INSTALLING TOOLS TO COMPILE AND RUN THE PROJECTS

Table of contents

1 Introduction

The development board EMF32GG-STK3700 features a EFM32GG990F1024 microcontroller from the Giant Gecko family. It means it is a Cortex M3. Its main features are:

Feature	Description
Flash (KB)	1024
RAM (KB)	128
GPIO	87
USB	Host
LCD	8x34
USART+UART	3/2
LEUART	2
Timer/PWMRTC	4/12
ADC	1(8)
DAC	2(8)
OpAmp	3

2 Peripherals

There is a lot of peripherals in the board.

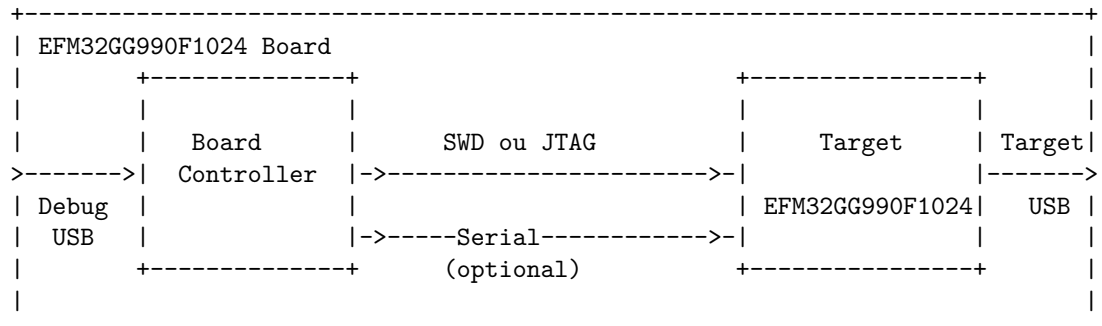
- LEDs: Using pins PE2 and PE3.
- Buttons: Using pins PB9 and PB10

- LCD Multiplexed 20x8 with a 4 character alphanumeric field, a 4 digit numeric field and other symbols at pins PA0-PA11, PA15,PB0-PB6,PD9-PD12,PE4-PE7
- Touch Sensor: Using pins PC8-11.
- Light Sensor: PD6,PC6
- LC Sensor: PB12,PC7
- NAND Flash: PB15, PE8-15, PC1-2, PF8-9, PD13-15
- USB OTG: PF11-12, PF5-6

It is possible to use the header connectors to add more.

3 Block diagram

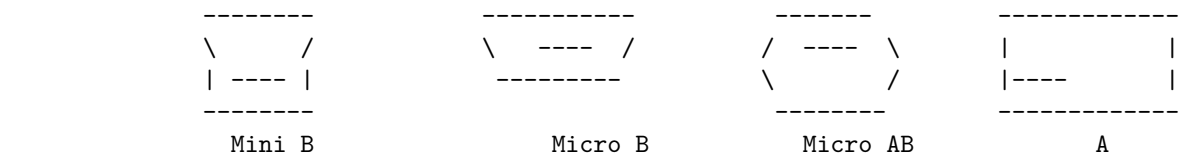
The board has the following block diagram.



4 Connections

The EMF32GG-STK3700 board has two USB connectors:

- Debug USB (Mini USB B Connector), connected to the Board Controller, for development and debugging
- Target USB (Micro AB-Type USB Connector), connected to the target EFM32F990F1024 microcontroller



For development, a cable (delivered) must be connected between the Mini USB connector on the board and the A-Type connector on the PC. Among the devices listed by the *lsusb* command, the following must appear.

```
Bus 001 Device 019: ID 1366:0101 SEGGER J-Link PLUS
```

The STK3700 board has two microcontrollers: one, called target, is a EFM32GG990F1024 microcontroller, and the other, called Board Controller implements an interface for programming and debugging.

For a ARN Cortex M microcontroller the following programming interfaces can be used:

Interface	# pins	Signals	Name	OBS
SWD	2	SWCLK, SWDIO	Serial Wire Debug.	
JTAG	4	TCK TMS TDI TDO	Joint Test Action Group	Not used

There is a connector on the board that permits the debugging using external SWD adapters.

There is a serial interface between the Target and the Board Controller. It is implemented using a physical channel with 2 lines. It appears to the Host PC as a serial virtual port (COMx in Windows or /dev/ttyACMx in Linux). In the STK3700 board the serial channel uses the UART0 unit (PE0 and PE1).

5 Power

The board can be powered thru the:

- Debug USB (DBG)
- Target USB (USB)
- On board battery (BAT)

There is a 3-position switch with positions BAT, USB and DBG. When the board is powered thru the USB debug port, the switch must be in the DGB position.

6 Tools

The essential tools needed for compiling the projects are:

1. Toolchain (Compiler/Linker/Debugger/Tools) supporting the microcontroller family
2. Files specific to the microcontroller from ARM and manufacturer (header files, libraries and start files)
3. Flash tools; 4. Make;

Highly recommended are the following tools:

1. Git: Version Control
2. Doxygen: Documentation tool from source code
3. An text editor like gedit (Linux), Notepad++(Windows).

To debug an application, a debugger is needed and it is part of the GNU C toolchain. It has a very crude CLI (Command Line Interface (CLI). So, an interface with a graphic User Interface (GUI) for the debugger is more comfortable.

Optionally, one can use an Integrated Developing Environments (IDE) instead of an text editor.

7 GNU Arm Toolchain

7.1 Introduction

The GNU toolchain is a package of applications that runs on a host and generates application for target microcontroller (cross-compiling). It consists of:

- compiler including preprocessor
- linker
- archiver
- utils like nm, ranlib, etc.

In Arm GNU Toolchain there are many versions of the GNU toolchain. To develop for microcontrollers without operating systems (bare-metal), the version with the none-eabi suffix is to be used.

7.2 Toolchain versions

The are versions of the toolchain for Windows, Linux and Mac(darwin) operation systems. The Linux and Darwin version can be based on a system with a Intel or an ARM processor. For Windows hosted systems there are self-installing executable and a zip file.

There are for 32-bit (AArch32) and 64-bit (AArch64))microprocessors. There are version for EABI and EABIHF programming interfaces (See below).

There are text files containing the checksum for MD5 (.asc suffix) and SHA26 (.sha356asc suffix). So it is possible to verify the integrity of the downloaded file.

The name convention for the files is show below.

`arm-gnu-toolchain-<Release Version>-<Host>-<Target Triple>.tar.xz`

The only target of interest is none, i.e., bare metal system or in other words, without an operating system. Which version to use depends on the operating system of the host computer and on the microcontroller to be used.

The table below shows a listing of the applications in the toolchain. To avoid conflict with the applications targeted to the host system, all applications in the GNU Arm Toolchain have a prefix. In case of bare metal systems, the prefix is *arm-none-eabi-*

Application	Description
addr2line	Convert addresses into file names and line numbers
ar	Create, modify, and extract from archives
as	The portable GNU assembler
c++	C++ compiler
c++filt	Demangle C++ and Java symbols
cpp	The C Preprocessor
elfedit	update ELF header and program property of ELF files
g++	C++ compiler
gcc	C compiler
gcc-13.2.1	C compiler (alias)
gcc-ar	a wrapper around ar adding the –plugin option
gcc-nm	a wrapper around nm adding the –plugin option
gcc-ranlib	a wrapper around ranlib adding the –plugin option
gcov	coverage testing tool
gcov-dump	offline gcda and gcno profile dump tool
gcov-tool	offline gcda profile processing tool
gdb	The GNU Debugger
gdb-add-index	Add index files to speed up GDB
gfortran	Fortran compiler
gprof	Display call graph profile data
ld	The GNU linker
ld.bfd	The GNU linker that uses BFD libraries
lto-dump	Tool for dumping LTO object files
nm	list symbols from object files
objcopy	copy and translate object files
objdump	display information from object files
ranlib	generate an index to an archive
readelf	display information about ELF files
size	list section sizes and total size of binary files
strings	print the sequences of printable characters in files
strip	discard symbols and other data from object files

7. Application Binary Interface (ABI)

There are many ABIs sponsored by ARM:

- ABI (Application Binary Interface). Obsolete. Also called OABI (Old ABI)
- EABI (Embedded Application Binary Interface):
- EABIHF (Application Binary Interface for devices with hardware floating point)

In the EABI, the parameters in and out of a function are in the integer registers. When a processor has a hardware floating point unit, optionally the parameter can be passed using the floating point registers.

8 Installing procedure for Linux machines

8.1 Introduction

In Linux system, the installation process is straightforward. Some software like Make, Git and Doxygen can be installed using the Software Manager used. For example, in Debian machine, just enter the commands

```
sudo apt install make git doxygen
```

It is also possible to use a graphical interface like Synaptic to install packages.

The newer versions of the toolchain uses the XZ compactor. To unpack it, it is needed to install the xz-utils package.

```
sudo apt install xz-utils
```

TODO : liblzma Unpacker tool XZ files for Linux. Python v3.8 is needed by GDB M

8.2 Installing the compiler toolchain

A toolchain for ARM is the ARM GNU CC, which includes compiler, assembler, linker, debugger, and many utilities. It can be downloaded from ARM GNU GCC or installing using a package manager like apt. The toolchain is a version of the GNU C ported to ARM processors in a project backed by ARM itself.

Under the Section x86_64 Linux hosted cross toolchains: AArch32 bare-metal target (arm-none-eabi), one can find the following files:

> To verify the integrity of the downloaded files one can use the md5sum application and compare the result with the one in the .asc file. The .sha256asc contains the checksum generated by sha256sum.

The package file, named gcc-arm-none-eabi-XXXXX-x86_64-linux.tar.bz2 (XXXXX is the version information) must be downloaded into a directory, for example, $HOME/Downloads. Linux distributions have an special directory for the installation of packages outside the package manager used by the distribution. Generally it is the /opt directory.

```
# Unpack int cd /opt/ tar -xvf
$HOME/Downloads/gcc-arm-none-eabi-XXXXX-x86_64-linux.tar.bz2
# Change name to a name without version information mv
gcc-arm-none-eabi-XXXXX-x86_64 /opt/gcc-arm-none-eabi
```

The name was change to a name without version information, in order to avoid extra work, when upgrading the toolchain.

The next step is to add it to the PATH variable, to it can be run without the need of a full path. This is done by editing the $HOME/.bashrc file and adding the following lines to the end.

```
# # Embedded ARM GCC # ARMGCC_HOME=/opt/gcc-arm-none-eabi
PATH=$ARMGCC_HOME/bin:$PATH
```

After installing the toolchain, the following applications are available as commands:

```
arm-none-eabi-addr2line   arm-none-eabi-gcc-nm          arm-none-eabi-lto-dump
arm-none-eabi-ar          arm-none-eabi-gcc-ranlib      arm-none-eabi-nm
arm-none-eabi-as          arm-none-eabi-gcov            arm-none-eabi-objcopy
arm-none-eabi-c++         arm-none-eabi-gcov-dump       arm-none-eabi-objdump
arm-none-eabi-c++filt     arm-none-eabi-gcov-tool       arm-none-eabi-ranlib
arm-none-eabi-cpp         arm-none-eabi-gdb             arm-none-eabi-readelf
arm-none-eabi-elfedit     arm-none-eabi-gdb-add-index   arm-none-eabi-size
arm-none-eabi-g++         arm-none-eabi-gfortran        arm-none-eabi-strings
arm-none-eabi-gcc         arm-none-eabi-gprof           arm-none-eabi-strip
arm-none-eabi-gcc-13.2.1  arm-none-eabi-ld
arm-none-eabi-gcc-ar      arm-none-eabi-ld.bfd
```

To verify the installation, enter the name *arm-none-eabi-gcc* in a command prompt. If the output contains "no input files", the installation is correct. If not, the procedure above must be repeated adding the correct folder to PATH.

8.3 Files specific to the EFM32

The files needed for development for a specific microcontroller are:

```
• headers: that define the registers to access the hardware.
• library files: libraries needed by gcc (can be superseded by
the ones in the toolchain) • start files: contains initialization
routines and interrupt vectors.  • linker script: instructs the
gcc how to correctly build the program
```

They are contained (and a lot more) in a Software Development Kit (SDK), called *Gecko_SDK*, provided the Silicon Labs. For a while, the Gecko_SDK was no more supported as a standalone product. It was (and is) contained if the Simplicity Studio, a free IDE provided by Silicon Labs. Simplicity Studio is based on Eclipse, and one of its features is that it downloads the support files as they are needed for a project reducing the demands on the capacity of the hard drives.

> *Gecko_SDJ is HUGE, about 300 MBytes zipped and 2,4 GBytes after unzipping.

Gecko_SDK is now available again at github. An older version can still be found at Silicon Labs Support Documents. But to reduce the demand on disk area, one can use a minimal *Gecko_SDK* that contains only the files needed to compile the projects. Just clone Mininal Gecko_SDK repository.

To get the most recent GeckoSDK, one can clone the github repo Geck SDK.

```
cd GECKOSDKDIR
git clone https://github.com/SiliconLabs/Gecko_SDK
```

To install from compacted files, follow the steps below, replacing DOWNLOAD and GECKOSDKDIR with the full path of the Download and Gecko_SDK folder:

1. Download the package (Gecko_SDK.zip or Gecko_SDK_minimal.zip) using a browser into a DOWNLOAD folder. 2. Using command line, change to the folder where it will be installed.

   cd GECKOSDKDIR

2. Unzip

   unzip DOWNLOADS/Gecko_SDK.zip or unzip DOWNLOADS/Gecko_SDK_minimal.zip

In the GECKOSDK/platform/Device/SiliconLabs/EFM32GG/Include/ folder one can find the headers with the register and field definition. For example, for the EFM32GG990F104 microcontroller, there is efm32gg990f1024.h file. The symbols are defined according the CMSIS Standard. There are also header files for peripherals like efm32gg_gpio.h, efm32gg_adc.h and others, which are already included by the device specific header file, so there is no need to explicitly include them.

Although it is possible and quite common to include the device specific file directly, a better approach is to use a "generic" header, and define which device using a pre-preprocessor symbol during compilation. Copy then the file em_device.h from the same folder to the project folder and define the symbol EFM32GG990f1024 with the -DEFM32GG990f1024 compiler parameter. During compilation the correct file will be included. This is the approach followed in the projects.

There is other header file, called em_chip.h locate in the Gecko_SDK/platform/emlib/inc folder that defines the CHIP_Init function. This function must be called at the very beginning and it corrects some bugs in core implementation. Since it calls other header files from the same folder, a better idea is to add this folder to the include path with the -IGecko_SDK/platform/emlib/inc/ parameter.

The package Gecko_SDK contains the start files and linker scripts. But since they must be adapted to a project, they are already part of the project files.

8.4 Tool to flash the device

To transfer the binary memory image to the flash memory of the microcontroller, one can use the Segger J-Link software. It is also possible to use OpenOCD, but it is not (yet) tested.

First, the JLink Debian package must be downloaded from Segger. Choose *64-bit DEB Installer*. Optionally, it is possible to install the Ozone package, an GUI

interface for JLink. Look down below in the page. Again, download the 64-bit DEB Installer.

To install them on Linux, run the following commands, but take note that the version information changes often.

```
sudo dpkg -i JLink_Linux_V622b_x86_64.deb
sudo dpkg -i ozone_2.54_x86_64.deb
```

After installing JLink, the following application are available in the /opt/SEGGER/JLink folder, with the corresponding links in /usr/bin folder.

```
JFlashSPI_CL        JLinkExe            JLinkGDBServer
JLinkLicenseManager JLinkRegistration   JLinkRemoteServer
JLinkRTTClient      JLinkRTTLogger JLinkSTM32          JLinkSWOViewer
JTAGLoadExe         Ozone
```

If the links in the /usr/bin folder are not created, adjust the PATH to include the folder /opt/SEGGER/JLink as done for the toolchain.

8.5 Debugging Tools

The J-Link package installed as above contains the programs needed to create a debug session.

9 Installing procedure for Windows machines

9.1 General instructions

In Windows, there is always the possibility to

- Install all packages directy in Windows * Use a virtual machine with a Linux distribution and install the linux packages * Install the Windows Subsystem for Linux (WSL/WSL2)

The approach shown here, is to install Windows versions of the different tools.

Recent version of the GNU Arm Toolchain demands the installation of the mingW-64 package. Them *mingw-64* is a successor of an old package called mingw, that adds to windows machines support for GNU tools. It includes

9.2 Install MingW-64

To install mingw-64, access MinGW-64 Downloads and click on the WinLibs link. It redirects to the standalone build of GCC and MinGW-w64 for Windows site.

There are versions for different runtimes (MSVCRT and UCRT) and different threading libraries (POSIX, WIN32, MCF). Good choices are UCRT and POSIX.

Unpack the downloaded file into a folder MINGWDIR (or whatever name was choosen).

Added the MINGWDIR/bin folder to the PATH. See below.

9.3 Adding a folder to the PATH

An important step in the installation of a software in Windows, that must be accessed thru a command line, is to add it to the PATH environment variable. This variable is a list of folders searched by Windows when trying to locate a command.

1. In Search, search for and then select: System (Control Panel)
2. Click the Advanced system settings link.
3. Click Environment Variables. In the section System Variables find the PATH environment variable and select it. Click Edit. If the PATH environment variable does not exist, click New.
4. In the Edit System Variable (or New System Variable) window, clock the New button (or if it is wrong defined, Edit button).
5. Enter the full path of the folder where the software was installed and click the OK button.
6. Close all windows opened for the editing and all command prompt windows.
7. Open a command prompt window and enter the program name and press Enter. This command prompt window must be open AFTER the change of the PATH variable, so it can get the new value.
8. A message like "program is not recognized as an internal or external program." shows that the PATH is not correctly defined and the procedure must be repeated with the correct path.

9.4 Installing make

There are three source of make for Windows machines.

- Make for Windows
- Chocolatey: The Package Manager for Windows
- Builtin Make in Mingw

Using the *Make for Windows* approach, the steps are:

1. Download the *Complete package, except sources* Setup 2. Locate it and double click it, to start execution of the installer.

Using the *Chocolatey* approach

1. Install *Chocolatey* following these instructions
2. Enter the command below in a Power Shell prompt as an administrator.

   choco install make

In both case, to verify the installation, enter the command

```
make --version
```

If the output contains something like *make is not recognized as an internal or external program*, the installation is wrong and must be repeated. It is also possible that it is not in PATH.

9.5 Installing the toolchain

It is possible to install the toolchain from a compressed file or using an installer.

The corresponding files can be found in Section *Windows (mingw-w64-i686) hosted cross toolchains AArch32 bare-metal target (arm-none-eabi)*

```
arm-gnu-toolchain-VERSION-SUBVERSION-RELEASE-mingw-w64-i686-arm-none-eabi.zip
arm-gnu-toolchain-VERSION-SUBVERSION-RELEASE-mingw-w64-i686-arm-none-eabi.zip.asc
arm-gnu-toolchain-VERSION-SUBVERSION-RELEASE-mingw-w64-i686-arm-none-eabi.zip.sha256asc
arm-gnu-toolchain-VERSION-SUBVERSION-RELEASE-mingw-w64-i686-arm-none-eabi.exe
arm-gnu-toolchain-VERSION-SUBVERSION-RELEASE-mingw-w64-i686-arm-none-eabi.exe.asc
arm-gnu-toolchain-VERSION-SUBVERSION-RELEASE-mingw-w64-i686-arm-none-eabi.exe.sha256asc
```

For now

```
VERSION = 13
SUBVERSION = 2
RELEASE = rel1
```

The file of intereset is *arm-gnu-toolchain-VERSION.SUBVERSION.RELEASE-mingw-w64-i686-arm-none-eabi.zip* altough its executable version with the .exe suffix can be used.

To use the executable option, download the file *arm-gnu-toolchain-VERSION.SUBVERSION.RELEASE-mingw-w64-i686-arm-none-eabi.zip* or a newer version form GNU Arm Embedded Toolchain Downloads. Using the File Manager, locate it in the Downloads folder and double click on it. Enter the installation folder when asked.

Using the compressed filed approach, the file *gcc-arm-none-eabi-10.3-2021.10-win32.zip* (or a newer version) must be downloaded from GNU Arm Embedded Toolchain Downloads. Using the File Manager, locate it in the Downloads folder and double click on it. Enter the installation folder when asked.

A good idea is to install in a folder without version information. Doing so, upgrading the toolchain will not demand extra steps to change the configuration.

The next step is to add the folder to the PATH as described in section 9.2.

9.6 Files specific to the EFM32

The files needed for development for a specific microcontroller are:

- `headers: that define the registers to access the hardware.`

- `library files: libraries needed by gcc (can be superseded by the ones in the toolchain)`
- `start files: contains initialization routines and interrupt vectors.`
- `linker script: instructs the C compiler on how to correctly build the program.`

They are contained (and a lot more) in a Software Development Kit (SDK), called Gecko_SDK, provided the Silicon Labs. It is now part of the Simplicity Studio, an free IDE provided by Silicon Labs. Simplicity Studio is based on Eclipse, and one of its features is that it downloads the support files as they are needed for a project.

But recently, Gecko_SDK is available at github again and it is still maintained.

> Recent modifications of GeckSDK have broken the Makefiles. The Cortex-M files are now found in > the GECKOSDKDIR/platform/CMSIS/Core/Include and not in GECKOSDKDIR/platform/CMSIS/Include anymore. > The Makefiles must be modified accordling by setting

Alternatively an old version can still be found at Silicon Labs Support Documents.

> Gecko_SDK is HUGE, about 300 MBytes zipped and 2,4 GBytes after unzipping.

To spare disk space, one can clone a Mininal Gecko_SDK repository, that consists of a stripped down version of Gecko_SDK, that contains only the files needed to compile the projects.

To install them, follow the steps below, replacing DOWNLOAD and GECKOSDK with the full path of the Download and Gecko_SDK folder:

1. Download the package (Gecko_SDK.zip or Gecko_SDK_minimal.zip) using a browser into a DOWNLOAD folder. 2. Using command line, change to the folder where it will be installed.

   cd Gecko_SDK

2. Unzip using command line

   unzip DOWNLOADS/Gecko_SDK.zip or unzip DOWNLOADS/Gecko_SDK_minimal.zip or the builtin unzip tool in Windows by double clicking on the downloaded file. Choose the folder where the Gecko_SDK is to be unpacked, name hereafter GECKOSDKDIR.

In the GECKOSDKDIR/platform/Device/SiliconLabs/EFM32GG/Include/ folder one can find the headers with the register and field definition. For example, for the EFM32GG990F104 microcontroller, there is efm32gg990f1024.h file. The symbols are defined according the CMSIS Standard. There are also header files for peripherals like efm32gg_gpio.h, efm32gg_adc.h and others, which are already included by the device specific header file, so there is no need to explicitly include them.

Although it is possible and quite common to include the device specific file directly, a better approach is to use a "generic" header, and define which device using a pre-preprocessor symbol during compilation. Copy then the file em_device.h from the same folder to the project folder and define the symbol EFM32GG990f1024 with the -DEFM32GG990f1024 compiler parameter. During compilation the correct file will be included. This is the approach followed in the projects.

There is other header file, called em_chip.h locate in the Gecko_SDK/platform/emlib/inc folder that defines the CHIP_Init function. This function must be called at the very beginning and it corrects some bugs in core implementation. Since it calls other header files from the same folder, a better idea is to add this folder to the include path with the -IGecko_SDK/platform/emlib/inc/ parameter.

The package Gecko_SDK contains the start files and linker scripts. But since they must be adapted to a project, they are part of the project files.

9.7 Tool to flash the device

To transfer the binary memory image to the flash memory of the microcontroller, one can use the Segger J-Link software.

First, the JLink Windows installer must be downloaded from Segger. Choose Windows 64-bit Installer. Optionally, it is possible to install the Ozone package (again choose the Windows 64-bit Installer), an GUI interface for JLink. It can be found below in the same page.

To install them on Windows, double click on the downloaded files.

After installing JLink, the following application are available in the C:/SEGGER/JLink folder.

```
JFlashSPI_CL    JLinkExe        JLinkGDBServer
JLinkLicenseManager JLinkRegistration   JLinkRemoteServer
JLinkRTTClient  JLinkRTTLogger JLinkSTM32      JLinkSWOViewer
JTAGLoadExe     Ozone
```

Adjust the PATH to include this folder as described in Section 9.2.

9.8 Debugging Tools

The J-Link package installed as above contains the programs needed to create a debug session.

9.9 Installing git

There is a Windows version. When one clicks on Download, it will be redirected to a github repository. In the releases folder, one can find *Git-2.34.1-64.exe* (or a newer version) that is an installer for a 64-bit Windows machine. Download it and double click on it.

9.10 Installing Doxygen

In the Doxygen site one can find a installer for Windows called *doxygen-1.9.2-setup.exe*. Download it and double click on it to start the installer.

10 Using the tools

10.1 Connecting

Connect the board to the PC using the Mini USB cable. When connected a blue LED should lit.

Starting JLinkExe prompt as shown.

```
$JLinkExe SEGGER J-Link Commander V6.22b (Compiled Dec  6 2017 17:02:58)
DLL version V6.22b, compiled Dec    6 2017 17:02:52


Connecting to J-Link via USB...O.K.
Firmware: Energy Micro EFM32 compiled Mar  1 2013 14:08:50
Hardware version: V7.00 S/N: 440112411
License(s): GDB
VTref = 3.329V
```

Configure the interface and type “connect” to establish a target connection, ‘?’ for help

```
J-Link>si swd
J-Link>speed 4000
J-Link>device EFM32GG990F1024
J-Link>connect
```

See the transcript below.

```
$JLinkExe
J-Link>si swd
Selecting SWD as current target interface.
J-Link>speed 4000
Selecting 4000 kHz as target interface speed
J-Link>device EFM32GG990F1024
J-Link>con
Device "EFM32GG990F1024" selected.


Connecting to target via SWD
Found SW-DP with ID 0x2BA01477
Found SW-DP with ID 0x2BA01477
Scanning AP map to find all available APs
AP[1]: Stopped AP scan as end of AP map has been reached
AP[0]: AHB-AP (IDR: 0x24770011)
```

```
Iterating through AP map to find AHB-AP to use
AP[0]: Core found
AP[0]: AHB-AP ROM base: 0xE00FF000
CPUID register: 0x412FC231. Implementer code: 0x41 (ARM)
Found Cortex-M3 r2p1, Little endian.
FPUnit: 6 code (BP) slots and 2 literal slots
CoreSight components:
ROMTbl[0] @ E00FF000 ROMTbl[0][0]: E000E000,
CID: B105E00D, PID: 000BB000 SCS
ROMTbl[0][1]: E0001000, CID: B105E00D, PID: 003BB002 DWT
ROMTbl[0][2]: E0002000, CID: B105E00D, PID: 002BB003 FPB
ROMTbl[0][3]: E0000000, CID: B105E00D, PID: 003BB001 ITM
ROMTbl[0][4]: E0040000, CID: B105900D, PID: 003BB923 TPIU-Lite
ROMTbl[0][5]: E0041000, CID: B105900D, PID: 003BB924 ETM-M3
Cortex-M3 identified.
J-Link>
```

A command line makes all easier.

```
JLinkExe -if swd -device EFM32GG990F1024 -speed 2000
```

Considering that the debugging session will be done with GDB, the only important commands for JLink are related to flashing a program:

- *exit*: quits JLink
- *g*: start the CPU core
- *h*: halts the CPU core
- *r*: resets and halts the target
- *erase*: erase internal flash
- *loadfile*: load data file into target memory

10.2 Flashing

To use the JLinkExe to write the flash contents one has to use the following command:

```
$JLinkExe -Device EFM32GG990F1024 -If SWD -Speed 4000 -CommanderScript Flash.jlink
```

The Flash.jlink file has the following contents:

```
r h loadbin <path>,<base address> r
```

10.3 Debugging

To use GDB as a debugging tool, the GDB Proxy must be started first using the command

```
 JLinkGDBServer -if SWD -device EFM32GG995F1024 -speed 4000
```

In other window to start the gdb the following command is used:

```
$arm-none-eabi-gdb <execfile>
(gdb) monitor reset
(gdb) target remote localhost:2331
(gdb) break main
(gdb) monitor go
```

To debug using GDB see Debugging with GDB. The most used commands are:

- *cont*: continues the execution
- *break*: sets a breakpoint in a function or in a line
- *list*: lists source file
- *next*: steps skipping over functions
- *step*: steps entering functions
- *display*: Displays the contents of a variable at each prompt.
- *print*: Prints the contents of a variable

The monitor commands enable direct access to JLink functionalities, as shown below.

- *monitor reset*: resets the CPU
- *monitor halt*: halts he CPU

The commands can be abbreviated by typing just enough characters. For example, *b main* is the same of *break main*.

11 Sample project

11.1 Structure

A sample project for a EFM32GG microcontroller consists of:

- application source files (e,g, main.c)
- startup file (startup_efm32gg.c), copied from
- CMSIS system initialization file (system_efm32gg.c) copied from
- linker script (efm32gg.ld) copied from .....
- header file (em_device.h) copied from
- Makefile, a build automation tool
- Optionally, README.md, a description of project
- Optionally, Doxyfile, configuration file for doxygen, a documentation generator

11.2 Makefile

The make application can automate a lot of task in the software development. The recipes are specified in a Makefile. The Makefile provides a lot of options, which can be specified in the command line. In the Makefile, it is possible (and sometimes nedded) to adjust compilation parameters and specify where the required folders and applications are. For example, OBJDIR is specified

as gcc and is the folder where all object files and the executable are generated. PROGNAME is the project name

- *make all* generate image file in OBJDIR
- *make flash* write to flash memory in microcontroller
- *make clean* delete all generated files
- *make dis* disassemble output file into OBJDIR)/PROGNAME.S
- *make dump* generate a hexadecimal dump file into OBJDIR/PROGNAME.dump
- *make nm* list symbol table in standard output
- *make size* list size of output file
- *make term* open a terminal for serial communication to board
- *make debug* start an GDB proxy and the GDB debugger
- *make gdbproxy* start the GDB proxy
- *make docs* generates docs using doxygen

12 Create a Debug Session

In the GCC toolchain, the tool for debugging is the GNU Project Debugger (GDB), in the form specific for ARM (arm-none-eabi-gdb). It run on the PC and since it can not communicate directly with the microcontroller, a relay mechanism is needed in the form of a GDB proxy. The GDB proxy can be the Segger JLinkGDBServer or the openocd.

This is a two step process:

1. In a separated window, start the GDB Proxy

   $JLinkGDBServer <parameters>

2. In other window, start GDB and stabilishes a connection

   arm-none-eabi-gdb <executable file> (gdb) target remote localhost:2331

The ARM GNU GDB has a Command Line Interface (CLI) as default and a Text User Interface (TUI), which can be activated with a -ui parameter. There are many Graphic User Interface (GUI) applications for GDB including ddd and nemiver.

Annex A - Determining stack usage

First, use the -fstack-usage option for compiling. It makes the compiler generate a .su file for each .c file. Each line of it, contains the function name and the size of stack.

```
led.c:21:6:LED_Init         16  static
led.c:42:6:LED_On           16  static
led.c:51:6:LED_Off          16  static
led.c:60:6:LED_Toggle       16  static
```

```
led.c:69:6:LED_Write        16  static
```

This is not enough, because on must know who call who (callgraph) to determine the stack usage.

To get the call graph, there are many options to generate the needed information:

- Use -fdump-rtl-dfinish option * Use -fdump-ipa-cgraph * Use cflow to generate the call graph

And the use a tool, to combine the information of callgraph and stack usage.

An important note is that functions called thru function pointers or interrupts are not considered.

A list of these tools:

- Worst case stack
- Stack Usage
- Avstack
- Cflow.py

Worst case stack

Needs the fdump-rtl-dfinish option during compilation.

It needs all input files (with suffixes .o, .su, and .c) in the same directory. So copy all .c files to gcc folder and then run

```
python3 wcs.py
```

Additional information can be added thru .msu files.

Stack Usage

Needs the -fdump-ipa-cgraph option during compilation.

First all files (.su and .cgraph) must be combined.

```
find . -name "*.cgraph" | grep -v stack-usage-log | xargs cat > stack-usage-log.cgraph
find . -name "*.su" | grep -v stack-usage-log | xargs cat > stack-usage-log.su
```

And then generates the report using

```
python3 stack-usage.py --csv stack-usage.csv --json stack-usage.json
```

Avstack

It uses a Perl script to read .su files and disassemble the object .o files.

cflow

It is possible to generate the callgraph using cflow

```
(cflow -l  -b --omit-arguments -D$(PART) $(addprefix -I
,${INCLUDEPATH}) $(SRCFILES) 2>&1) | egrep -v "^cflow"
```

And the use the cflow.py to generate the report.

References