by Anders Lundgren and Lotta Frimanson, IAR Systems

Optimize for minimal power consumption

How can I find the power leaks and how can I optimize my software to minimize power consumption in my embedded system? This article will go more in depth with power debugging and discuss use cases where power debugging can be useful.

What is power debugging?

Power debugging is a methodology that provides software developers with information about how the software implementation in an embedded system affects system level power consumption. By coupling source code to power consumption, testing and tuning for power optimization is enabled.

Long battery life-time is a very important factor for many embedded systems in almost any market segment; medical, consumer electronics, home automation and many more. Power consumption has traditionally been a design goal that only the hardware developers have been able to influence. However in an active system, the power consumption depends not only on the design of the hardware, but also on how it is used, and how it is used is controlled by the system software.

IAR Systems' innovative technology integrates the system's power consumption in the development tools for software in embedded systems. This means that we put a tool in the hands of software developers that makes it possible for them to get an insight into how power consumption can be minimized in embedded software. That is what we call power debugging.

The IAR C-SPY debugger visualizes the power consumption data both statically and dynamically in different views and provides both power profiling and debugging opportunities for an application.

How does power debugging work?

The technology for power debugging is based on the ability to sample the power consumption and correlate each sample with the program's instruction sequence and hence with the source code. One difficulty is to achieve high precision with sampling. The ideal would be to sample the power consumption with the same frequency the system clock uses, but power system capacitances will reduce the reliability of such measurements. Simply put, the measured power consumption will be 'smeared' in relation to what is actually being consumed by the CPU and peripherals.

In practice this is not a problem. From the software developer's perspective it is more interesting to correlate the power consumption with the source code and various events in the program execution than with individual instructions, so the resolution needed is much lower than one sample per instruction.

Power is measured by the debug probe, IAR I-jet in this case, by measuring the voltage drop across a small resistor in series with the supply power to the device. See figure 1. The voltage drop is measured by a differential amplifier and then sampled by an AD converter.

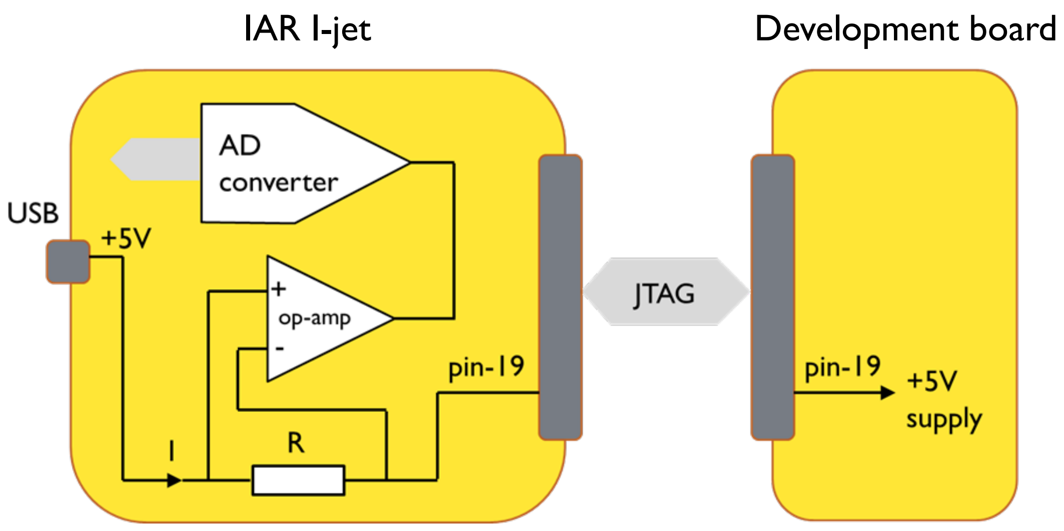


Figure 1: Power measurement

The key to accurate power debugging is a good correlation between the instruction trace and the power samples. The best correlation can be done if complete instruction trace is available, as is the case for ARM MCU's with ETM support. The drawback with using ETM is that it requires a special debug probe and ETM support in the device itself.

Less accurate but still giving good correlation is to use the PC sampling facility available in the ARM Cortex-M3/M4 cores, see figure 2. The DWT module implements the PC sampler, it samples the PC (Program Counter) periodically around 10000 times per second and triggers an ITM packet for each sample taken. The ITM is the formatter for events originating from the DWT. It packetizes the events and timestamps them. The debug probe samples the power consumption of the device using an AD converter. By time stamping the sampled power values and the PC samples it is possible for the debugger to present power data on the same time axis as graphs like interrupt log and variable plots, and to correlate power data to source code, see figure 3.

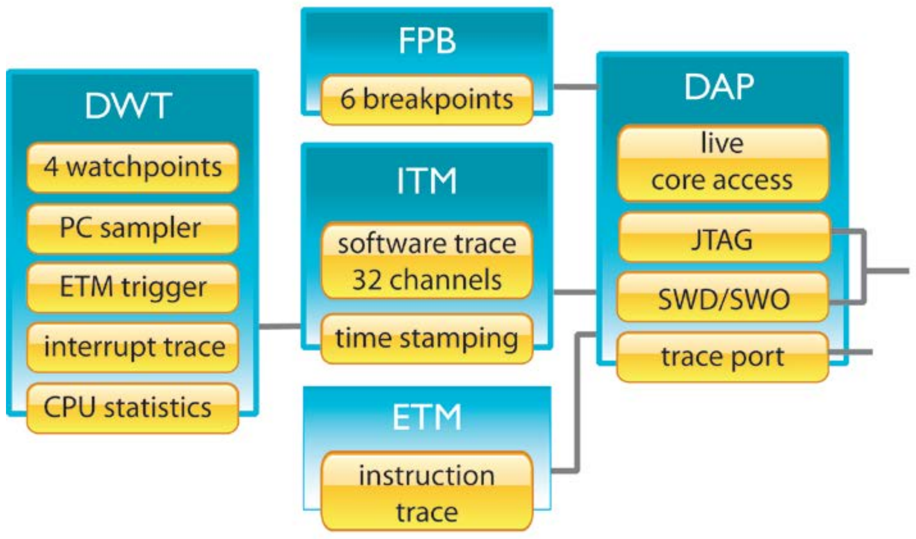


Figure 2: The debug module on Cortex-M3/M4

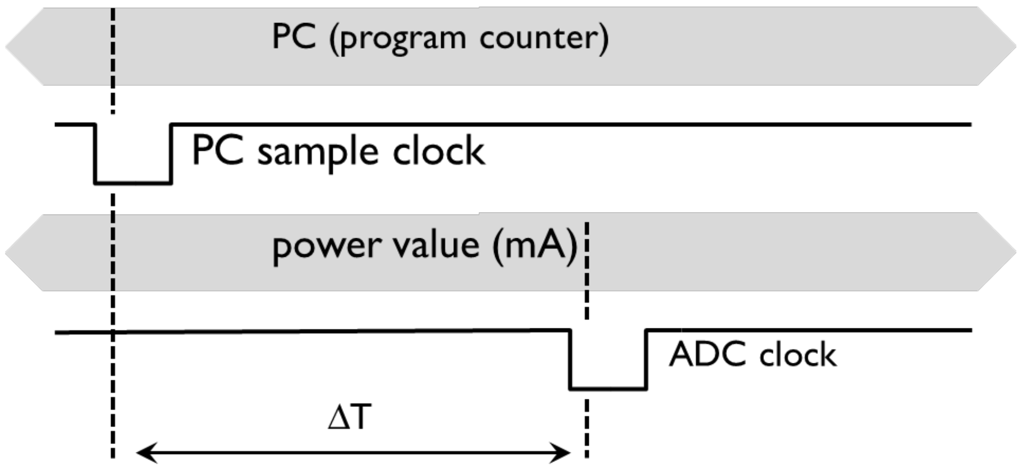
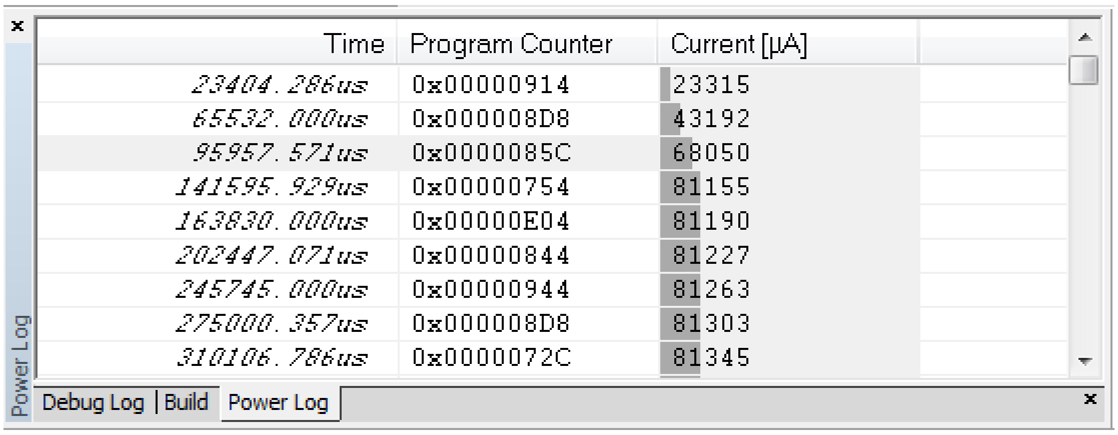


Figure 3: PC and power sample correlation

Power debugging in IAR C-SPY

As stated before, power debugging is based on the ability to sample the power consumption and correlate each sample with the source code. In IAR Embedded Workbench the power samples can be displayed in different formats. The **Power Log** window is a log of all collected power samples. This window can be useful to find peaks in the power sampling and since the power samples are correlated with the executed code, it is possible to double-click on a value in the **Power Log** window to get to the corresponding code. Depending on the power sampling frequency, the precision will be different, but there is a good chance that you find the code sequence that caused the peak.



Time	Program Counter	Current [µA]
23404.286us	0x00000914	23315
65532.000us	0x000008D8	43192
95957.571us	0x0000085C	68050
141595.929us	0x00000754	81155
163830.000us	0x00000E04	81190
202447.071us	0x00000844	81227
245745.000us	0x00000944	81263
275000.357us	0x000008D8	81303
310106.786us	0x0000072C	81345

*Figure 4: **Power Log** window*

Another way of viewing the power samples is via the **Timeline** window. In the **Timeline** window, the power samples are displayed in a time scale together with interrupt activities and up to four application variables that you can select.

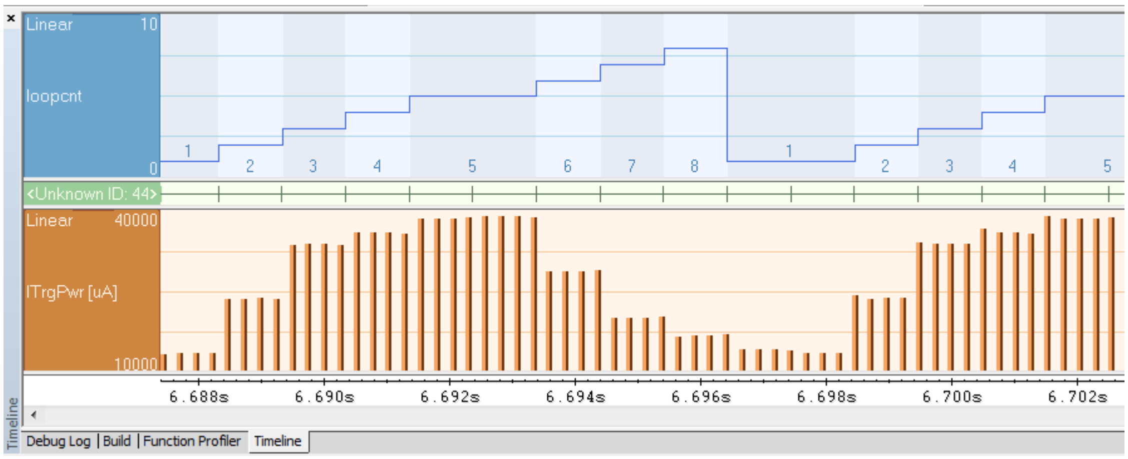


*Figure 5: **Timeline** window*

In embedded systems, peripheral devices often account for much of the power consumption, and software controls how they are used, regardless of whether they are integrated into the micro-controller or not. This view provides a very convenient way of viewing the power consumption in relation to both function calls and, if variables that are related to the status of a certain peripheral is used, also to activities that increase the power consumption on the board. The goal here is of course to see if the code can be optimized in the power domain.

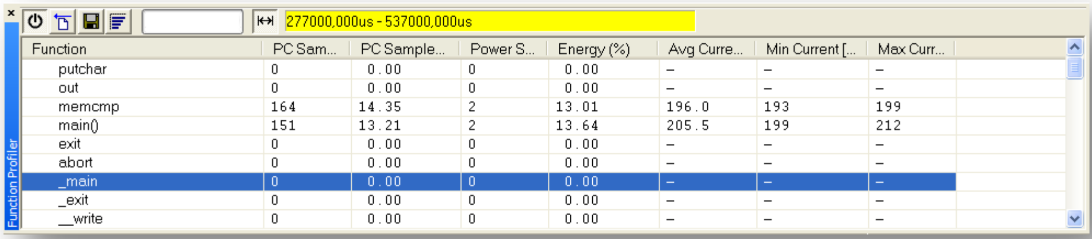
Also the **Timeline** window is correlated to both the **Power Log** window and the source code windows, so that you are just a double-click away from the source code that corresponds to the values you see on the time line.

Power profiling

In practice and in a task oriented system it is probably more interesting to see how a particular function affects power consumption than to see statement by statement how the power consumption changes. The function profiler will help you find the functions where most time is spent during execution, for a given stimulus. In this way, regions in the application where optimizations for power consumption should be done can be exposed.

On a Cortex-M3 device the debugger can utilize the PC sampling possibility in the DWT module. This allows the debugger to sample the PC and provide statistical profiling. The profiler finds the function that correlates to the sampled PC value and builds an internal data base of how often the functions are executed to generate function profiling information. The profiling information for each function in an application, will be displayed in a debugger while the application is running.

With power profiling we combine the function profiling with the power sampling to measure the power consumption per function and display that in the **Function Profiler** window.



Function	PC Sam...	PC Sample...	Power S...	Energy (%)	Avg Curre...	Min Current [...	Max Curr...
putchar	0	0.00	0	0.00	–	–	–
out	0	0.00	0	0.00	–	–	–
memcmp	164	14.35	2	13.01	196.0	193	199
main()	151	13.21	2	13.64	205.5	199	212
exit	0	0.00	0	0.00	–	–	–
abort	0	0.00	0	0.00	–	–	–
_main	0	0.00	0	0.00	–	–	–
_exit	0	0.00	0	0.00	–	–	–
__write	0	0.00	0	0.00	–	–	–

*Figure 6: **Function Profiler** window with power information*

The **Function Profiler** window will list the number of samples per function and also the average values together with max and min values. Once again we have a convenient way of finding peaks and abnormal behavior when it comes to power consumption in an embedded system. The system can appear to be fully functional and behave as expected in tests, but the power consumption can be much higher than it should and now we have a way to see that.

Optimizing code for power

In general, optimizing for power is very similar to optimizing for speed. The faster a task is executed, the more time can be spent in a low power mode. So by maximizing the idle time we are reducing the power consumption.

We will now look at some examples to illustrate the difficulty in identifying how a system unnecessarily consumes energy and where the system can be optimized for power. Typically it is not explicit flaws in the source code that is exposed, but rather opportunities to tune how the hardware is utilized. Sometimes, however, it may involve what might be described as pure bugs.

Waiting for device status

One common mistake that could cause unnecessary power to be consumed is to use a poll loop to wait for a status change of for example a peripheral device. Code constructions like the examples below execute without interruption until the status value changes into the expected state.

```
while (USBD_GetState() < USBD_STATE_CONFIGURED);
while ((BASE_PMC->PMC_SR & MC_MCKRDY) != PMC_MCKRDY);
```

Another related code construction is the implementation of a software delay as a `for` or `while` loop like in the following example:

```
i = 10000;                              // SW Delay
do i--;
while (i != 0);
```

This piece of code keeps the CPU very busy executing instructions that do not do anything except pass time.

In both of these situations, the code could be changed to minimize the power consumption. Time delays are much better implemented by using a HW timer. The timer interrupt is set up and after that the CPU goes down into a low power mode until it is awakened by the interrupt. Also a polling of a device status change should be solved with interrupts if possible, or by using a timer interrupt so that the CPU can sleep between the polls.

Depending on the characteristics of the embedded system, it could be difficult to find these situations using power debugging. One way forward is to use the different power debugging windows to get to know the power profile of the application so that abnormal behavior can be easily identified.

DMA vs. polled I/O

DMA has traditionally been used to increase transfer speed. In the MCU world, chip vendors have invented a plethora of DMA techniques to increase flexibility, speed, and to lower power consumption. In some architectures, the CPU can even be put into sleep mode during the DMA transfer. Power debugging allows the developer to experiment and see directly in the debugger what effects these DMA techniques will have compared to a traditional CPU driven polled approach.

Low power mode diagnostics

Many embedded applications spend most of their time waiting for something to happen: receiving data on a serial port, watching an I/O pin change state, or waiting for a time delay to expire. If the processor is still running at full speed when it is idle, battery life is being consumed while very little is being accomplished. So in many applications the microprocessor is only active during a very small amount of the total time, and by placing it in a low power mode during the idle time, the battery life can be extended by orders of magnitude.

A good approach is to have a task-oriented design and to use an RTOS: in a task-oriented design, a task can be defined with the lowest priority, and it will only run when there is no other task that needs to run. This idle task is the perfect place to implement power management. In practice, every time the idle task is activated, it puts the processor into a low power mode. Many microprocessors and other silicon devices have a number of different low power modes, in which different parts of the processor can be turned off when they are not needed. The oscillator can for example either be turned off or switched to a lower frequency, peripheral units and timers can be turned off, and the CPU stops executing instructions. The different low-power modes have different power consumption based on which peripherals are left on.

A power debugging tool can be very useful when elaborating with different low-level modes. The **Function Profiler** could be used to compare the power measurement for the task or function that brings the system down to the low power mode when different low power modes are used. Both the mean value and the percentage of the total power consumption could be useful in the comparison.

CPU frequency

Power consumption in a CMOS MCU is theoretically given by the formula:

$P = f \times U^2 \times k$

where f is the clock frequency, U is the supply voltage and k is a constant.

Power debugging allows the developer to verify the power consumption as a factor of the clock frequency. A system that spends very little time in sleep mode at 50MHz is expected to spend 50% of the time in sleep mode when running at 100MHz. The power data in the debugger will allow the developer to verify expected behavior and if un-linear dependency on the clock frequency exists chose the operating frequency that gives the lowest power consumption.

Interrupt handling

Let’s take an example involving interrupts to illustrate another situation when it is difficult to identify that a system consumes unnecessary energy.

Figure 7 shows a schematic diagram of the power consumption of an event driven system where the system at t_0 is in an inactive mode and the current is I_0. At t_1 the system is activated whereby the current rises to I_1 which is the system's power consumption in active mode with one used peripheral device. At t_2 the execution becomes suspended by an interrupt which is handled with higher priority. Peripheral devices that were already active are not turned off, although the thread with higher priority is not using them. Instead, more peripheral devices are activated by the new thread, resulting in an increase in current I_2 between t_2 and t_3 where control is handed back to the thread with lower priority.

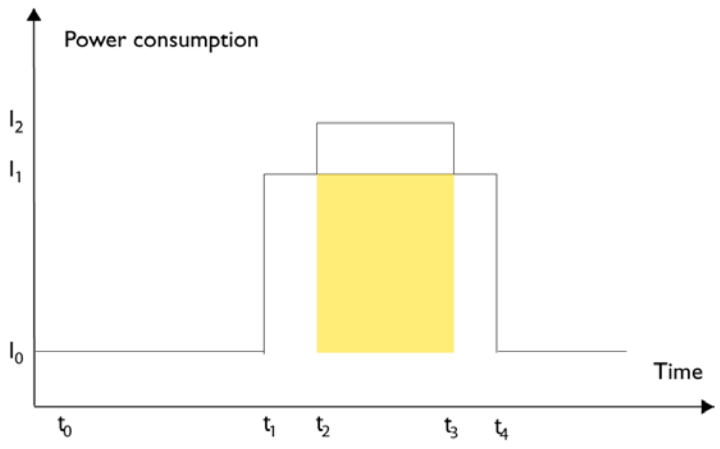


Figure 7: Schematic diagram of the power consumption

The functionality of the system could be excellent and it can be optimized in terms of execution speed and code size. But in the power domain more optimizations can be done. The yellow area represents the energy that could have been saved if the peripherals that are not used between t_2 and t_3 had been turned off, or if the priorities of the two threads could have been changed.

Using power debugging would have made it easy to discover the extraordinary increase in power consumption that occurs when the interrupt hits and identify it as abnormal. A closer examination of the **Timeline** window could have identified that unused peripheral devices were activated and consuming power for a longer period than necessary. Naturally there would have to be a review of whether it is worth to spend extra clock cycles to turn on and off peripherals in a situation like the example describes.

Finding conflicting hardware setup

To avoid floating inputs it is a common design practice to tie unused MCU I/O pins to ground. If the software by mistake configures one of the grounded I/O pins as a logical ‘1’ output, a current as high as 25mA may be drained on that pin. This high unexpected current is easily observed by reading the current value from the power graph; it is also possible to find the corresponding erratic initialization code by looking at the power graph at application startup. A similar situation will arise if an I/O pin is designed to be an input and is driven by an external circuit, but the software incorrectly configures the input pin as output.

Analog interference

Mixing analog and digital circuits on the same board has its own challenges. Board layout and routing becomes important to keep the analog noise levels low to ensure accurate sampling of the analog signals. Doing a good mixed signal design requires careful hardware considerations and skills. Software design can also affect the quality of the analog measurements. Performing a lot of I/O activity at the same time as sampling analog signals will cause many digital lines to toggle state at the same time, a candidate for introducing extra noise into the AD converter.

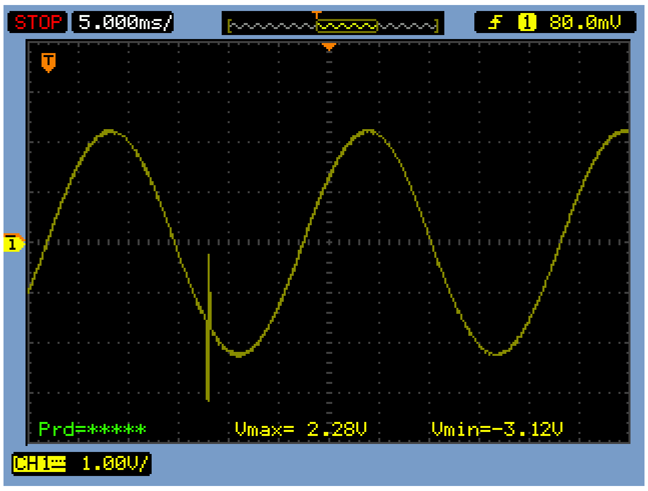


Figure 8: Power spike due to stepper motor interfering with AD-sampling

Power debugging can aid in investigating interference from digital and power supply lines into the analog parts. Interrupt activity can easily be displayed in the **Timeline** window together with power data, by studying the power graph right before the AD-converter interrupts. Power spikes in the vicinity of AD conversions could be the source of noise and must be investigated. All data presented in the timeline window is correlated with the executed code, simply double-clicking on a suspicious power sample will bring up the corresponding C source code.

Conclusion

The power debugging methodology described above gives the embedded developer the opportunity to examine the application program code and flow with regard to power consumption. With this knowledge, they can synchronize and optimize the source code in order to minimize the energy requirements. Using this method, engineers can ensure that their project is as energy efficient as possible without compromising the performance of the application.