



**Budapest University of Technology and Economics
Department of Measurement and Information Systems**

Operating Systems (VIMIAB00)

Embedded operating systems laboratory

Student's guide

**Version: 0.99
March 18, 2016**

NASZÁLY Gábor
naszaly@mit.bme.hu

Contents

| | | |
|--------|--|----|
| 1. | Preface..... | 3 |
| 2. | Introduction..... | 5 |
| 2.1. | Embedded systems in a nutshell..... | 5 |
| 2.2. | Microcontrollers..... | 6 |
| 2.3. | Developing in the C language (as usual)..... | 7 |
| 2.4. | Developing in the C language (for microcontrollers)..... | 8 |
| 2.4.1. | Differences in the development process..... | 8 |
| 2.4.2. | Startup code..... | 9 |
| 2.4.3. | Using standard I/O..... | 10 |
| 2.4.4. | Accessing I/O registers..... | 10 |
| 2.4.5. | Interrupts..... | 10 |
| 3. | Embedded real-time operating systems..... | 12 |
| 3.1. | The embedded RTOS as a software architecture..... | 12 |
| 3.2. | Task states..... | 13 |
| 3.3. | Task structures..... | 14 |
| 3.4. | Context switching..... | 14 |
| 3.5. | Time management..... | 14 |
| 3.6. | Shared resources..... | 15 |
| 3.7. | Priority inversion..... | 17 |
| 3.8. | Desktop OS vs. RTOS..... | 19 |
| 4. | FreeRTOS basics..... | 20 |
| 5. | EFM32 Giant Gecko Starter Kit..... | 21 |
| 5.1. | Properties of the development board..... | 21 |
| 5.2. | Connecting to a computer..... | 22 |
| 5.2.1. | Checking devices..... | 22 |
| 5.2.2. | Connecting to the virtual serial port..... | 24 |
| 6. | Simplicity Studio 4..... | 27 |
| 6.1. | Launcher window..... | 27 |
| 6.2. | Simplicity IDE..... | 28 |
| 6.3. | Debug perspective..... | 29 |
| 6.4. | Energy Profiler..... | 32 |
| 7. | Description of the project..... | 34 |
| 7.1. | Downloading the project..... | 34 |
| 7.2. | Importing the project..... | 34 |
| 7.3. | The structure of the project..... | 37 |
| 7.3.1. | The [src] folder..... | 37 |
| 7.3.2. | The [inc] folder..... | 38 |
| 7.3.3. | The [FreeRTOS] folder..... | 39 |
| 7.3.4. | The [Drivers] folder..... | 41 |
| 7.3.5. | The [BSP] folder..... | 41 |
| 7.3.6. | The [emlib] folder..... | 42 |
| 7.3.7. | The [CMSIS] folder..... | 42 |
| 8. | Test questions..... | 43 |

1. Preface

The topic of these laboratories are the embedded operating systems. Among them especially those operating systems that run on classic embedded systems. On classic embedded systems we mean those electronic devices that have been designed for a very special purpose. Their computing power is generally way behind the ordinary computers. In the other hand they face real-time requirements (while ordinary computers usually do not have to deal with such requirements). For this reason the literature calls the operating systems of the classic embedded systems as real-time operating systems (RTOS).

One can hear nowadays a lot about operating systems like embedded Linux (or Android or Windows CE). These operating systems run on devices whose computing power and application area are much closer to ordinary computers than to classic embedded systems. These kind of embedded operating systems are not investigated during the laboratories. We are concentrating on RTOSes mentioned above.

During the laboratories we are about to use the Silicon Laboratories¹ EFM32 Giant Gecko Starter Kit (EFM32GG-STK3700A²) (more specifically the development board (BRD2200A³) contained in it). This board has been built around a contemporary 32 bit microcontroller (EFM32GG990F1024⁴).



Figure 1. EFM32 Giant Gecko Starter Kit development board

During the development we will use a collection of tools called Simplicity Studio⁵. This has also been developed by Silicon Labs. It consist of a lot of components. Among them the Eclipse IDE based Simplicity IDE can be considered as the central element.



Figure 2. Logo of Simplicity Studio

The operating system chosen is the FreeRTOS⁶. Loyal to its name it is a free real-time operating system for embedded systems developed by Real Time Engineers Ltd. Its source code is also open. The

¹ Or just Silicon Labs or Silabs: <http://www.silabs.com/>

² <https://www.silabs.com/products/mcu/lowpower/Pages/efm32gg-stk3700.aspx>

³ https://www.silabs.com/Support%20Documents/TechnicalDocs/BRD2200A_A03.pdf

⁴ <http://www.silabs.com/products/mcu/32-bit/efm32-giant-gecko/pages/EFM32GG990F1024-BGA112.aspx>

⁵ <http://www.silabs.com/products/mcu/Pages/simplicity-studio.aspx>

⁶ <http://www.freertos.org/>

operating system is licensed under a modified version of the GNU General Public License. The modification taking the form of an exception. The exception permits the source code of applications that use FreeRTOS and are distributed as executables to remain closed source, thus permitting the use of FreeRTOS in commercial applications without necessitating that the whole application be open sourced.



Figure 3. Logo of FreeRTOS

2. Introduction

2.1. *Embedded systems in a nutshell*

Classic embedded systems are electronic devices designed for very special purposes. They are usually connected to the environment tightly by constantly monitoring the environment by their sensors and also making effect to the environment by their actuators. They have usually some kind of a user interface also. And they usually communicate with other devices too.

We can spot them in a vast variety of application area: automotive electronics, entertainment systems, medical equipment, measurement devices, communication equipment or even space engineering.

To fulfill their purpose they usually have some kind of a central processing element. This can be a circuit which is capable only for a very special task but for that it is very effective. Such a circuit can be realized in devices like FPGAs⁷ whose inner electrical configuration can be dynamically changed. This solution is used often during the development of a desired integrated circuit. After the design seems to be stable it is usually worth to realize the circuit in an ordinary IC (whose inner configuration is established during the manufacturing process). This latter kind of circuits are called application specific integrated circuits (or just ASICs for shorthand).

However it is sometimes more suitable to use some kind of a processor as the central processing element. This processor could be a microcontroller, a digital signal processor (DSP) or rarely a general purpose processor.

If we are about to use some kind of a processor, then we will eventually need some software (firmware) too. If the complexity of the task to be solved is under a certain level then this software could be realized using simple software architectures. For example it can start with an initializing part followed by an endless loop. If responsivity is of the essence this architecture can be amended by using interrupts. However this solution could become very complicated and hard to maintain over a certain level of complexity. For this reason it is necessary to decompose the complex task into smaller parts. These parts have to be able to communicate with each other and they also need to be able to synchronize their execution to each other. The operating system – as a software architecture – can provide a solution for the above mentioned needs.

For a simple embedded system using an ordinary desktop OS would be too much. What we need is basically a scheduler and the provision of communication and synchronization between threads. Furthermore as these classic embedded systems often face real-time requirements (e.g. reactions to certain events have to be accomplished within a given deadline), we name the operating systems for them RTOS.

Instead of the somewhat complex definition for embedded systems above one can say that every electronic device except ordinary computers could be considered as embedded systems. This is a more permissive definition. This can cover devices what cannot be covered with the classic definition: like smartphones. And indeed, we think the operating systems running on these devices (like Android, embedded Linux or Windows CE) as embedded systems. However these devices are closer to the ordinary computers according to their application areas and properties. During the laboratories we consider an RTOS under embedded operating system.

⁷ Field-Programmable Gate Array

2.2. Microcontrollers

As we are about to use an embedded system which utilizes a microcontroller as its central processing element, it is worth to investigate microcontrollers a little bit.

Microcontrollers are much like ordinary processors. However with some important differences. Maybe the most important difference is that they incorporate a vast number of various peripheral devices besides a central processing core. First of all there are code (or program) and data memories integrated into them. Code memories are usually based on some non-volatile technology (like flash). While data memories are usually based on volatile technology (like SRAM). This means their contents are lost after the supply power is turned off. In the other hand these memories are much faster (and unfortunately also much expensive) than non-volatile memories. Because of the difference in price data memories are usually an order smaller in capacity compared to code memories for a given microcontroller.

In nearly all of the case we can find various timer and counter units also in microcontrollers. They are used to schedule the execution of certain parts of the software, to keep track of time or to count some events.

To maintain some connectivity with the environment we can very often find so called GPIO (General Purpose Input/Output) peripherals in microcontrollers. With the help of these circuits it is possible to use certain pins as digital inputs or outputs. Probably the most frequent use cases for GPIO are driving LEDs and getting values generated by push buttons. But because their general nature practically any kind of more complex functionality can be accomplished using GPIO.

To maintain some connectivity with analog signals one can find analog comparators, analog to digital and digital to analog converters (ADCs and DACs).

It is also important to communicate with other integrated circuits. For this reason a lot of communication peripherals for the various communication protocols can be found in microcontroller like U(S)ART, I²C, SPI, USB, Ethernet, CAN, LIN and FlexRay to name the most frequent ones.

We can communicate with the peripherals integrated into the microcontroller (read or write data, configure their operation or read their status) through so called I/O registers.

Another difference compared to ordinary processors is the much lower computing performance, pin number, clock frequency, memory capacity, thermal dissipation, physical dimensions...

Talking about microcontroller we often say that a microcontroller is an 8 bit, a 16 bit or a 32 bit microcontroller. This bit value tells us the size of data on which the microcontroller can execute its instructions.

There is another property: is the given device utilizing a so called Harvard or Neumann architecture? A Harvard architecture means that the code and data memories are separated (usually this is the case with microcontrollers). In contrast a Neumann architecture means that the data and code memories are not separated (this is usually the case with ordinary processors).

Processors can be described also by the complexity of their instruction set. We can speak of so called CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer) devices. The sooner means that the instruction set contains a large number of instructions and these instructions are complex. Whereas the latter corresponds to an instruction set with small number of simple instructions. Microcontrollers are usually RISC while ordinary processors are usually CISC. A RISC architecture often means also that the given device is unable to execute instructions on data in memory. At first data has to be loaded into the processor's inner registers. Any manipulation can happen only on these registers. When it is done the result has to be stored back to the memory. Hence this operation is called as load and store.

A vast number of semiconductor manufacturers makes microcontrollers (like Atmel, Freescale, Infineon, Intel, MIPS, Microchip Technology, NXP Semiconductors, Renesas Electronics, Silicon

Laboratories, STMicroelectronics, Texas Instruments and others). It is important to mention one additional player on the market: ARM Limited. ARM designs processor cores but they does not have any manufacturing capacity. The processor cores designed by ARM are licensed instead as intellectual properties (IPs) to other members on the market. These other companies then design some other circuitry (like the peripherals) around the processor core and manufacture the final device. Before ARM all manufacturers designed their own cores. After the appearance of ARM on the scene these cores haven't disappeared but the majority of the manufacturers has now also ARM based devices in their product range.

2.3. *Developing in the C language (as usual)*

In an ordinary (not embedded) case developing in the C language is something like described below.⁸ At the beginning we need source files (with “.c” extensions) and header files (with “.h” extensions). Sometimes certain part of the code is written in assembly. In this case we have also source files with “.s” extensions. It is the duty of a compiler to compile the C language code to machine instructions (also called as object code). The result is stored in object files (with “.o” extensions). The assembler accomplishes a similar task: it makes object code from the assembly language sources. (More precisely the C compiler is first compiles the C language code into an assembly language source which is translated to machine instructions by the assembler.) It is possible that some components are already present in object code form. In this case these object codes are usually packed together into library archive files (with “.a” extensions).

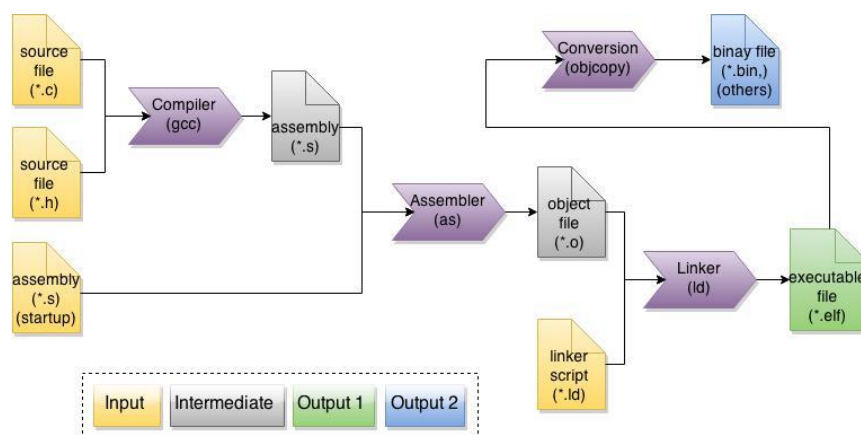


Figure 4. The process of developing in the C language

After all the needed object files and library archives are present, we need to put them together. Simply copying them together is not enough. This is because in the object files (and archives) one can refer to a variable (or call a function) defined in another object file (or archive). For this reason in these cases the object code only refers to the variable or function in question. When we put all object code together we have to resolve those references. This process is called linking and the component doing this task is called the linker. If all references can be resolved the linker produces an executable file (for example in the ELF format having “.elf” as the extension⁹).

The tools mentioned so far (compiler, assembler, linker) are called together as a tool-chain because the output of one tool is the input to another. To manage the whole process (e.g. to invoke the right

⁸ This description assumes GNU based development environment. However most probably other development environment are very similar.

⁹ Executable and Linking Format (formerly Extensible Linking Format)

tool with the right parameters in the right time) we can use the GNU make. This tool gets the needed information from so called makefiles. They tell make what files depends on what other files and also tells make how to produce a file from its dependencies. If a file is modified after a make process make only generates files which are depending directly or indirectly on the modified file (instead of dumbly produce all files all the time).

After the link phase all the references have been resolved. However the memory addresses in the executable file are relative addresses only. To execute the file (load it into the memory, make absolute¹⁰ addresses from relative addresses and give the execution to the entry point of the application) we need the so called loader component of the operating system.

Instead of the loader we need a debugger if we want to observe the behavior of the application, want to search for possible bugs. With the help of the debugger we can pause (break) the execution of the application at any time and then we can continue from that point. It is also possible to reset the application and start its execution again from the entry point. The application can be paused also when certain conditions are met by using breakpoints. In most of the cases we use breakpoints to stop the application at a given instruction. With a more sophisticated debugger it is also possible to stop the application when other conditions are met (for example access to a given variable). It is also possible by using a debugger to execute the C language instructions of the application one-by-one (stepping). If the next instruction is a function call, sometimes we want to follow that call (step into), sometimes we want to treat the function call as one instruction (step over). Sometimes we are in the middle of a function but want to execute it to its completion (step out). Sometimes it is also possible to step through assembly instructions and not on C language instructions (instruction stepping). But with a debugger not just the execution of the application can be altered. We can also observe the content of the memories, the values of the variables. To make debugging efficient it is needed that some debug symbol are stored in the executable file. The ELF format can store such debug symbols.

2.4. Developing in the C language (for microcontrollers)

2.4.1. Differences in the development process

Developing for microcontrollers has a lots of common with developing to ordinary processors. However there are also some differences. An embedded application¹¹ often does not contain any operation system at all. In this case we simply cannot say that the loader component of the OS loads the application into memory as there is no loader, no OS. But even if there is an RTOS used within the application it is too simple (basically a scheduler) to have such a component as a loader.

The above means that we need to find some other way to load the application. This can be done by a so called programmer component which loads the application to the code memory. (Because the memory for code is usually of a non-volatile type also the initial values of the constants and initialized global variables are copied to the code memory. To the data memory we do not load anything. We only need to reserve space in data memory for the used constants and variables.

The programmer itself does not know what part of the application should be loaded to what part of the memories. This relocation information is stored in the executable file. But what is the source of this information? If we want to separate the functions strictly we can say that the so called locator is

¹⁰ This is a simplification. An ordinary computer uses virtual memory nowadays. This means that all processes see a virtual address space which is unique for the given process. To actually make phisycal memory addresses out of the virtual memory addresses they have to be translated. This translation is done by the MMU (Memory Management Unit) during runtime.

¹¹ In the world of embedded systems we mean the whole firmware (with any RTOS if present) as the "application".

the component what puts the relocation information to the executable file. In reality it is rarely present as a separate tool. It is part of the linker. The information comes from so called linker script files. These script files are maintained by the developer as the relocation information can depend on the application.

It is worth to mention that the generated executable file is not suitable to be downloaded to the device memories. It has to be converted to an applicable file format. Such a format could be the Intel HEX format. It is used for programming microcontrollers long ago (~1970). The conversion between the executable and downloadable formats is done by `objcopy`.

In general the debugger is a pure software tool. In the embedded case however there is a need to some hardware component also to be able to connect to the target device. This way the debugger function is put together by hardware and software components. In the past the hardware components of the debuggers were a separate device. Nowadays many development board contains the debugger integrated to the panel. However debuggers with more complex functionalities tend to be separate devices even nowadays.

In general coding, compiling, linking, loading, debugging and running the application happens on the same platform. With embedded systems the development process usually requires a computer (because embedded systems usually do not have big enough displays and keyboards for development, and usually there is no development environment what could be run on them...). However running and debugging happens on the target system. A compiler running on platform "A" but compiling code for platform "B" we call cross-compiler.

2.4.2. Startup code

We used to the fact that the entry point of a C application is the `main()` function. This is what we expect also when developing to an embedded system. In general the operation system is responsible during the loading phase to call `main()`. However in the case of embedded systems what we know is only the fact that usually the processor starts executing the instructions in the beginning of the code memory.

We call that part of the application as the startup code what gets executed after the power is turned on and ends with calling `main()`. The very beginning of the startup code contains the so called interrupt vector table (see 2.4.5 for more on interrupts). After the vector table the startup code starts to execute its duties. Among them the most trivial is that at the end it calls `main()`.

To be able to call functions we need a stack. The stack is usually allocated within the data memory. To the current top of the stack is pointed by a special register of the processors: the stack pointer (SP). The stack pointer has to be initialized to a right value before the first function call. This value depends on what part of the memory is to be used as the stack and to which direction the stack pointer grows (toward lower or higher memory addresses).

We also expect from a C environment to initialize those global variables that do not have a starting value with zeros. This is accomplished by a little loop within the startup code.

We have a somewhat similar expectation regarding to those global variables that have a starting value (and also to the constants). Certainly with the difference that we expect them to be initialized with the given starting values. Furthermore we do not want the embedded system to forget these starting values after a power cycle. For this we need to store these starting values in a non-volatile memory. However as non-volatile memories are usually complicated to write from the application (like flash) or sometimes even impossible (in case of a one-time programmable ROM) and also because they are relative slow we do not want to access variables in them. It would be nice to access them in a RAM. To satisfy these two contradictory requirements we store these starting values in non-volatile memory but use them in a RAM. For this to work the starting values have to be copied from the non-volatile memory to the RAM. This is also accomplished by a little loop within the startup code.

After the above mentioned duties are done the startup code is basically ready to call `main()`. However there is still a question what should be happen after `main()` returns. In an ordinary case after `main()` returns the execution is given back to the OS. However in the embedded case we do not have a fancy operating system to return to. `Main()` simply returns to the location in the startup code from where it has been called. Normally the microcontroller would start to execute all the instructions after `main()` is called. However usually only that part of the code memory is filled with meaningful instructions that belong to our application. We do not know what values are stored in the code memory after our application. We should treat those values as memory garbage. It would not be wise to let the microcontroller to execute this garbage. For this reason there is usually an endless loop after `main()`. A somewhat more sophisticated solution is to put the microcontroller into some sleep state. Usually before these we disable all interrupts. This is needed because interrupts per definition are able to grab execution even from an endless loop and usually also wakes up the device.

2.4.3. Using standard I/O

Within a C language application we usually want to use the functions provided by the standard I/O library (like `putchar()`, `getchar()`, `printf()`, `scanf()`). In an ordinary case it is again the duty of the OS to define the standard input and output (usually to a console or to a file).

However in the case of a classic embedded system we either do not have an OS at all, or the OS is an RTOS which is a very lightweight operating system (basically a scheduler) and does not provide such functions as defining the C standard I/O streams. Furthermore in the ordinary case a console or a file is usually present. However in the embedded case it is not so straightforward what peripheral could be used as the standard input or output. And even if there is a suitable peripheral it is not guaranteed to be present on another embedded system. For these reasons the standard C streams (`stdin`, `stdout` and `stderr`) are not initialized on embedded systems. It is the duty of the developer to initialize them properly.

Initializing these streams basically means telling the C standard I/O library what functions to use when it wants to send or read a character to or from the chosen peripheral.

So we need to choose first a peripheral what can be used as a standard input, output or both. After that we also need to implement those functions that can write to and read from that peripheral.

Maybe the most frequently used peripheral in embedded systems for standard I/O purposes the UART (serial line). This can be connected to a PC. On the PC we need to run a so called terminal emulator. This application basically displays the characters received from the communication port, and send the characters typed by the user. To an output only stream LCDs also used frequently.

2.4.4. Accessing I/O registers

Inside a microcontroller we can find a lot of peripherals besides the processor core. We can communicate with them (read or write data, configure their operation or read their status) through so called I/O registers. During the laboratories we won't access I/O registers directly. We will use the library functions instead. However it is worth to at least mention them.

I/O registers are specific to a given microcontroller. However the C language tries to be platform independent. For this reason basically there is no standard way to access I/O registers. To make things work, microcontrollers usually maps their I/O registers to a dedicated part of the data memory address space. This way with a properly constructed pointer we can access the I/O registers.

2.4.5. Interrupts

During the laboratories we won't use interrupts either. However for the sake of completeness it is worth to mention them in a few words.

If an interrupt request is received by the processor (and the given interrupt is enabled) the execution of the current instruction will be finished but after that the processor starts to service the interrupt. The process is highly architecture dependent but usually results in visiting the right place in the interrupt vector table (which is usually situated in the first part of the code memory). On some architectures the table contains a jump instructions for the interrupts pointing to an interrupt service routine (ISR) which is expected to service the interrupt. On some architectures only the address is present in the table (the processor knows that it is an address and not an instruction and jumps to the given address). The above mentioned is not contrary to the fact that usually after reset the execution is starting at the beginning of the code memory. The reset can be considered as a special interrupt to which the very first entry in the vector table belongs.

If the vector table is filled with right values, the execution is then goes to the ISR belonging to the interrupt. It is architecture dependent but in most of the cases these routines cannot be implemented as ordinary C functions. There are microcontrollers what have separate registers for ordinary code and for interrupts. However simpler devices have only one instance from a given register. Because both the ordinary code and the interrupt service routines can use the same registers the ISRs should take care to preserve the value of any register that is modified during the execution of the ISR (e.g. save them upon entering the ISR and restore them upon exiting the ISR). Another special requirement can be towards an ISR to re-enable interrupts before exiting the ISR. This could be needed for those architectures that automatically disable interrupts upon executing the ISRs but do not re-enable them automatically after returning from the ISR.

Each architecture handles interrupts a little bit different way. This is not a problem when we are developing in assembly language. However it will be a problem with C development. The C language (as always) tries to be platform independent. For this reason there is no standard defined way to handle interrupts in C. For this reason per architecture and per C compiler there are different special extensions to the C language. Such language extensions can be the inline assembly, or some special keywords to instruct the compiler to make the right code for the interrupt service routines.

3. Embedded real-time operating systems

3.1. The embedded RTOS as a software architecture

The following code example shows the structure of the embedded OS as software architecture:

```
void interrupt Device1 (void) {
    !! Handle Device 1 time critical part
    !! Set signal to Device1_task
}

void interrupt Device2 (void) {
    !! Handle Device 2 time critical part
    !! Set signal to Device2_task
}

...

void Device1_task (void) {
    !! Wait for signal to Device1_task
    !! Handle Device 1
}

void Device2_task (void) {
    !! Wait for signal to Device2_task
    !! Handle Device 2
}

...

void main (void) {
    !! Initialize OS
    !! Start OS scheduler
}
```

ISRs do the handling of time critical parts of the devices. This is because even the lowest priority IT is able to interrupt the highest priority task¹². In turn we should place only that part of code to an ISR that can't be done in a task!

Probably one of the main advantages of an embedded OS is the possibility to give tasks priorities. The more important a task is, the less response time is needed for it. Further advantage is that the embedded OSes provide services for shared resource handling, inter-task communication and synchronization.

However the disadvantage is the need of greater computing time and memory.

On the following figure you can see an example time diagram for executing tasks and ISRs:

¹² the terms of *task*, *process* and *thread* is often confusing even in the terminology. This guide interprets these as follows:

- *Task*: is a functional unit to do something,
- *Process*: is an executorial unit. The processes are protected against each other, they can't see each others memory. Thus the inter-process communication is relatively complicated and context switching needs a great overhead,
- *Thread*: is also an executorial unit. Sometimes it is called „lightweight process“. It refers to the fact that there is much less protection between threads as between processes. Threads can see each others memory, so they can communicate easily and context switching doesn't mean such a great overhead.

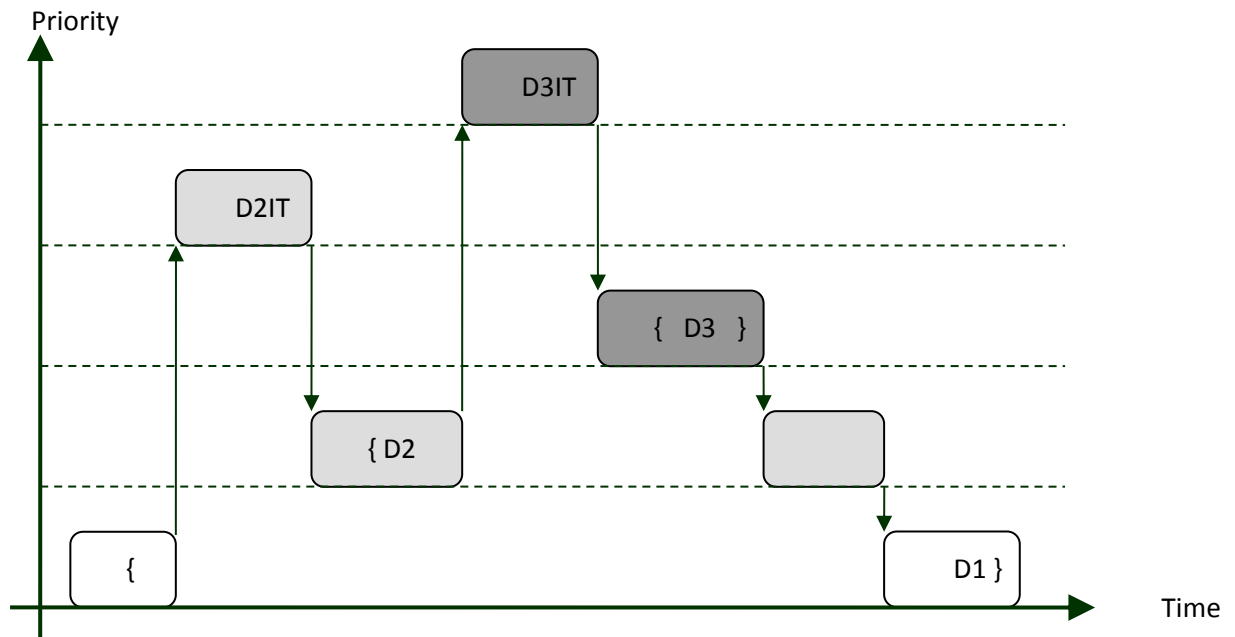


Figure 5. An example running diagram for tasks and interrupts

3.2. Task states

In general a task has three main states: running, ready to run and waiting.

A task is in the “waiting” state if it is waiting for an event (e.g. for the pass of a given time, for a semaphore etc.). As soon as the awaited event occurs the task becomes ready. It will become running if it has the greatest priority among the ready tasks. At a given time only one task can be running (in general: there can be as many running task as the amount of processors on the system). If exists the “run” → “ready to run” transition we are talking about a preemptive OS (otherwise we call the OS non-preemptive). If an OS allows more than one task on a given priority level, the scheduling is time-slicing and round-robin among these tasks.

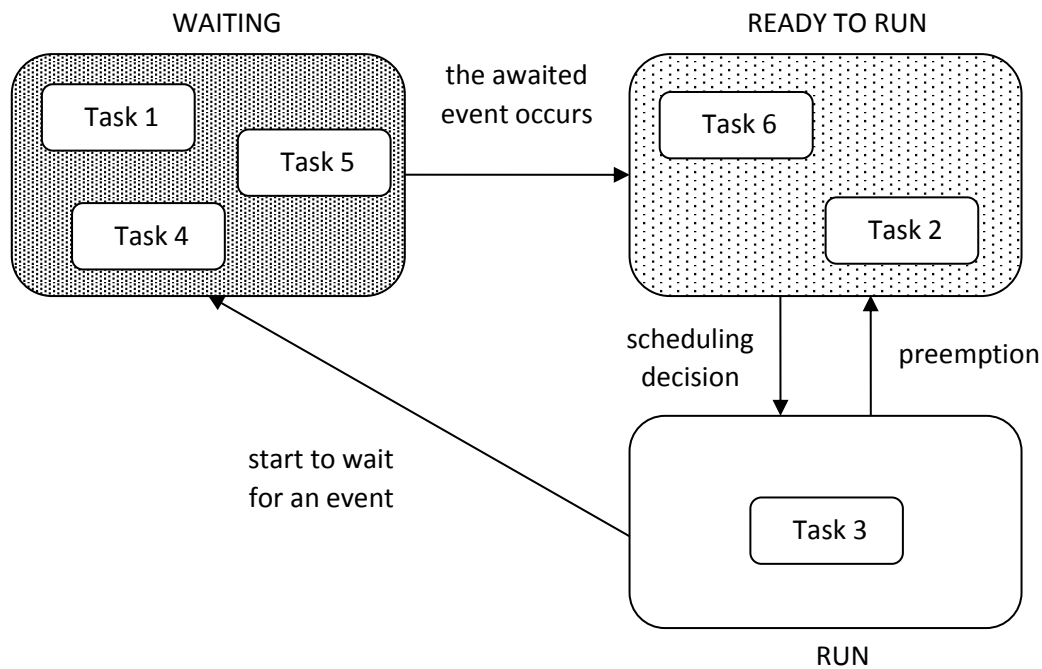


Figure 6. The three basic task states

3.3. Task structures

The ordinary task structure is mainly an infinite loop. It can be optionally preceded by an initializing block of code. To give the other tasks the chance to run, we have to place such OS calls into the infinite loop what send our task to the waiting state.

There is another structure called single-shot. In this case there isn't any infinite loop. The task runs until its completion (and in the end it have to delete itself). During its running it may cause events what make other tasks ready to run. Thus we get a chain-like running of tasks.

3.4. Context switching

In the case of multitasking many tasks run virtually at the same time (in reality at a given time the execution is only on one task, but the switching is done so frequently that it seems all the tasks are running simultaneously. How can it be that our tasks can operate without disturbing each other? The answer is that every task has its own "context": the value of processor registers, the stack of the given task... Certainly these exist in the case of non-multitasking too. But in this case there is only one of these. If we have multiple tasks we have to provide each task with its own context. Furthermore this context has to be saved if the OS wants to schedule another task, and it has to be restored if the task get back the right of execution. The saving of one task's context and then the restoring of an others called "context-switching".

3.5. Time management

Time management is needed if we want to suspend the execution of tasks for a given period of time, if we want to give a timeout value for waiting OS calls (like trying to reserve a semaphore), if we want timed function calls (once or periodically) etc.

For time management purposes the OS configures a hardware timer (we call it the *heartbeat timer*). This timer produces interrupts periodically (*system ticks*). But how long should these ticks be? In one

hand the shorter the period is the more accurate is the tracking of time. On the other hand the greater the period is the less overhead is caused by the timer IT. The typical value for the frequency of system ticks is 10 Hz – 1000 Hz (resulting in 1ms – 100ms period).

The accuracy of time management is one tick, as the figure demonstrates:

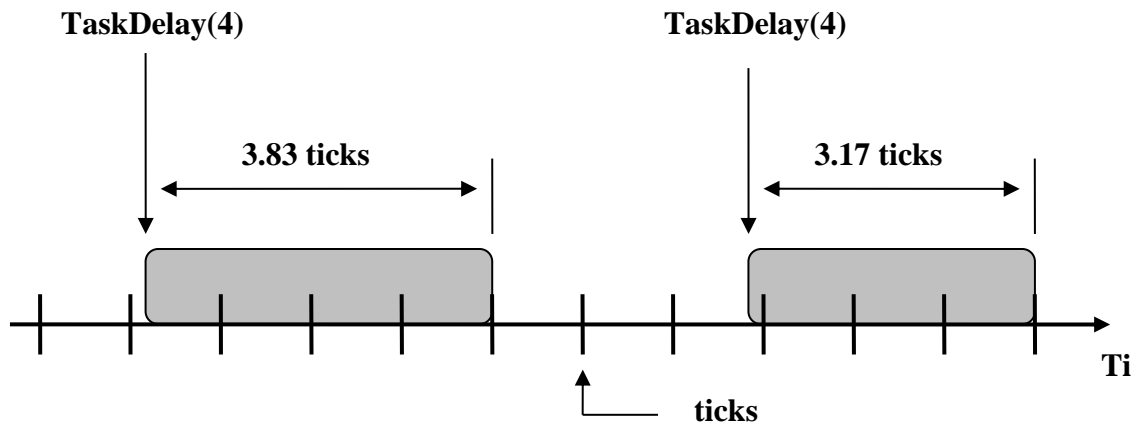


Figure 7. The accuracy of timing services

A further conclusion can be made: if we want to wait **at least n** system ticks, we have to give **n+1** for the parameter to waiting OS calls!

3.6. Shared resources

In the case of concurrent programming rises the problem of “shared resources”. From this point of view not only multitasking counts as concurrent programming. If we got only one task but has ISRs too, and both of our task and ISRs access a shared resource there could be also problems.

The problem is the following: if our execution units (tasks, ISRs) don’t handle shared resources in an atomic way, the resource can become inconsistent as our execution units interrupts each other.

Consider the following example: an ISR updates a 16 bit value repeatedly. And there is a task what reads out this value and does something depending on the value. But if our code runs on an 8 bit platform, reading a 16 bit value cannot be done by a single assembly instruction. Let’s suppose that the interrupt occurs in the middle of reading. In this case one part of the read value belongs to the old state and the other belongs to the new. If we are using a higher level programming language (like C), we even can’t see that a reading statement compiles to many assembly instructions.

Another example is when we want to write out text (for example to the serial line). If more than one task want to print messages and don’t handle the serial port in an atomic way the characters belonging to separate messages could be mixed.

The problem of shared resources has a very bad nature. Even if there is the chance for this problem to occur, it happens very rare. And when it happens, it produces very strange errors.

The part of the code that accesses the **shared resource** is called **critical session**. We have to handle the shared resource between the critical sessions in an **atomic** way. In other words we have to achieve **mutual exclusion** for the shared resource. For this purpose there are many options:

- disabling / enabling interrupts
- disabling / enabling scheduler
- using a *lock-bit*
- using semaphores

The simplest and easiest method is to disable and enable ITs. (Furthermore if one of the execution units is an ISR this is the only working option). If we choose this method we have to disable interrupts the shortest needed time. This is because the purpose of an ISR is to react as soon as possible to something. An important parameter of every OSes is the worst-case time needed to response to an IT.

If the problem rises only among tasks we can disable and enable the scheduler. Although this solution works use it rare because disabling the scheduler negates the advantage of using an OS.

There is a further simple opportunity to solve the problem: using a so called lock-bit. For example 1 represents that our resource is free and 0 means it is already used. Then we have to test at the beginning of the critical session whether our resource is free or not. If it is used we wait until it becomes free. If it is free (or has become free after we waited for it) we set the bit to zero (indicating that the resource is used). This procedure is called “test-and-set”. At the end of our critical session we set the bit to 1, indicating that the resource is now free. We shall notice that the test-and-set operation has to be uninterrupted! (Otherwise the following can occur: task A tests the bit and sees that it is 0. Then the OS switches to task B. It also test for the bit and sees that the resource is free. Then it set it to 1 and does something after that. At a given point the OS gives back execution to task A. Task A was interrupted between “test” and “set”. So it sets the bit (already holding value 1) to 1. Then does something with the shared resource. Unfortunately our resource is in an inconsistent state by now because task B already used it but hasn’t finished its critical session.) An atomic test-and-set assembly instruction (TAS) exists on some platform. If this is not the case, we have to disable ITs before “test” and re-enable them after “set”.

There is a more sophisticated solution called semaphore. A semaphore is basically a lock-bit. The difference is that it is an OS service. This means if a task have to wait a semaphore the OS put it to a waiting state. (Meanwhile if a task wait for a lock-bit to become free, it doesn’t go to waiting state rather it is running and endlessly checks whether the lock-bit is free or not.) Semaphores were invented originally for trains. For programming purposes it was invented by the Dutch Edgser Dijkstra in the mid 60’s. There are two basic operations on it: P() and V()¹³. The first is basically the TAS operation (it is also called sometimes Take(), Wait() or Pend()). The second is the releasing of the semaphore (sometimes called Give(), Signal() or Post()). The P() operation decreases the value of the semaphore by 1 (if it is greater than 0). If the semaphore is 0, P() waits until it is freed up. The operation V() increases the value by 1. If the maximal value of the semaphore is 1, then we are talking about a **binary semaphore**. When the maximal value is greater than 1 we call it a **counting semaphore**. We have to be careful with semaphores. They are great tools against the problem of shared resources, but we have to avoid misusing them:

- we forget to wait / release a semaphore
- we wait or release a wrong semaphore
- a *deadlock* (called also *deadly embrace*) can occur (task A waits for semaphore 1 what is used by task B, and task B is waiting for semaphore 2 what is used by task A)
- we block a semaphore too long (thus we let other tasks waiting to that semaphore starve)
- *priority inversion*

Against the first two mistakes we can offend ourselves with proper coding. Another solution is to make dedicated functions what handle a resource. The semaphore (what protects this resource) is handled only by these functions. So if we call these functions to access the resource, the protecting semaphore is automatically used and handled in the proper way).

Against deadlock an easy solution is to reserve all the needed semaphores at once. Another way is to reserve all the semaphores in a predefined order. (This can be expressed in general: give each

¹³ P: passeren [Dutch], to pass. V: vrygeven [Dutch], to release.

resource a number. And the tasks should ask only for those resources whose number is greater than the task already owns.)

Against starving proper coding can give a solution.

3.7. Priority inversion

One of the unwanted side effects of using semaphores is called “priority inversion”. You can see it on figure 8. Consider the following: there is three tasks with low, medium and high priorities and one semaphore. Let’s suppose the lowest priority task will run first. After beginning of its execution it reserves the semaphore. Then the high priority task becomes ready to run. Because we have a preemptive scheduler the high priority task will get the right to run. At a given time task high also wants to reserve the semaphore. Unfortunately it is already reserved by task low. Thus the task with high priority becomes waiting. Then the OS schedules task low again (as it is the only ready to run task). After a given amount of time the task with medium priority becomes ready. This means that the kernel schedules it. In our case this task executes until its completion. After task med finished task low continues executing. At a given point task low doesn’t need the semaphore anymore and releases it. By freeing up the semaphore task high becomes ready to run. And because it has higher priority than task low the kernel does a context switch to task high. At a given point task high releases the semaphore. (Because there isn’t any task waiting for the semaphore with a greater priority than task high no task switch occurs.) So task high continues execution until it finishes its job. After task high finishes task low will run and after a time it will finish too.

Note that task med behaves completely (and task low partly) as it had a higher priority than task high! And if there were more tasks on intermediate priority levels in the worst-case scenario every intermediate task are able to defer the execution of task high! This is why the name “priority inversion”.

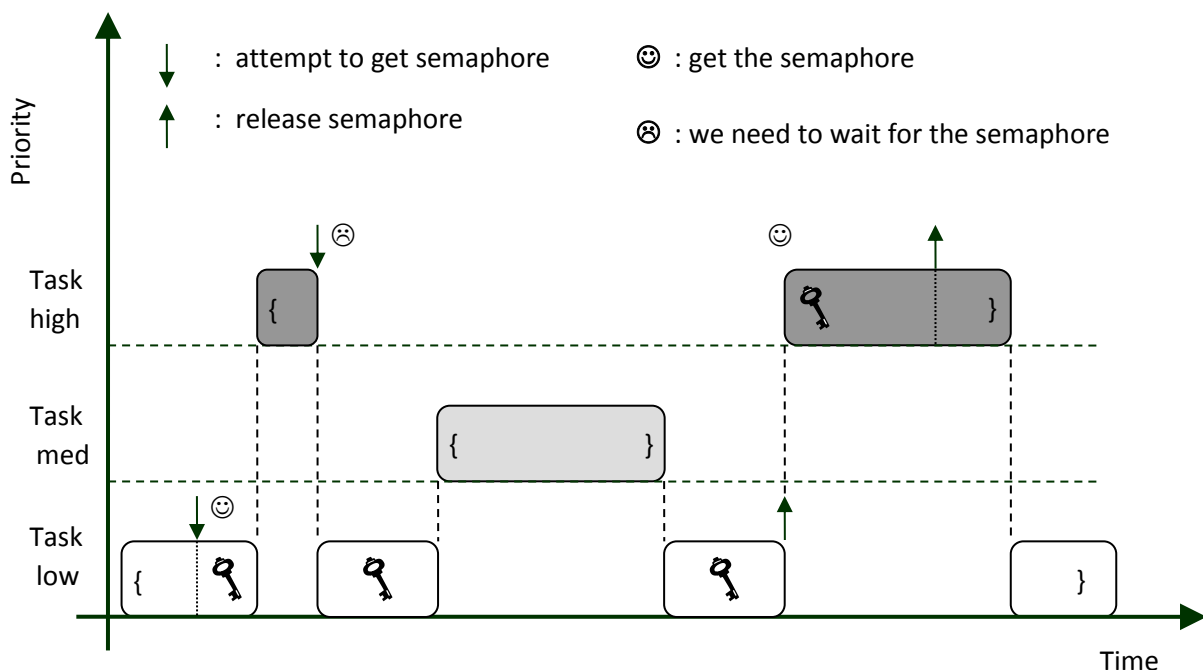


Figure 8. An example diagram demonstrating priority inversion

There are solutions (so called protocols to solve this problem):

- *priority ceiling protocol – PCP*
- *immediate priority ceiling protocol – IPCP*
- *priority inheritance protocol – PIP*

Here we discuss the PIP solution. The operation of the protocol is the following: if a task tries to get a semaphore which is already taken by a lower priority task then we raise the lower priority task to the priority of the higher priority task (hence the name inheritance). This way we can prevent interrupting the execution of the lower priority task when a task whose priority is higher than the lower priority task but lower than the higher priority task becomes ready to run. We do this because otherwise blocking the lower priority task would also delay the execution of the higher priority task as it is waiting for the semaphore. The operation is displayed on Figure 9.

As you can see task med was unable to defer the execution of task high in this case. (If there were more intermediate priority tasks they were also unable to defer task high.) There is only one thing what we was unable to avoid (because it is not possible): the lower priority task is able to block task high for the time it uses the semaphore.

Those semaphores that implement a protocol to mitigate the effect of priority inversion are called mutual exclusion semaphore (or just **mutex**). The second difference from an ordinary semaphore is that these semaphores have an “owner” (the task which currently owns the semaphore). Only the owner of the semaphore is allowed to release it!

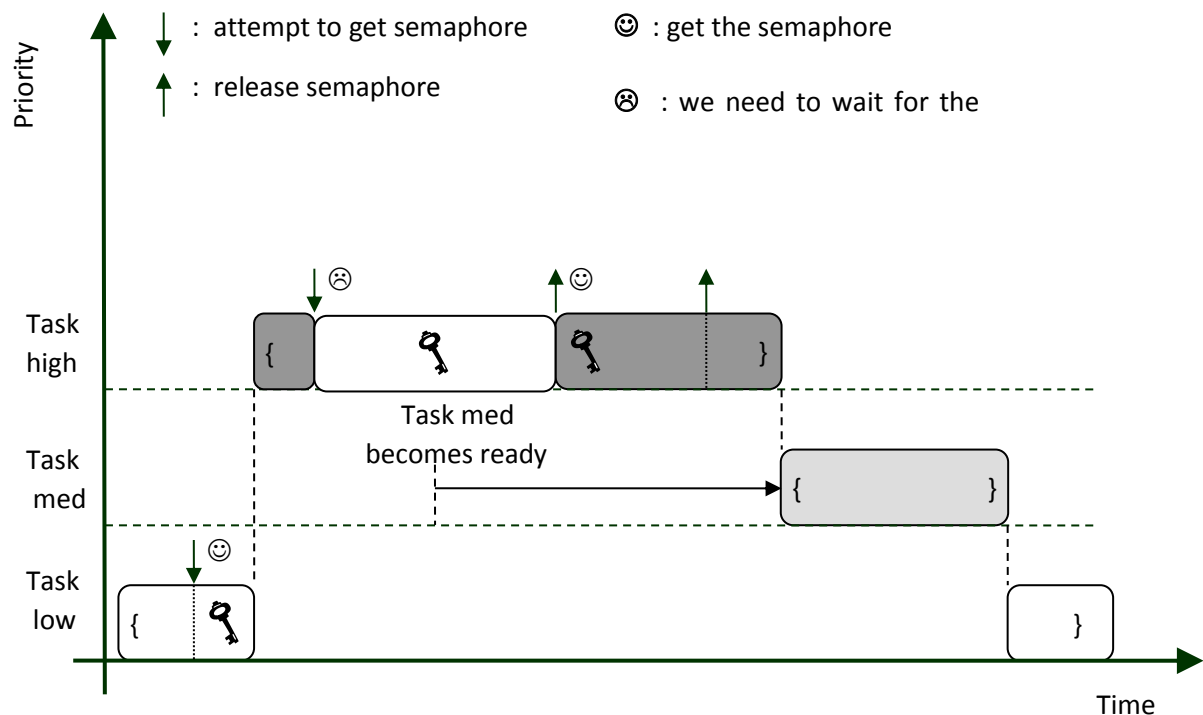


Figure 9. Solving the problem of priority inversion by priority inheritance protocol

3.8. Desktop OS vs. RTOS

| | Desktop OS | RTOS |
|---|--|--|
| Starting | The OS boots first and then it loads the applications to the memory. | The code of our application starts executing and it will launch the OS's scheduler. |
| Structure | The OS is a separate entity from the applications. | The OS and the application is one entity. |
| Protection (protecting the tasks against each other, and protecting the kernel against the tasks) | Strong. | Weak or absent. |
| Scalability, Configurability | Weak (e.g. it has various editions). | Strong (there are plenty of configuration options to get rid of all the unnecessary features). |
| Size | Big (~GB). | Small (n kB – n MB). |

Table 1: comparison of desktop and embedded OSes

4. FreeRTOS basics

Loyal to its name it is a free real-time operating system for embedded systems developed by Real Time Engineers Ltd. Its source code is also open. The operating system is licensed under a modified version of the GNU General Public License. The modification taking the form of an exception. The exception permits the source code of applications that use FreeRTOS and are distributed as executables to remain closed source, thus permitting the use of FreeRTOS in commercial applications without necessitating that the whole application be open sourced.

Further properties:

- The scheduler can be preemptive or co-operative.
- The preemptive and co-operative scheduling can be mixed.
- Optionally it is allowed for more than one task to have the same priority. In this case time-slicing scheduling is implemented among them.
- The source code is optimized for easy porting (currently there are more than 30 architectures officially supported).
- Designed to be small, simple and easy to use.
- Many example application is bundled with the OS to make the first steps easier.
- OS services (often used):
 - Tasks
 - Binary semaphores
 - Counting semaphores
 - Mutexes
 - Queues
 - Event groups /or Event flags/
 - Software timers
 - Memory management
- OS services (less often used)
 - Co-routines
 - Recursive mutexes
 - Direct to task notifications
 - Stack overflow detection
 - Trace features
 - Run time statistics

FreeRTOS is an actively developed operating system. For this reason detailed and up to date description can be found on the official website:

www.freertos.org

We advise you to browse this site! For the laboratories the following parts are important:

- [FreeRTOS] / [About FreeRTOS]
- [FreeRTOS] / [Features / Getting Started]: except for co-routines, direct to task notifications and software timers as these components are not investigated during the laboratories.

Further required learning materials are the FreeRTOS slides presented on the lectures of this subject. Additionally, chapter 7.3.3 contains some useful information on the source code of the FreeRTOS.

5. EFM32 Giant Gecko Starter Kit

5.1. Properties of the development board

The Silicon Laboratories EFM32 Giant Gecko Starter Kit (EFM32GG-STK3700A) contains the development board (BRD2200A) used during the laboratories featuring a contemporary, 32-bit microcontroller (EFM32GG990F1024). The components of the board can be seen on the figure below.

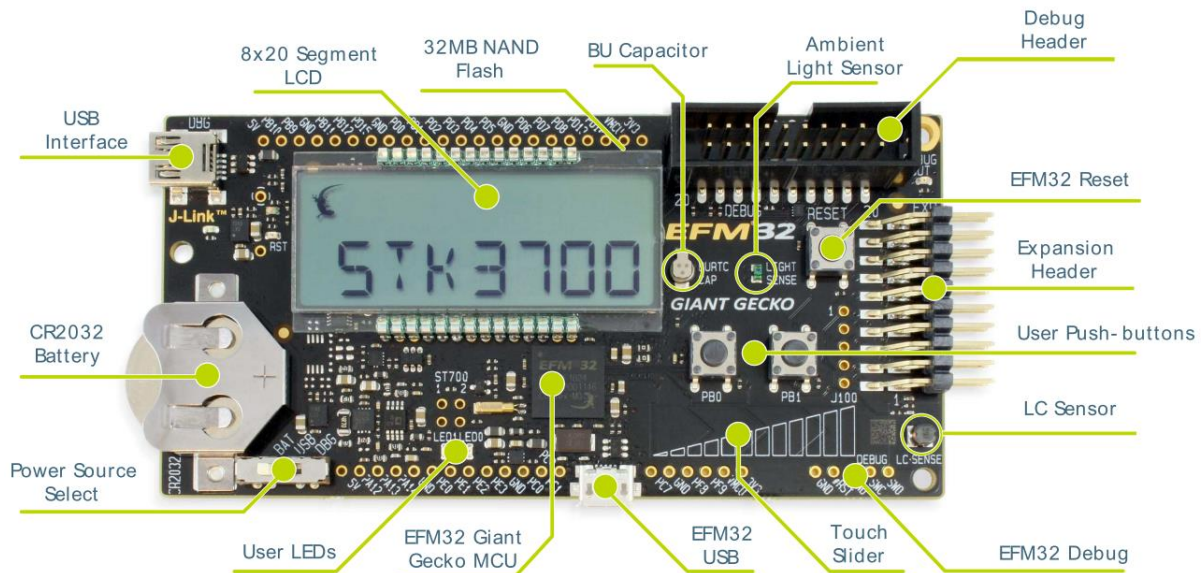


Figure 10. Components of the EFM32 Giant Gecko Starter Kit development board

The central component of the panel is the EFM32GG990F1024 microcontroller. It features the following:

- ARM Cortex-M3¹⁴ processor core
- 1024 kB Flash
- 128 kB RAM
- 48 MHz maximal clock frequency

There are a lot of other component around the microcontroller:

- 2 x user LEDs
- 2 x user buttons (+ the reset button)
- capacitive touch sensor
- metal detector
- light sensor
- LCD display (alphanumeric + special symbols)
- USB micro connector (host or device)
- CR2032 battery holder
- power source selection switch (battery, USB, debugger)
- 2x10 pin expansion header

¹⁴ <http://www.arm.com/products/processors/cortex-m/cortex-m3.php>

On the top-left corner we can find a second USB (mini) connector which is used to connect to the debugger integrated to the panel. We can connect to one of the UART peripheral of the microcontroller also on this connector. Which will be presented as a virtual COM port under the PC.

5.2. Connecting to a computer

During development we can connect the board to a computer via **the USB connector with DBG subscription** (at the upper left corner, USB-mini). Through this interface the following services can be reached:

- uploading the application to the microcontroller
- debugging the uploaded application
- monitoring the power consumption of the board
- virtual serial (COM) port
- pen drive function (for an alternate way of application uploading)

Because we are using the debug USB connector **we have to select DBG as the source of power** (the power switch is located at the lower left corner of the board).

5.2.1. Checking devices

The board represents itself to the computer as a composite USB device. Under this we can find the following other devices:



Figure 11. Devices belonging to the development board (*Device Manager / View: Devices by connection*)

These devices provide the services described above as follows:

| | |
|------------------------------|--|
| J-Link driver | application upload, debug, monitor power consumption |
| JLink CDC UART Port (COM<#>) | virtual serial port |
| SEGGER MSD Volume USB Device | pen drive function |

Before using the card it is worth to check whether these devices operate as expected.

The most important device is the *J-Link driver* because we use this to upload the application to the microcontroller and then to debug its execution (and also to observe the power consumed).

The virtual serial (UART) port is also important because we will use this as the standard output. **(Remember the number assigned by Windows to this serial port, we will need this!)**

The pen drive function is optional. It provides an alternative way to upload the application. After building the application we can copy the created binary file to the pen drive. After the file has been copied it is automatically uploaded to the program memory of the microcontroller.

The following figure shows where to find these devices on the default view of the *Device Manager* (devices by type).

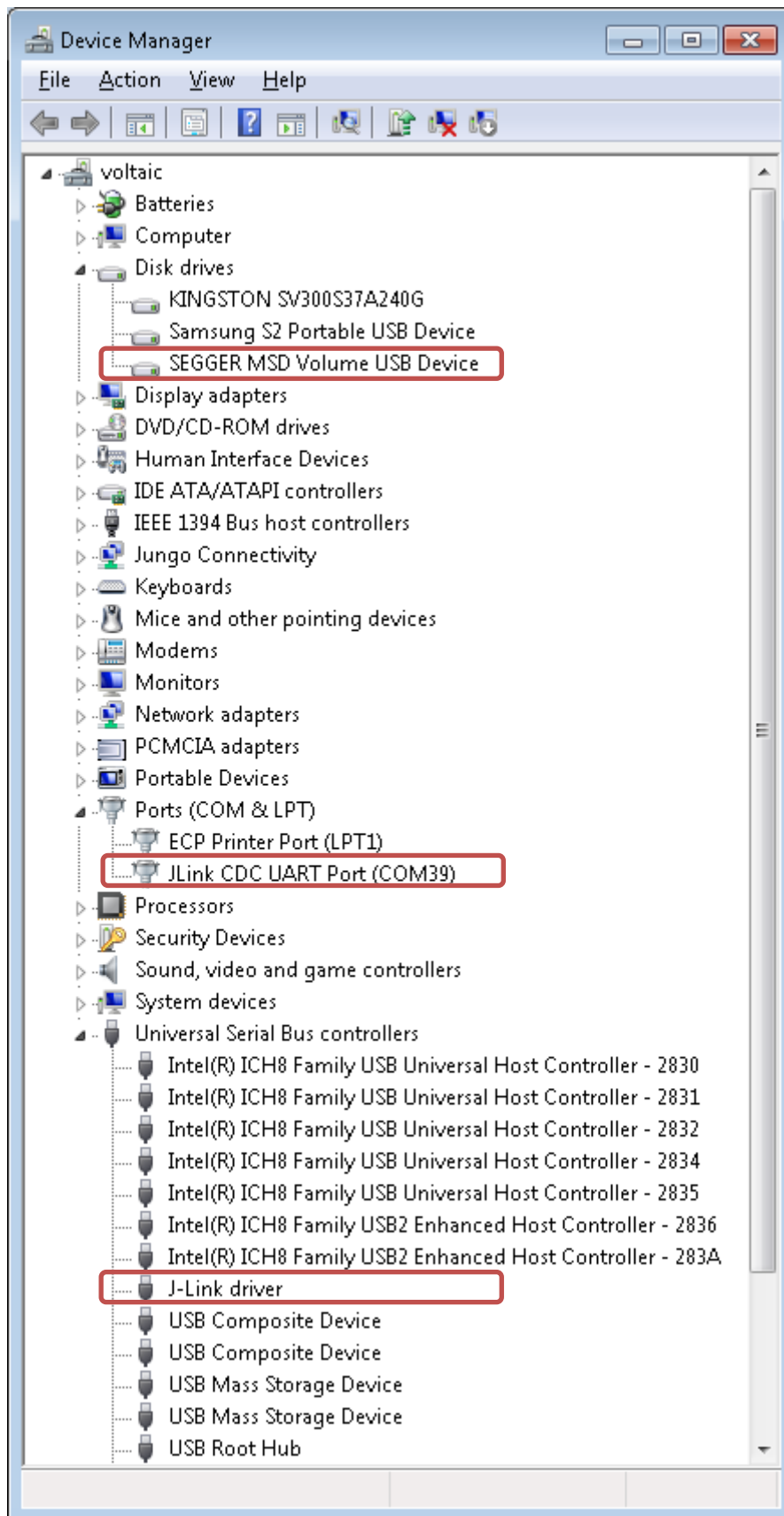


Figure 12. Devices belonging to the development board (*Device Manager / View: Devices by type*)

5.2.2. Connecting to the virtual serial port

The interface called as “serial port” (COM port in computers) usually refers to two different standards:

- UART (Universal Asynchronous Receiver/Transmitter)
- RS-232

The protocol of the communication is specified by UART:

- Baud: data transmission rate
- Number of data bits: 5, 6, 7, 8 or 9
- Parity bit: no, even, odd
- Number of Stop bits: 1, 1.5 or 2

RS-232 defines the physical aspects:

- signal levels
- the connectors used (D-sub)

The traditional RS-232 connector is usually missing from today’s computers. However there are plenty of USB connectors. For these reasons the UART peripherals are connected to the computers via an electrical circuit which communicates with the computer through USB. The device drivers of these circuits then create a virtual COM port on the computer. This virtual COM port can be used by traditional applications the same way as a physical COM port.

The easiest way to communicate over a COM port is to use a so called terminal application. These applications send the characters typed into their window over the serial line. Characters received from the serial line are displayed in the window. Such an application is *putty*. Maybe you know it as an SSH terminal. However it can be used as a serial terminal as well.

Whatever terminal software is used it is vital to set the same UART options as the ones used by the peer device. In the case of the Giant Gecko Starter Kit these are:

| | |
|---------------------|---------|
| Baud | 115 200 |
| Number of data bits | 8 |
| Parity bit | none |
| Number of Stop bits | 1 |
| Flow control | none |

So we have to set *putty* to use the same options as above (and to select the COM port assigned by Windows to the development board). The main settings can be made under *Session* while the advanced parameters can be set under *Serial* (furthermore the settings can be saved for later use, see *Saved Sessions*).

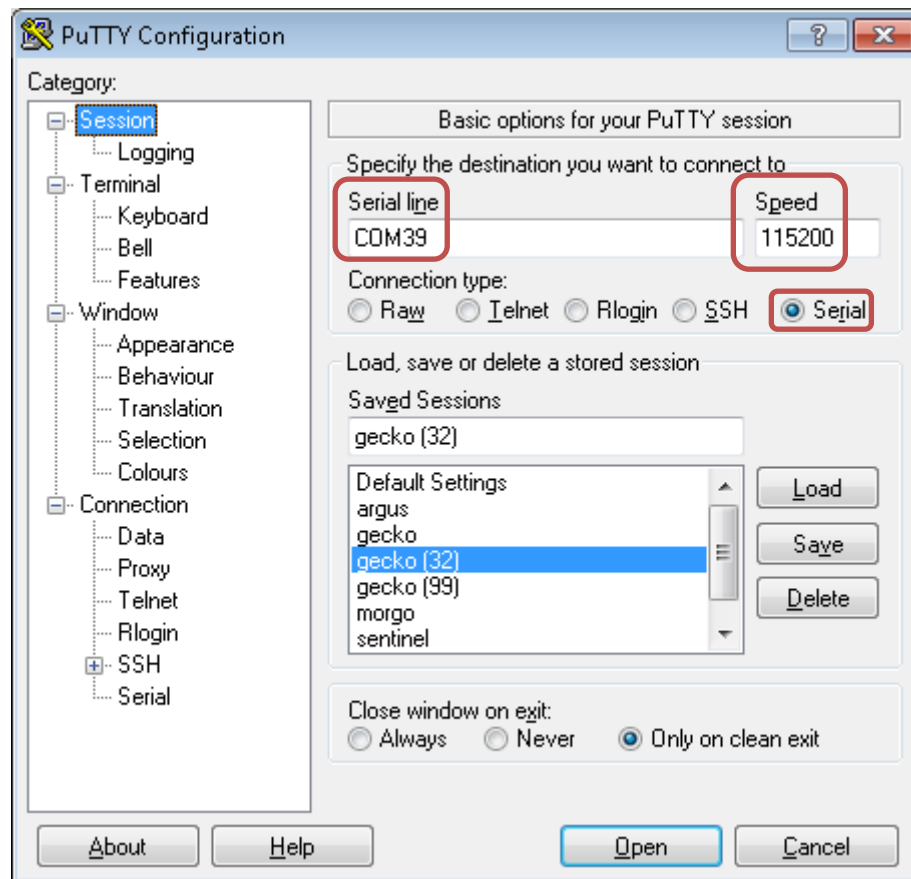


Figure 13. The main parameters of the serial communication in *putty*

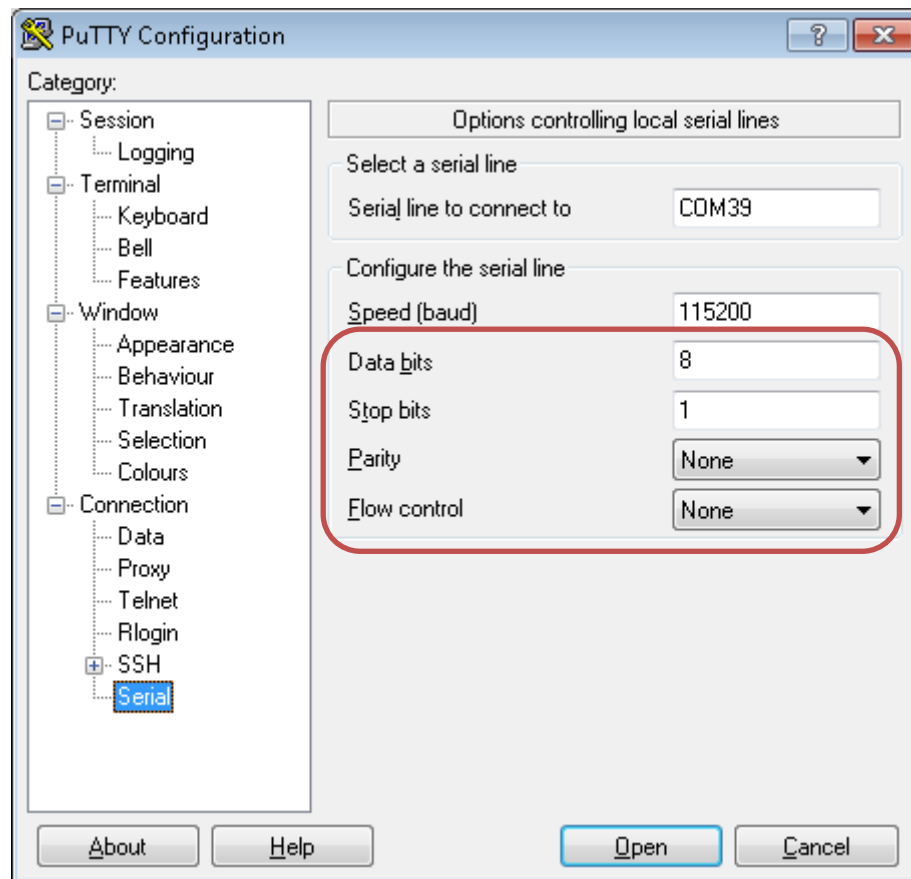


Figure 14. The advanced parameters of the serial communication in *putty*

6. Simplicity Studio 4

6.1. Launcher window

The development environment incorporates a number of components: *Compatible Tools*, *Documentation*, *Demos* and *Software Examples*... These components can be reached through the *Launcher* window. This window dynamically shows the components depending on the device selected.

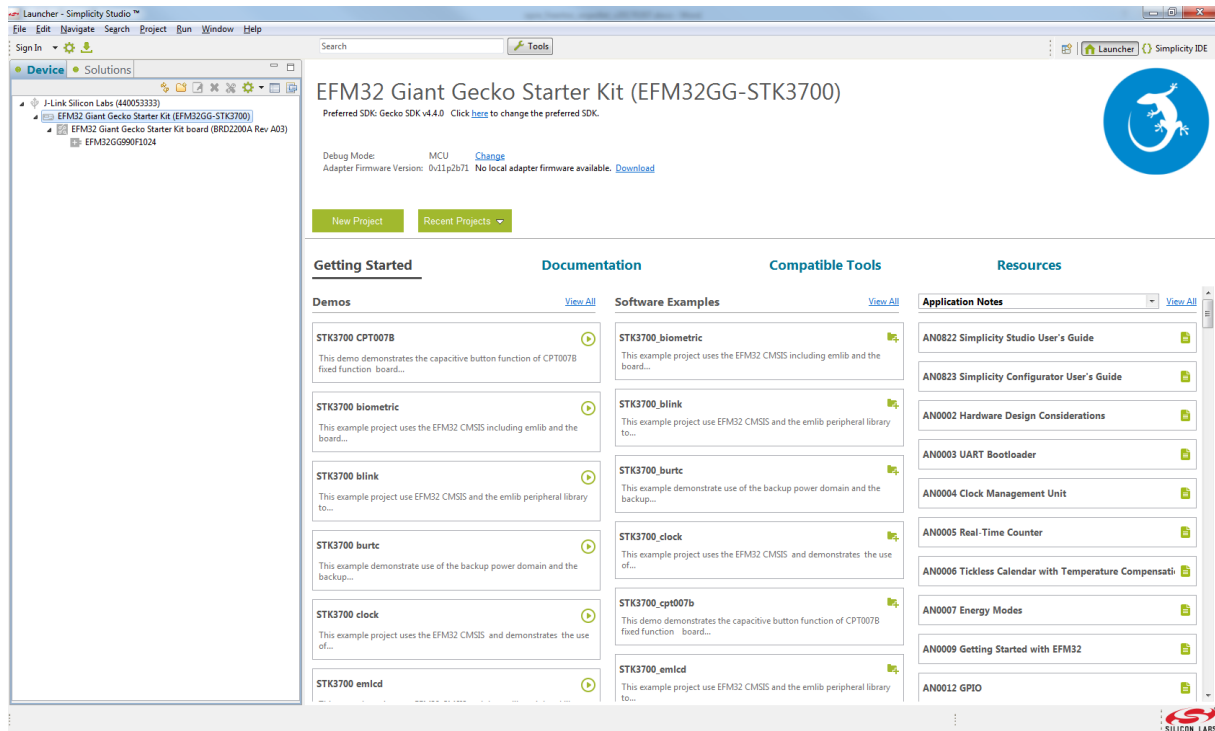


Figure 15. The *Launcher* window of Simplicity Studio 4

A device can be selected in two ways. If the device is already connected to the computer we can find it under the *Device* tab. The content displayed is hierarchical. For example we can see on the figure below that a *J-Link Silicon Labs* debugger circuitry is connected to the computer (via USB). On the other side of the debugger there is an *EFM32 Giant Gecko Starter Kit (STK3700)*. This kit consists of only one board (*BRD2200A*). And finally the microcontroller on the board is an *EFM32GG990F1024*.

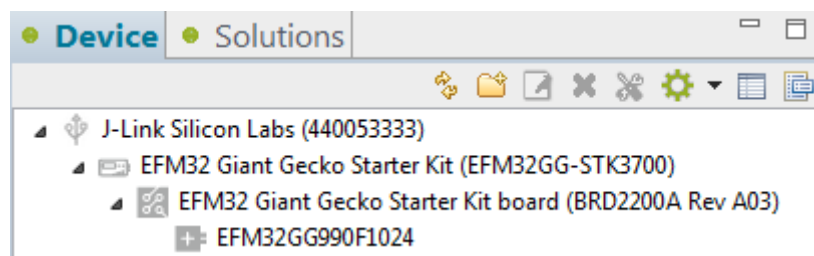


Figure 16. An example to the devices connected to the computer

The other way is useful if we want to select a device (and to access related components like its documentation) which is not currently connected to the computer. In this case select tab *Solutions*! All of the supported devices can be added to this tab manually.

It is worth to note that the components offered are depend on the actual selection. Staying with the example above the results are different if we select the whole starter kit, or only just the board or

only just the microcontroller. For example the documentation offered in the latter case involves only the microcontroller related materials (like its *datasheet*, *reference manual* or *errata*). However, if we select the board then besides these documents the ones related to the board are also offered (like its *schematic* or the *bill of materials*). And if we select the whole starter kit then all of the above mentioned materials are offered plus the ones related to the whole starter kit (like its *user's manual*).

The similar is true for the demo applications and software examples too. We can access them only if at least the board has been selected. This makes sense as the demo applications usually blink some LEDs, display something on the LCD, and so on... These actions make sense only with respect of a board. (The only difference between *Demos* and *Software Examples* is that meanwhile *Demos* are pre-built *Software Examples* are accessible as projects. This means that *Demos* can be uploaded to and run on the board (optionally with energy consumption monitoring) while *Software Examples* can be imported into the *Simplicity IDE*.)

6.2. Simplicity IDE

The central component of the development environment is the *Simplicity IDE*. It is based on the widely used and open-source Eclipse IDE. However, it has been tailored to the needs of developing applications to Silicon Labs devices.

It is a common property of Eclipse based IDEs that they offer various *perspectives* during the various phases of development. For example during the coding phase we will use the perspective named *Simplicity IDE* (if we use the term *Simplicity Studio* we refer to the whole development environment).

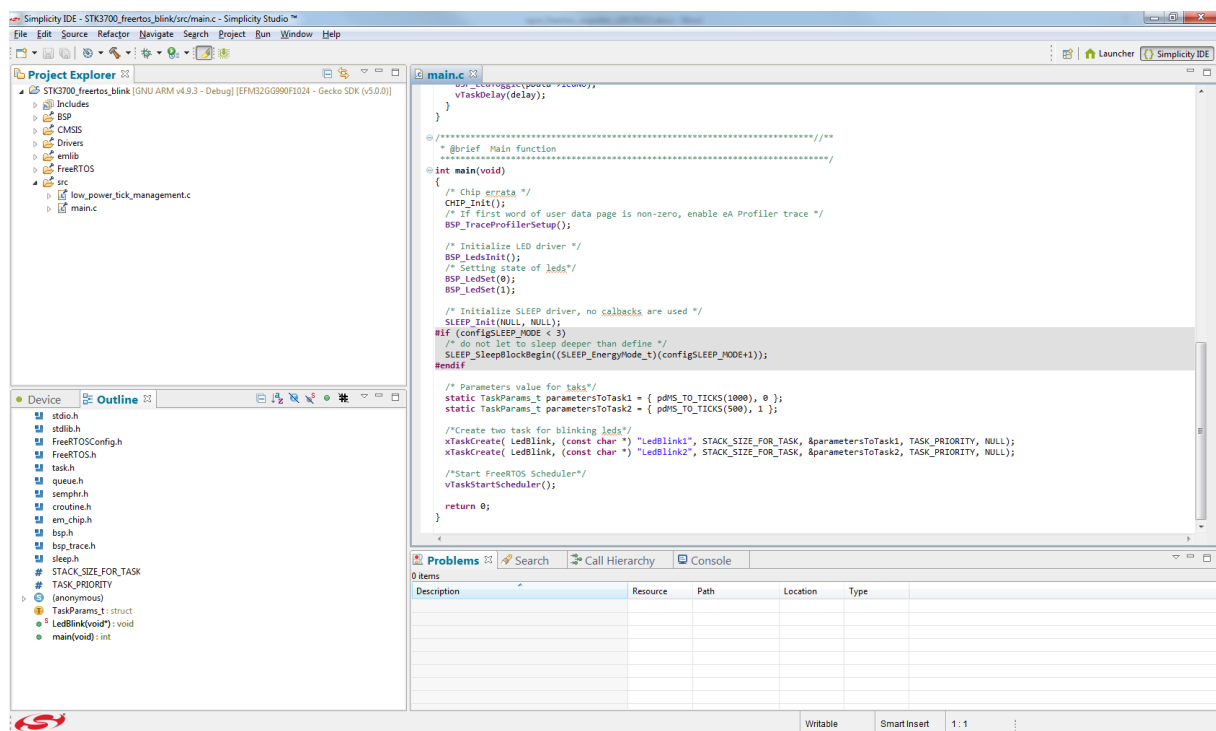


Figure 17. The *Simplicity IDE* perspective used during the coding phase

On the picture below we can see the most frequently used tools on the toolbar of this perspective. The project can be built by pressing the “hammer” icon. The other two icon is used to upload the application to the microcontroller (the project will be automatically built before uploading if it hasn’t been built already, or has changed since the last build). The difference between these two icons is what follows the uploading. If we press the “bug” icon the development environment enters debug mode

(and switches to the *Debug* perspective) after upload. This way we can observe the execution of our application and find bugs. If we press the third icon the development environment enters energy monitoring mode (and switches to the *Energy Profiler* perspective). This perspective lets us observe the energy consumed by the board during the execution of our application.



Figure 18. The most frequently used tools

During build command-line tools are executed in the background. Their output can be observed in the *Console* tab. The build process is governed by *make* using the information contained in *makefiles*. These files are generated by *Simplicity IDE* considering the project under building. The actual steps of the build process (like *compiling*, *linking*, conversion between different binary formats, displaying information about the sections in a binary file...) are done by various executables of the *GCC toolchain*. The compiler used is called as *cross-compiler* referring to the fact that the compiler runs on the computer but compiles to a different architecture (here ARM).

6.3. Debug perspective

After entering debug mode the layout of the development environment also changes to help debugging. Besides showing the C language source of the application we can also see a view with the *disassembled* instructions. The *Debug* tab shows the contents of the stack (called *stack trace*). The *Variables* tab shows the variables of the application, the *Expressions* tab can be used to form and evaluate expressions, the *Breakpoints* tab shows the *breakpoints* placed into the application. The *Registers* tab displays the contents of the core CPU registers and also the registers of other peripherals integrated into the microcontroller. And finally the *Memory* tab gives access to the contents of the memories within the microcontroller.

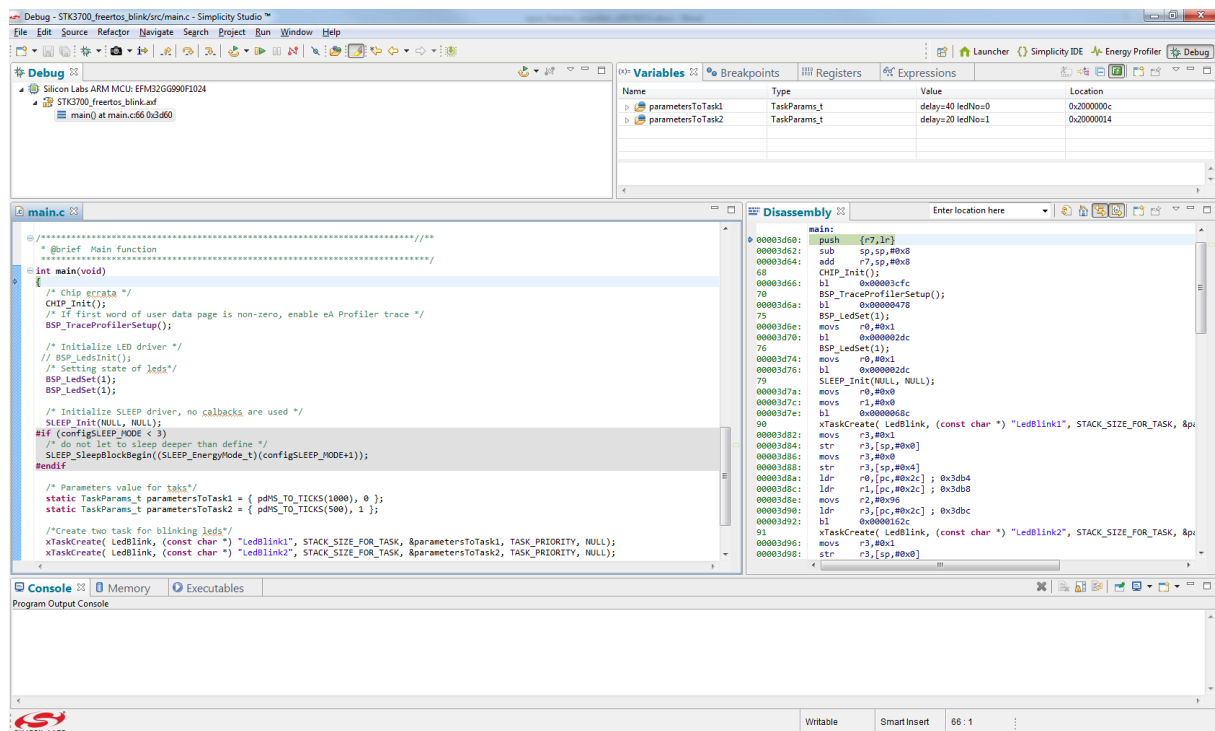


Figure 19. The Debug perspective of Simplicity Studio

After entering debug mode the application is stopped at `main()` which is defined as the entry point of a C application. If the application is stopped then the instruction to be executed next is highlighted (both on the C language and the disassembled views, see figure below).

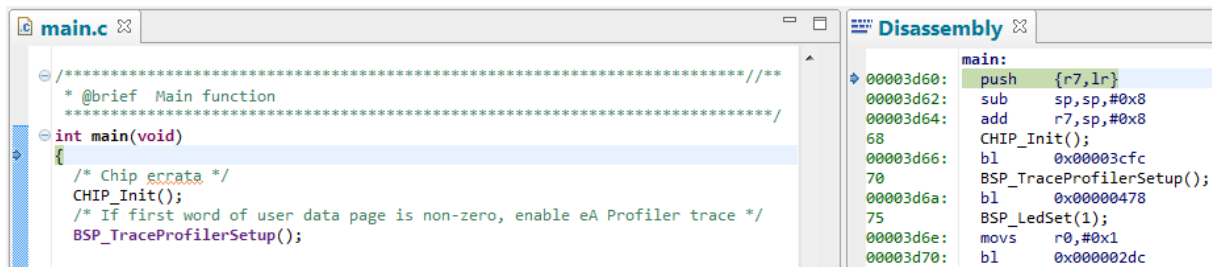


Figure 20. The next instruction to be executed (right after entering debug mode)

At this point it is worth to note that a C language statement usually compiled into more than one assembly instructions. Furthermore, if compiler optimizations are enabled then C statements cannot be always correlated to assembly instructions.

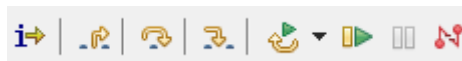


Figure 21. Most frequently used debug tools

Let's see the most frequently used tools during the debug phase! If we want to execute a currently stopped application use the "big green arrow" icon (*Resume*, F8). After pressing this button the application will run until it is stopped by pressing the "pause" icon (*Suspend*) or the execution has reached a breakpoint.

We also have the opportunity to step-by-step execute instructions! There are two ways to do that depending on how function calls are handled. If we want to enter into the function then use the icon depicting "a little yellow arrow pointing into the space between two instructions" (*Step Into*, F5). If we want to handle the function call as a single C statement then use the icon depicting "a little yellow arrow jumping over an instruction" (*Step Over*, F6). If we have entered into the body of a function but we are not interested in the remaining instructions of that function then press the icon depicting "a little yellow arrow pointing out from two instructions" (*Step Return*, F7). This will execute all of the remaining instructions (including the return from function).

In the previous paragraph under the term "instruction" we meant a C statement. If we want to step-by-step execute over actual assembly instructions we have to turn on *Instruction Stepping Mode* (the icon with "i").

We can also reset the microcontroller by pressing the button named *Reset the device* (the icon with a green and a yellow arrow). After reset the application is stopped at the entry point. In this case under *entry point* we mean the very first instruction to be executed (and not just simply the beginning of the `main()` function). The real entry point of the application is not `main()`. There are some prerequisites that has to be arranged before the application can call the `main()` function. This is the reason why the start of `main()` is not the real entry point. See chapter **Error! Reference source not found.** on the startup code for more information on this topic.

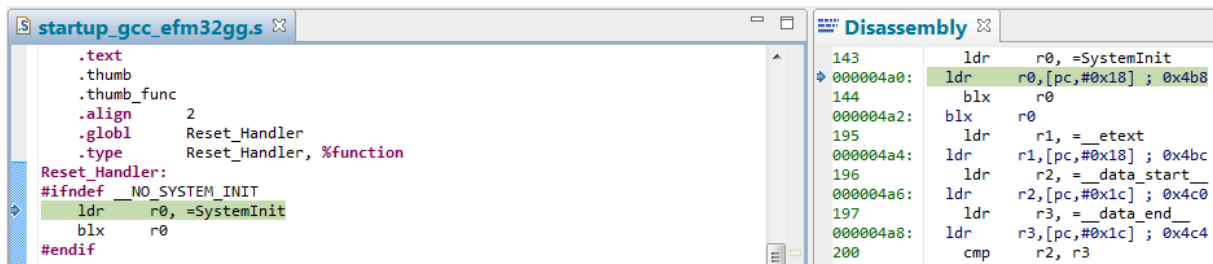


Figure 22. The entry point of the application right after reset

And finally, if we want to exit debug mode then press the *Disconnect* button (the red icon). After we exited debug mode the layout switches back to the *Simplicity IDE* perspective.

It is important to note that switching back to *Simplicity IDE* perspective can be accomplished also by pressing the button corresponding to that perspective (the button can be found at the upper right corner of the window). However, if we change perspectives in this way we won't exit debug mode! And it is easy to forget that the development environment is still connected to the board in debug mode! This could cause trouble if we want to enter debug (or energy monitoring) mode again later!

The remaining part of this sub-chapter contains supplementary information to those who find it strange that the very first instruction to be executed is not located at the very beginning of the program memory (at address 0) but at a different location (in our example at address 0x4A0). How can it be? The answer is the way how ARM Cortex-M processors start.

In the case of ARM Cortex-M processors the value stored at the very first (i.e. at address 0) word of the program memory is used to initialize the *Stack Pointer* (SP) of the CPU. The value stored at the second word (i.e. at address 4, as one word is 32-bit wide) is the address of the *Reset Handler*. After reset the processor first initializes its stack pointer then jumps to the address of the reset handler. So the very first instruction to be executed is really the first instruction of the reset handler. The figure below shows the first few words of the program memory in our example.

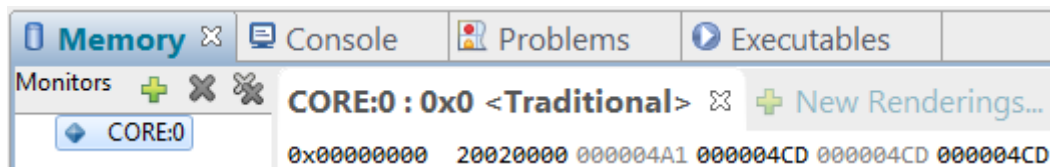


Figure 23. The first few words of the program memory

Currently the second word is in the focus of our interest: 0x000004A1. It is nearly the value we expected. If only that 1 at the end wasn't there... At this point we have to understand how ARM processors handle program memory addresses.

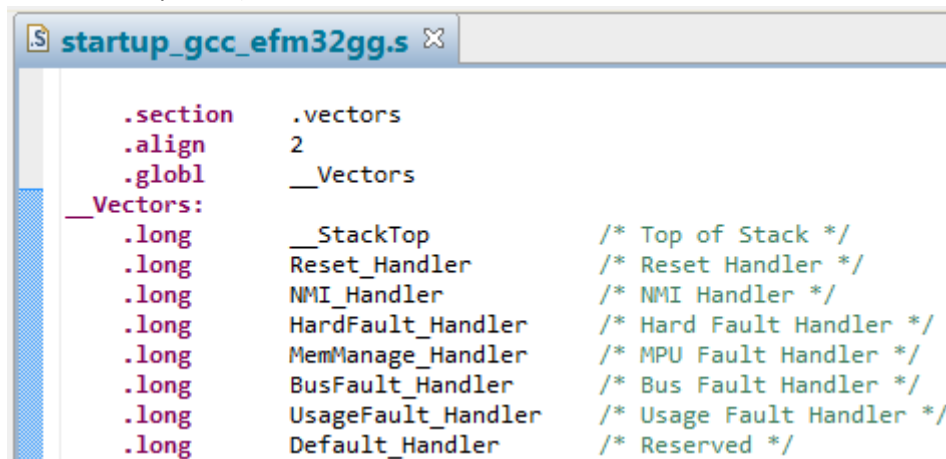
The *Least Significant Bit* (LSB) of the address is used to store the instruction set information. The traditional ARM cores support only the so called *ARM Instruction Set*. In this set the size of the instructions is 32 bit (so 4 bytes are needed from the program memory to store an instruction). To reduce the memory footprint required by applications using the *ARM Instruction Set* the designers at ARM developed the *Thumb Instruction Set*. This is a subset of the original instructions consisting of the most important ones and coded into 16 bits (so now only 2 bytes are needed to store an instruction). As some instructions are missing and some are limited (compared to their original counterparts) sometimes more instructions are needed then with the *ARM Instruction Set* but in most of the cases the memory footprint can be significantly reduced.¹⁵

¹⁵ <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACBCAAE.html>

If the execution jumps to a location in the program memory it is vital to know which instruction set should be used. This information is coded into the LSB of the address to be used for the jump. If it is 1 then Thumb otherwise the original instruction set should be used. In both of the cases (16 or 32 bit sized instructions) the memory address is surely even. Which means that the LSB could be safely used to store the instruction set information. The processor internally treats the LSB upon jump as 0 anyway.

As ARM Cortex-M processors implement only the *Thumb Instruction Set* all labels used as program memory addresses during a jump should have LSB set to 1. Otherwise an exception (*Hard Fault*) will emerge.

At this point we have only one question open. How to store the address of the reset handler (with LSB set to 1) into the second word of the program memory? Let's see the following code snippet (belonging to the startup code)!



```

.section      .vectors
.align       2
.globl       __Vectors
__Vectors:
    .long     __StackTop          /* Top of Stack */
    .long     Reset_Handler       /* Reset Handler */
    .long     NMI_Handler         /* NMI Handler */
    .long     HardFault_Handler   /* Hard Fault Handler */
    .long     MemManage_Handler   /* MPU Fault Handler */
    .long     BusFault_Handler    /* Bus Fault Handler */
    .long     UsageFault_Handler  /* Usage Fault Handler */
    .long     Default_Handler     /* Reserved */

```

Figure 24. The beginning of the vector table

In the code various *sections* can be defined. Such a section is `.vectors`. Vectors usually store program memory addresses (with the exception of the first, storing the initial value of the stack pointer). The vectors as a whole is called the *vector table*. The vector table is loaded into the beginning of the program memory. The so called *linker script* tells which sections go where. These addresses are used when an *interrupt* or a *fault* happen (these two are called together as *exceptions*) to jump to some part of the code to handle the exception. The *reset* event counts as an exception (with the greatest priority).

On the figure above we can see that the second word will really be the address of the reset handler. But how do we manage to make it odd? Under normal circumstances labels in the assembly code correspond to the address of the next instruction which is always even! The solution is a special GNU ARM assembly keyword: `.thumb_func` (see figure Figure 22.).

6.4. Energy Profiler

The *Energy Profiler* can be used with certain boards (like the one used in the laboratories) to monitor the power consumption of the card on-the-fly. (Furthermore under certain circumstances it is also even possible to correlate power usage to the code.)

On the figure below we can see the power consumption of an application blinking two LEDs (all the four logical combinations are generated). On the plot we can easily tell how many LEDs are turned on at a given time (we can even spot the fact the one of the LEDs consumes a little bit more power).

If we want to exit the energy monitoring mode then we have to choose *End Session* from the *Run* menu (or after clicking on that little black arrow in the middle, above the plot). After energy monitoring

stopped the layout is not switched back automatically to the *Simplicity IDE* perspective. We have to do that manually if we want.

It is important to note that switching back to *Simplicity IDE* perspective can be accomplished also by pressing the button corresponding to that perspective (the button can be found at the upper right corner of the window). However, if we change perspectives in this way we won't exit energy monitoring mode! And it is easy to forget that the development environment is still connected to the board to monitor power consumption! This could cause trouble if we want to enter energy monitoring (or debug) mode again later!

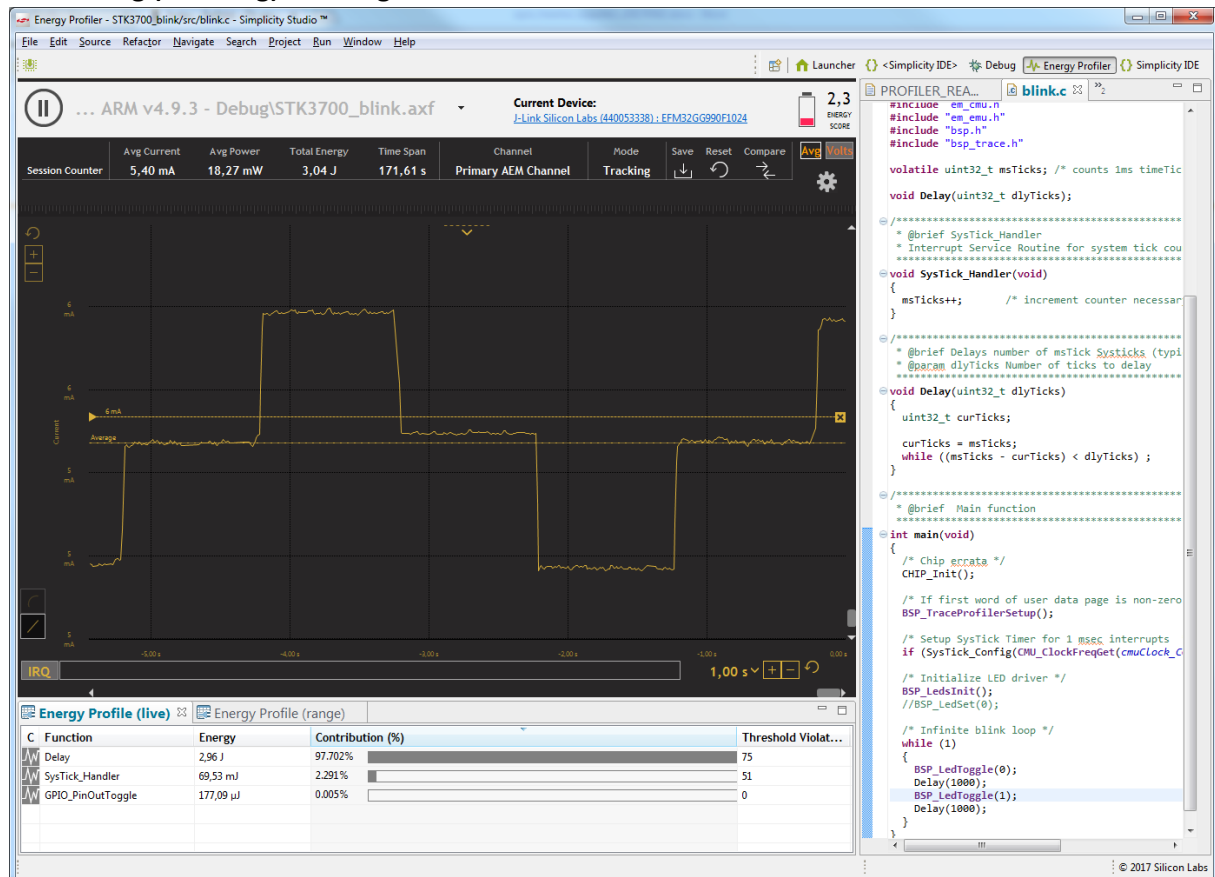


Figure 25. Energy Profiler (monitoring an application blinking LEDs)

Note: unfortunately this component of the development environment still needs some polishing. If we zoom on the y-axis many times then it is possible to display the same value to more than one grid lines (for example 5 mA and 6 mA are both displayed to two grid lines on the picture above). It is a little bit confusing. Hopefully the problem will be solved in the future (for example by displaying more decimal digits).

7. Description of the project

Development environments based on the Eclipse IDE store their settings and basic information on current projects in a *workspace* directory (more specifically in a subdirectory within the *workspace*, named [*.metadata*]). In most of the cases – but not necessarily – the *workspace* contains also the projects themselves (usually in a directory named the same as the project).

The project directories are constructed with a similar logic. They contain the more detailed settings of the project (in a file named *.project* and additionally – in case of C language development – in a file named *.cproject*). In most of the cases – but not necessarily – the project directories store also the files belonging to the projects.

Project can be created and deleted (either by just removing it from the list of current projects or actually deleting it from the filesystem). Furthermore, already existing projects not being on the list of current projects can be imported..

7.1. Downloading the project

We have made a project for the laboratories with the required settings and files. This can be downloaded in ZIP format. In the ZIP file a single directory (the project) can be found. Let's extract this directory into the *workspace* folder of Simplicity Studio (there is a shortcut on the Desktop to the *workspace*).

7.2. Importing the project

Although after the extraction of the project folder into the *workspace* directory it can be found on the filesystem, it is not yet placed into the list of current project. For this we need to import the project first.

Let's switch to the *Simplicity IDE* perspective! Then right click in *Project Explorer* tab. A popup menu will appear. Choose the submenu named *Import* and then the menu item named *Import...*

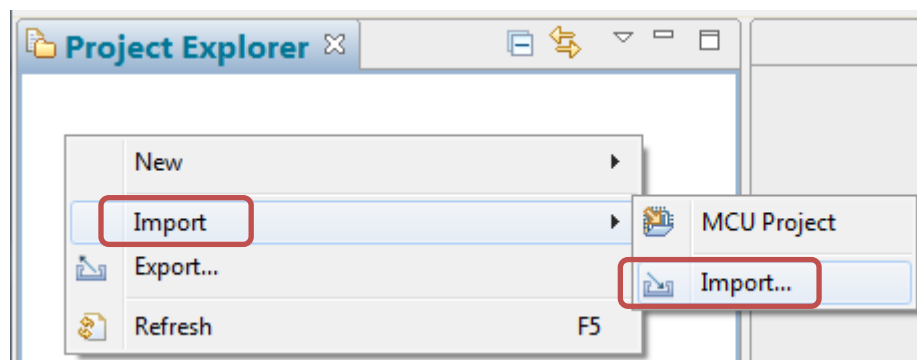


Figure 26. Importing the project (first step)

A dialog will appear where the method of the import process can be selected. Select *General / Existing Projects into Workspace* (then *Next*)!

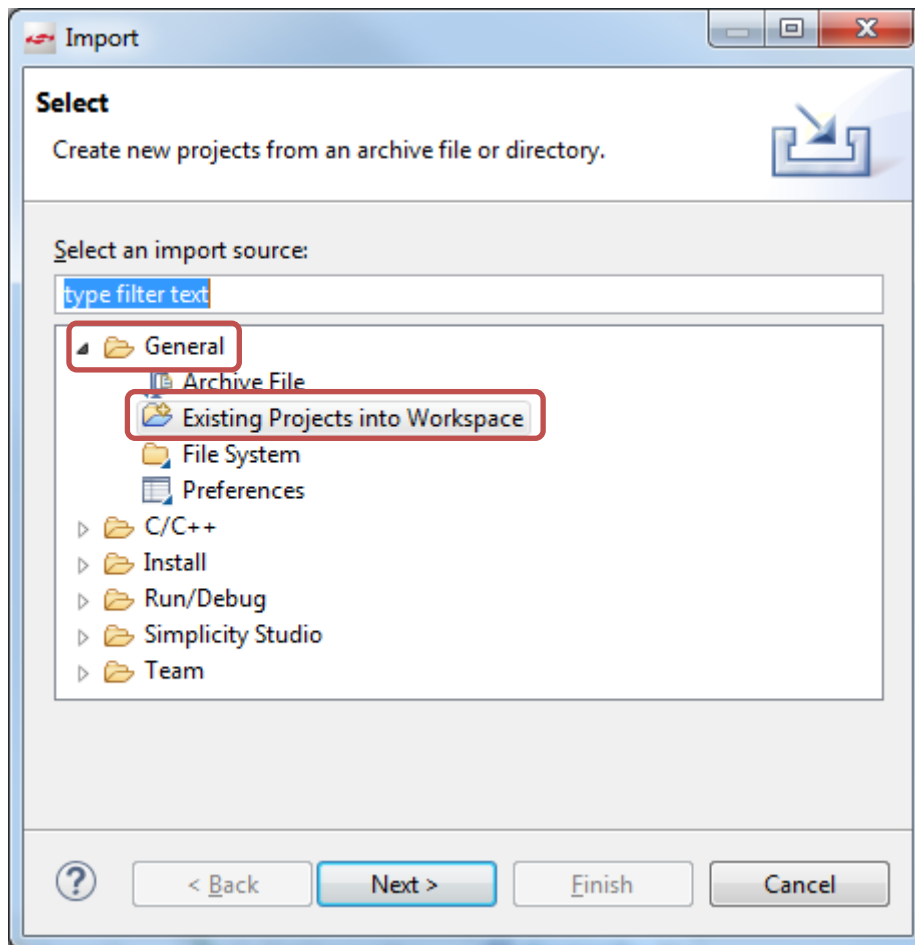


Figure 27. Importing the project (second step)

Now we have to specify the location of the project to be imported: click on *Browse...* and select the *workspace* folder (this folder is usually already selected, so in most of the cases after *Browse...* we can just click on *OK*). If everything goes well at this point the project should appear on a list in the dialog. Check this project (in case if it wasn't already checked)! Warning: the option *Copy projects into workspace* should not be checked (as the contents of the project is already located within the *workspace*)!

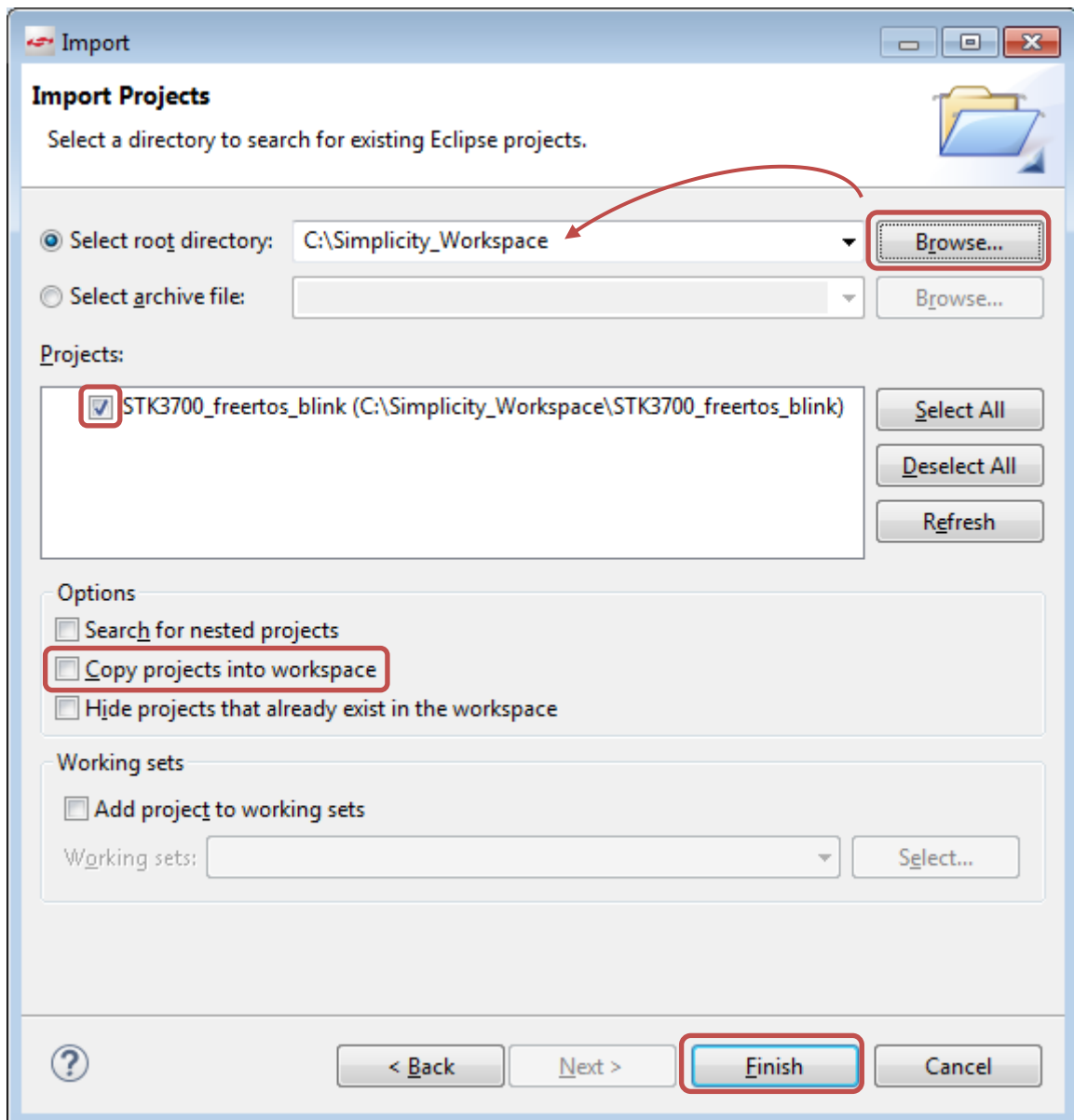


Figure 28. Importing the project (third step)

After a successful import the project appears on the list of current project in *Project Explorer*:

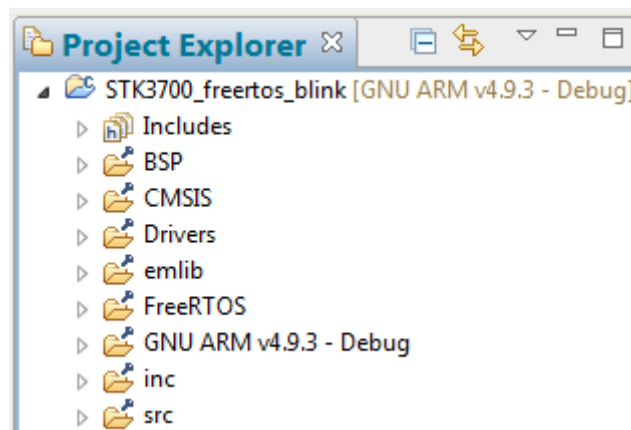


Figure 29. The project after import

7.3. The structure of the project

The source code belonging to the project can be found within the directories below:

| | |
|----------|---|
| src | Skeleton files for the exercises |
| inc | Header files that are modified for the project or should be modified during the exercises |
| FreeRTOS | The source code of FreeRTOS |
| Drivers | Some useful utilities |
| BSP | API for the devices on the development board |
| emlib | API for the peripherals integrated into the microcontroller |
| CMSIS | API for the core of the microcontroller |

Besides the above directories two additional items can be found under the project:

| | |
|------------------------|--|
| Includes | This is not a directory on the filesystem. This item simply shows the configured paths for header file lookup. |
| GNU ARM v4.9.3 - Debug | This directory contains various files generated during the build process. |

7.3.1. The [src] folder

This directory contains the various source file templates for the exercises. Among these only one is allowed to be part of the build at any given time (the others have to be excluded from the build). This makes sense as we want to build only the application belonging to the current exercise. Furthermore, the linker would raise error if we built more than one of these templates as there would be more than one `main()` function).

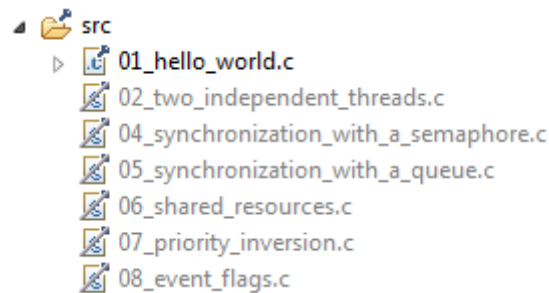


Figure 30. Contents of the [src] directory

If we have finished an exercise we have to exclude from build the template file belonging to that exercise and have to include the next one. Note: *there is no option to include a source file to the build. There is option only for exclude as by default all source files are included. So if we want to include a previously excluded file to the build what we have to do is to disable the exclusion.*

To exclude a file (or a whole directory) from the build (or to remove the exclusion) we can use the dialog that appears after right click on the given resource and select *Resource Configurations / Exclude from Build...* menu item:

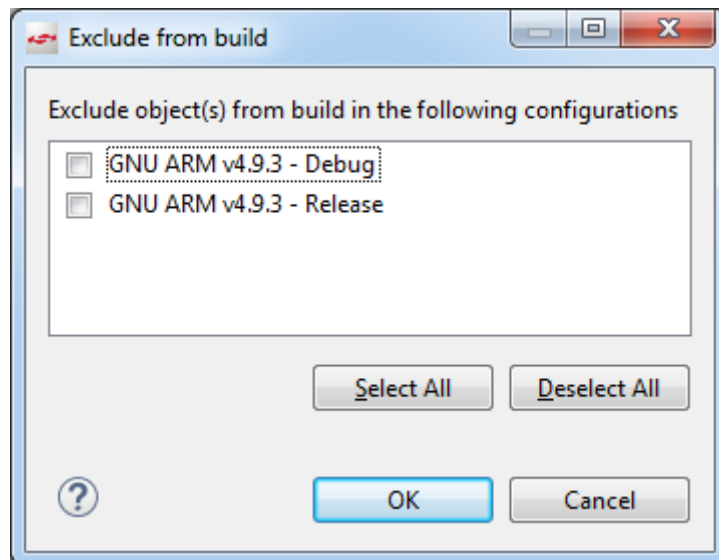


Figure 31. Dialog to exclude from build (or to remove exclusion) for a given file or folder

If we want to exclude the give object choose *Select All*. To remove the exclusion choose *Deselect All*. Then press *OK*. Note: *there are two ways to build a project (Debug and Release). We will always use the Debug variant. However, it is easier to press the Seleact All and Deselect All buttons than clicking on the individual checkboxes. And probably it is also more coherent if there is no difference between the two ways with respect of the excluded sources..*

7.3.2. The [inc] folder

Some header files were modified for the correct operation of the project. To preserve the original contents of these files at their original palces we have copied them into the [inc] folder.

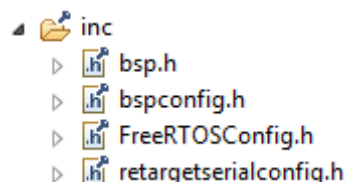


Figure 32. Contents of the [inc] directory

The header files `bsp.h` and `bspconfig.h` have been modified to support the push buttons on the board. This was needed as for some reason the original BSP API lacks the support for buttons. We have implemented button support in source file `bsp_stk_buttons.c` (see chapter 7.3.5 describing the [BSP] directory).

The header file `retargetserialconfig.h` has been modified to make it possible to use UART0 (which is accessible through the virtual serial port) as the standard I/O peripheral. For some reason the original API supports a number of UARTs but not UART0.

And finally we can find also `FreeRTOSConfig.h` here. This is because the configuration header of the FreeRTOS is always application dependent. It simply wouldn't make sense if we put it into a common location.

This directory is added to the beginning of the *include path* of the project. This makes it possible for the compiler to find these files. And as it was insertet into the beginning the compiler will use these files (and not the original ones on different paths):

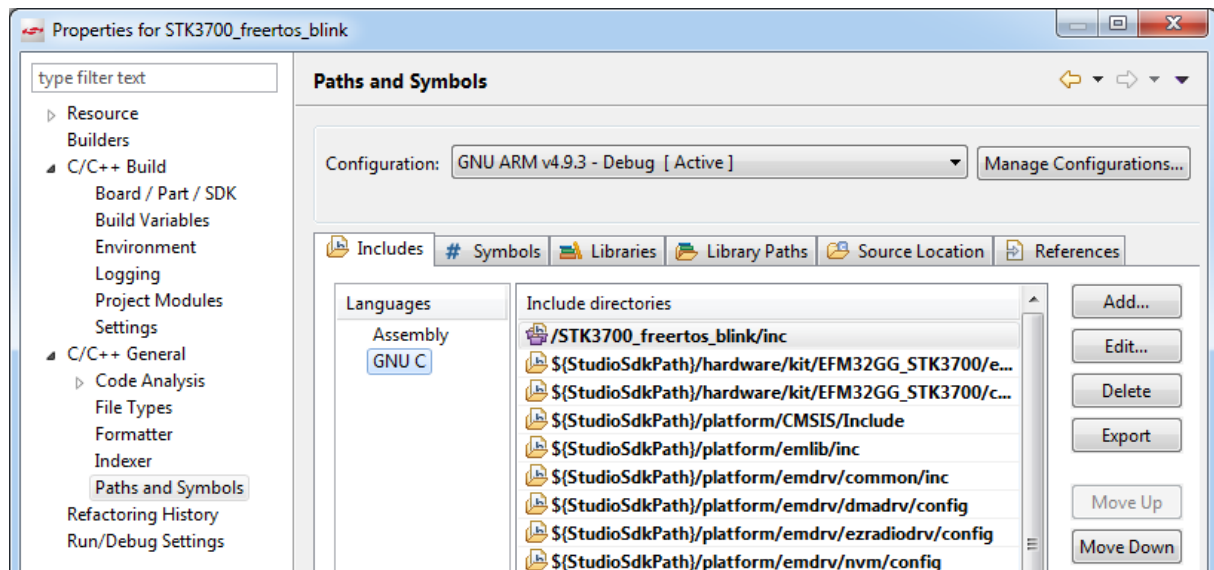


Figure 33. Contents for the [inc] directory

7.3.3. The [FreeRTOS] folder

The directory [FreeRTOS] holds the source files of FreeRTOS. The files not needed during the laboratories are excluded from build.

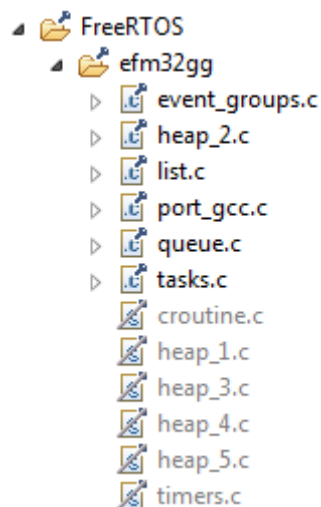


Figure 34. Contents of the [FreeRTOS] directory

The absolute minimum number of files to use FreeRTOS are the following:

- tasks.c Task handling (creation, deletion, scheduling)
- list.c Linked-list implementation
- port.c¹⁶ Architecture (and compiler) dependent code

If we want to schedule tasks it is trivial that tasks.c is needed. And it is also self evident that the lowest layer of the OS (port.c) is also needed. The list implementation is required for the internal operation of FreeRTOS as the operating system puts various items (like tasks and synchronization

¹⁶ In our case the name of the compiler has been also put into the name of the file for the sake of clarity. However, this is not usual.

objects) into a list. This means that the list implementation is not an API towards the application program but used by the scheduler. While it is tailored heavily for the scheduler's needs, it is also available for use by application code.

Other files needed during the laboratories:

| | |
|-----------------------------|--|
| <code>queue.c</code> | Message queues, semaphores and mutexes |
| <code>event_groups.c</code> | Bits for signaling events |
| <code>heap_2.c</code> | A dynamic memory management implementation |

If our application wants to use message queues, semaphores or mutexes then we need `queue.c`. It is worth to note that only queues have been implemented. Semaphores and mutexes are treated only as special use cases of the more general queue implementation.

To synchronize the execution of the tasks we can also use event signaling bits. Their implementation can be found in `event_groups.c`.

FreeRTOS supports operation without dynamic memory management. This has the cost that our application has to statically allocate memory for task stacks (and TCBs¹⁷)! This involves even the idle task! This is a little bit tedious (although it has advantages from the real-time point of view). The dynamic memory management provided by the C language is convenient but not well suited for real-time applications (it is usually slow and to make it even worse it has non-deterministic execution time).

It is on us to determine the real-time needs of our application. The developers of FreeRTOS provided us with a number of dynamic memory allocation schemes in the imaginary axis of “real-time friendly execution time vs. convenience”. We can choose among them. The one selected for the laboratories is the simplest implementation among the ones that support freeing up memory. (This is needed for example when a task is deleted. If the application does not delete OS object during its execution we can go even with the simplest implementation which lacks the support to free memory up.)

Other files not used during the laboratories:

| | |
|-------------------------------|--------------------------------------|
| <code>croutine.c</code> | So called co-routines |
| <code>timers.c</code> | Software timers |
| <code>heap_(1,3,4,5).c</code> | Other dynamic memory implementations |

Header files:

Header files (with the exception of `FreeRTOSConfig.h`) haven't been copied into the project directory. They can be found at their original path:

¹⁷ A TCB is the Task Control Block. It stores various information of a given task (like its priority).



Figure 35. Header files of FreeRTOS

The general header `FreeRTOS.h` is always has to be included. Furthermore, its inclusion shall be located before the inclusion of any other FreeRTOS headers.

To manage tasks we need `task.h`¹⁸.

The interface for queues is placed into `queue.h`, while the interface for semaphores and mutexes are declared in `semphr.h`. It is worth to have a look at `semphr.h`. We can observe that the semaphore API is declared purely by using the queue API via C macros.

The interface for event bits are defined in `event_groups.h`.

The other header files are not needed for the laboratories.

7.3.4. The [Drivers] folder

This directory contains some useful utilities:

| | |
|-------------------------------|---|
| <code>retargetio.c</code> | It is a very narrow software layer which makes it possible to retarget the standard input and output. |
| <code>retargetserial.c</code> | Retargetting the standard I/O to one of the U[S]ART peripherals of the microcontroller. |
| <code>udelay.c</code> | Software delay loops (the amount of delay can be expressed in μs). |

During the laboratories we will use UART0 as the standard I/O device. This is the one that is accessible through the DBG USB connector.

Software delay loops are needed only for the priority inversion examples to produce the artificial timing conditions needed to show the problem. (A software delay loop is basically a long enough for loop to achieve the desired delay. This means if a task waits by the help of these loops it remains in the running state!)

7.3.5. The [BSP] folder

BSP is the abbreviation of *Board Support Package*. So this folder contains source files to handle devices on the development board. `bsp_stk_leds.c` provides functions to manage LEDs, while `bsp_stk_buttons.c` helps with push buttons.

Notes:

¹⁸ Note, the name of the header is in singular while the name of the corresponding C source is in plural!

- LEDs are not required during the exercises but can be used freely (for example to help debugging the code).
- The functions for push buttons are missing from the original API for some reason. They have been implemented by us based on the LED API.
- There are various other BSP files but we won't use them so they haven't been copied into the project.

7.3.6. The [emlib] folder

Files in [emlib]¹⁹ provide support for devices inside the microcontroller (excluding the core). The project contains only those sources that are needed during the laboratories. It is not vital to know this software layer as we will use it only in an indirect way.

There are two exceptions to this rule:

- In one of the exercises the CPU shall be put into an energy saving mode. To accomplish this the API for the EMU (*Energy Management Unit*) is required (the name of the corresponding header is `em_emu.h`).
- The EM Library defines a so called *assertion* macro and a function (see `em_assert.h` and `em_assert.c`). This is obviously not for the support of a peripheral but useful. By *assertion* we mean that we want to be sure that a given condition is fulfilled. We use it when it is vital for a part of the code that a given condition is true (otherwise fatal errors could happen). In this case we put an *assertion* call before that code and pass the condition as the argument to the *assertion*. If the condition is true then nothing happens. The *assertion* returns without taking any action. But if the condition is false *assertion* stops the program (in the most simplest implementation by entering an endless loop). This assures that the code protected with the *assertion* will be executed only if the required condition is fulfilled. We won't use the *assertion* macro or function directly but will surely meet them...

7.3.7. The [CMSIS] folder

And finally the lowest software layer in the hierarchy is CMSIS (*Cortex Microcontroller Software Interface Standard*). This interface has been specified by ARM. The core of the microcontroller used is an ARM Cortex-M3. Many microcontrollers (including the ones from different vendors) may use ARM cores. This has raised the need to define a standard interface for managing the core to enhance portability between microcontrollers with same cores.

We won't meet with this layer directly with the exception of the case when after entering debug mode we reset the device from the development environment. After reset the execution of the application is stopped at the reset handler which is implemented in the startup code (and as such be a part of CMSIS). More information on the startup code can be found in chapter 6.3.

¹⁹ The naming has historical reasons. The EFM32 family of microcontrollers (and thus the firmware library made for them) was designed by a Norwegian company called Energy Micro. This company was later acquired by Silicon Labs.

8. Test questions

Embedded operating system in general:

1. What are the three main task states? What are the possible state transitions between them (assuming a preemptive scheduler)?
2. Describe the endless-loop task structure!
3. Describe the single-shot task structure!
4. If the tasks have shared memory they can communicate with each other easily. However there can be problems with shared memory. What could be wrong?
5. Let's consider two tasks communicating with each other by using a global variable. How can be avoided the problem of shared resources in this situation?
6. What is the semaphore? How can we use them?
7. The OS uses a periodic timer (called the heartbeat timer) to keep track of time. The time interval between two consecutive heartbeats is called a time tick. How accurate are the time related services of the OS?

FreeRTOS:

8. How can we configure FreeRTOS?
9. When does the FreeRTOS scheduler start?
10. What are the FreeRTOS tasks?
11. How can two FreeRTOS task communicate with each other?