... the world's most energy friendly microcontrollers

Programming Internal Flash Over the Serial Wire Debug Interface

AN0062 - Application Note

Introduction

This document explains how to access the debug interface of the EFM32 and how to use this interface to program devices (load applications into flash). It also explains how to lock and unlock debug access to the MCU to protect the contents of the internal flash and SRAM.

This application note includes:

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

1 Debug Interface Overview

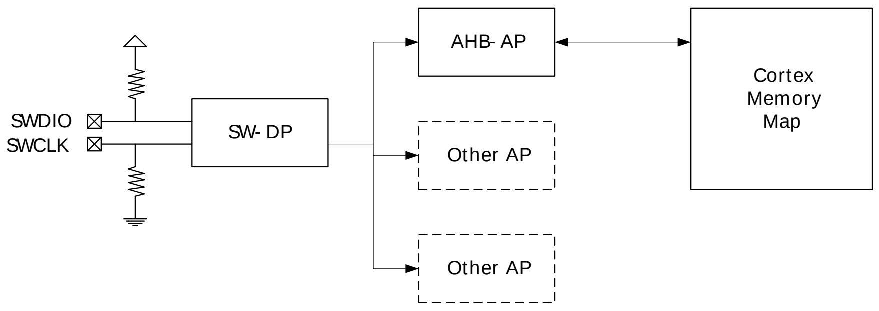
1.1 Serial Wire Debug

Serial Wire Debug (SWD) is a two-wire protocol for accessing the ARM debug interface. It is part of the ARM Debug Interface Specification v5 and is an alternative to JTAG. The physical layer of SWD consists of two lines:

- SWDIO: a bidirectional data line
- SWCLK: a clock driven by the host

Connecting to these pins allow an external device (such as a debug probe) to communicate directly with the Serial Wire Debug Port (SW-DP). The SW-DP in turn can access one or several Access Ports (APs) that give access to the rest of the system. The important AP on the EFM32 is the AHB-AP which is a bus master on the internal AHB[1] bus of the Cortex-M3. In other words the AHB-AP can access the internal memory map of the core. Since the internal flash, SRAM, debug components and peripherals all are memory mapped, this AP can control the entire device including programming it.

Figure 1.1. Serial Wire Debug interface



1.2 Debug Pins

The EFM32 has three pins used for debugging. Two of them are the SWDIO and SWCLK pins used by SWD. The last pin is called Serial Wire Output (SWO) and is used for debugging output. SWO is an asynchronous, one-directional protocol used by the internal debug components in the core to output various debug information. This pin is not required to program the device.

Out of reset both SWDIO and SWCLK are connected to SW-DP and configured with a weak internal pull-up and pull-down, respectively. It is possible to disable both debug pins by configuring the GPIO_ROUTE register. This can free up the pins to use them as GPIO. The SWO pin is initially disabled.

Note

Keep in mind that if the debug pins are disabled, a debugger will no longer be able to connect to the EFM32. During development, it is therefore a good idea to make sure there is a delay between reset and disabling the pins (typically a few seconds). In that way the debugger has time to connect and halt the CPU before the pins are disabled. If you have disabled the debug pins and are unable to connect to your device, see Section 4.3 (p. 13) for instructions on how to perform the debug unlock sequence.

When designing the debug header for a product it is common to also include the RESET pin and one pin for sensing the supply voltage in addition to ground.

[1]AHB (Advanced High-performance Bus) is the internal bus of the Cortex core

2 Serial Wire Programmers Model

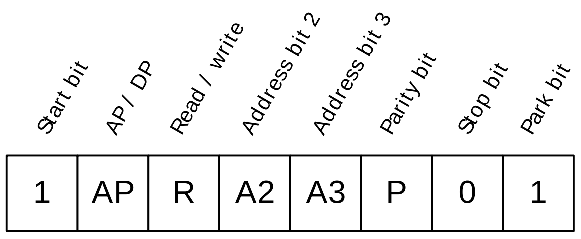
This chapter describes the SWD protocol and how to communicate with the SW-DP and AHB-AP.

2.1 The SWD Protocol

In SWD terminology the *host* refers to the system controlling the debugger, i.e. the PC / debug probe. The *target* is the system which is under debug, i.e. the EFM32.

SWCLK is a clock signal which is always driven by the host. Both sides will drive the SWDIO line to send data. A high value on SWDIO indicates a logical '1', a low value is a logical '0'. The protocol specifies when each side will drive the SWDIO line. Three different phases are specified. Each transaction begins with the host sending a request. The target answers with an acknowledge which is followed by a data phase. Who controls the line during the data phase depends on the type of request issued by the host. If the host issued a *write* request, the host will drive the line. On a *read* request the target will drive the line to transmit data from the target to the host. In all phases data is transmitted LSB (least significant bit) first. The target will both sample and put data on the line on a rising clock edge.

Figure 2.1. SWD request



The request phase consists of 8 bits. The meaning of each bit in the request is illustrated in Figure 2.1 (p. 3) . The start bit is always 1. The next bit specifies whether the transaction is a DP (Debug Port) or AP (Access Port) transaction. If this bit is zero, the transaction is a DP access. Bit 2 is the read/write bit. If this bit is 1 the transaction is a read access (from target to host). Bit 3 and 4 are address bits A2 and A3. These bits specifies which out of four registers are selected for the transaction. Register selection is described in Section 2.2 (p. 5) . Bit 5 is a parity bit. The parity bit is used by the target to verify the integrity of the request. This bit should be 1 if bits 1-4 contains an odd number of 1's. If the number of 1's are even, the parity bit should be zero. Bit 6 is the stop bit. This bit is always zero. Bit 7 is the park bit. This bit is always one.

The ack phase consist of three bits. An OK response has the value 1. Since values are put on the line LSB first an OK response looks like a 1 followed by two 0's on the line.

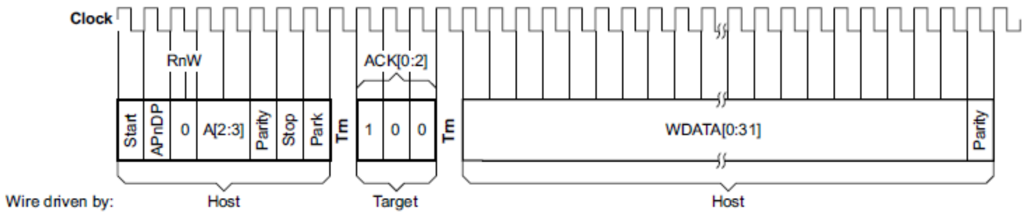
Once the host has received an OK response the data phase can begin. The data phase consists of 32 data bits (one word) followed by 1 parity bit. The parity bit is calculated based on all the 32 data bits. If the number of 1's in the data word is odd, the parity bit should be 1.

The host must continue to clock the interface for at least 8 cycles after the data phase to make sure the transaction is clocked through the SW-DP. This can be done either by:

- starting a new transaction
- inserting idle clock cycles

During idle clock cycles the SWDIO is driven *low* by the host.

Figure 2.2. Successful Write Command (from [adi5])



2.1.1 Turnaround Periods

Every time the SWDIO changes data direction, a one-cycle turnaround period is inserted which both sides should ignore. This means there is always a turnaround period between the request and acknowledge. On a write request, there is a turnaround period between acknowledge and the data phase. On a read request there is a a turnaround after the data phase.

2.1.2 Initialization

Before using the SW-DP an initialization sequence must be performed to establish communication and bring the SW-DP to a known state.

1. Perform a line resest
2. Send the JTAG-to-SWD switching sequence
3. Perform a line reset
4. Read the IDCODE register

A line reset is performed by clocking at least 50 cycles with the SWDIO line kept HIGH by the host.

The reason for the JTAG-to-SWD sequence is that the Debug Port implementation is actually a SWJ-DP. SWJ-DP is a wrapper around both SW-DP and JTAG-DP, the JTAG counterpart to SWD. The EFM32 does not include JTAG, but the switching sequence must still be performed as the default state required by the SWJ-DP specification is JTAG. The JTAG-to-SWD sequence is 0xE79E transmitted LSB first.

Figure 2.3. Read IDCODE SWD request

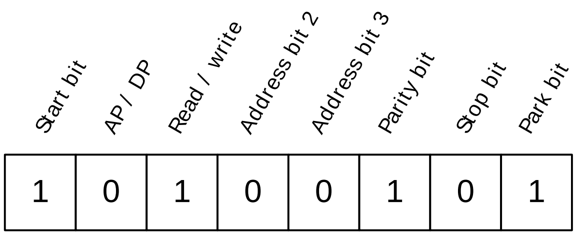


Figure Figure 2.3 (p. 4) shows the request command to read the IDCODE register. After this request has been transmitted the target will reply with OK followed by the IDCODE value (and parity bit). After this the debugger can use the SW-DP normally.

2.1.3 Errors

The SWD protocol specifies two other ACK responses than OK. If the target responds with a WAIT response, the host must retry the operation later. If the target responds with FAULT, an error has occured and one of the sticky bits in CTRL/STAT is set. The host can check the sticky error bits to see what kind of error has occured. It must clear the sticky bits before using any AP commands, because the target will always respond with FAULT as long as one of the sticky error bits are set.

If the target sees a protocol error it will immediately stop driving the line, i.e. it will not respond to any request. If the host sees that the target does not drive the line it must perform the initialization sequence again.

2.2 Serial Wire Debug Port Registers

Table 2.1. SW-DP registers

Address	Read	Write
0x00	IDCODE	ABORT
0x04	CTRL/STAT [1]	CTRL/STAT [1]
0x08	RESEND	SELECT
0x0C	RDBUFF	N/A

[1]WCR register if CTRLSEL bit of SELECT is 1, see [adi5]

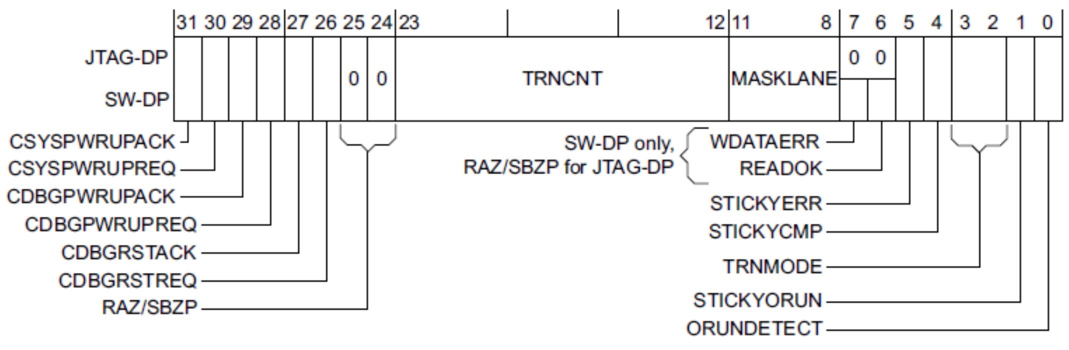
This section will give a brief overview over the SW-DP registers. A full description of each register and the meaning of the individual bits can be found in [adi5].

Table 2.1 (p. 5) shows an overview over the SW-DP registers. In the SWD request two address bits are given, ADDR[3:2]. Bits 1 and 0 of the address are always 0. These bits select which register is accessed. In addition the type of access (read or write) also affect which register is accessed. As an example, say the transaction is a *write* access where address bits 2 and 3 are 0 and 1, respectively. I.e. the address is 0b1000, or 0x8. This will choose the SELECT register.

The IDCODE register contains a identification value that identifies the SW-DP. On the EFM32 devices with a Cortex-M3 or Cortex-M4 core this register should read 0x2BA01477. For devices with a Cortex-M0+ core the register should read 0x0BC11477.

The CTRL/STAT register provides control of the DP and several status bits. The debugger must write a 1 to the CDBGPWRUPREQ and CSYSPWRUPREQ bits before using the AHB-AP. The STICKYERR bit is set if an error occurs in a AP transaction. While the STICKYERR bit is set any SWD request will return a FAULT response. To clear the STICKYERR bit, use the ABORT register.

Figure 2.4. Control/Status Register (from [adi5])



The SELECT register is used to select which Access Port (AP) is used when performing an AP transaction (AP bit of request is 1). The DP can be connected to several APs and this register is used to select which one is accessed. The APSEL field (bits [31:24]) is used to choose between different APs (refer to Figure 1.1 (p. 2)). On the EFM32, only one AP is used, the AHB-AP which is selected when APSEL is 0.

As with DP accesses, the request can only select 1 of 4 AP registers with the two address bits. To allow more registers to be implemented in an AP, the APBANKSEL field of the SELECT register enables the debugger to specify 4 additional address bits, allowing up to 64 registers to be defined by each AP.

The RDBUFF register returns the result of the previous AP transaction without generating a new memory access. See Section 2.3 (p. 6) .

2.3 AHB-AP

Table 2.2. AHB-AP registers

Address	Read	Write
0x00	CSW	CSW
0x04	TAR	TAR
0x08	N/A	N/A
0x0C	DRW	DRW
0xFC	IDR	N/A

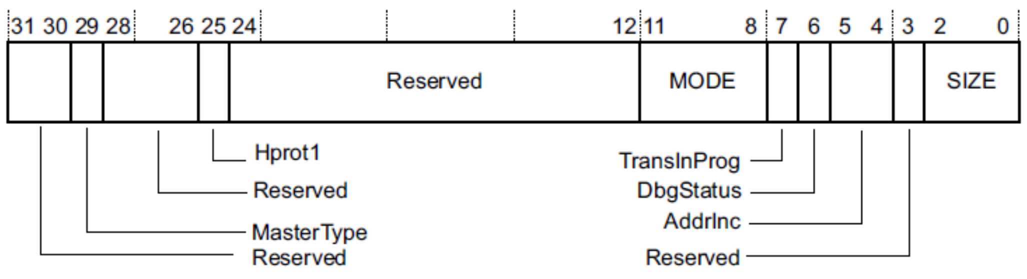
The AHB-AP is the only AP present on the EFM32. It is an implementation of the general MEM-AP, that gives access to the internal memory map of the core. Since memory, peripherals and debug components are all memory mapped, the AHB-AP can be used to both program and debug the MCU.

The AHB-AP is selected by writing 0 to the APSEL field of the SELECT register. This has to be performed before using the AHB-AP as the reset state of the SELECT register is undefined. Before using the AHB-AP the connection should also be verified by reading the IDR register. The IDR register is at address 0xFC. Thus, to read this register, the APBANKSEL field should be set to 0xF. The IDR register can then be read by reading register 3 in that register bank (ADDR[3:2] == 0b11). The IDR register should return the value 0x24770011 on devices with a Cortex-M3 or Cortex-M4 core. On devices with a Cortex-M0+ it should return 0x04770031. If it instead returns the value 0x16E60001, debug access is locked and the AHB-AP is not available. See Section 4.3 (p. 13) for information on how to unlock a device.

When reading from any AP register, the value in the data phase is from the *previous* transaction. That means that in order to read out the value of a register, it is necessary to perform two reads and discard the first result. If performing several sequential reads, it is enough to discard the first value and perform one extra read. The DP register RDBUFF can also be used. This register will return the result of the last AP read operation without generating a new memory access.

The basic operation on the AHB-AP is to read or write from some memory location. The two main registers used for this purpose is the Transfer Address Register (TAR) and the Data Read/Write Register (DRW). The TAR register contains the address of the resource. The addressing scheme is the same as the internal memory map of the core. The DRW register is used to either write or read from the address held in TAR. To write a value to an internal memory address, first write the address to the TAR register, then write the value to DRW. To read a memory address, first write the address to TAR, then read the value in DRW.

Figure 2.5. Control/Status Word Register



The Control/Status Word (CSW) register contains configuration and status signals about the connection to the internal memory bus. Before using AHB-AP to read/write to the internal memory, the SIZE field should be set to configure the size of the memory transfer. The TransInProg bit can be checked to see if a memory transaction is currently in progress.

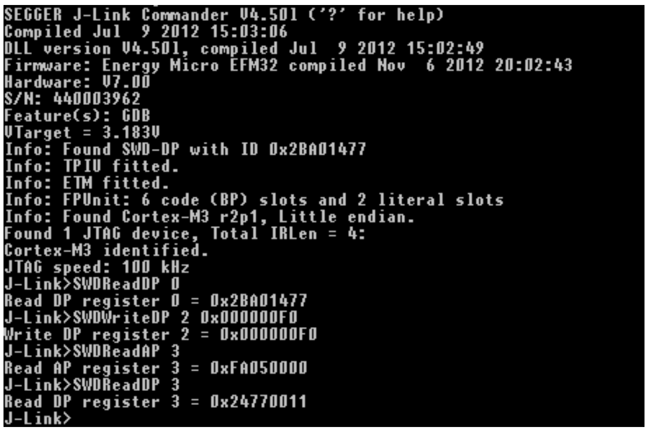
In summary, when using the AHB-AP:

1. Set the CDBGPWRUPREQ and CSYSPWRUPREQ bits of CTRL/STATUS (power up the debug interface)
2. Write 0x000000F0 to SELECT (select AHB-AP, bank 0xF)
3. Read the IDR register
4. Verify that the IDR value is one of the valid values
5. Write 0x00000000 to SELECT (select AHB-AP, bank 0x0)
6. Set the SIZE field of CSW to 0x2 (32-bit transfers)
7. Start using TAR/DRW to access internal memory

2.4 SEGGER J-Link

All Energy Micro Development and Starter Kits come with an integrated SEGGER J-Link debugger. The J-Link debugger handles the SWD connection to the EFM32 and communicates with a PC over a USB cable. The interface on the PC is in the form of a shared library, the JLinkARM.dll. Applications such as the energyAware Commander and IDEs can use this library to connect to the EFM32 directly. To use JLinkARM.dll in an application, an SDK (Software Development Kit) must be obtained from Segger.

Figure 2.6. Read IDCODE and IDR with JLink.exe

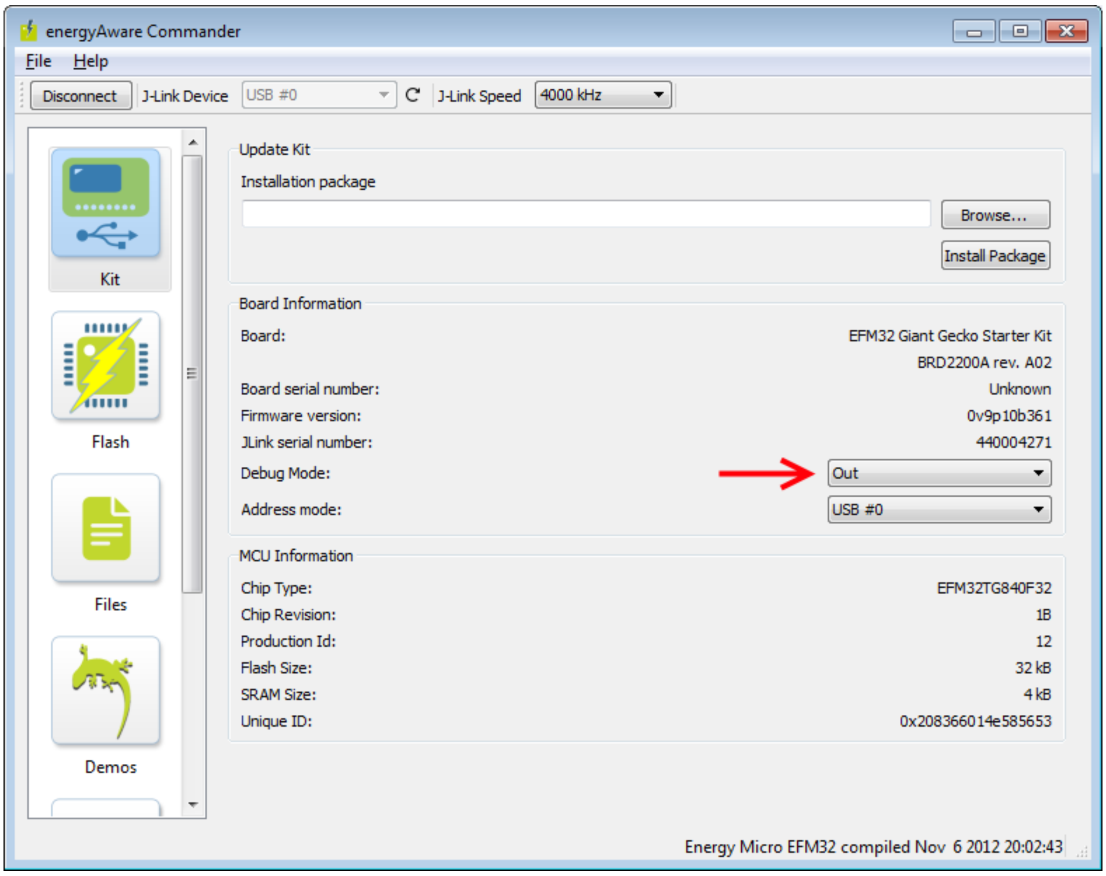


```
SEGGER J-Link Commander V4.50l ('?' for help)
Compiled Jul  9 2012 15:03:06
DLL version V4.50l, compiled Jul  9 2012 15:02:49
Firmware: Energy Micro EFM32 compiled Nov  6 2012 20:02:43
Hardware: V7.00
S/N: 440003962
Feature(s): GDB
VTarget = 3.183V
Info: Found SWD-DP with ID 0x2BA01477
Info: TPIU fitted.
Info: ETM fitted.
Info: FPUnit: 6 code (BP) slots and 2 literal slots
Info: Found Cortex-M3 r2p1, Little endian.
Found 1 JTAG device, Total IRLen = 4:
Cortex-M3 identified.
JTAG speed: 100 kHz
J-Link>SWDReadDP 0
Read DP register 0 = 0x2BA01477
J-Link>SWDWriteDP 2 0x000000F0
Write DP register 2 = 0x000000F0
J-Link>SWDReadAP 3
Read AP register 3 = 0xFA050000
J-Link>SWDReadDP 3
Read DP register 3 = 0x24770011
J-Link>
```

The SEGGER distribution also contains a Command Line Interface (CLI) to directly send SWD commands to the EFM32. The program is called JLink Commander (JLink.exe). Figure 2.6 (p. 7) shows a sample JLink Commander session in where the IDCODE and IDR registers are read. Note that when reading the AP IDR register, a dummy read is performed first, followed by reading the actual value from the DP RDBUFF register.

The Energy Micro kits can also be used to debug external boards. If 'Debug Mode' in energyAware Commander is set to 'Out', the SWD lines are disconnected from the on-board MCU and instead put out on the Debug Header. The Debug Header can be directly connected to the target board with a standard 20-pin flat cable, or if the external board does not contain a 20-pin debug header, it is enough to connect SWDIO, SWCLK, RESET, GND and Vtarget.

Figure 2.7. Configure kit to debug external target



Update Kit

Installation package

Board Information

Board:	EFM32 Giant Gecko Starter Kit BRD2200A rev. A02
Board serial number:	Unknown
Firmware version:	0v9p10b361
JLink serial number:	440004271
Debug Mode:	Out
Address mode:	USB #0

MCU Information

Chip Type:	EFM32TG840F32
Chip Revision:	1B
Production Id:	12
Flash Size:	32 kB
SRAM Size:	4 kB
Unique ID:	0x208366014e585653

Energy Micro EFM32 compiled Nov 6 2012 20:02:43

3 Flash Programming

To load a program on the EFM32 it must be written to flash. This chapter will explain how to use the MSC registers to write data to the internal flash of an EFM32.

3.1 Halting the CPU

Before writing a new program to Flash, the CPU should first be reset and halted. There are two reasons for doing this:

1. Get the CPU and peripherals into a known state
2. Prohibit the CPU from accidentally running partial code while writing the program

Three registers are used to get to this state. All of these registers are described in [cm3trm]

- The Debug Halting Control and Status Register (DHCSR).
- The Application Interrupt and Reset Control Register (AIRCR).
- The Debug Exception and Monitor Control Register (DEMCR).

The process is as follows:

1. Write 0xA05F0003 to DHCSR. This will halt the core.
2. Write 1 to bit VC_CORERESET in DEMCR. This will enable halt-on-reset
3. Write 0xFA050004 to AIRCR. This will reset the core.

Now the CPU will be halted on the first instruction and all peripherals and registers (except for the debug registers) will have their reset value.

3.2 Writing to Flash

Writes to flash (and erase operations) are controlled by the MSC (Memory System Controller) registers. These registers are described in the EFM32 Reference Manual. The minimum write unit is one word (32 bits). The following process outlines how to write a word to flash:

1. Enable flash writing by setting the WREN bit in MSC_WRITECTRL
2. Write the destination address to MSC_ADDRB
3. Load the internal write register by writing a 1 to bit LADDRIM in MSC_WRITECMD
4. Write the word to MSC_WDATA
5. Initiate the write by writing a 1 to bit WRITEONCE in MSC_WRITECMD

This will initiate the write process. The process takes about 20 µs and the status can be checked by polling the BUSY flag in MSC_STATUS.

The Memory System Controller also supports automatic address increments when writing multiple words. Use the WRITETRIG bit instead of WRITEONCE to start the write process in order to trigger address increments. The WDATAREADY flag in MSC_STATUS is used to signal when MSC_WDATA is ready for the next word. Note that address increments only works within a flash page. When writing to the last word of a page, the address will wrap around to the start of the current page.

Different EFM32 families have different page sizes. Page sizes are documented in the Reference Manual for each device family. It is also possible to read out the size of memory and page size directly from the Device Information (DI) page on the MCU. Table 3.1 (p. 10) shows page sizes for the current EFM32 families at the time of writing.

Table 3.1. EFM32 family page sizes

EFM32 Family	Max Flash Size	Page Size
Zero Gecko	32 kB	1 kB
Tiny Gecko	32 kB	512 bytes
Gecko	128 kB	512 bytes
Leopard Gecko	512 kB	2 kB
Wonder Gecko	256 kB	2 kB
Giant Gecko	1 MB	4 kB

The process to erase a page is very similar to writing a word. The start address of the page should be written to MSC_ADDRB and the bit ERASEPAGE is used instead of WRITEONCE/WRITETRIG to start the erase operation. The process can be monitored with the BUSY bit in MSC_STATUS.

When a page is erased all bits are 1, i.e. all words are 0xFFFFFFFF. When writing to flash it is only possible to clear bits (set them to 0). It is possible to write twice to the same word as long as the bits are either untouched or changed from 1 to 0. Note that you should not write more than two times to the same word between each erase, even if you follow this rule.

3.3 Using a Flashloader

There are two main strategies that can be used when programming flash. The first option is to write directly to the MSC registers over the SWD interface. This is the simplest approach, but also the slowest.

A faster method is to first write a small program directly to RAM and then let this program control the MSC registers. Such a program is called a flashloader. This approach is faster because the debugger does not have to continiously check the MSC_STATUS register before writing a new word. A large buffer can be written directly to RAM and the flashloader can take care of polling and feeding the MSC, in addition to signaling the debugger when the buffer can be refilled.

When using a flashloader, the debugger must communicate with the flashloader to know when it is safe to send more data. The communication can be implemented by using fixed-address variables that both parties check. The flashloader can be written directly to RAM.

The source code examples provided with this application note (see Chapter 5 (p. 16)) implements both strategies for programming the target. To see the speed benefits of using a flashloader, consider the following benchmark: on an EFM32GG990F1024, a 512 kB firmware image was programmed and verified by both methods.

Without flashloader: 12 kB/s.

With flashloader: 85 kB/s.

3.4 Lock Bits

The Lock Bits (LB) page can be used to lock specific pages of the flash to prevent them from beeing overwritten. Each flash page has a corresponding bit in the LB page. If the Lock Bit for a particular page is cleared (set to zero), any write or erase command to that page will be ignored. The lock bits are grouped in words, so the first word of the LB page contains lock bits for the first 32 pages of the main flash.

The LB page has its own Lock Bit as well, which is also part of the LB page itself. When this bit is cleared the LB page cannot be erased or written to, which will fix the set of locked flash pages. However, note that when in Debug Mode (when DBGPWRUPACK in CTRL/STAT is set), the LB page is writeable regardless of the LB page Lock Bit. This means that a debugger can connect and change the state of the lock page, even though the LB Page Lock Bit is cleared.

The Lock Bits are documented in the EFM32 Reference Manual.

4 Debug Lock and Unlock

After a device has been programmed it is often desirable to disable debug access to the system, to hinder that others read out or tamper with the device firmware. This chapter will explain how to lock (disable) debug access to the EFM32 and also how to unlock a previously locked chip.

4.1 Lock Debug Access

To lock debug access to the EFM32, clear the Debug Lock Word (DLW) and then perform a hard reset (pin reset) of the device. When the MCU comes out of reset debug access will be locked which means that the debugger can no longer access the internal Cortex memory bus.

The Debug Lock Word is part of the Lock Bits Page.

4.2 Verify Debug Status

When the MCU is locked, access to the internal memory through AHB-AP is no longer available. Instead, only a special set of registers, which is part of the Authentication Access Port (AAP) are accessible. How to access these registers vary slightly between different EFM32 families.

On EFM32 families with an M3 or M4 core, the AAP registers replace the AHB-AP registers when the device is locked. This means that when the debugger tries to read the IDR value of the AHB-AP it will instead see the IDR value from AAP.

On families with an M0+ core, the AAP registers are memory mapped to the internal memory space and the AHB-AP is always available. However, when the device is locked the AHB-AP can only access the AAP registers and nothing else. When the device is *not* locked the debugger can not access the AAP registers. The AAP registers are documented in the DBG section of the EFM32 Reference Manual.

To verify if an M0+ device is locked, the debugger must therefore try to read the IDR value of the AAP at the memory mapped location. If the debugger can read the AAP IDR value, the device is locked.

Note

The IDR value of the AAP is 0x16E60001 on all devices.

The following examples illustrate how to verify if an EFM32 device is locked. The commands can be used in J-Link Commander (JLink.exe).

4.2.1 Verify Debug Lock on M3/M4 Devices

```
// First Write 0x000000F0 to SELECT to select
// the last register bank of AP #0.
SWDWriteDP 2 0x000000F0

// Dummy-read the fourth register in this
// bank (A[3:2] == 0b11), this is the IDR register.
SWDReadAP 3

// Read the RDBUFF register to get the
// actual contents of IDR
SWDReadDP 3
```

If the output of the last command is 0x16E60001 the device is locked and only the AAP registers are available. If the value is 0x24770011 the device is NOT locked and the AHB-AP is accessible.

4.2.2 Verify Debug Lock on M0+ Devices

```
// First Write 0x00000000 to SELECT to select
// the first register bank of AP #0 (AHB-AP)
SWDWriteDP 2 0x00000000

// Write address of AAP_IDR to the TAR register
SWDWriteAP 1 0xF0E000FC

// Dummy read the DRW register. This will
// generate a memory access to read IDR
SWDReadAP 3

// Read the RDBUFF register to get the
// actual contents of IDR
SWDReadDP 3
```

If the output of the last command is 0x16E60001 the device is locked and only the AAP registers are available. If the value is anything else the device is NOT locked and the AHB-AP can access the internal memory.

4.3 Unlock Debug Access

When an EFM32 is locked the AAP registers can be used to unlock the device. Unlocking a device is only possible by performing a Device Erase, where all data both in flash and RAM (and cache) will be erased.

Note

The User Data page will NOT be erased by a Device Erase. Never place secret data such as encryption keys in this page.

Table 4.1. AAP registers

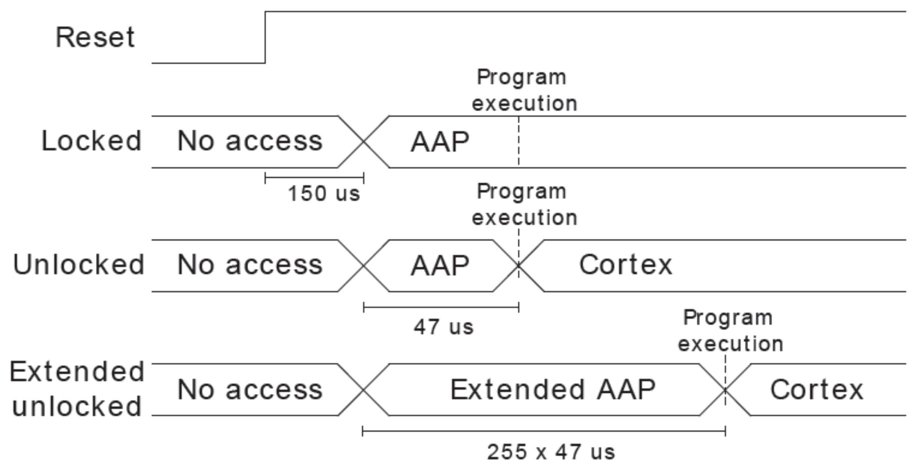
Address	Read	Write
0x00	N/A	AAP_CMD
0x04	N/A	AAP_CMDKEY
0x08	AAP_STATUS	N/A
0xFC	IDR	N/A

To unlock a locked device, follow this process:

1. Write 0xCFACC118 to AAP_CMDKEY to enable writes to AAP_CMD
2. Write 1 to the DEVICEERASE bit of AAP_CMD
3. Check the ERASEBUSY flag in AAP_STATUS to see when the Device Erase operation is complete
4. When the erase is complete reset the device by pin reset or by using the SYSRESETREQ bit of AAP_CMD

4.4 AAP Window

Figure 4.1. AAP Window



When an EFM32 is reset, there is a short time before the core starts executing code where the AAP registers are available regardless of the Debug Lock State. This window is normally 47 µs but it can be extended by sending a special sequence on the SWDIO/SWCLK before RESET pin is released. If the debugger holds RESET low, sends the expansion sequence and then releases RESET, the AAP window will become 255 * 47 µs, giving the debugger more time to access the AAP registers. The AAP Window Expansion Sequence consists of 4 pulses on SWDIO with SWCLK HIGH followed by 4 pulses on SWDIO with SWCLK LOW, see Figure 4.2 (p. 15) .

Note

The AAP Expansion Sequence is not available on Gecko (EFM32G) devices. On these devices the debugger must operate fast enough fit all the commands to AAP within the 47 µs window.

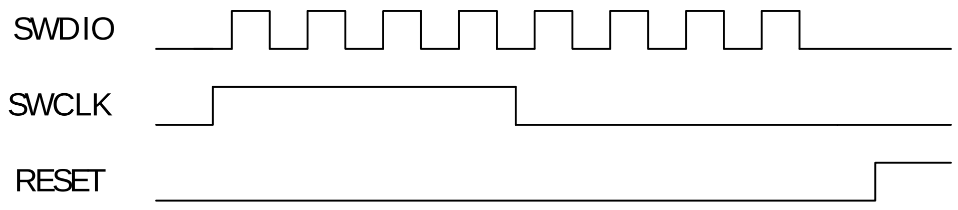
The AAP window is useful when you lose debug access by some *other* method than Debug Lock (e.g. disabling debug pins or entering EM4 very early in your program). In such a case it is possible to recover debug access by using the AAP to unlock the device during the AAP window.

There are slight differences when using AAP in the AAP window as opposed to when the device is locked. When using the AAP window to unlock a device, the actual Device Erase (Unlock) operation will not start before the AAP window completes. This implies that it is not possible to poll the ERASEBUSY flag to see when the operation is complete. Furthermore, the device will stay in the AAP window as long as AAP_CMDKEY is set to the valid unlock key. This means that after setting the DEVICERASE bit in AAP_CMD, the AAP_CMDKEY register must be cleared again in order to exit the AAP window.

To unlock a device using the AAP window, the process is the following:

1. Hold RESET low while sending the AAP window expansion sequence (not possible on Gecko)
2. Release RESET and verify that the debugger can access the AAP by reading IDR
3. Write 0xCFACC118 to AAP_CMDKEY to enable writes to AAP_CMD
4. Write 1 to the DEVICEERASE bit of AAP_CMD
5. Write 0x00 to AAP_CMDKEY to allow exit from the AAP window
6. Wait at least 100 ms for Device Erase to complete
7. Issue a pin reset to the target

Figure 4.2. AAP Expansion Sequence



5 Source Code Example

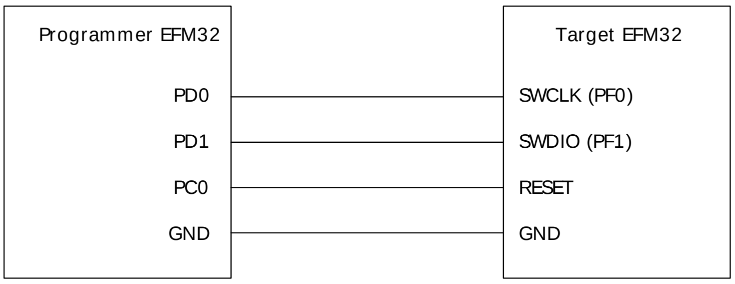
This application note comes with a source code example which implements a simple programmer by bit-banging GPIO pins. Projects are included for the EFM32GG-STK3700 and EFM32GG-DK3750.

The following features are demonstrated:

- Program by writing directly to MSC
- Program by using a flashloader
- Lock target
- Unlock target

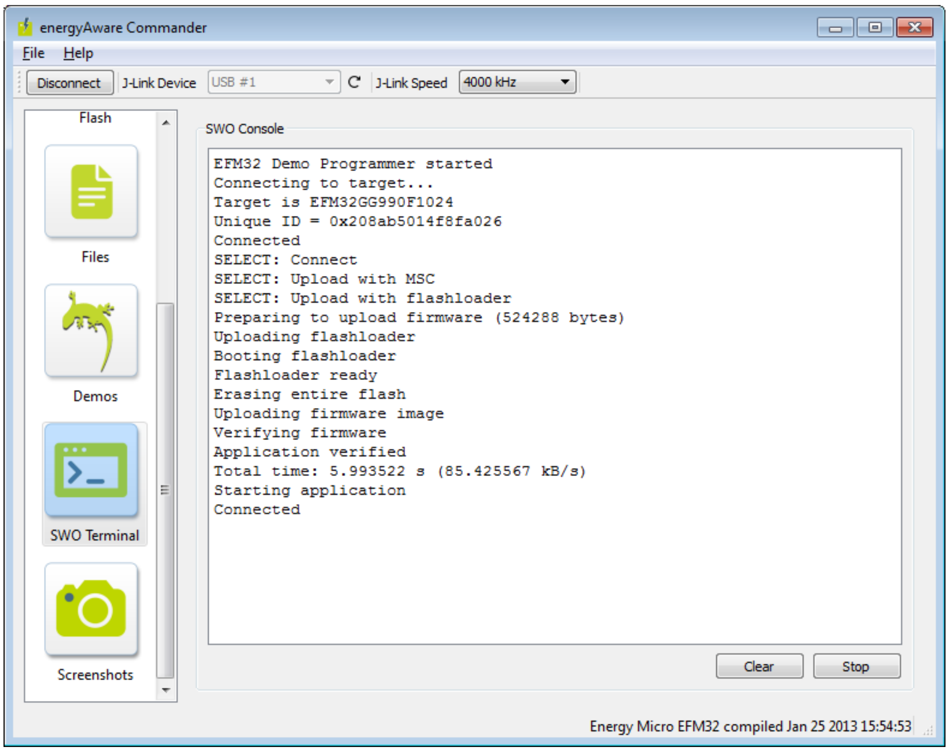
To use the programmer connect RESET, SWCLK, SWDIO and GND from the target to the configured pins on the programmer. Figure 5.1 (p. 16) shows the default configuration when using the EFM32GG-STK3700. The voltage level (e.g. 3.3V) should be the same on both host and target. The pins can be configured in dap.h

Figure 5.1. Default Programmer Connection EFM32GG-STK3700



The program is operated by using PB0 to select an action and PB1 to execute the current action. The segmented LCD will display information about the current action. LED1 is on when a target is connected. LED0 is on whenever the programmer is busy with an operation, e.g. programming the target. More information is output over SWO, which can be viewed by connecting to the kit with the energyAware Commander and selecting the SWO tab.

Figure 5.2. SWO output with energyAware Commander



```
EFM32 Demo Programmer started
Connecting to target...
Target is EFM32GG990F1024
Unique ID = 0x208ab5014f8fa026
Connected
SELECT: Connect
SELECT: Upload with MSC
SELECT: Upload with flashloader
Preparing to upload firmware (524288 bytes)
Uploading flashloader
Booting flashloader
Flashloader ready
Erasing entire flash
Uploading firmware image
Verifying firmware
Application verified
Total time: 5.993522 s (85.425567 kB/s)
Starting application
Connected
```

5.1 Flashloader and Firmware Projects

Separate projects are included for the flashloader and sample firmware. Linker options in the programmer project includes the binary output of these project in the programmer output file.

5.2 Source Code Overview

The low level implementation of the SWD protocol is in *dap.c*. Higher level functions, such as halting the target or reading the page size are found in *utils.c*.

The files *errors.c* and *errors.h* implements a simple exception-type system for C. This allows errors to be raised by a low-level function and be catched several levels up in the call hierarchy. While this allows the source code to be simpler and easier to read, it is not always safe to use. Please read comments in errors.c carefully.

Direct writes to flash via the MSC registers are handled by *flash_write.c*, while *use_flashloader.c* handles programming by using the flashloader. Standard functions for Debug Lock and Unlock are in *debug_lock.c* and specialized functions to handle Gecko (no AAP expansion) and Zero (memory mapped AAP) unlock sequences are put in separate files.

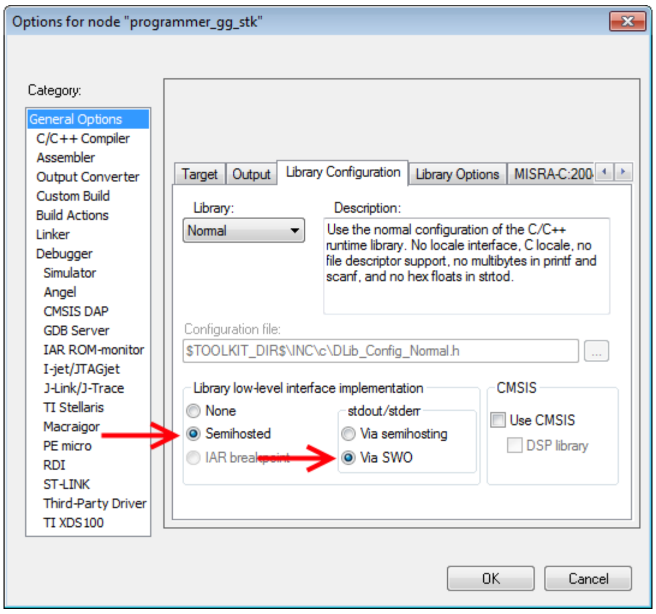
To implement AAP window unlock for Gecko devices, the EBI is used instead of GPIO to make the bit-bang sequence faster. This is only needed for this particular case, but when using Gecko Unlock with EBI, the programmer *must* use pins PE8 and PE9 for SWCLK and SWDIO.

state_machine.c handles the user interface and allows the user to select different actions. The kit specific features (LEDs, buttons) are put in *kits.c*.

5.3 Compiling the Projects

The programmer project needs to be compiled with low level IO functions mapped to SWO. See Figure 5.3 (p. 18) .

Figure 5.3. Configure SWO semihosting in IAR



Options for node "programmer_gg_stk"

Category:

General Options
C/C++ Compiler
Assembler
Output Converter
Custom Build
Build Actions
Linker
Debugger
Simulator
Angel
CMSIS DAP
GDB Server
IAR ROM-monitor
I-jet/JTAGjet
J-Link/J-Trace
TI Stellaris
Macraigor
PE micro
RDI
ST-LINK
Third-Party Driver
TI XDS100

Target | Output | Library Configuration | Library Options | MISRA-C:200

Library: Normal

Description: Use the normal configuration of the C/C++ runtime library. No locale interface, C locale, no file descriptor support, no multibytes in printf and scanf, and no hex floats in strtod.

Configuration file: $TOOLKIT_DIR$\INC\c\DLib_Config_Normal.h

Library low-level interface implementation: None [✓ Semihosted] IAR breakpoint

stdout/stderr: Via semihosting [✓ Via SWO]

CMSIS: Use CMSIS, DSP library

OK Cancel

The programmer project depends on the binary output from the flashloader and firmware (blink) projects. These projects must therefore be compiled first. The binary images are loaded into the final output by using linker options. The linker options used in IAR are:

```
--image_input flashloader\Debug\Exe\flashloader.bin,flashloader_bin,flashloader_section,4
--image_input blink\Debug\Exe\blink.bin,fw_bin,fw_section,4
```

The symbol names *fw_bin* and *flashloader_bin* must be included in the "keep symbols" configuration to the linker in order to prevent the linker from eliminating them.

In order to achieve the fastest programming, code optimizations should be set to speed when compiling. If the macro SWD_FAST is defined, faster versions of the low-level functions are compiled. These come with some limitations, please read the comments in dap.h.

Pin configurations are in dap.h. Change this file if you want to use other pins. The default are PD0,PD1 and PC0 (for SWCLK, SWDIO and RESET) when using the STK project, and PE8,PE9 and PC0 when using the DK project.

A Acronyms

AAP	Authentication Access Port
AHB	Advanced High-performance Bus
AHB-AP	Advanced High-performance Bus Access Port
AP	Access Port
DAP	Debug Access Port
IDE	Integrated Development Environment
MSB	Most Significant Bit
MSC	Memory System Controller
LB	Lock Bit
LSB	Least Significant Bit
SWD	Serial Wire Debug
SWJ-DP	Serial Wire and JTAG Debug Port
SWO	Serial Wire Output
SW-DP	Serial Wire Debug Port

6 References

[adi5] . *ARM Debug Interface v5 Architecture Specification*.

[adi51] . *ARM Debug Interface v5 Architecture Specification Supplement (ADI v5.1)*.

[cm3trm] . *Cortex-M3 Technical Reference Manual*.

7 Revision History

7.1 Revision 1.02

2013-11-26

Fixed a bug where the programmer would use the wrong value for TAR wrap-around on Zero Gecko

7.2 Revision 1.01

2013-10-14

New cover layout

7.3 Revision 1.00

2013-07-16

Initial revision.

B Disclaimer and Trademarks

B.1 Disclaimer

B.2 Trademark Information

C Contact Information

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

Please visit the Silicon Labs Technical Support web page:
http://www.silabs.com/support/pages/contacttechnicalsupport.aspx
and register to submit a technical support request.

Table of Contents

List of Figures

List of Tables

silabs.com

ZERO
ARM Cortex-M0+

TINY
ARM Cortex-M3

GECKO
ARM Cortex-M3

LEOPARD
ARM Cortex-M3

GIANT
ARM Cortex-M3

WONDER
ARM Cortex-M4