AN0007.1: MCU and Wireless SoC Series 1 Energy Modes

This application note describes strategies to reduce current consumption as well as how to enter different energy modes.

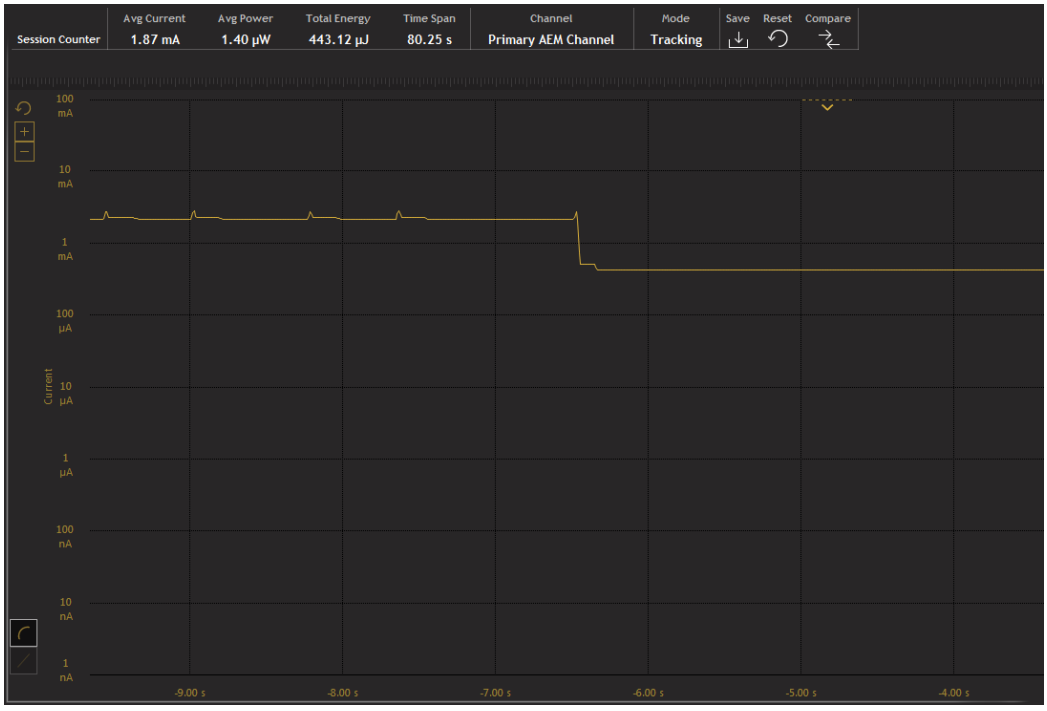
Additionally, the prime number calculation code used in current consumption benchmarking is included.

This application note includes:

- This PDF document
- Source files (zip)
  - Example c-code
  - Multiple IDE projects

KEY POINTS

- Lower energy consumption by altering energy modes, peripherals, oscillators, clocks, and compilers.
- Measure low current on starter kit using an external source with USB connected
- See software examples to understand different energy modes and MCU current consumption



1. Device Compatibility

This application note supports multiple device families, and some functionality is different depending on the device.

MCU Series 1 consists of:

- EFM32 Jade Gecko (EFM32JG1/EFM32JG12/EFM32JG13)
- EFM32 Pearl Gecko (EFM32PG1/EFM32PG12/EFM32PG13)
- EFM32 Giant Gecko (EFM32GG11)

Wireless SoC Series 1 consists of:

- EFR32 Blue Gecko (EFR32BG1/EFR32BG12/EFR32BG13)
- EFR32 Flex Gecko (EFR32FG1/EFR32FG12/EFR32FG13)
- EFR32 Mighty Gecko (EFR32MG1/EFR32MG12/EFR32MG13)

2. Energy Saving

2.1 General

In battery-powered microcontroller applications, saving energy is essential. By reducing current consumption, the mean time between battery charging and replacement can be significantly increased. Microcontroller software design should follow these principles to reduce current consumption:

- Use appropriate Energy Modes
- Exploit low energy peripherals
- Turn off unused modules / peripherals
- Disable clocks to unused modules / peripherals
- Reduce the clock frequency
- Lower the operating voltage
- Optimize code

Use of these principles is explained in the sections that follow.

2.2 Use Appropriate Energy Modes

The most effective way to save energy is to spend as little time as possible in active mode. Five tailored energy modes allow the microcontroller to run in the most energy optimal state at any given time.

2.3 Make Use of Low Energy Peripherals

All peripherals are built with energy consumption in mind and are available in various energy modes. Whenever possible, select an appropriate peripheral and let it do the work while the CPU goes to sleep (or performs other tasks). A few examples:

- Use the RTC and go to sleep instead of waiting in some kind of loop
- Use DMA to transfer data between memory and the U(S)ART
- Use the Low Energy Sensor Interface (LESENSE) to monitor a sensor instead of waking up and polling

See the System Overview chapter in the Reference Manual for a given device to see which peripherals are available in the different energy modes.

2.4 Turn off Unused Modules / Peripherals

At any given time in every microcontroller application, there are modules / peripherals which are not used. Turn these off to save energy.

This also applies to the CPU itself. If the core is idle (e.g. waiting for data reception), it can be turned off and energy is saved. This is one of the main features of the different EFM32 energy modes.

When disabling peripherals, remember to take startup and stop conditions into consideration. For example, if turned off completely, the ADC requires some time to warm up before a conversion can be started. Similarly, a USART synchronous transfer in progress should be allowed to complete so that the shift register of the receiver is not left in an indeterminate state.

2.5 Disable Clocks to Unused Modules / Peripherals

Even though a module / peripheral is disabled (e.g. TIMER0 is stopped), energy will still be consumed by various circuits in that module if its clock is running. Therefore, it is important to turn off the clocks for all unused modules. This is illustrated in the example code that accompanies this application note. Further details are provided in the Clock Management Unit (CMU) chapter of the specific device's reference manual.

2.6 Reduce the Clock Frequency

Current draw scales with clock frequency. In general, a certain task or peripheral should run at the lowest possible frequency. For example, if a timer is to request interrupts every few milliseconds, it should be clocked at a few kHz rather than several MHz. This is easily be implemented with the prescaling functionality in the CMU.

Likewise, one approach to CPU frequency selection is that it should be so low that the CPU is not idle (some margin should be added). However, in many cases it is by better to complete the current tasks quickly and then enter a suitable energy mode until new tasks must be handled. The different energy modes are optimized for this purpose and described in the Energy Management Unit (EMU) chapter of the specific device's reference manual.

2.7 Lower the Operating Voltage

By lowering the operating voltage, energy consumption is further reduced. The Gecko family of microcontrollers can run with full functionality on low voltages. The absolute minimum ratings be found in the data sheet for each device.

2.8 Optimizing Code

Optimizing code usually leads to lower energy consumption by increasing the program speed and efficiency. A faster program spends less time in active mode, and each task in a more efficient program takes fewer instructions to execute. A simple way to optimize your code is to build it with the highest optimization settings in release mode rather than in debug mode. In the [**Development Perspective**] of Simplicity Studio, go to [**Project**]>[**Build Configurations**]>[**Set Active**] and select [**Release**] for your compiler.

Compiler selection can also have an impact on energy efficiency. For example, the IAR compiler tends to generate more efficient code than GCC. To use the IAR toolchain in Simplicity Studio, make sure IAR Embedded Workbench is installed on your computer. In the [**Development Perspective**] of Simplicity Studio, go to [**Project**]>[**Properties**]>[**C/C++ Build**]>[**Settings**]. Under the [**Configuration:**] drop down menu, select [**IAR ARM - Release**]. If you do not see this option, click [**Manage Configurations...**]>[**New...**], select the [**IAR ARM - Release**], and click [**OK**] twice.

Next, increase IAR's optimization settings, under [**Tool Settings**]>[**IAR C/C++ Compiler for ARM**]>[**Optimizations**], select [**High, Balance**] for the [**Optimization level:**]. Under [**IAR Linker for ARM**]>[**Optimizations**], check all options ([**Inline..., Merge..., Perform..., Even...**]), and then click [**OK**].

Note: As a starting point, this should lower energy consumption, but it may not be the most optimized setting for a given project. Try different optimization settings such as [**High, Speed**], [**High, Balance**], other optimization option combinations, even other compilers, and compare the results.

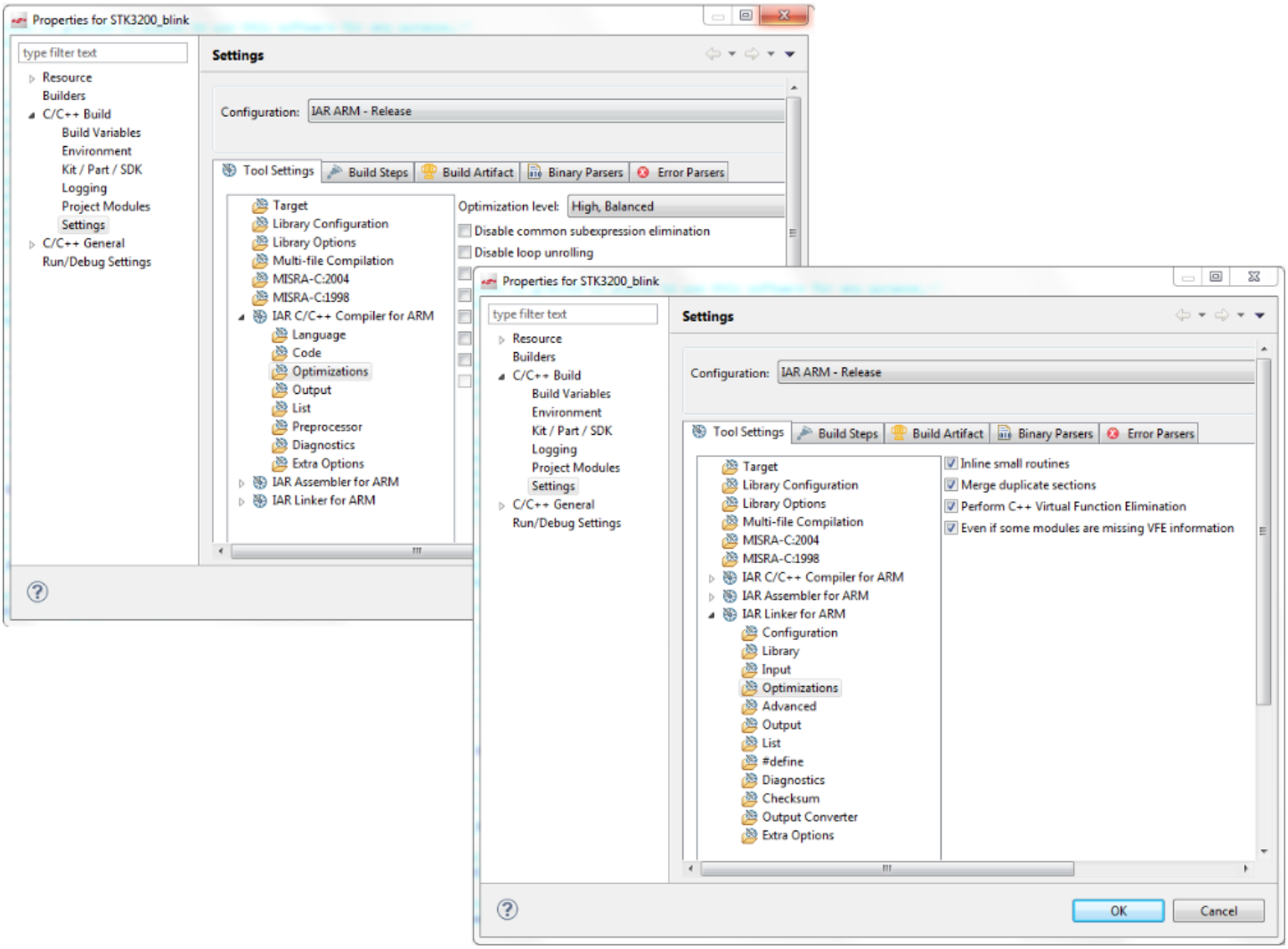


Figure 2.1. Setting the Optimization Settings in Simplicity Studio when using IAR

3. Energy Modes

CPU activity and the availability of peripherals and clocks differ in each of the energy modes. These are discussed briefly in the section that follow.

Functions provided by emlib make it easy to configure peripherals for low energy consumption and to switch between energy modes.

3.1 Run Mode (Energy Mode 0)

This is the default mode. In this mode, the CPU fetches and executes instructions from flash or RAM, and all peripherals may be enabled.

3.2 Sleep Mode (Energy Mode 1)

In sleep mode, the clock to the CPU is disabled. All peripherals, as well as RAM and flash, are available. By using the Peripheral Reflex System (PRS) and DMA, several operations can be performed autonomously. For example, the timer may trigger ADC conversions at a regular interval. When conversions are complete, the results are moved by the DMA to RAM. When a given number of conversions have been performed, the DMA may request and interrupt to wake the CPU.

Sleep mode is entered by executing either the "Wait for Interrupt (WFI)" or the "Wait for Event (WFE)" instruction.

Use the emlib function `EMU_EnterEM1()` to go into sleep mode.

3.3 Deep Sleep Mode (Energy Mode 2)

In deep sleep mode, no high frequency oscillators run, which means that only asynchronous and low frequency peripherals are available. This mode further improves energy efficiency while still allowing a range of activities, including use of:

- the Low Energy Sensor Interface (LESENSE) to monitor a sensor,
- the LCD controller to drive a LCD,
- the LEUART to receive or transmit a byte of data,
- the I^2C to perform address match check,
- the RTC to wake the CPU after a programmed number of ticks,
- an Analog Comparator (ACMP) to compare a voltage to a programmed threshold, and
- a GPIO to check for transitions on an I/O line.

Deep sleep mode is entered by first setting the SLEEPDEEP bit in the System Control Register (SCR) and then executing either the "Wait for Interrupt (WFI)" or the "Wait for Event (WFE)" instruction. Use the emlib function `EMU_EnterEM2()` to enter deep sleep mode.

3.4 Stop Mode (Energy Mode 3)

Stop mode differs from deep sleep mode in that no oscillator (except the ULFRCO or AUXHFRCO) is running.

Modules / functions, if present on a device, are generally still available in stop mode when the appropriate clock source remains active:

- I^2C address check
- Watchdog
- GPIO interrupt
- Pulse counter (PCNT)
- Low-energy timer (LETIMER)
- Low-energy sensor interface (LESENSE)
- Real time counter and calendar (RTCC)
- Analog comparator (ACMP)
- Voltage monitor (VMON)
- Ultra Low Energy Timer/Counter (CRYOTIMER)
- Temperature sensor

Stop mode is entered the same way as deep sleep mode, except that the low frequency oscillators must be manually disabled. The emlib function `EMU_EnterEM3()` handles all of this, as well as re-enabling high and low frequencies oscillators that were active prior to entering stop mode if the boolean restore parameter is true.

3.5 Hibernate Mode (Energy Mode 4)

Some EFM32 and EZR32 devices can wake from shut off mode using sources that run from one of the low frequency oscillators in addition to the usual asynchronous pin-based sources. This capability is called hibernate mode on EFM32 and Wireless SoC Series 1 and is enabled with dedicated control register logic. Writing the 0x2, 0x3, 0x2, 0x3, 0x2, 0x3, 0x2, 0x3, 0x2 sequence to the EM4ENTRY bit field in the EMU_EM4CTRL register places the device in hibernate mode when the EM4STATE bit is set; otherwise, the device enters shut off mode as usual.

In Hibernate Mode, the majority of peripherals are shutoff to reduce leakage power. A few selected peripherals are available. System memory and registers do not retain values. GPIO PAD state and RTCC RAM are retained. Wake-up from EM4 Hibernate requires a reset to the system, returning it back to EM0 Active.

Hibernate mode wake up is possible from the same shut off mode power-cycling, nRESET, and user-specified pin sources, as well as the:

- RTCC
- CRYOTIMER
- rising or falling edge of any monitor power supply (VMON)
- measurement of a temperature outside of defined limits (TEMPCHANGE)

The emlib `EMU_EnterEM4H()` function handles hibernate mode entry.

3.6 Shut Off Mode (Energy Mode 4)

Shut off mode is the lowest possible energy state for an EFM32 and EZR32 Series 0, EFM32 or Wireless SoC Series 1 microcontroller. Power is switched off to most of the device, including internal RAM, and all clocks are disabled. Only the recovery logic, and, if explicitly enabled, GPIO pad state, are retained. Waking from shut off mode always entails a reset. Current draw in shut off mode can be as low as 20 nA when the reset is sourced from either the RESETn pin or via one of a small group of device-specific pins that can be explicitly enabled for this purpose. Some devices provide alternatives to pin-based wake up; however, waking from these sources requires one of the low frequency oscillators to remain enabled, increasing current draw. See the following table for the shut off mode wake up options that are available on a given device.

Table 3.1. Shut Off Mode Wake Up Sources

Wake Up Source	Device						
	EFM32PG1/JG1	EFM32PG12/JG12	EFM32PG13/JG13	EFM32GG11	Wireless SoCs (EFR32xG1)	Wireless SoCs (EFR32xG12)	Wireless SoCs (EFR32xG13)
Power-on Reset	Yes	Yes	Yes	Yes	Yes	Yes	Yes
nRESET pin	Yes	Yes	Yes	Yes	Yes	Yes	Yes
User-specified pin(s)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CRYOTIMER	Yes	Yes	Yes	Yes	Yes	Yes	Yes
RFSENSE	—	—	—	—	Yes	Yes	Yes

Shut off mode is entered by writing the sequence 0x2, 0x3, 0x2, 0x3, 0x2, 0x3, 0x2, 0x3, 0x2 to the EM4CTRL field in the EMU_CTRL register of the Energy Management Unit (EMU) on EFM32 and EZR32 devices. EFM32 and Wireless SoC Series 1 derivatives use the same sequence, but it is written to the EM4ENTRY field in the EMU_EM4CTRL register. The emlib `EMU_EnterEM4S()` function simplifies this by writing the shut off mode entry sequence to the device-appropriate register and bit field.

3.7 Waking Up

Wake from sleep (EM1), deep sleep (EM2), and stop (EM3) modes occurs when a module that is still active, e.g. has asynchronous circuitry like GPIO or has an enabled clock source, requests an interrupt. Operation in run mode (EM0) resumes, first with the service routine for the interrupt that caused the wake up and then with the normal program code at the point immediately after its entry into low-power mode.

A reset is required to wake from either shut off mode or hibernate mode. To facilitate timed and event-based wakeups from EM4, the device provides an Ultra Low Energy Timer/Counter (CRYOTIMER) which is able to run even in EM4 Shut Off mode. Selected GPIO pins can also trigger a reset. Some additional software configuration is required before going into EM4 when waking from a source other than the nRESET pin or a power-on reset.

3.7.1 GPIO Wake from EM4

Waking from a pin other than nRESET requires configuration of the GPIO registers as follows:

- The pin must be set to the input pull filter.
- Active polarity of the reset signal must be set in GPIO_EXTILEVEL for EFM32 and Wireless SoC Series 1 devices.
- Wake up from EM4 must be enabled in GPIO_EM4WUEN.

3.7.2 VMON Wake from EM4

Waking from hibernate mode (EM4H) is possible on EFM32 and Wireless SoC Series 1 when a voltage monitored by the the EMU Voltage Monitor (VMON) rises above or falls below a user-programmed threshold. The VMON has two two comparators for AVDD, as well as comparators for DVDD and IOVDD0, any of which can be used to wake from hibernate when the related EMU registers are programmed:

- with a valid threshold or thresholds (in case of the primary AVDD monitor) and
- to enable the monitor and wake from EM4 when the monitored voltage rises above or falls below the programmed threshold(s).

3.7.3 TEMPCHANGE Wake from EM4

Along with its voltage monitoring capability, the EMU on EFM32 and Wireless SoC Series 1 incorporates a temperature sensor that takes measurements at 250 ms intervals and that is active in all modes except shut off (EM4SH). Similar to VMON events, temperatures measured outside of user-programmed limits (TEMPCHANGE) can trigger wake from hibernate when the EMU_TEMPLIMITS register is programmed with valid high and low thresholds and EM4 wake up is enabled.

3.7.4 CRYOTIMER Timed Wakeup from EM4

EFM32 and Wireless SoC Series 1 with the CRYOTIMER can perform a timed wakeup from hibernate (EM4H) or shut off (EM4SH) mode. When writing the EM4 entry sequence to the EMU_EM4CTRL register, the ULFRCO, LFRCO, or LFXO must be kept active by setting the RETAINULFRCO, RETAINLFRCO, or RETAINLFXO bit, respectively, in order to clock the CRYOTIMER.

To perform a timed wakeup from EM4, some registers must be configured before going to sleep:

- An oscillator must be selected. If different from ULFRCO software must also make sure the oscillator is running and is stable before going to sleep.
- Wakeup from CRYOTIMER interrupt must be enabled.

Likewise, the CRYOTIMER registers must be set so that:

- the CRYOTIMER is enabled
- the CRYOTIMER interrupt must be enabled
- EM4 wake up requests are enabled
- an active clock source with optional prescaling is selected.

3.7.5 RTCC Timed Wakeup from EM4

EFM32 and Wireless SoC Series 1 with the Real Time Counter and Calendar (RTCC) can also perform a timed wakeup from hibernate (EM4H). The CMU, EMU, and RTCC must all be configured appropriately before writing the EM4 entry sequence to EMU_EM4CTRL:

- The ULFRCO, LFRCO, or LFXO must be enabled and set to remain enabled upon EM4 entry.
- The LFECLK must be enabled, and one of the low frequency oscillators must be selected as its source.
- The RTCC must be enabled and set to allow EM4 wake ups.
- At least one RTCC interrupt must be configured and enabled.

4. Clock and Oscillator Control

4.1 General

As previously mentioned, current consumption is highly dependent on clock frequency. Selecting the correct oscillator and frequency is, therefore, a very important aspect of low energy application design and development. The following sections discuss different modes and settings for clocks and oscillators.

4.2 Enabling Oscillators / Setting Clock Source

Oscillators are enabled and disabled through the CMU_OSCENCMD register in the CMU. Each oscillator has one enable and one disable bit in this register (e.g. LFXOEN and LFXODIS). The CMU_STATUS register holds two flags for each oscillator — enabled and ready (e.g. LFXOENS and LFXORDY). Each enabled flag is set when the corresponding oscillator is turned on, whereas the ready flag is set when the oscillator is ready to be used as a clock source.

Note: Until the ready flag is set, the oscillator output may be incorrect, both with respect to frequency and duty-cycle. Use of an oscillator before this flag is set can result in unpredictable or undefined device behavior.

Out of reset, the High Frequency RC Oscillator (HFRCO) is set as source for the CPU core and high-speed peripherals (HFCLK domain). No oscillator is enabled or selected for the Low Frequency Clock (LFxCLK) domains (A, B, and, if present, C, E).

Changing the oscillator sourced for a particular clock domain is a three step procedure:

1. Enable the desired oscillator by setting its corresponding bit in the CMU_OSCENCMD register.
2. Wait until the oscillator's ready flag in the CMU_STATUS register is set.
3. Select the new oscillator for the clock domain in question. The high and low frequency clock domains on members of the EFM32 and Wireless SoC Series 1 families have dedicated CMU_HFCLKSEL and CMU_LFxCLKSEL registers for this purpose.

4.3 HFRCO Band Setting

The extreme frequency tuning range of the HFRCO is a major advantage, and should be used to minimize the energy consumption of any application. The following frequencies may be set [MHz]: 1 - 2 - 4 - 7 - 13 - 16 - 19 - 26 - 32 - 38. Frequency band is selected using the FREQRANGE field in the CMU_HFRCOCTRL register.

The HFRCO can be tuned to run at one of several different frequencies, the selection of which should be used to minimize energy consumption. The available frequency bands and register bit field used to select a band differ depending on the device. See the following table for details.

Table 4.1. HFRCO Frequency Bands and Control Register Bit Fields

Device	Frequency Bands (MHz)	CMU_HFRCOCTRL Bit Field
EFM32PG1/JG1	1, 2, 4, 7, 13, 16, 19, 26, 32, and 38	FREQRANGE
EFM32PG12/JG12		
EFM32PG13/JG13		
EFM32GG11		
Wireless SoC EFR32xG1		
Wireless SoC EFR32xG12		
Wireless SoC EFR32xG13		

4.4 Enabling or Disabling a Module Clock

A module's clock may be enabled or disabled using the corresponding bit in the appropriate CLKEN register in the CMU. The following registers are used to do this:

- CMU_HFBUSCLKEN0: High-speed and core modules like LDMA, AES, USB, external bus interface, etc.
- CMU_HFPERCLKEN0/1: Most peripherals (e.g. timers, ADC, USART, etc).
- CMU_LFxCLKEN0: Low energy peripherals (e.g. RTC, LEUART, LETIMER, etc).

Details on which register and bit fields are used to enable and disable module clocks are covered in the Reference Manual for a given device.

4.5 Clock Prescaling

Clock prescaling (dividing down by a programmable factor) is available for all high frequency and many low frequency clock domains. Even when a module is idle, portions of its logic, such as registers and the bus interface, are continuously clocked and draw current; therefore, whenever possible, clocks should be prescaled so that continuously clocked logic runs at the minimum frequency necessary for tasks to be completed or for a particular level of performance to be sustained.

Devices in the EFM32 and Wireless SoC Series 1 family providing clock prescaling through four different registers:

- CMU_HFCOREPRESC is used for the CPU core and related logic like the DMA and external bus interface
- CMU_HFPERPRESC provides prescaling of the baseline clock to all peripherals which take a high frequency clock. Many high frequency peripherals, such as the TIMERs, have local prescalers, as well.
- Low frequency peripherals in the LFACLK and LFBCLK domains have their own dedicated prescalers in the CMU_LFAPRESC0 and CMU_LFBPRESC0 registers, respectively.

Before being distributed to any of the modules in high frequency clock domains, the output of the selected high frequency oscillator is prescaled by the PRESC field in the CMU_HFPRESC register. This clock, the HFCLK, is effectively the baseline bus clock frequency of the system because it is supplied to anything tightly coupled to the core, such as the flash and LDMA. The CPU core also has its own prescaled clock, HFCORECLK, which is controlled by CMU_HFCOREPRESC. This decoupling allows the core to run at a lower frequency than the rest of the system and permits lower energy usage in cases where the core must maintain some minimum activity level while the LDMA or PRS is responsible for the bulk of system activity.

5. STK Low Power Measurement

5.1 General

Measuring low power on starter kits (STKs) requires a specific setup to reduce any off-chip current consumption. The following section discusses configuration of the board and instruments to accurately achieve and read the lowest power numbers.

5.2 Setup

To measure current consumption using a bench power supply and an ammeter, do the following:

1. Program the desired software into the on-chip flash memory (see Optimized Code).
2. Power board with a stable, external source connected to the VMCU and GND pins on either the breakout or expansion headers (see the User Manual for STK being used).

   Note: User manuals can be found in the [**Kit Documentation**] tile of Simplicity Studio.

3. Use an ammeter in-circuit to measure active current.
4. Move the power switch to the BAT position.
5. Keep the USB debug interface connected so that the analog switches on the board remain powered.

   Note: Current consumption measured with the Gecko STK may not be accurate when USB is connected.

6. Reset board.

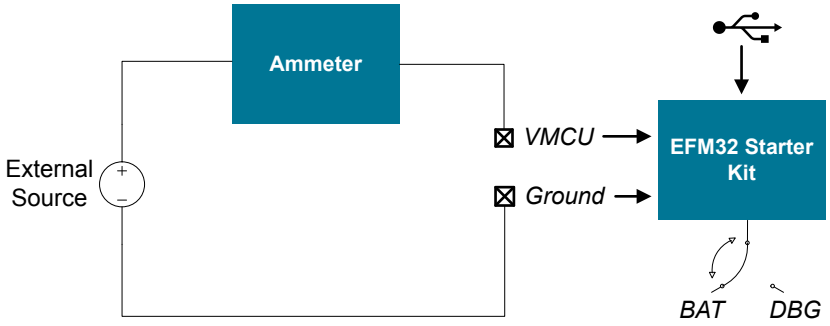


Figure 5.1. STK Low Power Measurement — Setup

Note: If a standalone power supply and ammeter are not available, the [**Energy Profiler**] in Simplicity Studio can also be used to measure active current.

6. Software Examples

The example accompanying this document includes three demoes: Energy Modes, EM4 Wakeup and Prime Number. The software example displays current demo name on LCD. Using button 1 to select the demo, and press button 0 to run the demo.

6.1 Energymodes Software Example

This example illustrates how to enter different energy modes, how to enable different oscillators, enable / disable clocks and set up prescaling. To illustrate the importance of only enabling needed clocks and oscillators, the software starts off by turning on everything. Then, more and more energy is saved by disabling clocks and oscillators, and by entering energy modes.

Note: When in debug mode, the MCU will not go below EM1. When running this example exit debug mode and reset the MCU after flashing it.

The program goes through the following states. After each change of settings, a few seconds of waiting is inserted to make the current consumption visible. Use the energyAware Profiler to see how the current consumption changes.

- All Clocks Enabled—Every oscillator on the EFM32 is turned on. The HFCLK source is set to HFXO, which is the fastest oscillator. In addition, clocks to all core modules and regular peripherals are enabled.
- All Clocks Disabled—The HFCLK source is set back to HFRCO. All unused oscillators are turned off, and clocks to unused modules / peripherals are disabled.
- Core Clock Downscaled—The core clock frequency is reduced by selecting the 7 MHz frequency band. In addition, the core clock is prescaled with factor 4, i.e. the core frequency is 1.75 MHz.
- Sleep Mode—The clock to the interface of the LE peripherals is enabled, Then the RTC is set up to issue an interrupt after a few seconds. Then Sleep Mode is entered until the RTC wakes up the device.
- Deep Sleep Mode—The clock to the interface of the LE peripherals is enabled. Then the RTC is set up to issue an interrupt after a few seconds. Then Deep Sleep Mode is entered.
- Stop Mode / Shut Off Mode—At the end of the program either EM3 or EM4 is entered and the program stays in this mode.

6.2 EM4 Wakeup Software Example

This example shows how to do a wake up from EM4 Shut Off Mode. The example will repeatedly enter EM4 and sleep for few seconds and CRYOTIMER timeout wakeup the device. It is also possible to wake up by pressing button 1 on the STK board.

Every time the MCU wakes up, the reset cause (reset pin, GPIO or CRYOTIMER) will be printed to the LCD.

6.3 Prime Number Software Example

The prime number software example is used for current consumption benchmarking. The code makes the device run from the High Frequency Crystal Oscillator (HFXO) and disables all other oscillators. It sets up the device to forever execute a prime calculation algorithm from flash.

7. Revision History

7.1 Revision 1.10

2017-06-28

Updated formatting.

Split AN0007 into AN0007.0 and AN0007.1 for MCU/Wireless MCU Series 0 and MCU/Wireless SoC Series 1, respectively.

Added the 1. Device Compatibility section.

Added the 2.8 Optimizing Code section to 2. Energy Saving

Added the 5. STK Low Power Measurement section.

Added the section.

7.2 Revision 1.09

2014-05-07

Updated example code to CMSIS 3.20.5

Changed to Silicon Labs license on code examples

Added example projects for Simplicity IDE

Removed example makefiles for Sourcery CodeBench Lite

7.3 Revision 1.08

2013-10-14

New cover layout

7.4 Revision 1.07

2013-05-08

Added software projects for ARM-GCC and Atollic TrueStudio.

7.5 Revision 1.06

2012-11-12

Adapted software projects to new kit-driver and bsp structure.

7.6 Revision 1.05

2012-08-13

Adapted software projects to new driver file structure.

7.7 Revision 1.04

2012-07-19

Added EM4 wakeup example with BURTC and data retention

Fixed a bug where Prime Example would be stuck in while loop

Added software support for Tiny and Giant Gecko STK's.

7.8 Revision 1.03

2012-04-20

Adapted software projects to new peripheral library naming and CMSIS_V3.

7.9 Revision 1.02

2012-03-13

Fixed makefile-error for CodeSourcery projects.

7.10 Revision 1.01

2010-11-16

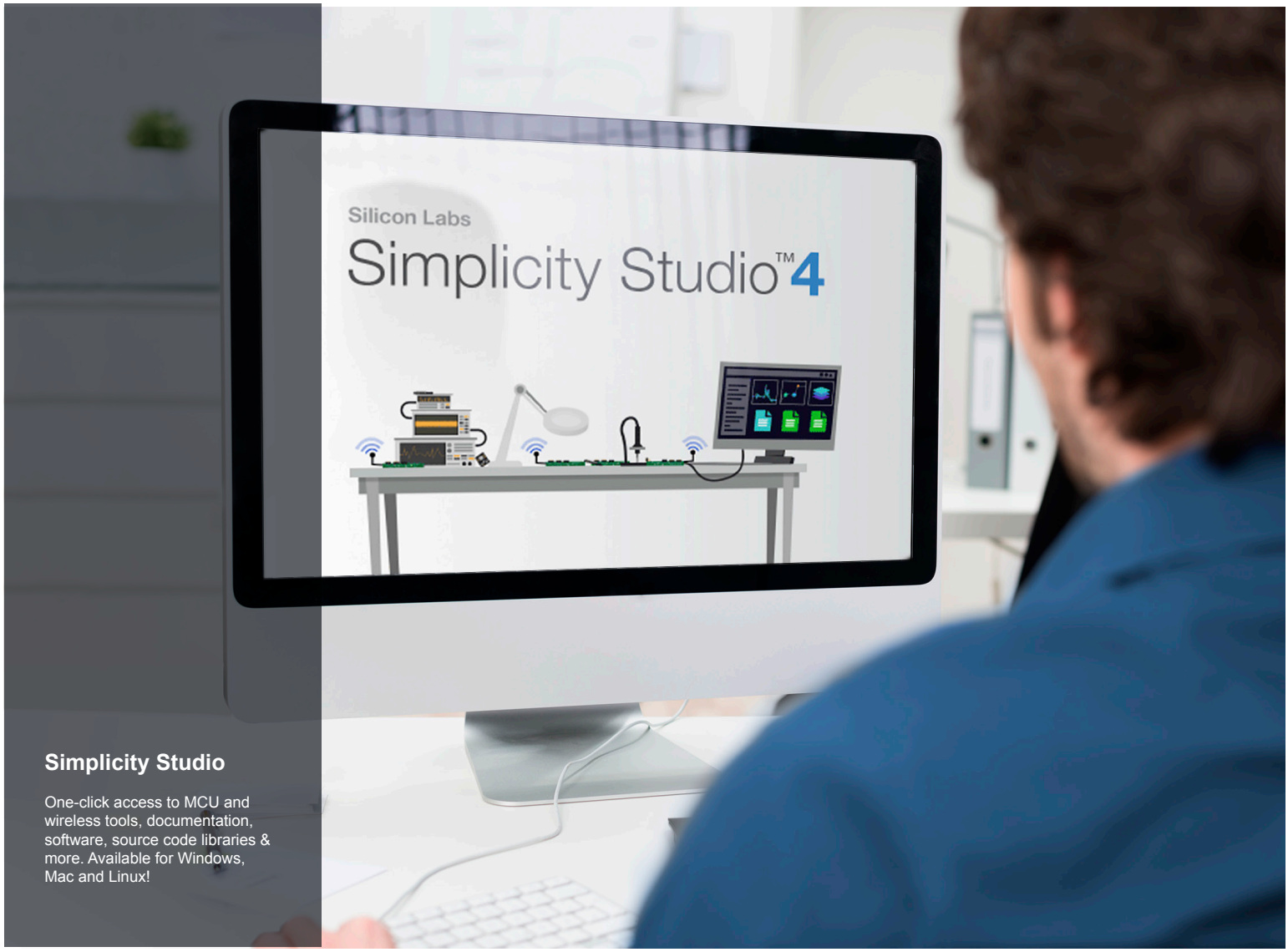
Changed example folder structure, removed build and src folders.

Added chip-init function.

7.11 Revision 1.00

2010-09-20

Initial revision.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com