Designing Energy-Efficient Systems with Cortex-M Microcontrollers

Reinhard Keil, ARM Germany GmbH
Frank Grobe, Hitex Development Tools GmbH

Energy-efficient embedded systems are essential for many application areas, even when it is not obvious at a first glance. Systems that replace mechanical components such as light switches, energy meters, or door looks are in permanent operation and need therefore energy efficiency. The automotive market needs energy-efficient electrical systems to reduce the gas per mile costs of the vehicles. Systems that enhance energy efficiency, for example a BLDC motor controller, need an energy efficient microcontroller to maximize the total effect. Of course, battery operated equipment is followed by the need for the battery to last longer for convenience, environmental, and cost reasons.

At the same time, software applications become more complex and demand higher performance, making the challenge of designing systems for low energy even more difficult. The following sections discuss various design considerations for the software architect and the application programmer to improve the energy efficiency of an embedded system.

This article is grouped into two parts that discuss:

1. **Design Considerations:** provides an in-depth overview of the various aspects that should be considered to create and energy efficient embedded system.
2. **Practical Implementation and Power Optimization:** demonstrates with a Cortex-M3 processor-based system how to analyze and optimized the energy profile of an application.

Part 1: Design Considerations

To create optimal embedded system several design aspects must be considered that can be grouped into:

- **System Design** which influences the selection of the microcontroller. Today’s microcontroller offers multiple power-saving modes and the computing power may be used to replace external hardware with software algorithms.
- **System Configuration** is increasingly complex in high-end microcontrollers. Devices offer various clock sources, allow different processor and peripheral clocks, and provide power-down features for individual peripherals.
- **Software Structure** impact the energy consumption of an embedded application and applications must effectively utilize the microcontroller power-saving modes. Therefore the “best-practice” structure for the application software must be used.
- **Development Tools** can optimize program code and memory accesses towards energy efficiency. For the application programmer, it is important to understand the impact of the various optimizations that are available.
- **Software Libraries** for communication protocol, graphical user interface, or other highly optimized algorithms can reduce the number of instructions executed and therefore increase energy efficiency.

Throughout this article the following terms describe the various operating states of the embedded system:

- *active* is the state where the embedded system is fully operating and interacting with most peripherals.
- *standby* is the state where the embedded system operates with reduced functionality, for example accepting only a command subset or displaying status information.
- *inactive* is the state where the embedded system is effectively powered-off and is waiting for a command or input to transition to *active* or *standby*. Even in this state the system may need to display status information.

System Design

Modern microcontrollers offer multiple power-saving modes (table 1) and the processor along with peripheral and bus systems can run with various clock frequencies.

Table 1: characteristics of typical power-saving modes of modern microcontrollers

Characteristics	**High** ←	**Energy Consumption**		→	**Low**
Power-saving mode	"Run"	"Sleep"	"Deep Sleep"	"Stop"	"Shut-Off"
Processor Clock	On	Off	Off	Off	Off
Processor Power	On	On	Off	Off	Off
Wake-up Time	-	0 µs	2 µs	200 µs	200 µs
High-Frequency Peripherals	Available	Available	Stopped	Stopped	Stopped
Low-Frequency Peripherals	Available	Available	Available	Stopped	Stopped
CPU and RAM Retention	On	On	On	On	Limited

Each silicon vendor has different names and characteristics for the various power-saving modes, which introduces another level of complexity for selecting the microcontroller device. Application programmers therefore frequently use just the "Sleep" power-saving mode, where only the CPU clock is turned off while all peripherals and the complete memory system is running at full speed. From "Sleep" mode the device wake-ups instantly on any interrupt or event.

More advanced power-saving modes switch off peripherals, memory, and clock signals and therefore require a wake-up time before the application software can continue to execute. This adds further complexity to the software but can have a major impact to the total power consumption of the embedded system.

Many microcontroller devices also offer various oscillators, tuned for accuracy or low-power, and a PLL to multiply the oscillator frequency for the processor, the peripherals, and the bus clocks. Often there is a high-speed and low-speed clock for peripherals implemented to optimize the power profile of the embedded system.

Initial Investigation of System Requirements

For an effective system design it is essential to evaluate the requirements of the application. The challenge for the software architect is to match the operating states of the embedded system to power-saving and clocking modes of the microcontroller hardware. Below some questions are listed that the software architect should evaluate before a specific microcontroller or system hardware is selected.

How long is the embedded system in each state? What delay is acceptable for the state transitions? The *standby* state may be omitted to simplify system design.

For each of the *active*, *standby*, and *inactive* state determine the following:

- What peripherals are required by the application in each state?
- What are the specific characteristics for these peripherals? Is, for example, an accurate communication baud rate required? What is the speed of input/output signals in each mode?
- What processor performance is required to run the application software?
- What are the requirements for the clock system? Is a precise clock required or is it possible to use a low precision, low-power, low-frequency clock?
- What code and data memory sizes does the application software have? Is it possible to separate the memory into portions used by the *standby* state and the *active* state?

Every embedded system must provide an *inactive* state and the easiest implementation is to power-off the whole system. However, electronic devices often need to display status information and must wake-up from *inactive* state by a remote command or a keyboard entry. Some microcontrollers offer a system wake-up from "Shut-Off" mode by monitoring transitions on I/O pins. Can this be used to wake-up from the *inactive* state?

Finally you should evaluate whether external peripherals and analog components can be potentially replaced by software algorithms. Removing hardware will save energy for the component, but the processor will consume additional energy to execute the software algorithm. In which operating states are these peripherals active and what is the impact on the required processor performance?

Selecting the Best Matching Microcontroller

Once these requirements are captured, the software architect can evaluate a matching microcontroller device. Of course, all required peripherals must be available on-chip or be easy to connect to the device. In this process, use the information in the device data sheets for the power characteristics in each power-saving mode. Also a short wake-up time from power saving modes is important to reduce energy consumption. But, the total energy consumption depends on many more factors, for example the memory system, the processing time needed (figure 1), and the I/O load. Hence, data sheet parameters should be used cautiously during this selection process.

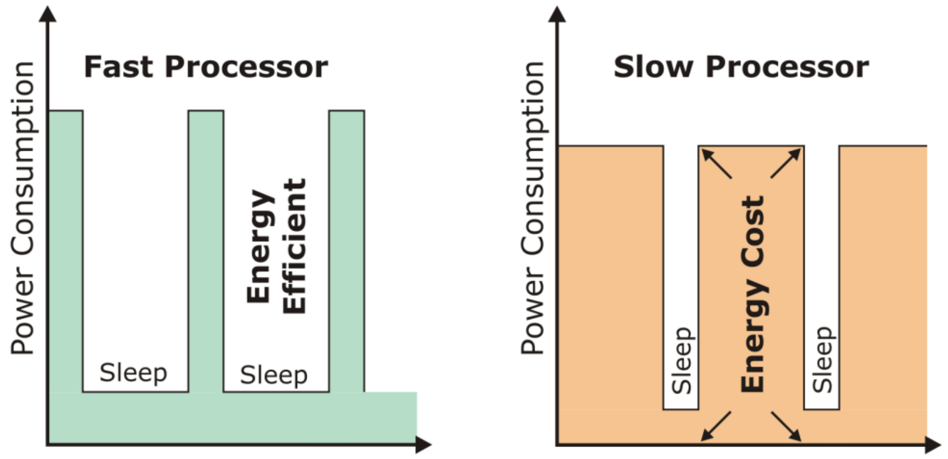


Figure 1: A faster microcontroller needs less processing time for algorithms, is therefore longer in power-saving modes and can be more energy efficient.

Long-term aspects must be considered, since application requirements are frequently modified over the life-time of a product. When designing a new product it is hard to capture all requirements and therefore microcontroller families which offer downgrade and upgrade devices are good choices. This simplifies software migration in the case that the microcontroller selected from the family is not the best fit long term.

System Configuration

Once the microcontroller has been selected, determine the optimum system configuration for each of the *active*, *standby*, and *inactive* state. The goal is to select the lowest possible overall power configuration and the main objective is to evaluate the best clock frequency for the processor and the peripherals. At this point a simplified general rule may be used: if you double the speed, you double the power consumption. However in certain applications a higher processor speed will achieve lower overall system power consumption since faster code execution means that the application may stay longer in power-saving modes and switch off peripherals.

Also important is the power supply available in the embedded system. Especially for battery operated equipment this question is essential: how fast can the microcontroller run across the whole voltage range? In real life, the supply voltage is often dictated by the battery. A higher active power consumption of a microcontroller means that the peak power consumption for the battery is higher and this can reduce the lifetime of batteries.

During the evaluation of the best system configuration the following guidelines are helpful:

- Try to architect the embedded system to run as slowly as practical.
- Allow the processor to dynamically change the speed of operation based on demand for computing power in each operating state.
- Disable peripherals and clocks when they are not needed in an operating state.

Identify the Optimum System Configuration

Modern microcontrollers have a variety of configuration settings and it is challenging for the software architect to find the best match for the application. To find a good match you should consider the following:

- Many microcontrollers have low frequency and high frequency clock sources. Often a low-power, low frequency on-chip oscillator is available, but the precision of this clock may be insufficient for the *active* state of the embedded system. Communication protocols, for example, need an accurate baud rate and require an external crystal-based oscillator. However, clock sources can change dynamically, and a low-power oscillator may be used in the *standby* or *inactive* state.
- Microcontroller peripheral clocks might vary from the processor clock. Try to run the peripherals as slowly as practical. The maximum speed of input/output signals and the need for specific communication baud rates are the main factors in this consideration.
- Peripherals might be configurable to use a direct memory access (DMA) unit for automated data transfers to and from memory. This eliminates the need for processing power for such data transfers.
- It might be possible to remove power and/or clock signals from peripherals to reduce the overall power-consumption.
- I/O port pins frequently have programmable characteristics with configurable pull-up or pull-down resistors. Chose a configuration that minimizes power consumption.

The configuration can dynamically change during execution of the application program. It is a good practice to assign each operating state a specific configuration as exemplified in table 2.

Table 2: Sample system configuration of a microcontroller application

Operating State	System Configuration
Active	▪ Processor is running from high-frequency external crystal-based oscillator ▪ All peripherals are powered and run from high-frequency clock ▪ All I/O Ports are powered and configured ▪ System Tick Timer is used as RTOS clock ▪ "Sleep" mode is used as power-saving mode
standby	▪ Processor is running from low-frequency on-chip oscillator ▪ Only selected peripherals are clocked: Comparator, Real-Time Clock ▪ Only few I/O port pins are enabled for status output ▪ Real-Time Clock provides a wake-up timer ▪ "Deep-Sleep" mode is used as power-saving mode
inactive	▪ Processor in "Stop" mode ▪ Wake-up via an input signal

Once the system configuration for the various operating states is decided, the application programmer may create functions for each configuration setup. It is important to understand how state transitions work. For example, the transition from *inactive* to *active* state may be identical to a power-on reset, except that a peripheral register indicates this by a special reset type. Transitions between other states might be initiated with a function call that incurs an oscillator or PLL setup delay. That must be considered by the application programmer.

Typically, "Sleep" and "Deep Sleep" modes are automatically terminated when an interrupt or event occurs. Therefore these modes are easy to implement.

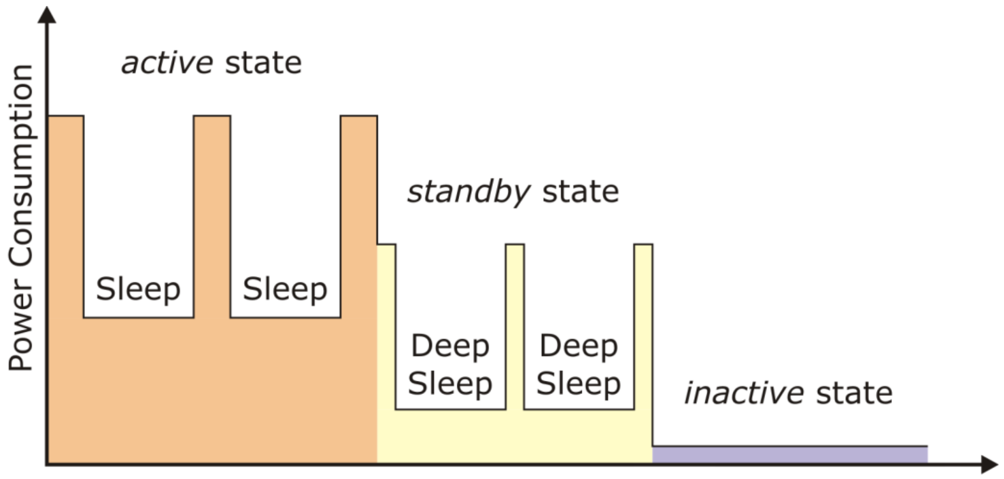


Figure 2: power profile of a microcontroller application

Figure 2 depicts a power profile of an application in the various operating states along with the usage of the power-saving modes. By using the "Sleep" and "Deep Sleep" modes a significant power saving can be achieved. During "Sleep" and "Deep Sleep" times, the processor is not active and does not execute any application program. To optimize the power profile of an application it is therefore important to enhance the software performance and several methods are outline in the next sections.

In a practical embedded system the power profile is affected by many electrical characteristics (such as the impedance of the power supply unit and blocking capacitors in the voltage supply). Fast dynamic currency changes are smoother in a real system since the charge changes of capacitors limit the effect of switching between different power saving states. The calculation of a power profile in real-world applications is complex and it is therefore easier to use measurement equipment that is designed for the validation of the system energy consumption.

Software Structure

The software structure has a major impact on the energy consumption of an embedded system. The application program should put the processor into the “Sleep” mode whenever possible. Software overhead should be minimized and there are several ways to achieve that. For example, instead of device polling use interrupt driven I/O and direct memory access (DMA).

For embedded systems two general software design patterns are typical:

- **Super-Loop** where the functions of the application are called one-by-one in a single loop.
- **RTOS** (Real-Time Operating System) where the application functions are split into individual tasks or threads.

Each design pattern has pros and cons and selecting the best matching pattern for the application is essential, not just for maximizing energy efficiency.

Super-Loop Design Pattern

The Super-Loop design pattern is a good choice for small embedded systems or the portion of the embedded application that runs in *standby* or *inactive* state. An endless loop calls the functional blocks of the application and the Super-Loop executes them in fixed order. Using device power-saving modes is easy: when interrupt events deliver timer ticks or perform the application I/O, the software just needs to invoke the “Sleep” mode at the end of the Super-Loop (listing 1).

Listing 1: Sample program using Super-Loop design pattern

```
void SysTick_Handler (void) {                // Interrupt Handler
  ;                                          // Get a periodic interrupt every 10ms
}

void SuperLoop (void) {
  Device_Initialization ();                  // Configure the device
  SysTick_Config (SystemFrequency / 100);    // Setup System Tick Timer

  while (1)  {                               // Endless loop (the Super-Loop)
    Get_InputValues ();                      // Read values
    Calculation_Response ();                 // Calculate results
    Output_Response ();                      // Output results
    Wait_For_Event ();                       // "Sleep" until next event or interrupt
  }
}
```

The fixed execution order of the Super-Loop simplifies the data communication between the functional blocks. Most applications based on this pattern just use shared or global variables for information exchange.

Unfortunately, the simplicity of the Super-Loop design pattern has several serious disadvantages for more complex applications. Since all time-critical operations must be processed within interrupt service routines (ISR), these functions become complex. ISRs are important for using power-saving modes, but each interrupt event in an application causes a wake-up from power-saving. And the Super-Loop needs to query each result of the ISR processing which may create time-consuming polling calls. Therefore the effect of the power-saving modes is minimal in more complex Super-Loop applications.

Super-Loop applications quickly become complex. Since all functional blocks need to execute in a reasonable time, more complex jobs must use local state machines to split up the job into smaller portions. When doing so, the power-saving mode switching becomes more complex and therefore ineffective too. In general, a large Super-Loop application is hard to extend and a simple change in the source code may have unpredictable side effects and such side effects are time consuming to analyze.

RTOS Design Pattern

The problems of the Super-Loop design pattern for more complex applications are well understood and a Real-Time Operating System (RTOS) solves many of these issues. In the RTOS design pattern, each functional block is designed as a single tasks or thread. Each task can be viewed as a single Super-Loop (figure 3). The RTOS assigns processing time to each task on demand based on two basic methods:

- Event-driven, pre-emptive task switching occurs only when an event of higher priority needs service.
- Time-sharing or round-robin task switching where each task is executed for a period of time. Time-sharing designs switch tasks more often than needed, but give the illusion that a task has sole use of a processor.

Typically an RTOS kernel allows waiting for events such as timeout, signal, or messages from other tasks. While a task is waiting, the RTOS can assign the processor to other tasks. When the RTOS has no other task to execute it runs the idle task that is waiting for events. In many cases an ISR is the most efficient way for interfacing to device peripherals and an RTOS that is designed for embedded applications should support therefore data communication with interrupt handlers.

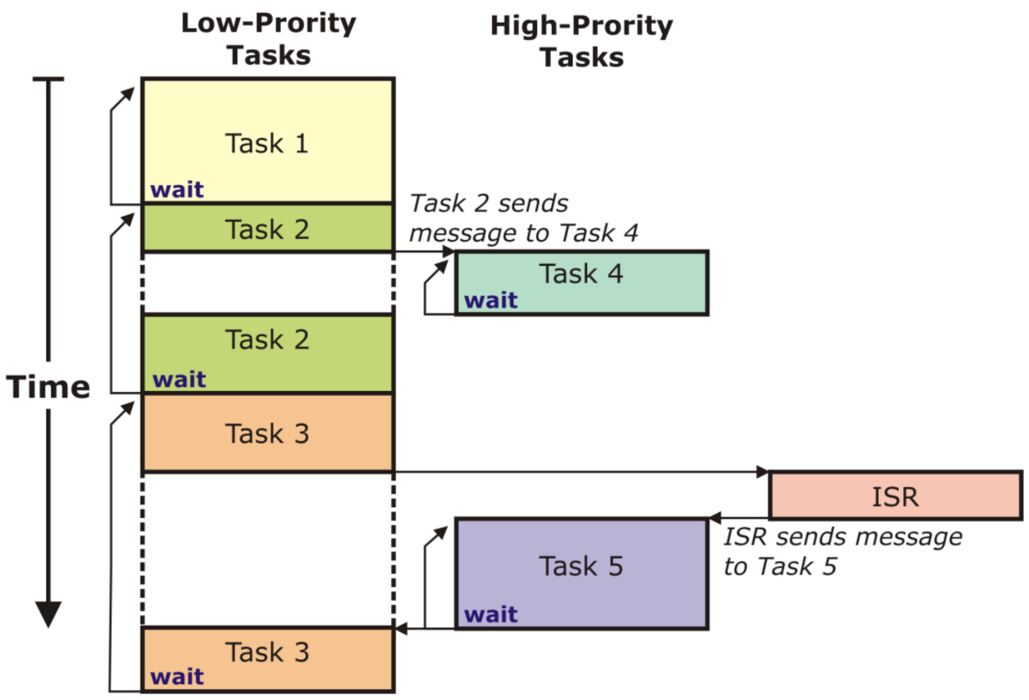


Figure 3: task and interrupt execution of an RTOS system

To achieve a power-efficient RTOS design, the idle task must invoke the power-saving modes of the microcontroller. In addition, the application software must be designed to use the resource management features of the RTOS:

- Every RTOS has a system clock derived from a system tick timer. Configure this system tick timer clock as slowly as practical. In a pre-emptive RTOS high priority tasks are running instantly when related events occur. Therefore, configuring the system tick timer should be based on the time intervals needed by the software applications.
- If possible, avoid time-sharing or round-robin task switching. Each task switch needs processor time and therefore it is better to complete a task or job. Instead, use task priorities for tasks or jobs that need to run instantly. The correct usage of pre-emptive task switching and the inter-task communication features is essential for an energy-efficient RTOS application.
- Select an RTOS that has been specifically design for embedded systems and can exchange messages and signals with an ISR. Effective peripheral communication is typically implemented using interrupt handlers and therefore data exchange from an ISR to a task is important.
- For more complex applications, choose middleware that uses the RTOS features for memory pools, message passing, timeouts, and signals.

Interrupts and DMA

When the processor has no more tasks to execute it should be put into a power-saving mode. For maximizing the time the application spends in power-saving it is essential to minimize the number of processor instructions executed. Embedded applications frequently use I/O operations that can be optimized using interrupts or DMA:

- Interrupt events typically indicate that a peripheral can receive or provide data. The ISR can directly perform the required I/O operation. Processing a block of data is in many cases more energy efficient than processing lots of individual items and therefore the ISR may collect I/O data in a memory block.
- DMA is a hardware unit that performs block oriented I/O operations. The data are directly copied to/from the peripheral from/to a memory block. Interrupts events are generated only at the end of each block transfer, but not for each data item. Using DMA improves performance, reduces interrupt overhead, and does not required software memory copy functions.

A good RTOS should provide support for block-oriented data handling (figure 4). To achieve high performance with block data a deterministic memory allocation function is needed. A FIFO mailbox buffer avoids overruns with high data traffic. Such a structure has minimal software overhead since the data buffer can be identical to the I/O buffer and therefore eliminates data coping.

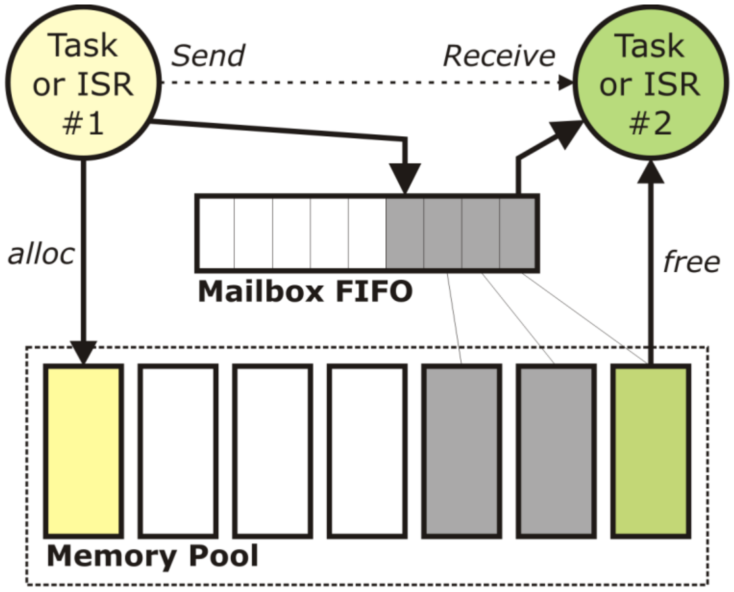


Figure 4: Block I/O operations in RTOS use memory pools and a mailbox FIFO for communication

Power-Saving Modes

The section **System Design** lists typical characteristics of power-saving modes and the section **System Configuration** exemplifies how such modes can match to the operating states of an embedded system.

When a microcontroller goes from a “Deep-Sleep”, “Stop”, or “Shut-Off” mode, in which the oscillators are disabled, to the “Run” mode, there is always a wake-up period in which the processor must wait for the oscillators to stabilize before starting code execution. Since no processing can be done in this period, the energy spent while waking up is wasted. Microcontroller applications often have real-time deadlines and the response to an event must happen within a very short time period. “Deep-Sleep”, “Stop”, and “Shut-Off” disable high-speed peripherals and therefore the RTOS system tick timer clock might be no longer available.

For the *standby* or *inactive* state, where “Deep-Sleep”, “Stop”, or “Shut-Off” modes are used, the RTOS design pattern is not appropriate. In this states the Super-Loop design pattern is a better choice and both patterns can be mixed in a single embedded application.

Another strategy to save energy is to turn off peripherals when they are not used. Many microcontrollers allow dynamic switching of power and clock for peripherals during the execution of the application software.

Development Tools

The development tools are another important factor for creating energy efficient systems. Every CPU instruction executed and every memory access has an energy cost and to maximize energy efficiency, programs should be optimized for fast execution. The compiler used for creating the application software should be specialized for embedded systems since some of the very aggressive performance-improving optimizations done by “workstation” compilers may cost energy.

What is Better: Assembly, C, C++, or UML?

When starting a new project, it makes sense to evaluate the "best practice" approach for developing the application program. Embedded developers are commonly using the following languages:

- **Assembly language** gives direct control over each processor instruction generated. In theory assembly programming allows utmost efficiency, however in practice writing assembly code is very time consuming and requires long experience with the processor architecture. Complex assembly applications are hard to maintain and programs cannot be ported to other processor architectures. Hence, assembly programming is no longer the choice for embedded application development. Assembly programming still makes sense where specialized algorithms are needed, for example for digital signal processing.
- **C** is the most popular language for creating embedded applications and is well standardized. C is available for all today's microcontroller architectures and includes a C standard library which provides a good collection of utility functions. The C language matches well to today's processor architectures and experienced C programmers can write efficient programs that compare well with assembly language.
- **C++** extends C with build-in support for object oriented programming, virtual functions, operator overloading, and exceptions amount other features. C++ adds to the C standard library a Standard Template Library (STL) that supports sophisticated algorithms. For more complex applications C++ is widely used in the embedded industry. Some C++ features are hard to understand and when misused create inefficient program code with large memory requirements. However, when correctly used, C++ applications are efficient.

  Around ten years ago, Embedded C++ (EC++) was designed to solve the shortcomings of C++ for embedded applications. EC++ is a C++ dialect that removes several language features in an attempt to encourage, or guarantee, efficient implementation by compilers. EC++ never became popular and is no longer maintained. Perhaps a reason is that even the standard C language allows object oriented programming techniques and therefore the EC++ dialect makes only little sense.
- **UML** (Unified Modelling Language) is an object modelling and specification language that includes graphical notation techniques to create visual models. UML allows code re-factoring and provides methods for data flow and time control that typically use an RTOS as underlying run-time environment. The visual models can embed C and C++ language statements. In fact, UML code generators create C or C++ source code. UML designs create a clean software structure which uses RTOS data communication features. This can be more energy efficient than designs that use shared memory. But the UML code generators along with the RTOS integration are still inefficient and therefore not yet popular. However, UML may become the future language for embedded application programming.

The question, which language fits best for energy-efficient embedded systems depends on the application to be deployed. The wide range of embedded applications makes it hard to give a single recommendation, but the above explanation gives you hints for choosing the best fitting language. Other languages, such as Java, create serious overhead and are therefore unpopular for microcontroller applications unless portability and speed of creation are the paramount concerns.

Optimizing Memory Accesses

Embedded systems are typically resource constrained and provide various types of memory. Each memory access has an attached energy cost and there is a huge difference between the memory types:

- On chip SRAM is the fastest memory available in a microcontroller and provides the best energy efficiency. On-chip SRAM is size restricted and is typically used for variables only. In case that a microcontroller application executes a short routine during the *inactive* state, storing this code portion in on-chip SRAM may increase power-efficiency since then even the on-chip FLASH ROM can be powered off.
- The next best memory type is on-chip FLASH ROM that is used for program code and constants. On-chip FLASH ROM is slower then SRAM and usually switched off by the microcontroller hardware when the device is entering “Deep Sleep”, “Stop”, or “Shut Off” power-saving modes. Using on-chip FLASH ROM is therefore easy and a good choice for energy efficient program code.
- Accessing off-chip memory (SRAM, DRAM, or ROM) has a high energy cost. Compared to on-chip memory, these memory types can be 10 times slower and the energy required for a memory access can be up to 40 times higher.
- Systems that provide only off-chip memory typically integrate on-chip cache memory for code and data to speed-up program execution. On-chip cache memory greatly improves speed, but the energy efficiency is not as good as on-chip SRAM. For maximum energy efficiency with large application, locate frequently accessed code or data parts to on-chip SRAM or FLASH.

For creating an energy efficient application it is crucial to place frequently accessed program code and data in on-chip memory that does not require external bus activity. Applications with large memory requirements typically have hot spots. Figure 5 shows a memory access profile of such an application. In this example more than 95% of all accesses are made to objects that occupy less than 14% of the total memory. While applications vary, the general trend is that more than 90% of accesses are to less 20% of the memory area. Utilities that analyze such a memory profile and locate frequently used data to on-chip memory types would be therefore useful.

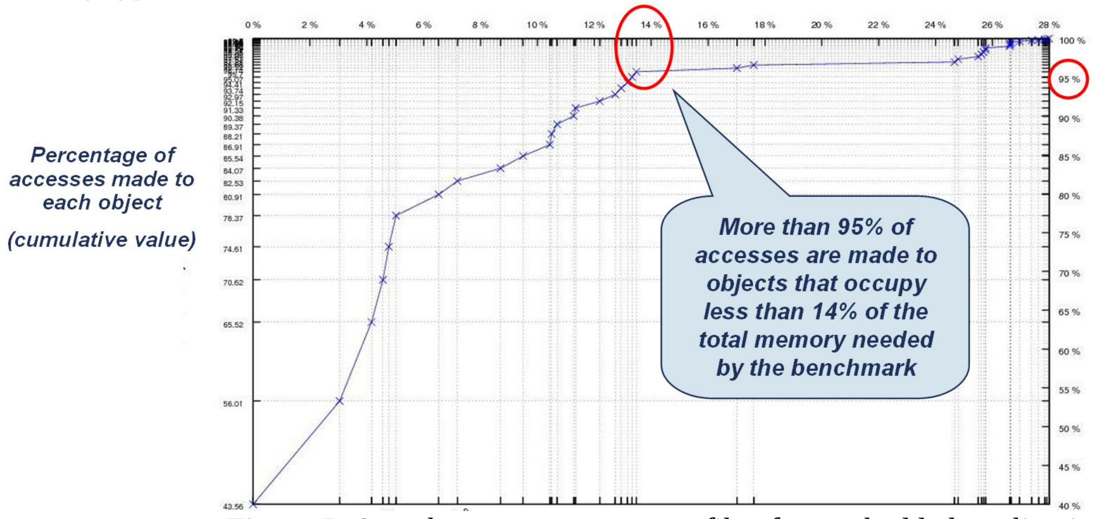


Figure 5: Sample memory access profile of an embedded application

For large applications it might be possible to separate the memory into portions used by *standby* state and by *active* state. For the portion used in the *standby* state use the on-chip memory resources of the device.

Compiler Optimizations

For optimizing the power profile of an application it is essential to reduce the number of instructions executed for a program algorithm. Therefore an optimizing compiler that is designed specifically for embedded systems is crucial. Today, the compiler vendors do not offer specific options that optimizing towards energy efficiency, but the following few guidelines are valid:

- Optimize for short execution time: compilers allow their users to favour execution speed or to minimize the code size. There are several techniques that speed-up program execution, but all of them reduce the number of instructions that are executed. In embedded system that provides only one type of program memory, optimization for short execution time is the best option.
- Modern compilers typically offer function in-lining, sometimes even across several program modules. This removes the function call overhead. C/C++ compilers typical support an `inline` keyword for function definitions which serves as a "hint" to the compiler that it should try to inline the function. The function in-lining mechanisms vary between compilers and the results are difficult to predict without carefully evaluating the tool chain.

Embedded systems with large off-chip program memory may provide an instruction cache to speed-up program execution. This cache reduces the number of off-chip memory accesses but is also size constrained. Program code executing from cache is more energy efficient than fetching from off-chip memory (the section **Optimizing Memory Accesses** describes the energy costs for memory accesses). Function in-lining and speed optimizations increases the total code size required, and when the executed code cannot be held in the cache for size reasons the program becomes inefficient. It is important to select a compiler that offers a good balance for such optimizations so that these negative effects are minimized.

Design Utilities

Silicon vendors frequently offer design utilities for their microcontroller portfolio. These help to configure the microcontroller system and the on-chip peripherals, but currently do not provide features for optimizing the energy profile. However, using such utilities can be a great help for the software architect and the application programmer.

Software Libraries

Today, complex applications are composed using software libraries and third-party middleware components. Middleware components implement typically communication stacks, flash file systems, or graphical user interfaces. When choosing such components for energy critical applications the following selection criteria should be used:

- A middleware component that interfaces to peripherals should use energy efficient I/O techniques such as interrupts or for bulk data transfers DMA.
- When the application is based on an RTOS, the middleware components should use the RTOS features for event control and block oriented data handling.

Other library components support specialized algorithms, in many cases based on digital signal processing (DSP) algorithms. Some CPU architectures provide DSP instruction set extensions. For example, the new ARM Cortex-M4 processor offers single instruction multiple data (SIMD) techniques (figure 6).

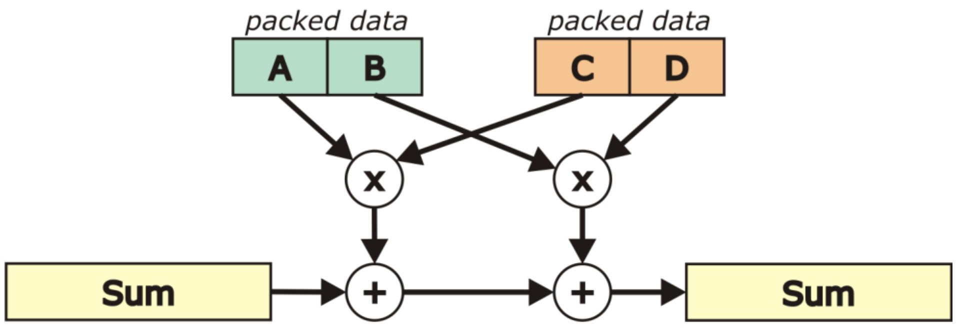


Figure 6: SIMD extensions perform multiple operations in one single-cycle CPU instruction, in this example: Sum = Sum + (A x C) + (B x D)

SIMD performs two or four identical operations with one single-cycle CPU instruction and even supports multiply-add operations that are frequently used within DSP algorithms. SIMD techniques operate with packed data and this can reduce also memory load instructions. Such instruction set enhancements drastically increase performance and energy efficiency at the same time since an algorithm can execute in fewer CPU cycles. However, library components must be optimized towards such instruction set extensions.

Part 2: Practical Implementation and Power Optimization

Low Power Requirements and Suitable Microcontrollers

Microcontrollers based on Cortex-M processors are leading in energy-efficiency. Through their high computing power, instructions are executed faster and the overall system resides in *Sleep Mode* and has the clocks powered-down for a longer time. Cortex-M processors also offer a *Deep-Sleep Mode* in which many processor peripherals are powered off. Peripheral voltage and clocks can be adjusted while the user-software is executing.

Optimization of the Application's Energy Consumption

In principle, two development steps are needed to optimize the energy consumption of an application. In the first step, the user-software influence on power consumption is analyzed and tuned for reducing power consumption. In the second step, realistic and unaltered power consumption measurements are performed.

The key aspect in the first step is the correlation between the power consumption changes and the software activity. To ensure a detailed view into the overall system behavior, the components of the development tool, the measurement equipment, and the debugger probes have to be adjusted and synchronized to work in cooperation.

The Embedded Trace Macrocell (ETM) of the Cortex-M processor offers a high-performing possibility to trace instructions. ETM correlates the program instructions to the power consumption precisely. Thereby it is required to record the ETM instruction trace and the current-/voltage progression simultaneously. ETM is not integrated in every Cortex-M processor, but each Cortex-M3 and M4 processor has a Serial Wire Viewer (SWV) interface, which can output interrupt events, code instrumentation, program counter (PC) values, or statistics about the *Sleep Mode*. Recording the start and end of each Interrupt Service Routine (ISR) is a simple and precise method to associate ISRs to the energy consumption trace. Two further trace methods are available for software portions not part of an ISR: PC sampling and code instrumentation. PC sampling records the actual PC value periodically, ensuring a gross correlation between program execution and energy consumption. A better and more accurate method is to record software events using code instrumentation. Various events, like activation of a PLL or periphery (for example ADC), can be recorded, making transparent the correlation between events and changes in energy consumption. Another possibility is to signal events by toggling a free I/O-pin. It is not possible to distinguish events with this method; however, based on its simplicity, it is often used.

After the software has been optimized and tuned for energy consumption in the first step, a thorough and precise energy consumption measurement is required in the second step. This is necessary, because the power consumption is falsified when the microcontroller is operating in debug and trace mode. Activating the debug unit of the Cortex-M microcontroller prevents using the *Sleep Mode* with all the features it offers. Debugging is not possible when *Sleep Mode* is fully active. In addition, the debugger probe itself adds to power consumption and falsifies the recorded energy demand. Therefore, it is absolutely necessary to disconnect the debug/trace unit from current/voltage recoding. Combined systems never measure the accurate power demand. Exclusively the measurements performed on separated systems ensure the exact and accurate power demand, which is a key criterion and has to be documented with each release.

Energy Analysis with PowerScale and ULINKpro

ARM/Keil and Hitex have combined the professional ULINK®*pro* debugger adapter with the Embedded Award-winning energy profiling tool PowerScale. The measurement hardware (ULINK*pro* and PowerScale) and the development tools (µVision and PowerScale GUI) have been synchronized for performing the first development step as described before.

µVision integrates the compiler, debugger, and RTOS into the development environment MDK-ARM. The µVision Debugger allows, in combination with ULINK*pro*, to trace consistently instructions using ETM, or using the high-speed SWV interface for recording frequently triggered ISRs. ULINK*pro* combined with SWV allows to trace events, to record *Sleep Mode* statistics, and to analyze ISR executions. In addition and when combined with ETM, application code-coverage and time execution analysis are possible.

PowerScale consists of a base unit, which can be connected to a PC via USB. Up to four probes can be connected to the unit measuring the current and voltage simultaneously on four different power domains (Figure 7). This allows measuring the real energy consumption. Long-time measurements are possible and collected data can be stored on the hard drive. The Active Current Measurement Technology has to be mentioned, which allows current measurements of a wide range located between 200nA and 500mA. Through statistical analysis, PowerScale GUI allows to evaluate energy- and current integrals for the selected software domains easily.

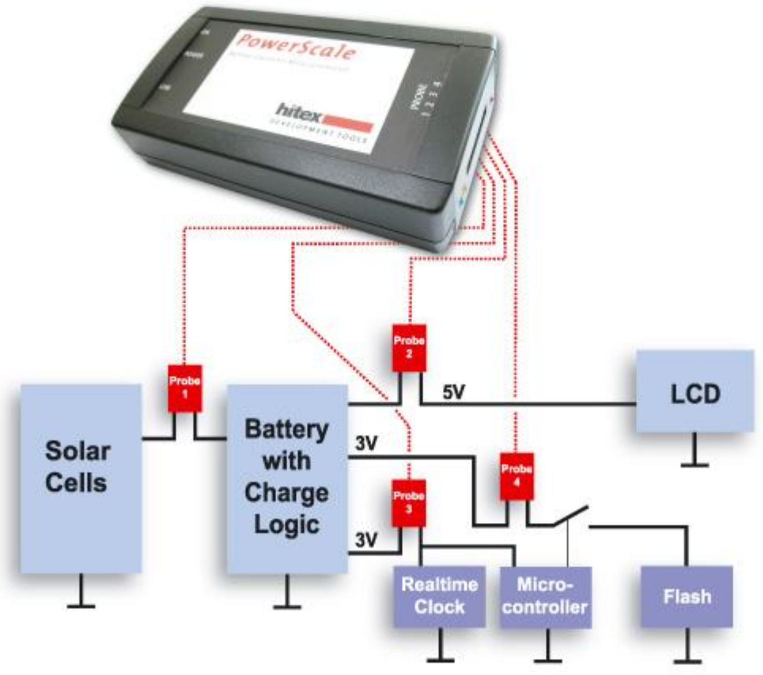


Figure7: PowerScale measures the current and voltage on up to four power domains.

The combination ULINK*pro*, PowerScale, and ETM Trace records the executed instructions and the current- and voltage usage of one or more domains. The timing adjustment of the entire system is done through a hardware synchronization signal (Figure 8). This allows identifying the position on the current-, voltage-, or energy diagram that corresponds to the traced instruction. And conversely, it allows identifying the traced instruction that caused any change from the current-, voltage-, or energy diagram. The coordination is ensured through the ability of the software components to communicate properly. A mouse-click into the µVision Instruction Trace window positions the cursor of the energy diagram displayed in the PowerScale GUI. In this way, the user-software impact on power consumption can be detected and optimized.

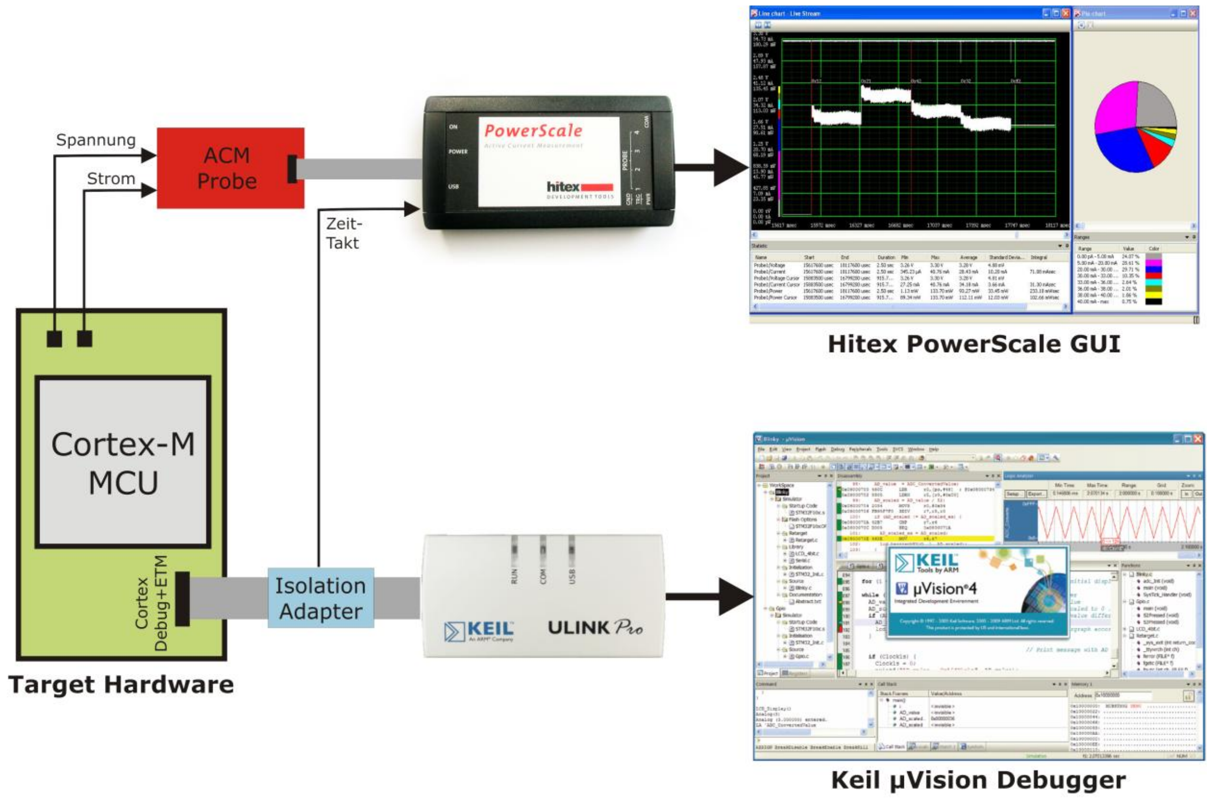


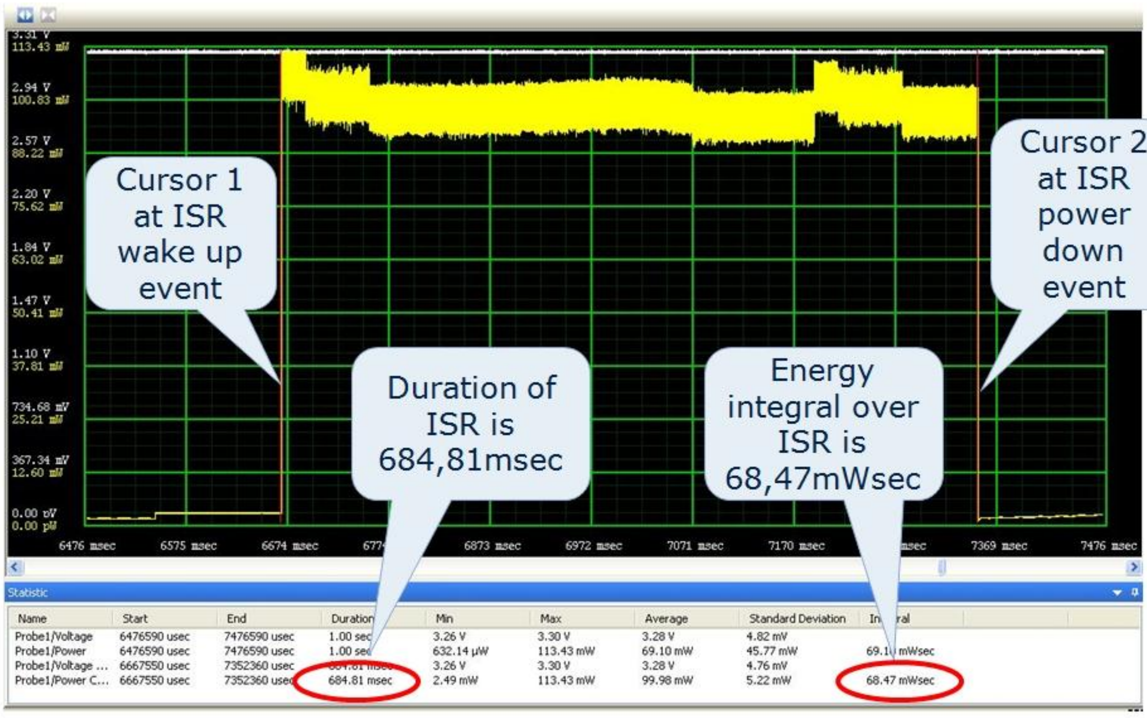
Figure 8: PowerScale and ULINKpro connected to the microcontroller system.

If ETM is not available on the microcontroller, the high-speed SWV functionality of ULINK*pro* can be used. ULINK*pro* and PowerScale are working in sync also in this setup. All three SWV recording methods can be used: ISR event recording, PC sampling, and dedicated software event recording. The high data transfer rate of ULINK*pro* makes PC sampling possible. Other systems available on the market record PC samples at rates below 10000 processor cycles. Because of the gross sampling grid, an association with the code is not possible. Important application parts that have a short execution time are missed from being recorded. Templates with appropriate functions are provided for generating dedicated software events. The developer simply inserts a particular macro at the key position in the software code. In this case also, the events recorded in the µVision Debugger Trace window are associated to the energy diagrams displayed in PowerScale.

For the second development step, ULINK*pro* is detached from the target hardware leaving PowerScale to record the finally unaltered measurement values. In this setup, the debug unit of the Cortex-M microcontroller is not in use, and also the power consumption of ULINK*pro* is not registered. The *Sleep Mode* and *Deep Sleep Mode* of the Cortex-M microcontroller can be activated to work in their full scope. Hereby, PowerScale can be used to measure the energy consumption on a long-time basis ensuring that the overall system fulfills the specifications.

Typical Application Examples

The benefits of the combined systems are demonstrated with the help of two applications. A typical application is the optimization of an ISR. An interrupt wakes-up the microcontroller form its low power mode. The ISR sets the PLL to the required frequency and continues executing the instructions. Thereafter, the microcontroller is put in *Sleep-Mode* again. In reality, the microcontroller performance and power consumption are not always proportional to the frequency. This is way the optimal frequency has to be determined through realistic and unaltered measurements. Under certain circumstances, it might be reasonable shutting down the PLL or not using an external clock, because they consume time and power in addition. Certainly, the ISR has to have a short processing time. Figure 9 shows the energy consumption diagram of an ISR. The ISR's start and finish points have been determined with the µVision trace capabilities, and Cursor 1 and Cursor 2 have been positioned accordingly. The Statistic window shows the run-time and the power consumption values of the ISR. Optimization can be achieved by varying the f0requency.



Name	Start	End	Duration	Min	Max	Average	Standard Deviation	Integral
Probe1/Voltage	6476590 usec	7476590 usec	1.00 sec	3.26 V	3.30 V	3.28 V	4.82 mV	
Probe1/Power	6476590 usec	7476590 usec	1.00 sec	632.14 µW	113.43 mW	69.10 mW	45.77 mW	69.10 mWsec
Probe1/Voltage ...	6667550 usec	7352360 usec	684.81 msec	3.26 V	3.30 V	3.28 V	4.76 mV	
Probe1/Power C...	6667550 usec	7352360 usec	684.81 msec	2.49 mW	113.43 mW	99.98 mW	5.22 mW	68.47 mWsec

25.08.2010

Figure 9: Optimization of an Interrupt Service Routine.
The time and power consumed are displayed right away.

A common and frequent fault is shutting down peripherals partially or even the wrong peripherals before switching the microcontroller into low power mode. Ensure to use the correct shut-down procedures to achieve the specified low power values indicated in the data sheets. High power consumption is not negligible and can have serious impact, for example on the life-time of a battery. The certainty to have reached the right operating mode can be better achieved through measurements than through studying the code or manuals. Another common fault is when current measurements were altered through using debugger units where PowerScale should have been used solely.

Summary

Designing embedded systems for low energy consumption is a complex task. The silicon industry offers a wide variety of microcontroller devices and many integrate specific features that increase energy efficiency. Unfortunately, no standards or benchmarks exist today for comparing the energy efficiency of microcontrollers. Even the application requirements have a huge impact and therefore it is hard to select the best matching microcontroller for an energy critical application.

Also the application software itself has a major impact to energy consumption. It is important to choose the system configuration carefully for the requirements of the application and to select the best matching software design pattern. Needless to say, development tools used for creating the application should be specifically designed for embedded programming and configured towards energy efficiency.

While creating the application software the programmer should use these guidelines:

- Run the clocks in the embedded system as slowly as practical.
- Use the best matching power-saving modes.
- Disable peripherals that are not required.
- Optimize the code towards fast execution speed.
- Power consumption, particular current, is not a digital value. Rise and fall times depends on the system. Measurement is required to validate the theory.

Of course these are simplified guidelines, but keeping them in mind during the software development is the key to an energy efficient embedded design.

How much energy needs my application? The answer to this question is very important for system designers of low power applications – in fact power profiling is impossible without knowing – and guides immediately to the question: **How to measure power consumption?**

Simple tools like a multimeter provide only average values and neglect dynamic effects. Unfortunately the effect of those dynamic effects on battery powered systems is influencing the battery duration and is therefore critical for the system.

More sophisticated tools like oscilloscopes are complex to use and sometimes difficult in practice (e.g. where to place the shunt resistor, how to setup a galvanic isolation). The setup of such a measurement environment takes time and the complex operation may yield to erroneous results.

PowerScale is designed to analyse and optimize the energy requirements of embedded systems. In conjunction with the ULINK*pro* it allows the embedded engineer to identify incorrectly configured peripherals and gives a power profile of the application software.

Literature

Cortex-M3 Technical Reference Manual, ARM Ltd.
CoreSight Architecture Specification, ARM Ltd.
The embedded world TECHNOLOGY REPORT 2010.

About the Authors

Reinhard Keil is the Director of MCU Tools at ARM. His responsibilities include the definition and strategy of tools for ARM Microcontrollers. He is founder of Keil Elektronik GmbH (the German root of Keil Software Inc.) and co-author of several key software products, such as Keil C51, Keil C166, and µVision. Reinhard continues to influence the microcontroller market and to advance the technology of the embedded space.

Frank Grobe is the Managing Director of Hitex Development Tools GmbH since 2005. In his previous carrier he held various positions in the semiconductor industry and in embedded market companies. As he started as application engineer he still got a strong focus to the application related view.