffer[TX_Buffer_Transmit_Ptr]);
    str_ptr = &BluetoothTransmitBuffer[TX_Buffer_Transmit_Ptr][0];

    LEUART0->CTRL |= LEUART_CTRL_TXDMAWU;
    DMA_ActivateBasic(
        LEUART0_TX_DMA_Channel,       // LEUART0 transmit channel
        true,                         // Use primary descriptors
        false,                        // No burst mode, LEUART0 does not support it
        (void *)&LEUART0->TXDATA,
        (void *)str_ptr,              // Beginning of output string
        str_length - 1);              // Transmit length
    INT_Enable();
```

Firmware Best Practices

- Int a;
- Are these three expression equivalent in c-code? NO

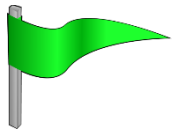
a = (5/2) * 4;	a = 5 * (4/2);	a = (5 * 4) / 2;
a = (2) * 4;	a = 5 * 2;	a = 20 / 2;
a = 8;	a = 10;	a = 10;

Integer Math

Firmware Best Practices

- Another example
- In “real” math, (5/2) * 3 = 7.5
- In Integer math, ordering matters towards precision. What operation in the above example should be done first?

```
a = (5/2) * 3;        a = 5 * (3/2);        a = (5 * 3) / 2;
a = (2) * 3;          a = 5 * 1;            a = 15 / 2;
a = 6;                a = 5;                a = 7;
```



If staying strictly in Integer Math, always do the multiplication first. Will limit the loss of accuracy

Firmware Best Practices

- Speeding up your Math
  - Know your compiler and processor
  - Additions and Subtractions are fast
  - Division is very slow, and floating point math is “dead slow”

Cortex-M3: Integer vs floating point addition

C-code for integer addition:

```
int ramBufferData[BufferSize];
int Summation;

Summation = 0;
for(j=0;j<BufferSize;j++) {
    Summation = Summation + ramBufferData[j];
}
```

Assembly code equivalent:

```
Summation = Summation + ramBufferData[j];
00005450:   ldr     r3,[pc,#0x7c] ; 0x54cc
00005452:   ldr     r2,[sp,#0x1c]
00005454:   ldrh.w  r3,[r3,r2,lsl #1]
00005458:   ldr     r2,[sp,#0x18]
0000545a:   add     r3,r2
0000545c:   str     r3,[sp,#0x18]
```

6 Assembly Instructions

88 Assembly Instuctions!

Cortex-M3: Integer vs floating point addition

C-code for float addition:

```
int ramBufferData[BufferSize];
float Summation;

Summation = 0;
for(j=0;j<BufferSize;j++) {
    Summation = Summation + ramBufferData[j];
}
```

Assembly code equivalent:

```
Summation = Summation + ramBufferAdcData[j];
00005452:  ldr     r3,[pc,#0x90] ; 0x54e0
00005454:  ldr     r2,[sp,#0x1c]
00005456:  ldrh.w  r3,[r3,r2,lsl #1]
0000545a:  mov     r0,r3
0000545c:  bl      0x00005804
00005460:  mov     r3,r0
00005462:  ldr     r0,[sp,#0x18]
00005464:  mov     r1,r3
00005466:  bl      0x0000569c
0000546a:  mov     r3,r0
0000546c:  str     r3,[sp,#0x18]
```

```
00005804:  ands    r3,r0,#0x80000000
00005808:  it      mi
0000580a:  rsbs    r0,r0,#0
0000580c:  movs.w  r12,r0
00005810:  it      eq
00005812:  bx      lr
00005814:  orr     r3,r3,#0x4b000000
00005818:  mov     r1,r0
0000581a:  mov.w   r0,#0x0
0000581e:  b       0x000000000000585a
0000585a:  sub.w   r3,r3,#0x800000
0000585e:  clz     r2,r12
00005862:  subs    r2,#0x8
00005864:  sub.w   r3,r3,r2,lsl #23
00005868:  blt     0x000000000000588c
0000586a:  lsl.w   r12,r1,r2
0000586e:  add     r3,r12
00005870:  lsl.w   r12,r0,r2
00005874:  rsb.w   r2,r2,#0x20
00005878:  cmp.w   r12,#0x80000000
0000587c:  lsr.w   r2,r0,r2
00005880:  adc.w   r0,r3,r2
00005884:  it      eq
00005886:  bic     r0,r0,#0x1
0000588a:  bx      lr
```

```
0000569c:  lsls    r2,r0,#1
0000569e:  itttt   ne
000056a0:  lsls.w  r3,r1,#1
000056a4:  teq     r2,r3
000056a8:  mvns.w  r12,r2,asr #24
000056ac:  mvns.w  r12,r3,asr #24
000056b0:  beq     0x0000000000005788
000056b2:  lsr.w   r2,r2,#24
000056b6:  rsbs    r3,r2,r3,lsr #24
000056ba:  itttt   gt
000056bc:  adds    r2,r2,r3
000056be:  eors    r1,r0
000056c0:  eors    r0,r1
000056c2:  eors    r1,r0
000056c4:  it      lt
000056c6:  rsbs    r3,r3,#0
000056c8:  cmp     r3,#0x19
000056ca:  it      hi
000056cc:  bx      lr
000056ce:  tst     r0,#0x80000000
000056d2:  orr     r0,r0,#0x800000
000056d6:  bic     r0,r0,#0xff000000
000056da:  it      ne
000056dc:  rsbs    r0,r0,#0
000056de:  tst     r1,#0x80000000
000056e2:  orr     r1,r1,#0x800000
000056e6:  bic     r1,r1,#0xff000000
000056ea:  it      ne
000056ec:  rsbs    r1,r1,#0
000056ee:  teq     r2,r3
```

```
000056f2:  beq     0x0000000000005774
000056f4:  sub.w   r2,r2,#0x1
000056f8:  asr.w   r12,r1,r3
000056fc:  adds.w  r0,r0,r12
00005700:  rsb.w   r3,r3,#0x20
00005704:  lsl.w   r1,r1,r3
00005708:  and     r3,r0,#0x80000000
0000570c:  bpl     0x0000000000005714
0000570e:  rsbs    r1,r1,#0
00005710:  sbc.w   r0,r0,r0,lsl #1
00005714:  cmp.w   r0,#0x800000
00005718:  bcc     0x0000000000005742
0000571a:  cmp.w   r0,#0x1000000
0000571e:  bcc     0x000000000000572e
0000572e:  cmp.w   r1,#0x80000000
00005732:  adc.w   r0,r0,r2,lsl #23
00005736:  it      eq
00005738:  bic     r0,r0,#0x1
0000573c:  orr.w   r0,r0,r3
00005740:  bx      lr
```

Firmware Best Practices

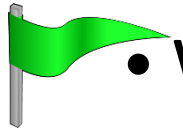
- Speeding up your Math
  - Know your compiler and processor
  - Additions and Subtractions are fast
  - Division is very slow, and floating point math is “dead slow”
  - Many processors have hardware support for multiplication

Firmware Best Practices

- Speeding up your Math
  - As the previous example, a single line of code can explode into many assembly instructions if the hardware does not support in hardware the operation
  - Division is the most common basic operation not supported in hardware
  - Modulo math is very helpful in circular buffers, used by the write and read pointers to wrap around the pointer values
  - What type of common operation is modulo math related to?
    - %, modulo operative, is actually a division

Firmware Best Practices

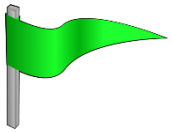
- Speeding up your Math
  - Write_Ptr = Write_Ptr++ % 0x04;
    - Costly in energy, time, and possibly code space
  - Modulo math can be replaced with a very fast and possible a single or two assembly instruction
  - What fast and low energy operation can we use instead of module math?



  - Write_Ptr = Write_Ptr++ & 0x03;
    - Best practice is to replace modulo operations with bit-wise "AND" operation where possible

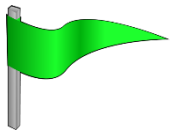
Firmware Best Practices

- Speeding up your Math
  - Multiplication and Divisions by power of 2 are usually single operations
    - How can power of 2 multiplications and divisions be a single operation?
    - Multiplication, shift to the left
    - Division, shift to the right
  - Where possible, keep constants to the power of 2 to enable the above multiplication and division by shifting



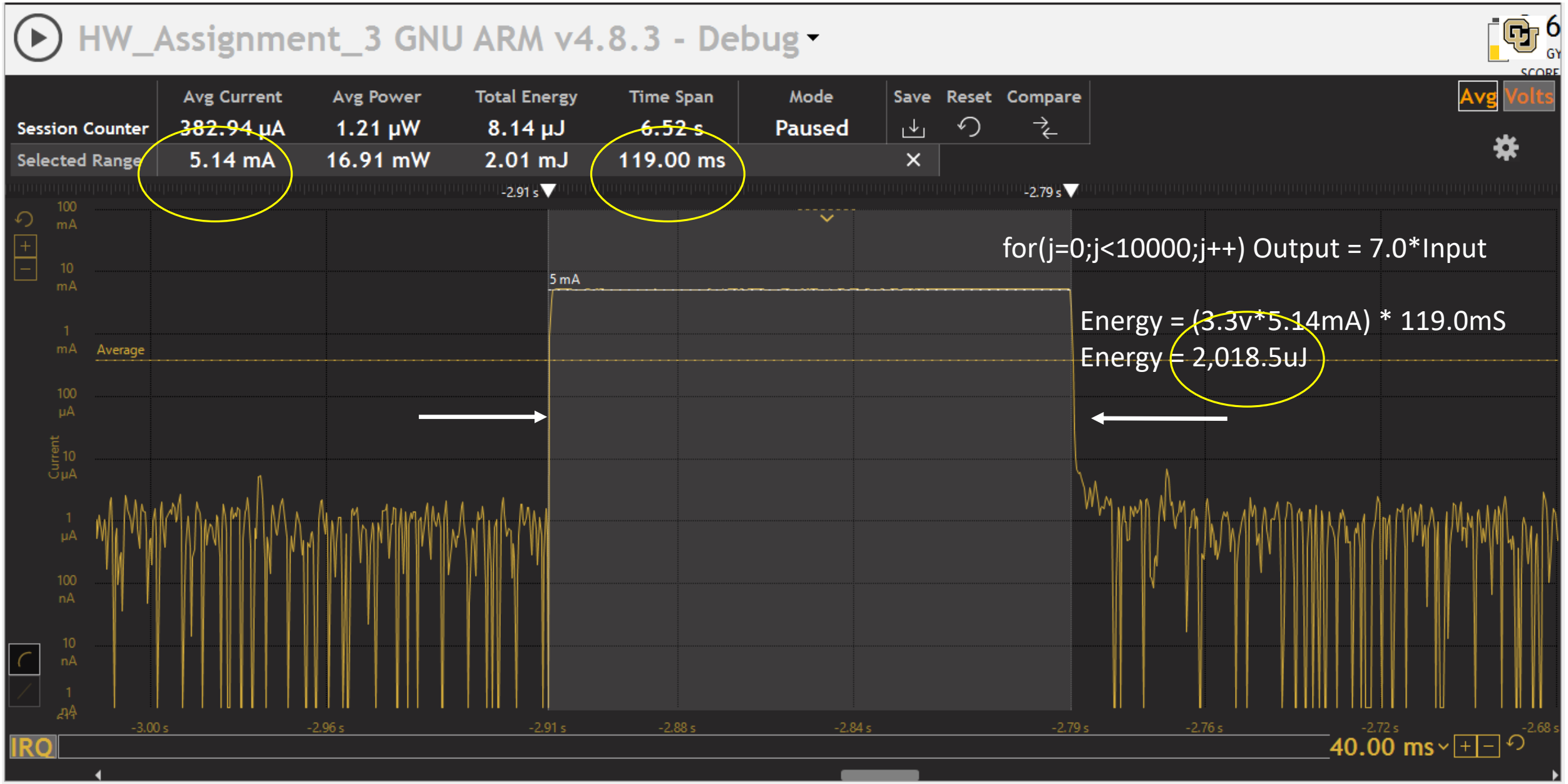
Firmware Best Practices

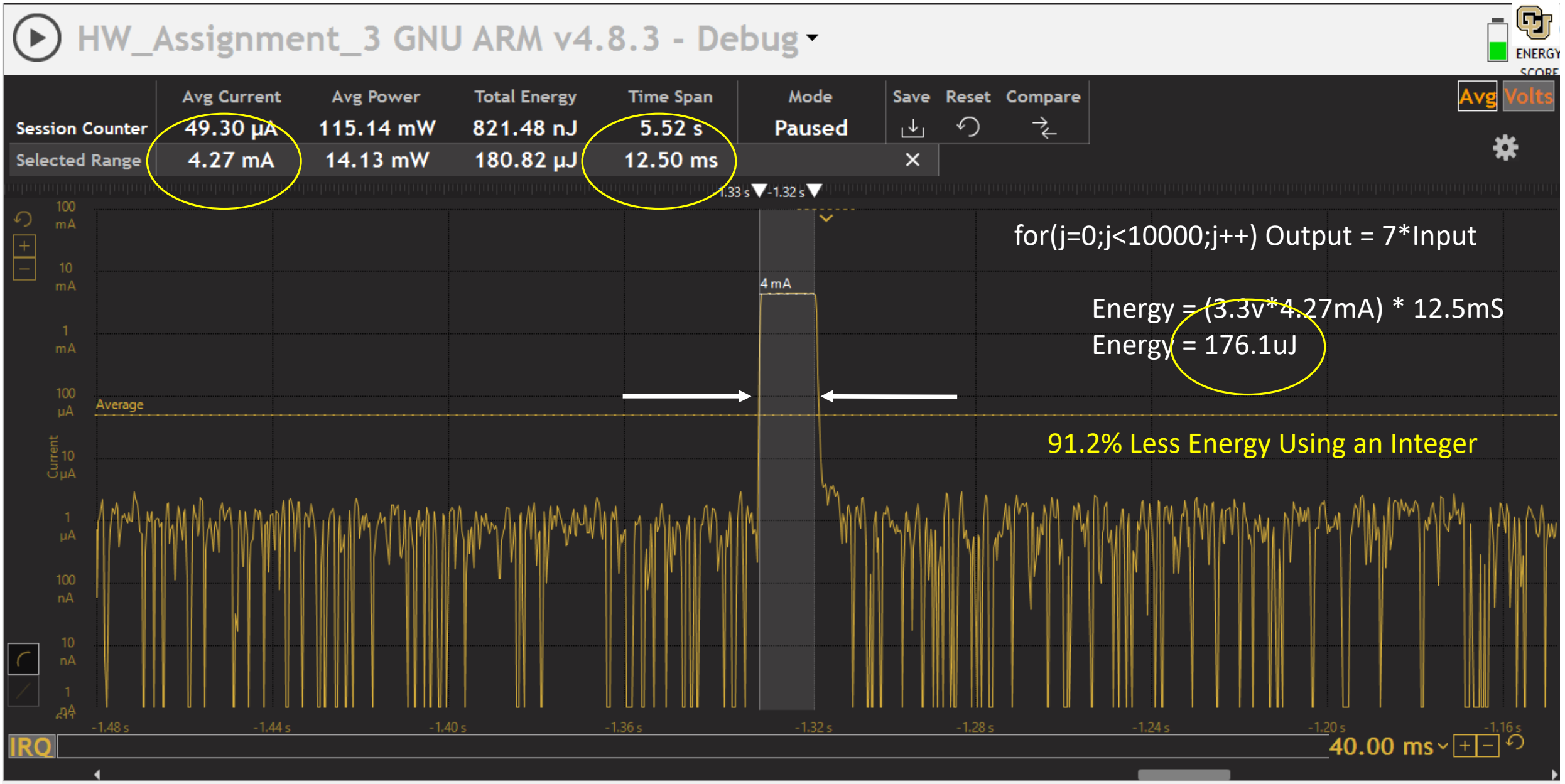
- Speeding up your Math
  - Constants are faster than using variables
    - They can be imbedded in assembly instructions
  - They must be “real” constants, and not variables
  - “real” constants in C are #define, not constant variables (const)
  - Putting constants in #define is good programming practice for readability and for Speeding up your Math



Firmware Best Practices

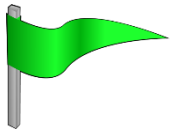
- Speeding up your Math
  - A floating point constant will generate a floating point instruction as much as a float variable
  - If a constant does not need to be float, make it an integer
  - Example:
    - Lets take a look at the constant 7 on a Cortex-M3 integer based MCU
    - The constant will be defined as 7.0 and 7


HW_Assignment_3 GNU ARM v4.8.3 - Debug
Avg Current
Avg Power
Total Energy
Time Span
Mode
Save
Reset
Compare
Avg
Volts
Session Counter
382.94 µA
1.21 µW
8.14 µJ
6.52 s
Paused
Selected Range
5.14 mA
16.91 mW
2.01 mJ
119.00 ms
-2.91 s
-2.79 s
for(j=0;j<10000;j++) Output = 7.0*Input
5 mA
Energy = (3.3v*5.14mA) * 119.0mS
Energy = 2,018.5uJ
Average
Current
IRQ
40.00 ms


HW_Assignment_3 GNU ARM v4.8.3 - Debug
ENERGY SCORE
Avg Current
Avg Power
Total Energy
Time Span
Mode
Save
Reset
Compare
Avg
Volts
Session Counter
49.30 µA
115.14 mW
821.48 nJ
5.52 s
Paused
Selected Range
4.27 mA
14.13 mW
180.82 µJ
12.50 ms
for(j=0;j<10000;j++) Output = 7*Input
Energy = (3.3v*4.27mA) * 12.5mS
Energy = 176.1uJ
91.2% Less Energy Using an Integer
4 mA
Average
Current
100 mA
10 mA
1 mA
100 µA
10 µA
1 µA
100 nA
10 nA
1 nA
-1.48 s
-1.44 s
-1.40 s
-1.36 s
-1.32 s
-1.28 s
-1.24 s
-1.20 s
-1.16 s
IRQ
40.00 ms

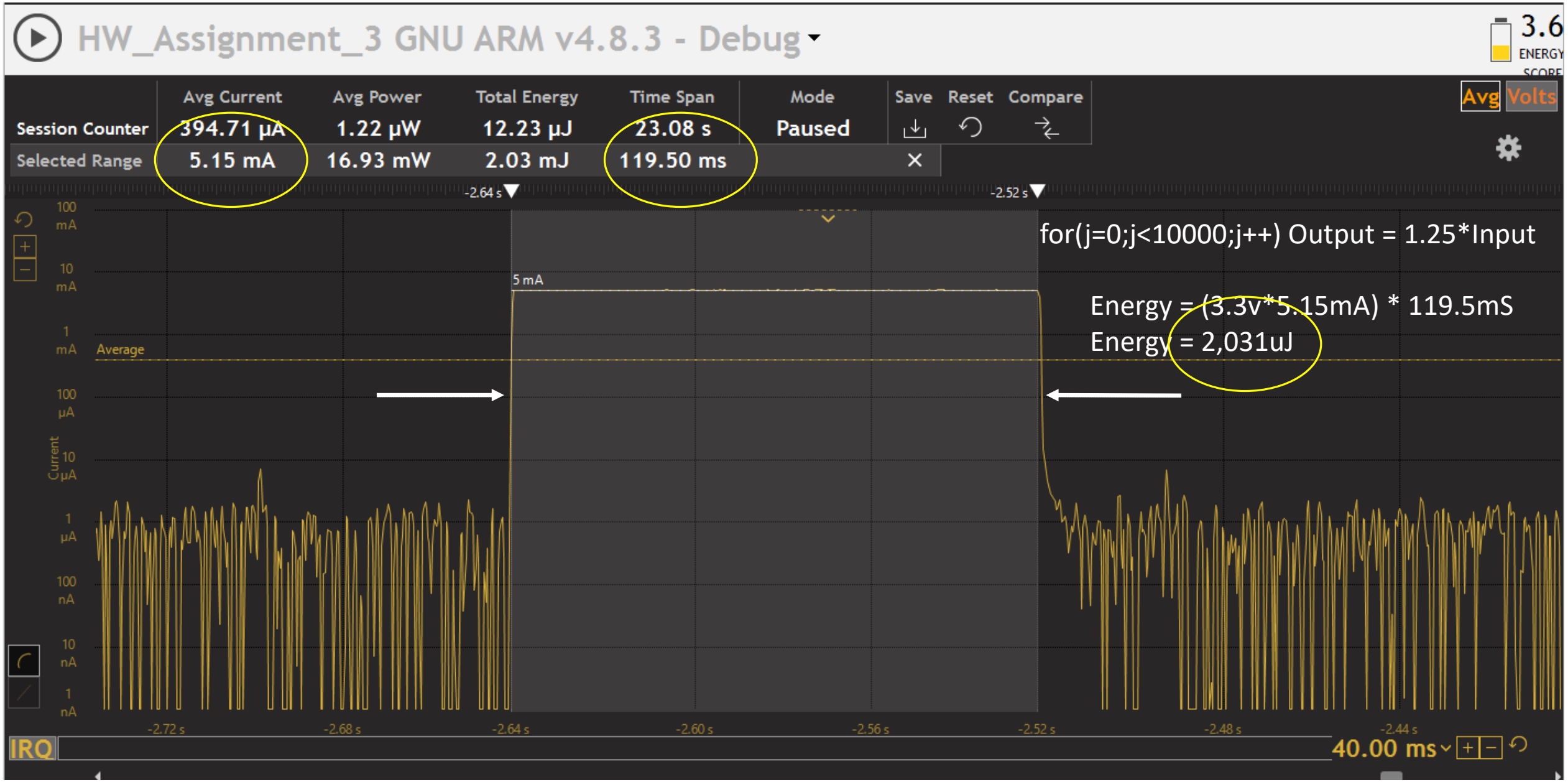
Firmware Best Practices

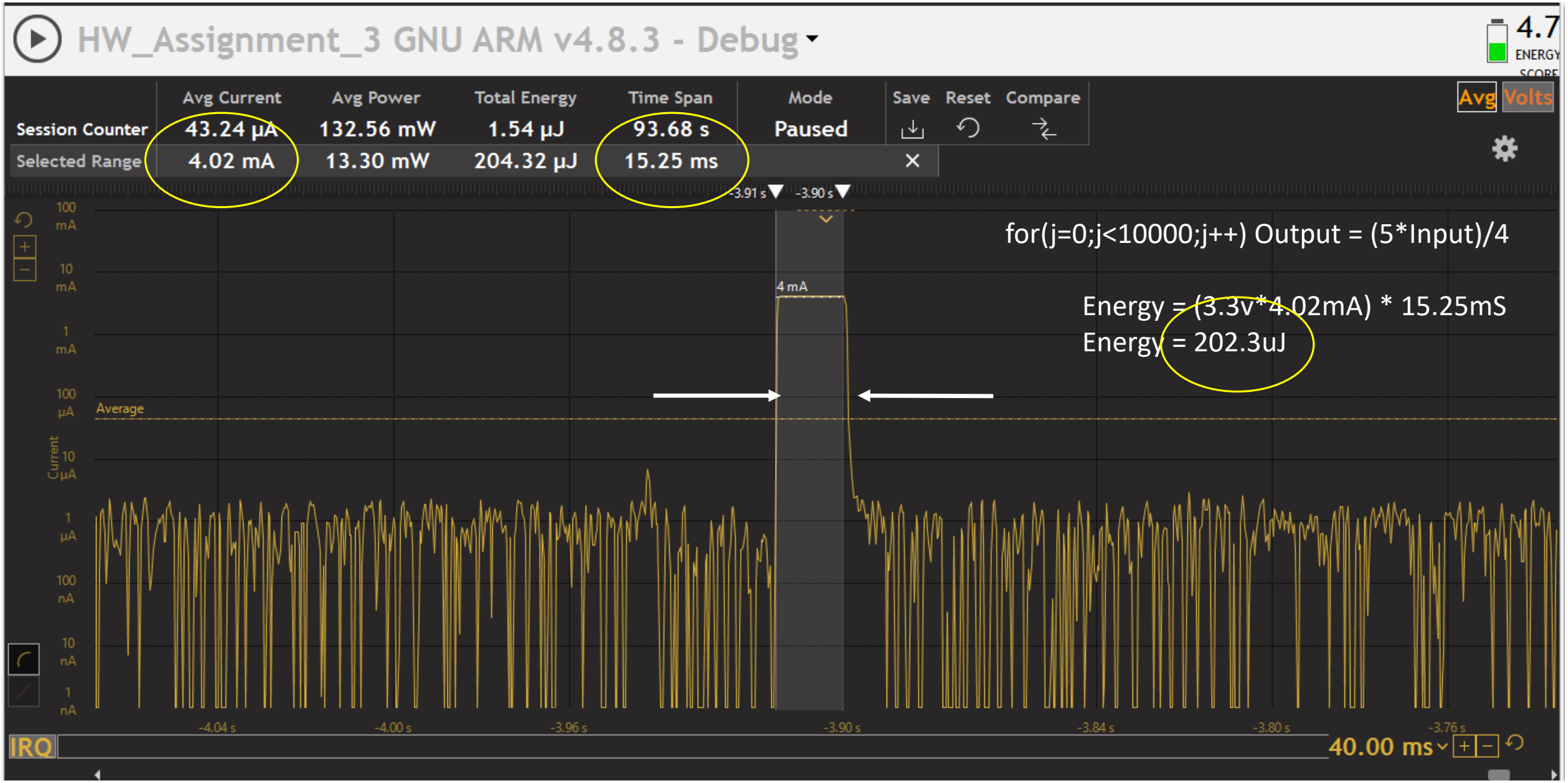
- Speeding up your Math
  - Using an integer 7 over the float 7.0 resulted in the following savings:
    - 2018.5uJ / 176.1uJ = 11.46x savings in Energy
    - 119mS / 12.5mS = 9.52x savings in Time
  - Where possible, specify constants in whole integers



Firmware Best Practices

- Speeding up your Math
  - Convert rational numbers into fractions when the divisor is a power of 2
  - For example, a constant of 1.25 could be represented by 5/4
  - Is Output = 1.25*Input equivalent to 5/4*Input in Integer math? NO!
    - In 1.25*Input, the rounding occurs after the multiplication (Input = 5, Output = 6)
    - 5/4*Input, the rounding occurs after 5/4 (Input = 5, Output = 5)
  - (5*Input) / 4 is equivalent. (Input = 5, Output = 6)


HW_Assignment_3 GNU ARM v4.8.3 - Debug
3.6
ENERGY
Avg Current
Avg Power
Total Energy
Time Span
Mode
Save
Reset
Compare
Avg
Volts
Session Counter
394.71 µA
1.22 µW
12.23 µJ
23.08 s
Paused
Selected Range
5.15 mA
16.93 mW
2.03 mJ
119.50 ms
-2.64 s
-2.52 s
for(j=0;j<10000;j++) Output = 1.25*Input
Energy = (3.3v*5.15mA) * 119.5mS
Energy = 2,031uJ
5 mA
Average
Current
100 mA
10 mA
1 mA
100 µA
10 µA
1 µA
100 nA
10 nA
1 nA
-2.72 s
-2.68 s
-2.64 s
-2.60 s
-2.56 s
-2.52 s
-2.48 s
-2.44 s
IRQ
40.00 ms


HW_Assignment_3 GNU ARM v4.8.3 - Debug
4.7
ENERGY
Avg Current
Avg Power
Total Energy
Time Span
Mode
Save
Reset
Compare
Avg
Volts
Session Counter
43.24 µA
132.56 mW
1.54 µJ
93.68 s
Paused
Selected Range
4.02 mA
13.30 mW
204.32 µJ
15.25 ms
-3.91 s
-3.90 s
for(j=0;j<10000;j++) Output = (5*Input)/4
Energy = (3.3v*4.02mA) * 15.25mS
Energy = 202.3uJ
4 mA
Average
Current
100 mA
10 mA
1 mA
100 µA
10 µA
1 µA
100 nA
10 nA
1 nA
-4.04 s
-4.00 s
-3.96 s
-3.90 s
-3.84 s
-3.80 s
-3.76 s
IRQ
40.00 ms

Assembly code for Output = 1.25*Input

```
000055a2:  ldr     r0,[sp,#0x4]
000055a4:  bl      0x00005978
000055a8:  mov     r2,r0
000055aa:  mov     r3,r1
000055ac:  mov     r0,r2
000055ae:  mov     r1,r3
000055b0:  mov.w   r2,#0x0
000055b4:  ldr     r3,[pc,#0x30] ; 0x55e4
000055b6:  bl      0x00005a44
000055ba:  mov     r2,r0
000055bc:  mov     r3,r1
000055be:  mov     r0,r2
000055c0:  mov     r1,r3
000055c2:  bl      0x00005e68
000055c6:  mov     r3,r0
000055c8:  str     r3,[sp,#0xc]
00005978:  teq     r0,#0x0
0000597c:  itt     eq
0000597e:  movs    r1,#0x0
00005980:  bx      lr
00005982:  push    {r4,r5,lr}
00005984:  mov.w   r4,#0x400
00005988:  add.w   r4,r4,#0x32
0000598c:  ands    r5,r0,#0x80000000
00005990:  it      mi
00005992:  rsbs    r0,r0,#0
00005994:  mov.w   r1,#0x0
00005998:  b       0x0000000000005818
00005818:  teq     r1,#0x0
0000581c:  itt     eq
0000581e:  mov     r1,r0
00005820:  movs    r0,#0x0
00005822:  clz     r3,r1
00005826:  it      eq
00005828:  adds    r3,#0x20
0000582a:  sub.w   r3,r3,#0xb
0000582e:  subs.w  r2,r3,#0x20
00005832:  bge     0x000000000000584e0
000584e:  it      le
00005850:  rsb.w   r12,r2,#0x20
00005854:  lsl.w   r1,r1,r2
00005858:  lsr.w   r12,r0,r12
0000585c:  itt     le
0000585e:  orr.w   r1,r1,r12
00005862:  lsls    r0,r2
00005864:  subs    r4,r4,r3
00005866:  ittt    ge
00005868:  add.w   r1,r1,r4,lsl #20
0000586c:  orrs    r1,r5
0000586e:  pop     {r4,r5,pc}
00005a44:  push    {r4,r5,r6,lr}
00005a46:  mov.w   r12,#0xff
00005a4a:  orr     r12,r12,#0x700
00005a4e:  ands.w  r4,r12,r1,lsr #20
00005a52:  ittte   ne
00005a54:  ands.w  r5,r12,r3,lsr #20
00005a58:  teq     r4,r12
00005a5c:  teq     r5,r12
00005a60:  bl      0x0000000000005c20
00005a64:  add     r4,r5
00005a66:  eor.w   r6,r1,r3
00005a6a:  bic.w   r1,r1,r12,lsl #21
00005a6e:  bic.w   r3,r3,r12,lsl #21
00005a72:  orrs.w  r5,r0,r1,lsl #12
00005a76:  it      ne
00005a78:  orrs.w  r5,r2,r3,lsl #12
00005a7c:  orr     r1,r1,#0x100000
00005a80:  orr     r3,r3,#0x100000
00005a84:  beq     0x0000000000005af8
00005a86:  umull   r12,lr,r0,r2
00005a8a:  mov.w   r5,#0x0
00005a8e:  umlal   lr,r5,r1,r2
00005a92:  and     r2,r6,#0x80000000
00005a96:  umlal   lr,r5,r0,r3
00005a9a:  mov.w   r6,#0x0
00005a9e:  umlal   r5,r6,r1,r3
00005aa2:  teq     r12,#0x0
00005aa6:  it      ne
00005aa8:  orr     lr,lr,#0x1
00005aac:  sub.w   r4,r4,#0xff
00005ab0:  cmp.w   r6,#0x200
00005ab4:  sbc     r4,r4,#0x300
00005ab8:  bcs     0x0000000000005ac4
00005aba:  lsls.w  lr,lr,#1
00005abe:  adcs    r5,r5
00005ac0:  adc.w   r6,r6,r6
00005ac4:  orr.w   r1,r2,r6,lsl #11
00005ac8:  orr.w   r1,r1,r5,lsr #21
00005acc:  lsl.w   r0,r5,#11
00005ad0:  orr.w   r0,r0,lr,lsr #21
00005ad4:  lsl.w   lr,lr,#11
00005ad8:  subs.w  r12,r4,#0xfd
00005adc:  it      hi
00005ade:  cmp.w   r12,#0x700
00005ae2:  bhi     0x0000000000005b22
00005ae4:  cmp.w   lr,#0x80000000
00005ae8:  it      eq
00005aea:  lsrs.w  lr,r0,#1
00005aee:  adcs    r0,r0,#0x0
00005af2:  adc.w   r1,r1,r4,lsl #20
00005af6:  pop     {r4,r5,r6,pc}
00005e68:  lsl.w   r2,r1,#1
00005e6c:  adds.w  r2,r2,#0x200000
00005e70:  bcs     0x0000000000005e9e
00005e72:  bpl     0x0000000000005e98
00005e74:  mvn     r3,#0x3e0
00005e78:  subs.w  r2,r3,r2,asr #21
00005e7c:  bls     0x0000000000005ea4
00005e7e:  lsl.w   r3,r1,#11
00005e82:  orr     r3,r3,#0x80000000
00005e86:  orr.w   r3,r3,r0,lsr #21
00005e8a:  tst     r1,#0x80000000
00005e8e:  lsr.w   r0,r3,r2
00005e92:  it      ne
00005e94:  rsbs    r0,r0,#0
00005e96:  bx      lr
```

116 assembly instructions

Assembly code for Output = (5*Input)/4

```
000055ca:   ldr     r2,[sp,#0x4]
000055cc:   mov     r3,r2
000055ce:   lsls    r3,r3,#2
000055d0:   add     r3,r2
000055d2:   cmp     r3,#0x0
000055d4:   bge     0x000055d8
000055d6:   adds    r3,#0x3
000055d8:   asrs    r3,r3,#0x2
000055da:   str     r3,[sp,#0xc]
```

Multiplication of 5 is:

- Left shift by 2 (multiplication by 4)
- Add input (now equivalent of multiplication by 5)

Now, shift to the right by 2 to divide by 4

Store result of (5*Input) / 4

Only 9 assembly instructions

Firmware Best Practices

- Speeding up your Math
  - Fractional math where the divisor is a power of 2 provided the following savings over the floating constant:
    - 2031uJ / 202.3uJ = 10.04x savings in Energy
    - 119.5mS / 15.25mS = 7.84x savings in Time
  - Where possible, use correct ordering of fractions where the divisor can be of power 2 instead of a rational multiplication