Design for Internet of Things
Prof. T V Prabhakar
Department of Electronic Systems Engineering
Indian Institute of Science-Bengaluru

Lecture - 31
ADC Driver Design and Development

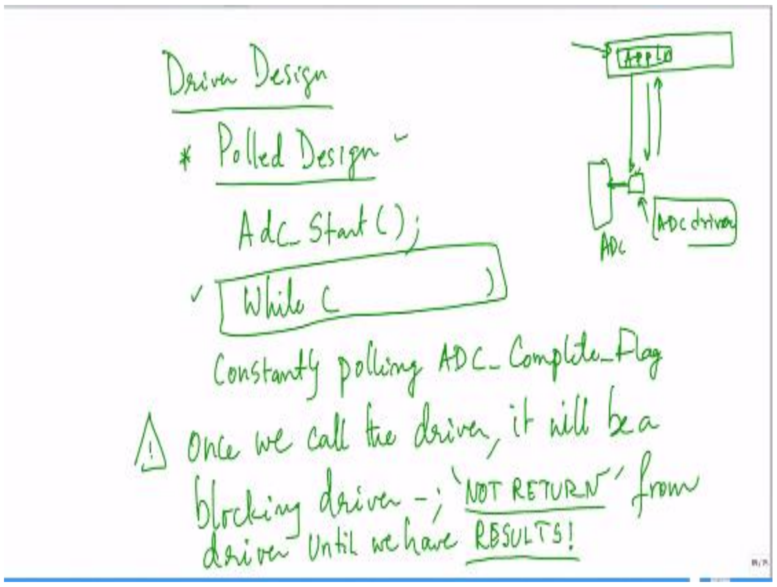
Welcome back. See low power software module is a very critical part because this is what will actually give you large lifetimes for your IoT notes, okay? And you know we have been, in the last modules we have been talking so much about ADC, right? And every time there is emphasis because sensing is an important thing in IoT.

So now the question is, when you talk about an ADC driver, that is you have an SOC, you have an ADC driver, how should you write this driver? How should one develop this driver? Why ADC? Even you take UART driver, how should you write the driver? That becomes another important question. We cannot get into the details of, let us say, callbacks and so on.

Because that is one way of, you know getting into details of making the application code agnostic to the hardware. Usually they use callbacks. We will not get into that, but we will show you some critical parts, which actually make and break your IoT node in terms of its lifetime. But before we go into any demo, you recall the last class we spoke about polling. We spoke about the while loop and the if else loop and so on.

We just gave a very overall view of it. Now is the time to actually see something and make an impact to yourself that yes, these are very critical parts. We will do that. But before we do that, let me just throw a little bit of light into driver development, okay; ADC Driver Development, UART Driver Development. Let us get some basic understanding and then let us see the demonstrations.

(Refer Slide Time: 02:21)


Driver Design

* Polled Design

Adc_Start();

✓ While ( )

Constantly polling ADC_Complete_Flag

⚠ Once we call the driver, it will be a blocking driver -; 'NOT RETURN' from driver Until we have RESULTS!

App
ADC
ADC driver

I have with me here two, one possible way of writing the driver. One is you do a polled driver, okay. You do a polled design. What do we mean by that? While loop is the critical part. Remember, this is the ADC driver. This has nothing to do with the application. Application is sitting right on top. Application is sitting right on top. I am, this is the ADC which is out here. There is a little piece of code here.

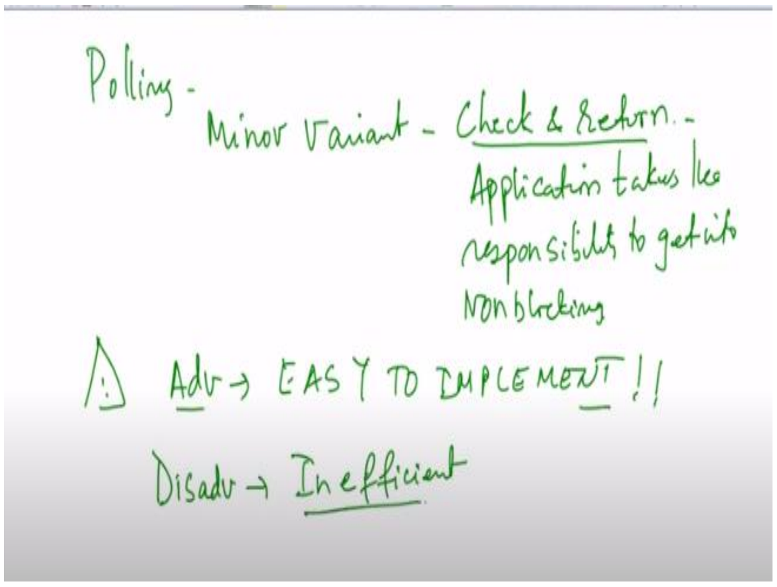
This is the ADC driver, ADC driver code, okay. And application is accessing via the ADC driver. It is actually getting the data from the ADC. This is the actual ADC, okay. Usually as I mentioned, there will be an abstraction here, it is called the hardware abstraction layer. And you are only talking about to the hardware abstraction layer, callback comes here, okay. So there will be a callback here.

Callback handler will come here. Application will be as one block. And this is the callback part of the hardware abstraction layer. So this callback we will talk here, and so on. Let us not complicate it. Let us just keep it pretty straightforward. And let us just say that you are, the ADC, even this is not important. We are just worried about this part. How is this driver developed as far as access from ADC is concerned?

You can write the C code for ADC driver using while loop. Very simple and quite easy to write. Once we call the driver, once this application calls the driver, driver will start trying to acquire data from ADC. Even if ADC is down it does not know. Even if it is up, it does not know. It does not know if the sensor is connected or not. It continues to go and get the data from the ADC till it gets the ADC conversion flag true.

That is still the ADC says I have finished. It goes on checking for the ADC and remains in that loop. Think about how the CPU is completely blocked, not able to do anything. And therefore no return is possible from this driver until we have the results. And results will come only when the ADC conversion flag gets set and says I finished the conversion. This is very simple to do. But as I said, it is not the best way to do.

(Refer Slide Time: 04:51)



Polling -

Minor Variant - Check & Return -

Application takes less responsibility to get into Non blocking

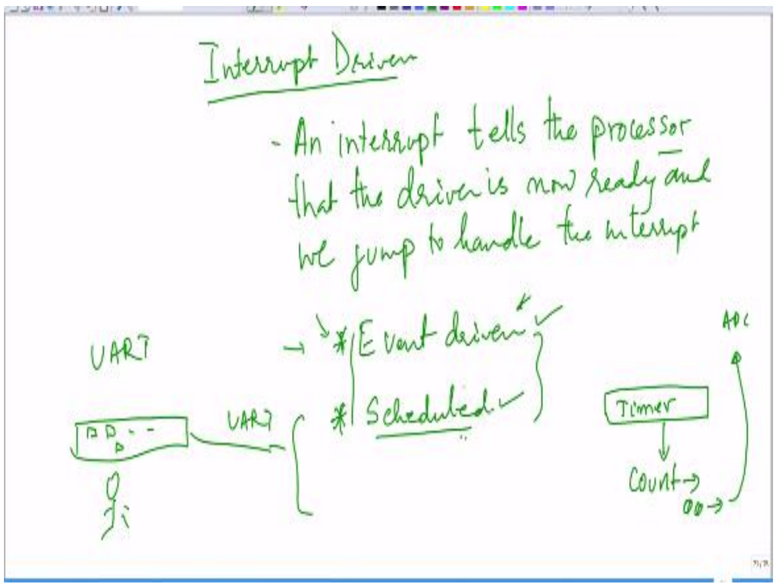
Adv → EASY TO IMPLEMENT !!

Disadv → Inefficient

There are some small variants to polling also, okay. What you can do is the variant is very simple. Here in this application, you can write a code, which says, go check. And if you do not come back, I am going to stop it. I am just going to exit it. You are not doing it at the driver level, but you are doing it at the application level. So that is what I meant by check and return. Application takes the responsibility to get it non-blocking, okay.

It is a minor variant, but actually, it is polling in a way. Only thing you are breaking out of the polling, because application is handling it all by itself. The main advantage of polling is it is highly, it is very easy to implement, but it is highly inefficient, okay. So this is a major problem for polling. Let us look at what are the other possibilities. The other very efficient way to do it would be interrupt driven, okay.

(Refer Slide Time: 05:53)


Interrupt Driven
- An interrupt tells the processor that the driver is now ready and we jump to handle the interrupt
UART
*Event driven
*Scheduled
UART
Timer
Count
00
ADC

Now the interrupt will tell the processor that the driver is now ready, and we have to jump and handle the interrupt, okay. This is a good thing. And there are two ways of interpreting an interrupt driven driver. One is called event driven, the other is called scheduled. Now think of scheduled as something like a timer based interrupt, okay. You have a timer. Your timer keeps counting down or counting up, count.

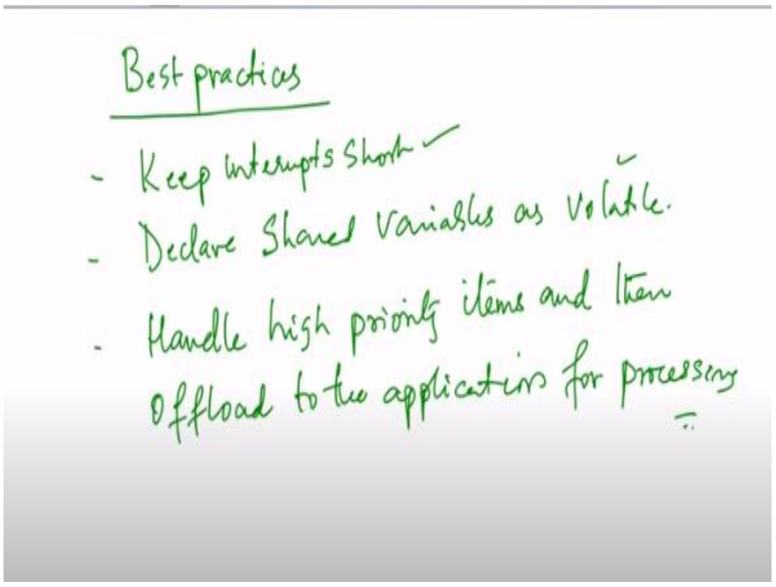
And based on the count, if it becomes 00 you go and check for the ADC. You go and check for the ADC, okay. And then you, hopefully ADC is ready at that instant. And then you wait for a certain time and then come back, okay. By that time, the ADC data is ready, you read it, otherwise you move on. That is you do not even do any other further processing. When the timer expires, check for the ADC.

If the flag is true, get the data. In fact, force the ADC to go and get the data if required. And then come out of it very quickly, okay. So it is more like, periodically you are going and writing. The driver itself is written to do scheduled way of accessing the data. This is one way. Another way is event. Event is another strange thing. It is easy to understand event from a UART perspective, okay.

A UART for instance, you take the case of let us say a keyboard, for instance, okay. And keyboard is connected via let us say this is your keyboard. And it is connected via let us say UART interface. And human, you really do not know when the key is pressed, right? It is an asynchronous event. So you will want to handle that kind of asynchronous events also.

You are not doing it periodically, you are not doing it in a scheduled way. You are only doing when there is a key pressed, then you are interrupting the processor, right? So these are broadly two ways of doing it. We should be able to see a demonstration of both as a complete solution. But to begin with, we will show you a few things, which will give you a clearer understanding of the impact of these ADC drivers on the energy consumption of the complete system. Alright.

(Refer Slide Time: 08:24)

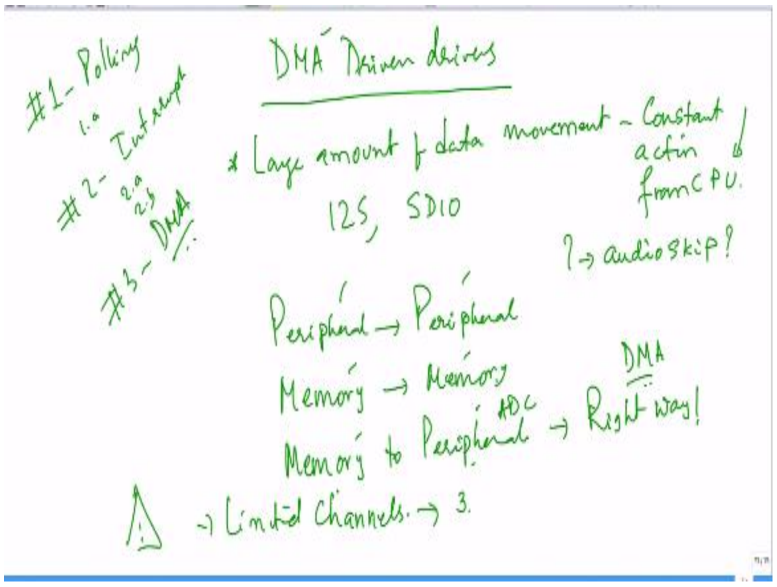


Best practices

- Keep interrupts short ✓
- Declare Shared Variables as Volatile.
- Handle high priority items and then offload to the applications for processing

So the other thing is, you have to keep some best practices in the back of your mind, keep the interrupts as short as possible. Do not write, do not execute 1000 lines of code in an interrupt service routine, keep it very, very short. And any variable that is shared across programs, which are shared variables, they keep them as volatile and always handled high priority items, then offload to the applications for further processing, okay. So these are important things that you should bear in your mind.

(Refer Slide Time: 08:56)



Now the other way of doing it would be do not either do interrupt or do not do polling, but use DMA very effectively. So this is the third way of doing it, right? So first one #1, #1 is polling, polling and 1.a is a small variant of it. #2 interrupt. Here you had 2.a and 2.b, please note. Now we are doing a third one, which is #3. This is DMA based. Whenever you want to move large amount of data, you typically use it with the DMA.

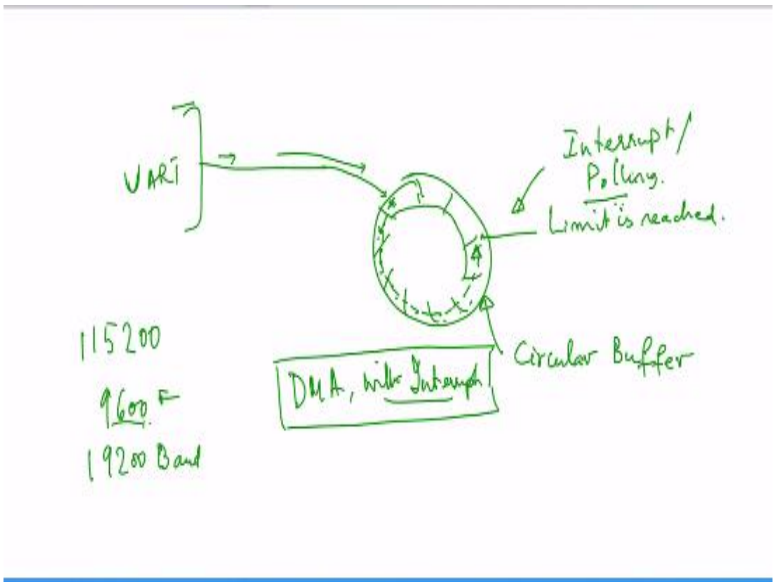
And microcontrollers, particularly a Nordic and so on, there are a very limited number of channels. There are something like three channels or so which are part of the Nordic controller, okay. When you, when we talk about large amount of data or any one of the systems where you are trying to get from let us say an audio chip using I2S or an SDIO based memory systems, typically you are expected to use DMA.

Because you cannot expect the CPU to keep the, you know constant action. It is not going to work that way. CPU cannot be locked up for so long a time. You want CPU to be interrupted after the DMA transfer has happened, let us say from an I2 on an I2S bus. You have copied it to memory. Then you want the CPU to come in and take action. The process of copying itself CPU should be avoided.

And for that you have DMA to do the job. What all can DMA do? It can do independently peripheral to peripheral, memory to memory, and memory to peripheral, okay? See think about peripheral to memory, memory to peripheral, both are possible. So ADC is like a peripheral. You can write it into memory and then you can interrupt the processor after copying it into memory.

This is the right way to do. This is the right way to do. Please use DMA every time you need to access data periodically you want to have the DMA trigger and then copy it into memory and then you want to interrupt the system.

(Refer Slide Time: 11:32)



So for that, I wanted to show you another picture of this particular aspect of what I think you should actually try. This is a nice picture of a UART, okay. The UART is essentially transferring data to this circular buffer. This is a circular buffer here, okay. You can see that the first byte is written here and second byte is written here, third byte and so on.

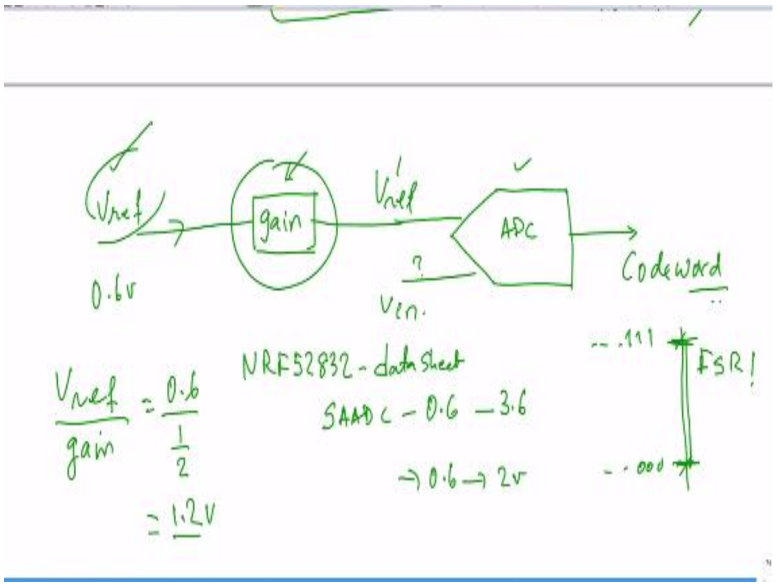
Every time it writes to this, this one circular buffer, CPU is not interrupted at all, okay. Only when it reaches a certain limit, see now only this much space is available. These two locations alone are free and it has come up till here. Now this is like reaching a limit. Once this limit is reached, you essentially trigger and ask the CPU to read all of this data and free up the circular buffer, okay.

And how do you do that? You generate an interrupt when this limit is reached. See the beauty, you are combining DMA with interrupt. How is it getting combined? Every time a character comes, it is copied to this memory without CPU intervention. Who is doing the copy? DMA is doing the copy. Next character written here, third here, fourth here and so on and so on and so on till this limit.

Once this limit comes, an interrupt is issued. Who issues? The DMA controller says now it is time for CPU to take it. Then you do it with interrupt or you can also do with polling by the way. And then the data is read. Therefore, combining DMA with interrupt in an example like what I showed with respect to UART is a very critical part of the whole demonstration.

So this is what you have to note in order to make it ultra low power for all your settings. Now let us revisit the ADC problem, okay.

(Refer Slide Time: 13:46)



The ADC problem is very simple here. I have taken an example, in fact Vasanth has set it up beautifully. He will show you a nice demonstration of the following. Here is the ADC which is part of the SOC and here is the code word output that you want. What is your goal? Your goal is you want to use the full scale range.

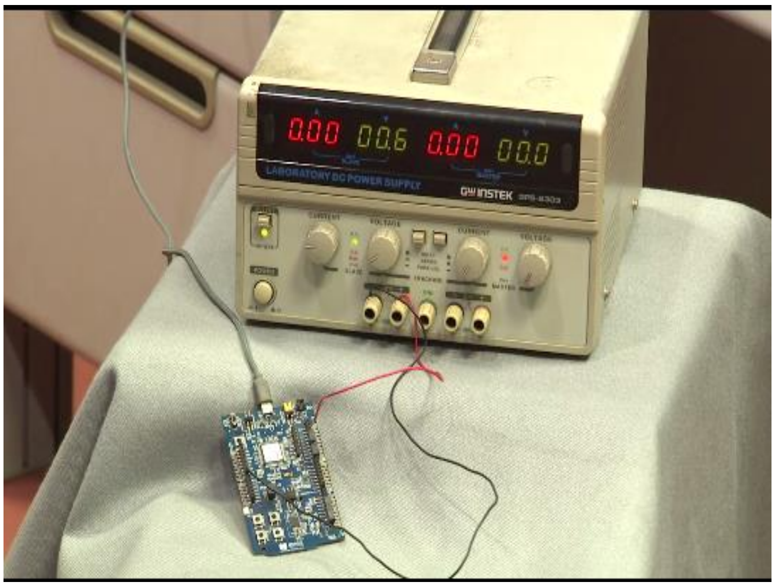
You want to use all 0s to all 1s depending on the number of bits, ADC bit resolution you have, you want you can cascade all the zeros there. So you want to use the full scale range. Now the chip actually provides a gain block, okay. It says V ref is always fixed to some value and that is typically 0.6 volts and then use this gain block and you tune this gain block depending on your sensor output.

If your sensor output is going to give you three volts, you adjust this gain block such that you will get the full scale range, okay. If your sensor is giving you 2 volts, then you adjust this gain block to give you, take that 0.6 and make it into 2 volts. So your V ref dash is now 2 volts instead of, let us say 3 volts. So your V ref prime largely depends on what your Vin is likely to be.

And that you have to look up your sensor and you will have to see what is the maximum voltage the sensor is going to give. And based on that, you will have to tune the gain block such that you will be able to use the full scale range in its completeness, okay? To demonstrate the idea of V ref and V ref prime that I showed you, which is right in this picture here.

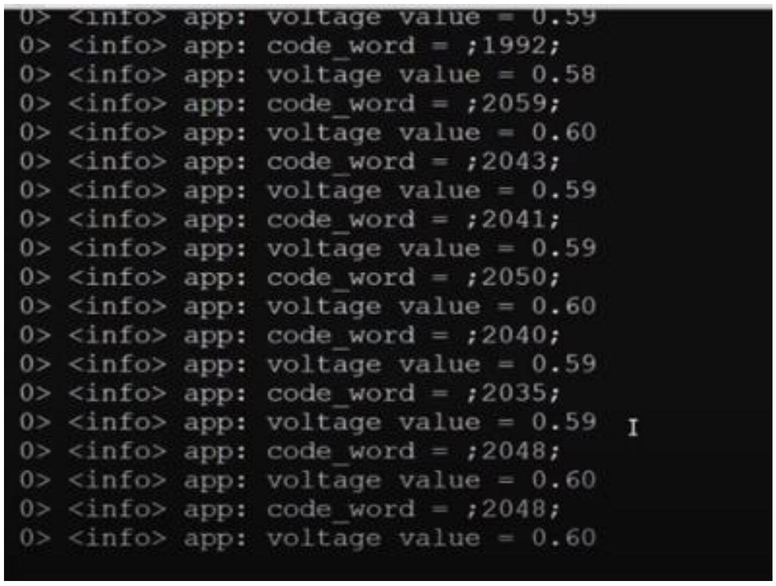
Let us now turn our attention to set 0.6 volts at the input of the power supply. So let us focus on the power supply now.

(Refer Slide Time: 15:53)



The power supply shows 0.6. You can read 0.6 there, which is the second window there, you can see 0.6. That power supply is the V in which is being connected to the development board. And now what I will show you is the output screen.

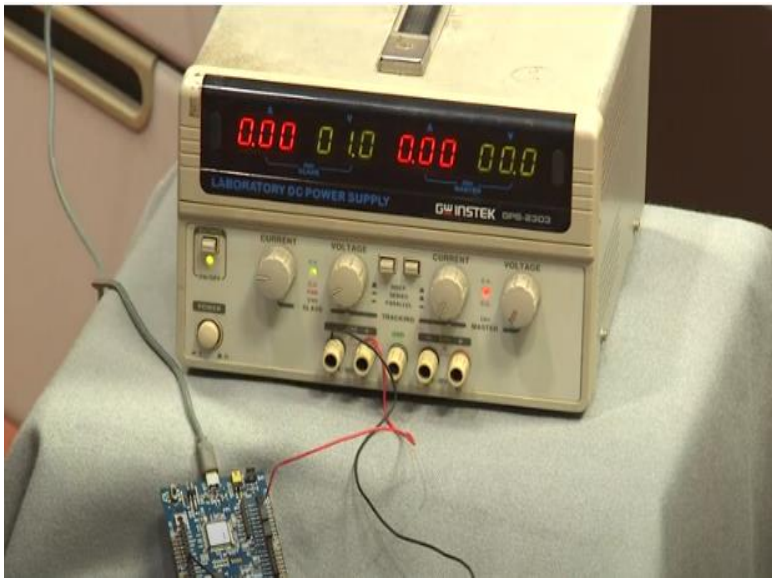
(Refer Slide Time: 16:09)



```
0> <info> app: voltage value = 0.59
0> <info> app: code_word = ;1992;
0> <info> app: voltage value = 0.58
0> <info> app: code_word = ;2059;
0> <info> app: voltage value = 0.60
0> <info> app: code_word = ;2043;
0> <info> app: voltage value = 0.59
0> <info> app: code_word = ;2041;
0> <info> app: voltage value = 0.59
0> <info> app: code_word = ;2050;
0> <info> app: voltage value = 0.60
0> <info> app: code_word = ;2040;
0> <info> app: voltage value = 0.59
0> <info> app: code_word = ;2035;
0> <info> app: voltage value = 0.59
0> <info> app: code_word = ;2048;
0> <info> app: voltage value = 0.60
0> <info> app: code_word = ;2048;
0> <info> app: voltage value = 0.60
```

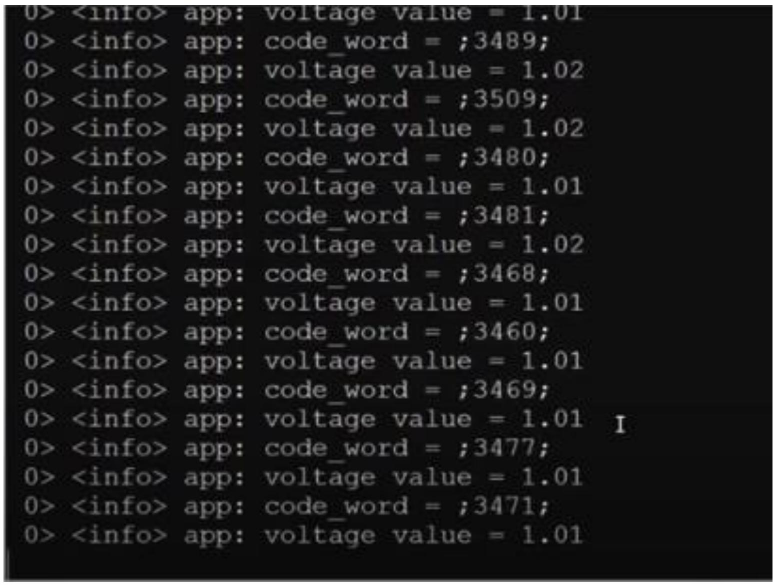
How much should it show? It should also show you that the input is indeed 0.6. So V in 0.6 is now converted safely into ADC equivalent and displaying it back again as 0.6 is a nice thing that it is actually showing you 0.6. Now let us go and modify that to 1 volt.

(Refer Slide Time: 16:35)



Let us now modify it to 1 volt. So 1 volt will be changed. Yes, now it is at 1 volt. If the input is at 1 volt, what does the ADC conversion indicate?

(Refer Slide Time: 16:48)

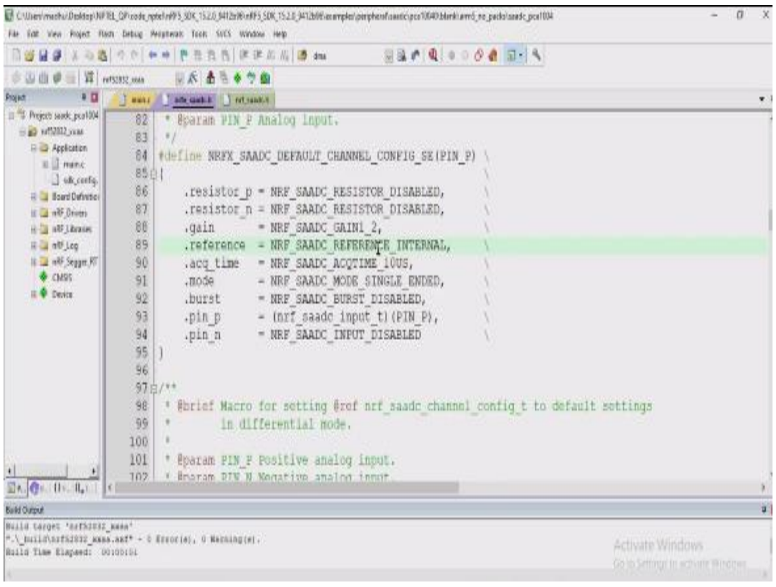


```
0> <info> app: voltage value = 1.01
0> <info> app: code_word = ;3489;
0> <info> app: voltage value = 1.02
0> <info> app: code_word = ;3509;
0> <info> app: voltage value = 1.02
0> <info> app: code_word = ;3480;
0> <info> app: voltage value = 1.01
0> <info> app: code_word = ;3481;
0> <info> app: voltage value = 1.02
0> <info> app: code_word = ;3468;
0> <info> app: voltage value = 1.01
0> <info> app: code_word = ;3460;
0> <info> app: voltage value = 1.01
0> <info> app: code_word = ;3469;
0> <info> app: voltage value = 1.01
0> <info> app: code_word = ;3477;
0> <info> app: voltage value = 1.01
0> <info> app: code_word = ;3471;
0> <info> app: voltage value = 1.01
```

Beautifully it indicates 1 volt as well. Now what we will do is we will make it 1.1 volt. Let us make it 1.1 small voltage increase 1.1. 1.1 is also successful. You will get 1.1. Now let us do 1.3. Let us see what happens to 1.3, 1.2 and 1.3. What happens if you do 1.3?

It is just saturating at 1.2, which is a clear indicator that this picture that I showed you is clearly working as a block V ref in, gain, V ref prime, V ref dash prime, which is clearly showing you that it is saturated to 1.2, which means this 1.2 can be nicely exploited by the full scale range, okay. Now let us shift to the code written in C which will actually show you several parts of this block.

(Refer Slide Time: 17:54)

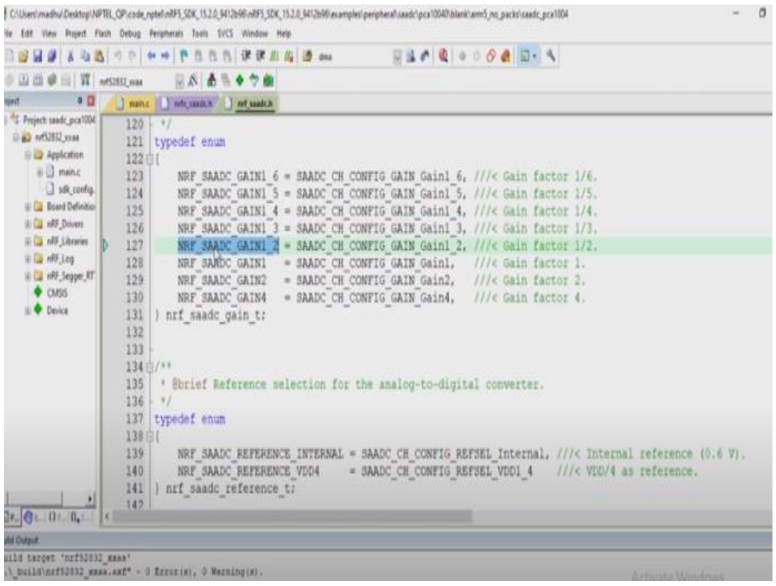


```c
 * @param PIN_P Analog input.
 */
#define NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_SE(PIN_P) \
{                                                   \
    .resistor_p = NRF_SAADC_RESISTOR_DISABLED,      \
    .resistor_n = NRF_SAADC_RESISTOR_DISABLED,      \
    .gain       = NRF_SAADC_GAIN1_2,                \
    .reference  = NRF_SAADC_REFERENCE_INTERNAL,     \
    .acq_time   = NRF_SAADC_ACQTIME_10US,           \
    .mode       = NRF_SAADC_MODE_SINGLE_ENDED,      \
    .burst      = NRF_SAADC_BURST_DISABLED,         \
    .pin_p      = (nrf_saadc_input_t)(PIN_P),       \
    .pin_n      = NRF_SAADC_INPUT_DISABLED          \
}

/**
 * @brief Macro for setting @ref nrf_saadc_channel_config_t to default settings
 *        in differential mode.
 *
 * @param PIN_P Positive analog input.
 * @param PIN_N Negative analog input.
```

```
Build target 'nrf52832_xxaa'
".\_build\nrf52832_xxaa.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:01
```

Now look at this code. Let us see the internal V ref which is set to 0.6, okay. You can see that is the setting which says it is 0.6 volts. There you go, this is done now. So your V ref is set to, V ref is now set to 0.6, but this is not what is applied to the ADC V ref. What is applied is the V ref prime. What is coming in the middle, the gain block. So let us go and look up the gain that he has set.

(Refer Slide Time: 18:27)



```
 */
typedef enum
{
    NRF_SAADC_GAIN1_6 = SAADC_CH_CONFIG_GAIN_Gain1_6, ///< Gain factor 1/6.
    NRF_SAADC_GAIN1_5 = SAADC_CH_CONFIG_GAIN_Gain1_5, ///< Gain factor 1/5.
    NRF_SAADC_GAIN1_4 = SAADC_CH_CONFIG_GAIN_Gain1_4, ///< Gain factor 1/4.
    NRF_SAADC_GAIN1_3 = SAADC_CH_CONFIG_GAIN_Gain1_3, ///< Gain factor 1/3.
    NRF_SAADC_GAIN1_2 = SAADC_CH_CONFIG_GAIN_Gain1_2, ///< Gain factor 1/2.
    NRF_SAADC_GAIN1   = SAADC_CH_CONFIG_GAIN_Gain1,   ///< Gain factor 1.
    NRF_SAADC_GAIN2   = SAADC_CH_CONFIG_GAIN_Gain2,   ///< Gain factor 2.
    NRF_SAADC_GAIN4   = SAADC_CH_CONFIG_GAIN_Gain4,   ///< Gain factor 4.
} nrf_saadc_gain_t;


/**
 * @brief Reference selection for the analog-to-digital converter.
 */
typedef enum
{
    NRF_SAADC_REFERENCE_INTERNAL = SAADC_CH_CONFIG_REFSEL_Internal, ///< Internal reference (0.6 V).
    NRF_SAADC_REFERENCE_VDD4     = SAADC_CH_CONFIG_REFSEL_VDD1_4    ///< VDD/4 as reference.
} nrf_saadc_reference_t;
```

```
uild target 'nrf52832_xxaa'
.\_build\nrf52832_xxaa.axf" - 0 Error(s), 0 Warning(s).
```

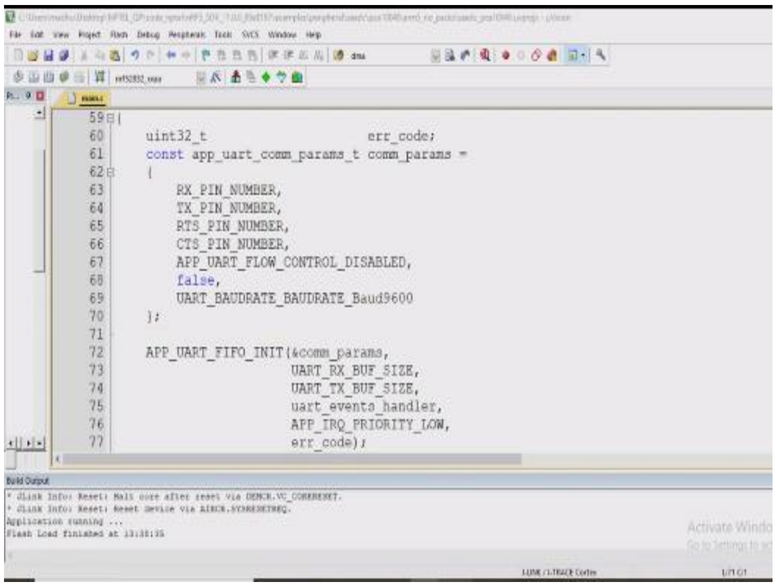
The gain block is showing as you can see that it is gain is 1/2, right? And now we go back and put that 1/2 here which is giving you 1/2, which is 0.6 divided by 1/2, which is 1.2. So that is essentially what the V prime. This is 1.2 volts by the way. This is making an assumption that the V in from the sensor is giving you 1.2 volts at its maximum.

So the big highlight from this code that we have shown you is please look up the data sheet, okay? Please look up the data sheet and several of these settings will be available. Reading the data sheet of the controller becomes very critical for you for planning your experiments and for planning your sensor interfaces to the SOCs. Okay folks. Now is another important demonstration of the UART, okay.

We mentioned that we will look at the UART also. See UART is connected to the controller over a baud rate, which is typically called 115200. This is the loopback baud rate which is supported. Well some of them do not even support this kind of high baud rate. What we know is 9600, 19200 baud and so on, right? Supposing some of the older systems can actually give you 9600 and 19200 and so on.

Now the character moves, if you have 9600 baud the character transfer rate is at 9600 baud, which is a slow baud, right? It takes a longer time for the character to move from the UART port to the circular buffer okay, which means the UART driver is now working at 9600 baud. So let us see what happens when you do it at 9600 baud and how do we set it?

(Refer Slide Time: 20:36)



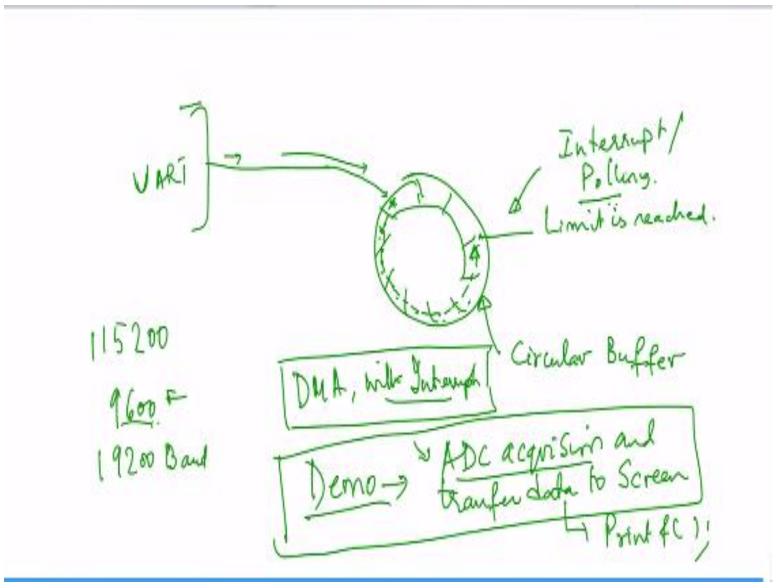
```
{
    uint32_t                     err_code;
    const app_uart_comm_params_t comm_params =
    {
        RX_PIN_NUMBER,
        TX_PIN_NUMBER,
        RTS_PIN_NUMBER,
        CTS_PIN_NUMBER,
        APP_UART_FLOW_CONTROL_DISABLED,
        false,
        UART_BAUDRATE_BAUDRATE_Baud9600
    };

    APP_UART_FIFO_INIT(&comm_params,
                       UART_RX_BUF_SIZE,
                       UART_TX_BUF_SIZE,
                       uart_events_handler,
                       APP_IRQ_PRIORITY_LOW,
                       err_code);
```

```
* JLink Info: Reset: Halt core after reset via DEMCR.VC_CORERESET.
* JLink Info: Reset: Reset device via AIRCR.SYSRESETREQ.
Application running ...
Flash Load finished at 13:38:35
```

Let us shift to the screen and there you are, you can see that he has set the baud rate to UART baud rate to 9600. Let us now compile this. Please note this is the driver. The UART driver is now programmed to fetch data at 9600 baud. Process is going on, compiled, code is dumped. And now let us see how he acquires this data and what is the time it takes and the current consumption it takes.

Let us take a very simple case of this demonstration. The driver has now been configured to 9600, you have already seen the process. What is the demo that we are going to show you? Demo is very simple. I have an ADC which is acquiring data okay, over an ADC port.

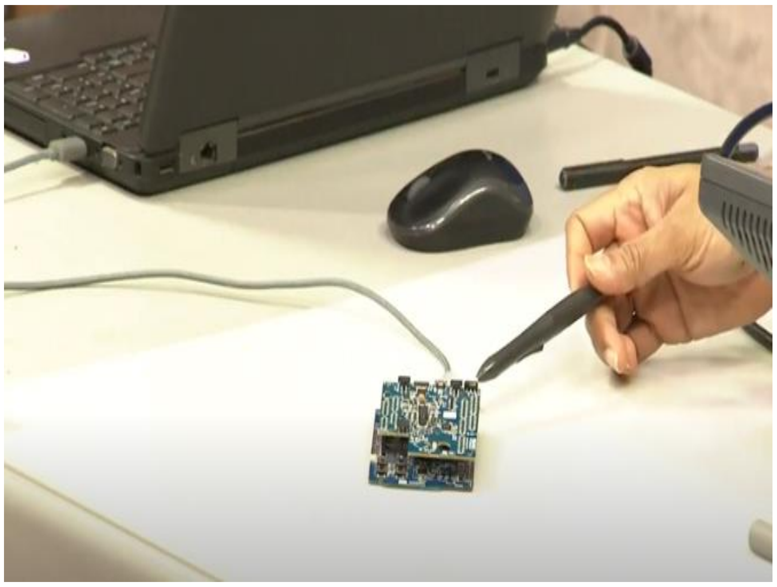
(Refer Slide Time: 21:25)



This is the demo of interest. We are acquiring data, ADC data, and we are transferring the data to the screen. That means we are going to use printf statement here, right? Printf is an output statement, we are going to use a printf statement. And we are dumping that character. We are collecting data from ADC and dumping it on the screen. Let us see how the current waveform looks like.

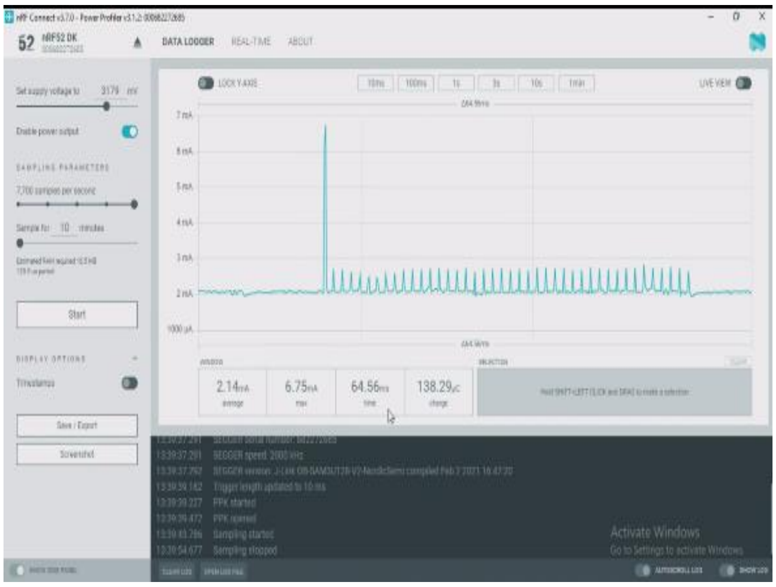
If you want to see the current waveform, you must go back to the demo that we did last time, power profiler kit and the development board.

(Refer Slide Time: 22:03)



We have here, the top one is the power profiler kit. The down one is the development board. Now this is the cable which is connected to the computer. So let us see the screen now folks.

(Refer Slide Time: 22:14)



Set supply voltage to 3179 mV

Enable power output

SAMPLING PARAMETERS

7,700 samples per second

Sample for 10 minutes

2.14mA	6.75mA	64.56ms	138.29µC
average	max	time	charge

Hold SHIFT+LEFT CLICK and DRAG to make a selection

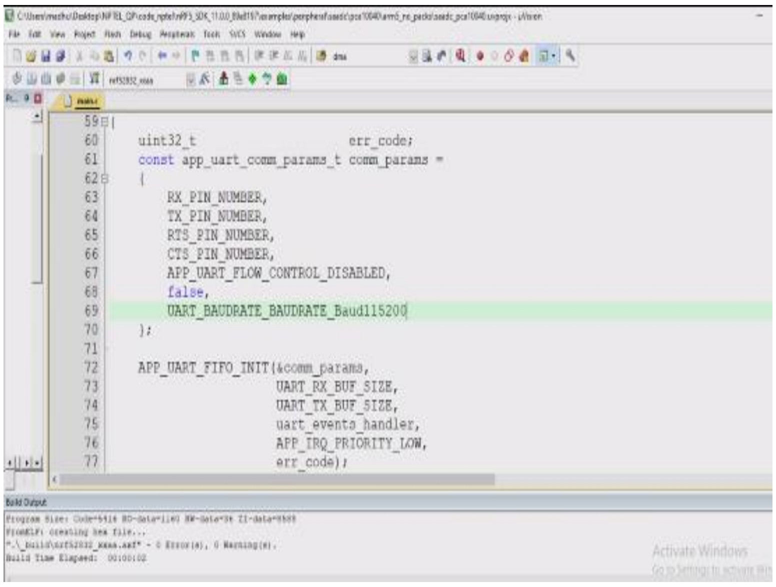
```
13:39:37.291   SEGGER serial number: 682272685
13:39:37.291   SEGGER speed: 2000 kHz
13:39:37.292   SEGGER version: J-Link OB-SAM3U128-V2-NordicSemi compiled Feb 2 2021 16:47:20
13:39:39.142   Trigger length updated to 10 ms
13:39:39.227   PPK started
13:39:39.472   PPK opened
13:39:43.786   Sampling started
13:39:54.677   Sampling stopped
```

The screen here is simply showing you that. What you see on the x axis is time, y axis you see is the spiky nature, one large spike, which is let us say a starting indicator or whatever. Now those small spikes are acquiring data from the ADC dumping to the screen, acquiring data dumping to the screen, okay. Fixed number of times, we are trying to do that. And you can see that the time is extended to some level, okay.

You can actually find out how much time it has taken to do this operation a fixed number of times from that axis from that point to the other point. So please make a note of the time that is the T1 time, right and T2 time. That is the T2 time, okay. Now look at the down picture. Look at the current consumption, okay. The current consumption is clearly indicated. And that is the value out there 6.75 milliamperes.

Folks, do not forget that energy is voltage times the current times the time (V x I x t). See the t there. It is quite a significant amount of time. Now let us shift the demonstration to 115200. And let us see what is the time it takes to do the same demonstration at a higher baud rate.

(Refer Slide Time: 23:53)

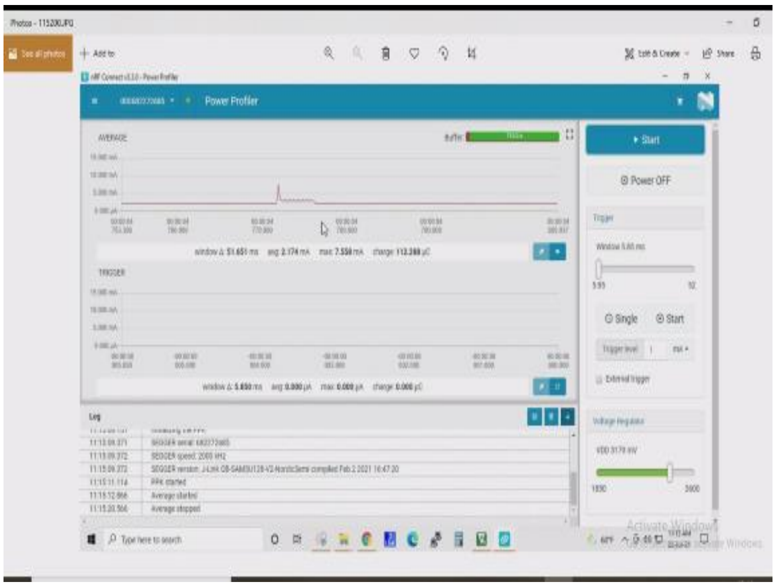


```
{
    uint32_t                     err_code;
    const app_uart_comm_params_t comm_params =
    {
        RX_PIN_NUMBER,
        TX_PIN_NUMBER,
        RTS_PIN_NUMBER,
        CTS_PIN_NUMBER,
        APP_UART_FLOW_CONTROL_DISABLED,
        false,
        UART_BAUDRATE_BAUDRATE_Baud115200
    };

    APP_UART_FIFO_INIT(&comm_params,
                       UART_RX_BUF_SIZE,
                       UART_TX_BUF_SIZE,
                       uart_events_handler,
                       APP_IRQ_PRIORITY_LOW,
                       err_code);
```

```
Program Size: Code=5516 RO-data=1160 RW-data=56 ZI-data=8588
FromELF: creating hex file...
".\_build\nrf52832_xxaa.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:02
```

I have 115200. Take this value, the driver is modified, again compiled and again dumped and again dumped and will directly show you the screen, okay of whatever is there.

(Refer Slide Time: 24:08)



This is the screen folks. You see now how much time is reduced. You can see that the T1 and T2 are highly reduced. Data is transferred at a much superior data rate. And therefore, if you go back and do energy computation, (V x I x t), you find that it is a much reduced energy consumption and therefore the battery lives for a longer time because of the high data rate transfer between the two systems. So keep this demonstration in mind when you do your low power software parts.