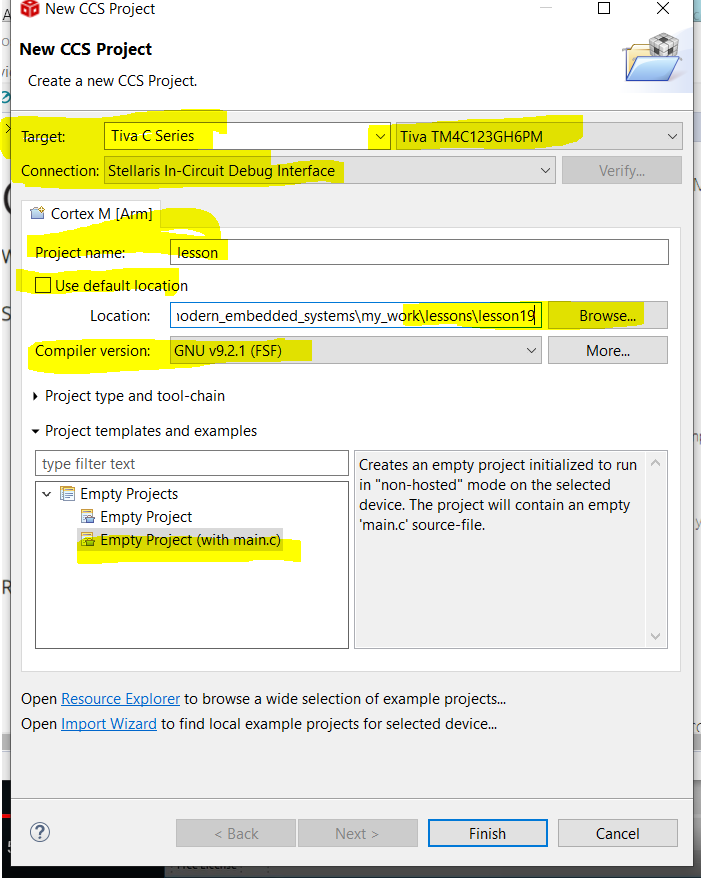
**Lesson#19: GNU-ARM Toolchain and Eclipse IDE**

* Set workspace in mmy\_work\_css and Use this setting when setting up the project



* After creation pf project, One very important thing, the following are the setting of GNU Compiler and GNU Linker
  + **GNU compiler:**
    - **Command:** **"${CG\_TOOL\_GCC}" -c**
    - **Command Line Pattern:** ${command} ${flags} ${output\_flag}${output} ${inputs}
    - **Flags:** -mcpu=cortex-m4 -march=armv7e-m -mthumb -mfloat-abi=hard -mfpu=fpv4-sp-d16 -DPART\_TM4C123GH6PM -I"D:/E Drive/E drive/Embedded/modern\_embedded\_systems/my\_work/lessons/lesson19" -I"C:/ti/ccs/tools/compiler/gcc-arm-none-eabi-9-2019-q4-major/arm-none-eabi/include" -Og -ffunction-sections -fdata-sections -g -gdwarf-3 -gstrict-dwarf -Wall
  + **GNU Linker:**
    - **Command:** "${CG\_TOOL\_GCC}"
    - **Command Line Pattern:** ${command} ${flags} ${output\_flag}${output} ${inputs}
    - **Flags:** -mfpu=fpv4-sp-d16 -DPART\_TM4C123GH6PM -Og -ffunction-sections -fdata-sections -g -gdwarf-3 -gstrict-dwarf -Wall -mcpu=cortex-m4 -mcpu=cortex-m4 -march=armv7e-m -mthumb -mfloat-abi=hard -mfpu=fpv4-sp-d16 -DPART\_TM4C123GH6PM -I"D:/E Drive/E drive/Embedded/modern\_embedded\_systems/my\_work/lessons/lesson19" -I"C:/ti/ccs/tools/compiler/gcc-arm-none-eabi-9-2019-q4-major/arm-none-eabi/include" -Og -ffunction-sections -fdata-sections -g -gdwarf-3 -gstrict-dwarf -Wall -specs=nosys.specs

|  |  |  |
| --- | --- | --- |
| Time | Capture | Create |
| 8.21 | Difference between the previous and this environment | Startup code File **tm4c123gh6pm\_startup\_ccs\_gcc** uses propietery names and are not compliant with CMSIS thus you need to change the interrupt names every time you use this in the vector table of the above mentioned file also the new vector table’s prototype as well  By default all the definitions of exception handlers are set to as endless while loop, which means when they occur they will hang the system and present the denial of CPU which is not a production-grade code  This generated code also contains the file **tm4c123gh6pm.lds** which contains the which is the linker script that tells linker where RAM and ROM is located and where to place various program sections  In lesson#14 we have seen the linker script  It allocates the stack as the LAST section in the RAM, which actually a mistake, because stack grows the lower addresses on ARM, so stackoverflow can damage the RAM sections above it  FACT: This was the most likely failure in the infamous TOYOTA UNINTENDED ACCELERATION CASE, read article-> “are we shooting ourselves in the foot with stack overflow” |
| 9.39 | Since the generated code are not usable then what we do? | 1. Copy lesson #18 **main.c, startup\_tm4c.c, TM4C123GH6PM.h, bsp.h, bsp.c** 2. Since u have 2 starup files now, so delete previously generated 1, **tm4c123gh6pm\_startup\_ccs\_gcc** 3. Now when you run the rebuild the project you will face errors |
| 13.22 | What are the generated error and how to deal with them | 1. core\_cm4.h file could not find as it was in IAR Workbench 2. Locate the **core\_cm4.h** where all the source file are being located copy the directory under which core\_cm4 is present named CMSIS, paste into the folder where you keep the lessons of this video course 3. But you need exclusively add this project in GNU Compiler by going into **Project->Properties->GNU Compiler->Directories ->Add path ->brows->select include file** 4. The problem of the above path is that you need to remove the absolute path, measn it will work only on your system, thus we need to convert to relative path 5. Eclipse IDE allows you to create relative paths by means of system variables. 6. Go to Variable-> **${PROJECT\_LOC}** that will create **paths** relative to the project locations then, after project\_loc 7. You need to append on path /../CMSIS |
|  | What and Why there are more errors persists and how to solve them | Since more error are coming from a source file, the reason for errors is that thethey source file is compatible to IAR toolset not with GNU Compiler  Use the startupcode from course lesson with gnu specific extension with the C language, this startup code must match with the linker script, so that is why you need to include the matching linker script together with the C code of source file into the project 19  **-startup\_tm4c\_gnu.c**  **- tm4c123gh6pm.lds** (linker matching script)  Remove previous startup C file and its linker script file which is  **- startup\_tm4c.c**  **- tm4c123gh6pm.lds** |
| 15.15 | Understanding of new startup code | Latest startup file is created from the CMSIS template for the specified device, the file has   1. Cortex-M Processor fault exceptions 2. Cortex-M Processor non-fault exceptions 3. external interrupts 4. GNU SPECIFIC EXTENSION WHICH TELLS THE COMPILER, to place the following symbol whihc is the vector table in the specified section 5. This specified section is the .isr\_vector() inside the SECTIONS in the new gnu linker script .lsd, here in this file it linked to the first section of the ROM which 0x00000000, specidying what ARM CPU needs, follows microcontroller specifications 6. You can match vector all vector table components from .c starup file to its linker file, where it interface with the memory 7. Instead of putting stack at the last section of the RAM, place it in the first section of the RAM, since Stack grows upwards to 0x20000000, so stack will not damage memory places after 0x20000000, The Additional bonus we can have, a stack overflowing into the unmapped memory in BELOW RAM will be detected automatically by executing the hardfault exception |
| 17.05 | Can we change the stcak size using linker script? | Yes we can do that by adjusting the stack size at the top of the linker script on the variable STACK\_SIZE= …. Variable  Variables that provided by the linker use with extern keyword in C file, because it is necessary to tell the compiler about the specific word  The vector table has stack\_end and then all handler functions of CORTEX-M exceptions and interrupts  In C startup file: Remember we use weak alias if any handler is not DEFINED in the file or other related file, here if any exception is not **DEFINED** it will be automatically replaced by Deafault\_Handler Implementation as **Cortex-M Processor non-fault exceptions** which is the GNU specific attributes weak and alias , weak tells that that if function is defined the alias will not be active, alias is the **default handler** function |
|  |  | Not all handlers are alias means they don’t have default handler, because the standard fault handler are actually defined in the startup code, so you don’t need to define it in the application |
| 18.40 | What is inside in the default handler? | Fault handlers are defined in in startup code  Definition of fault handlers: These fault handlers are not mindless for loops, the fault handlers definition use inline assembly which carefully avoiding the usage of stack and loading values through (startup fileline #421) r0 and r1 to assert\_handler(). Also the hard  Here you find another gnu specific handler line #57 **\_\_attribute\_\_((naked))** which instructs the compiler **not to do any stack operation for this function**  In bsp.c file You need to remove the **\_\_stackless keyword** before the function which in IAR toolset was a function to avoid the usage of stack, here since we use GNU toolset **\_\_attribute((naked))** will be placed  Another function call from main.c **\_\_iar\_builtin\_enable\_interrupt()** which is IAR tool specific which is used to enable the PRIMASK bit to enable counter need to be replaced, GNU function for that purpose is **\_\_enable\_irq()**  **CONGRATULATIONS: YOU HAVE SUCCEFULLY FINISHED FIRST PORT OF DEEPLY EMBEDDED CODE FROM ONE TOOLSET TO ANOTHER** |
|  | Get familiarize with debug of CCS and modify the behaviour of code | This concludes the lesson about switchiung the toolset from IAR ->GNU ARM and Eclipse based Code Composer Studio  The startup is much more closer to production quality than much closer to production quality than the typical code distributed by silicon vendors, the code is compliant with CMSIS and work with any toolset based GNU-ARM not just with CCS  You can adapt this for any ARM CORTEX-M Microcontroller |

**Lesson #20: Race Conditions: What are they and how to avoid them?**

|  |  |  |
| --- | --- | --- |
| **TIME** | **CAPTURE** | **CREATE** |
| **3.09** | What are race conditions and why and how to avoid them?  How to create a new project |  |

**Step#01:** Copy from previous lesson, open workspace from windows and delete the previous project and then file->import

A computer screen shot of a computer screen

Description automatically generated

**Step#02:** Browse->select pasted lesson folder->select file-> finish

A screenshot of a computer

Description automatically generated

|  |  |  |
| --- | --- | --- |
| Time | Capture | Create |
| 5.50 | Chang the code inside main.c of GREEN LED | Till now using the data bits register “**DATA\_Bits”** that have been explained in lesson#07  Now we replace it with and using the register whole data register that controls all the bits from 0-8  Here we are employing the bitwise operator, if you have ambiguity about it go to lesson#06  Basic structure of expression is built by reading the databait and logically OR with green led bit to ON the green led and apply local AND NOT on green led to turn off the led  The read modified sequence can also be performed by using the following operator  **|= OR()**  **&= ~ AND(NOT())**  The code is working as regular upon debug  But Blue LED seems to be on for longer period of time irregularly |
| 7.45 | Why Blue LED is blinking irregularly  Here we will only answer how do enable systick interrypt manually | * Lets set BP at where LED green is off main.c * Open disassembly view registers and monitor * Single step through assembly and monitor core registers * Here bic opcode clears the Green LED bit in the R2 register but before you clear this line lets trigger the SysTick interrupt manually * We trigger the interrupt manually: registers->NVIC->NVIC\_INT\_CTRL->NVIC\_INT\_CTRL\_PENDSTSET<-WRITE 1 HERE * In disassembly view place place brake point immediately after the bic instruction and 2nd in SysTick interrupt * Run the code |
| 9.15 | Why Blue LED is blinking irregularly  We found out that it was condition called **race condition.**  Since the interrupt has higher priority so it can go on changing the interrupt, here in our case, in main.c when led status of blue led being read, after that interrupt appeared from bsp.c and toggles the blue led, which when comesback with the toggled led, now when write operation is being performed the function have loaded up the previous value of R2 register before function interrupt | * Upon running: the systick handler code break point hit which means that we successuly trigger the interrupt manually * Its time to examine SysTick handler code which are ldr, ldr, eor, str * Now do assembly single step through systick handler code * Here I found that inside function instruction, bx code returns to the where the green led code is in main.c file inside while loop, on that line where we are turning green led off have bic instruction, bic clears the green led bit in R2, then stores the R2 in GPIO data register. * Theoratically only green LED must be turned off but actually Blue led also got turned off * Why does it happen? Because the assembly code that turns off the green led is a sequence of **read-modified-write** operation, **but the sequence was interrupted after the read, but before the write, the interrupt change the status of GPIO data register, by turning the blue led on, however interrupt code is unwaware of this change and still use the previous value of GPIO data register stored in R2** |
| 11.11 | **WHAT IS RACE CONDITION AND WHY IT IS SUCH A BIG DEAL?** | It can happen when 2 pieces of code shared a recource and preempt each other in a way that the result is depenedent on the sequence of executiong of the code  Now imagine having instead of having blue LED you have cooling system placed, when ever nuclear reactor reaches to some temperature your BLUE LED AKA Cooler turned ON and immediately due to the shared recourse, the result is catastrophical and result in death  These bug can appear very rarely thus it may appear in your application as well |
| 14.40 | How to avoid Race Condition? ->Mutual Exclusion | This method make sure that 1 piece pf concurrent code can be executed while accessing the shared resource  In the case of blinky program, you can implement Mutual Exclusion by simply disabling interrupt around turning Blue Led ON and around turning it off, by actually applying it on GREEN LED hwihc has BLUE LED toggling  The function you use to enable Interrupt was **\_\_enable\_irq()** and the function for disabling it is **\_\_disable\_irq()**  The section between enabling and disabling is called the critical section  This critical section serialized the action of shared recourse (GPIOF Data in this case )  With critical section is placed 3 pieces of code, the SysTick interrupt and 2 critical section can either run before or after each other but not in the middle |
| 16.17 | But how to avoid race condition in a first place ? By not sharing the recources -> by not sharing | What does the meaning of not sharing the recource?  The current code is accessing the DATA bits array of 255 GPIO Resgiter, the **lesson #07** ies devoted for explaining how did this array of 255 registers work  Array provides separate section for every possible combination of the 8 GPIO Bits so the DATA\_Bits[LED\_GREEN] is different than the DATA\_Bits[LED\_BLUE]  This means that there is no sharing of the common DATA register, there is no need for read-modify-write sequence  Here instruction of the code uses atomic write to the dedicated DATA\_Bits register |
|  |  | You finally understand why the hardware engineers design the GPIO REGISTERS IN THIS PRECULIAR WAY, they did exactly to separate GPIO Bits to avoid the need of sharing them and thus eliminating the need of potential race condition in the software  The data registers are designed in a way that they are designed to share to avoid potential race conditions  The designers of Texas instruments did the heavy lifting in hardware so software engineers life can be easy |

**Lesson #21: Foreground-Background Architecture ("Superloop")**

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **0.51** | We will learn about Background Architecture -> SuperLoop == main+interrupt | It is very important to understand before learning RTOS |
| **4.23** | Installl the keil mdk software and copy the lesson 22 | This lesson uses **qpc** file and new IDE KEIL MDK    The reason to change is that Miro ran into the problems with CCS 7.3 which was blocked by his anti-virus software  It is easier than to use Eclipse |
| **7.50** | The difference in code of here | Now the led remain ON for 1/4th of second and OFF fro 3/4th of a second  Here BSP\_delay dependent on SysTick interrupt, it produce more accurate timing because it doesn’t depend on the speed of compiler generated code  The systick interrupt is programmed to fire at the rate of BSP\_TICKS\_PER\_SECOND which is defined as 100 times per second in yje bsp.h header file  This BSP\_TICKS\_PER\_SECOND constant use to configure the SysTick interrupt  The SysTick interrupt only use to increment the 1\_tickCtr a kind of counter for loop  BSP\_tickCtr only simple reads and return the current value of I\_tickCtr only in the critical section of the program to avoid any race condition between BSP\_tickCtr and SysTick  From our previous lesson we know that, accessing such variable must only be performed in critical section (**INTERRUPT DISABLE** )  The function BSP\_delay() takes returning value from BSP\_tcikCtr and stores into **start,**  then a true while loop constantly reads the **(**BSP\_tickCtr() value – old initial value or the value from BSP\_tickCtr() which assign to start**)** which runs the code till the ticks doesnot become less than (BSP\_tickCtr-start)  The loop keeps running till the difference remain smaller than specified number of clock ticks remain  2s compliment handle the discontinuity properly which it receives when the values get rolled over 0  BSP\_delay() function also waste the number of clock cycles  The current program has all the characteristic of Foreground / Background architecture == main+ISRs which is very common in modern embedded systems  The architecture consist of two main parts:   * Background: The endless background loop inside the main.c * Foreground: Interrupt handlers which is SysTick handler |

**Explanation of Background and Foreground:**

Because A diagram of a process

Description automatically generated

The ISR run in foreground upon the Preempt

|  |  |  |
| --- | --- | --- |
| TIME | CAPTURE | CREATE |

|  |  |  |
| --- | --- | --- |
| 9.50 | How Foreground and Background communicate with each other ? | They communicate with each other by the shared variable (l\_tickCtr)  To avoid race condition due to the Preempted background loop, this shared variable must be **volatile** and must be protected by briefly disabling the interrupt around any access to them from the background  The shared variable must be declared as **volatile**. This tells the compiler that the variable can be modified outside the normal flow of control (e.g., by an ISR) and prevents the compiler from optimizing accesses to this variable.  **Interrupts must be briefly disabled** during access to the shared variable in the background loop to prevent a race condition. This ensures that the ISR does not preempt the background loop in the middle of an operation involving the shared variable.  Tasks that require **strict timing** (like reading sensor data at precise intervals or generating pulse-width modulation signals) should be handled in the **ISR** because the foreground is more predictable in terms of timing.  The timing of the execution of the various function calls from the background loop is not well defined meaning that every interrupt doesn’t necessarily be of exact time, because it depends on the time inside the loop that typically varies from one pass through the loop to another due to conditional branching in the code and interrupt activity  For this reason any ISR operation of strict time constraint cannot be reliably performed by background loop and must be pushed to the interrupt level running in the foreground, however this tends to make the interrupt longer and they might start interfering with the background loop and with each other  Despite its limitations, the **foreground-background architecture** is widely used because of its simplicity. Many embedded systems, especially in **consumer electronics, home appliances, toys, and remote controllers**, are designed using this approach since these systems often do not require complex scheduling or multitasking.  In summary, while the **Foreground-Background Architecture** is simple and suitable for many embedded applications, careful management of shared variables, interrupt lengths, and time-sensitive tasks is required to avoid timing issues and maintain system stability  This architecture Also used in Arduino |
| 11.35 | How superloop architecture implemented in Arduino | We take example of Blinky program of Arduino  At first glance you might not recognize it as superloop because Arduino hides it inside library  Lets take a look inside main function of Arduino |
| 13.45 | Lets re-organzie the code | Take all board specific codes like on and off and initialization of registers move to bsp.c since they are board specific and replace with function call, which is ebing defined in bsp.c  Inside bsp module we can a function to turn on and off leds of various colors  Once the functions are being defined you can simply **call**(led\_on()😉 them from background loop  Add the prototypes all the new bsp function in bsp.h file  Remove all the board specific stuff from main.c file such as led pin number, mcu header file and paste in bsp board specific file  The end-effect is, you main code is completely insulated from the board  The background code only specified **WHAT NEEDS TO BE DONE** and the bsp code specifies **HOW TO DO IT**  For saperating task in what and how?  Reason is : main code is smaller and self explanatory, you can run this code on a different board or you can use a different development toolchain, **but obviously you would need to provide different bsp implementation in the bsp.c file but you don’t need to change the single line of your main.c application code** |
| 15.10 | Lets examine the code now | Question is where our blinky program spends most of its time  BSP\_tickCtr <- BSP\_delay<-main  The program is doing nothing else than delaying its execution  When the event is arrived the code naturally progresses to handling the event because the code downstream the blocking call, provide the right context for an expected event  The code is sequential because the sequence of expected events is hard coded in the sequence of instruction  Blinky program expects ¼ second after turing the green led on, and another timeout event of duration ¾ of second after turning the green led off |

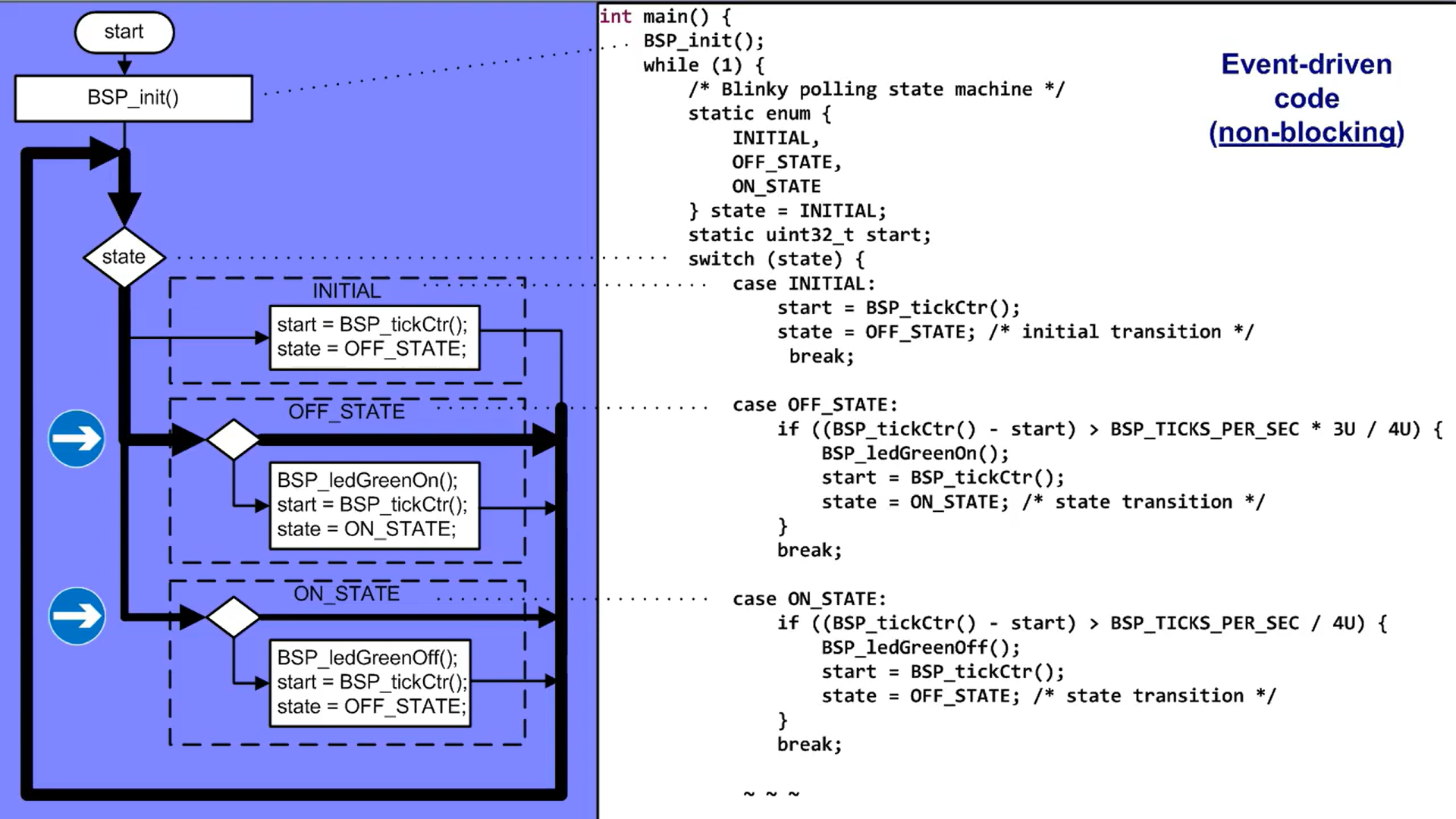
**Code Working:**

A screenshot of a computer

Description automatically generated

**Spend most of its time on BSP\_delay()**

|  |  |  |
| --- | --- | --- |
| Time | Capture | Create |
|  | How to arrange code differently in a non-blocking fashion, without the polling loops that busy-wait for specific events | When we enable non-blocking code we found out that, now the code does not spend most of its time in BSP\_intCtr, instead on main functions  The non-blocking main loop spends hundreds of thousands times per second, instead of only once per in a sequential code  This means that main loop can handle events as soon as they arrive the order in which they arrive  In other words the non-blocking code is driven by events, therefore I would call it event-driven  But there is a price to pay for this increased flexibility and timelessness of such non-blocking code and that is apparent here which is complexity  The event driven code is more complex, because the sequence of event this code can accept is no longer hardcoded in the sequence of instructions  The non-blocking code (main.c) structured here as a polling state machine but this is not common  In the majority of real life project you would see rather convoluted and deeply nested IF-THEN-ELSE branching based on the value of many global variables, also known as spaghetti code or a “big ball of mud”  This is part of state-machines which I will go in detail, but the important thing here is to note that they are fundamentally important |



A screenshot of a computer program

Description automatically generated

This concludes this lesson with foreground-background architecture, which is fundamental to understand all other architectures that use in embedded softwares

A diagram of a business process

Description automatically generated