Introduction

Welcome to Enginaator 2019 electronics engineering competition. This is the electronics stamina task – build an alarm clock. We at Artec Design have built you a development kit. This kit contains the PCB, components and documentation required for this task. The kit is yours – you can keep it after the competition.

The rules of the competition:

- All of the components needed for this task are included in the package. **Do not use any additional components** sourced elsewhere. You may use your own tools.
- You are encouraged to use all the information available to you. Google is your friend.
- You are not allowed to use any external help. **You may not ask any questions or have anybody else not part of your team do any work for this project**. You are allowed to read any information already present, but not information that gets created because you asked.
- **You must write all of the code yourself.** Only exceptions are the microcontroller HAL library provided by ST and pieces of code in this document. **Do not use any other libraries**. Do not copy-paste from internet or other sources. You are encouraged to research existing code on the net, just do not copy it.
- The tools needed to complete the task are provided by the organizers. Some tools (ie oscilloscopes etc.) are shared by teams. For shared tools, do not monopolize them; let the other teams use them too.
- Listen for the judges advice and commands.

The assignment is divided into multiple smaller tasks. Check for the scoring of the tasks, your aim is to get the highest total score (scoring summary is at the end of this document). If you mess up a part of the stuff, you still can work on other tasks and get some points. This is not all-or-nothing competition. The jury may award you partial points for a task, if the requirements are partially met.

This is a competition; the complexity of the tasks is deliberately quite high. Time is very limited and the organizers do not assume that you complete all of the tasks during the given time – on the contrary – we hope that you complete about a half of the tasks. Choose wisely, what you are going to develop. Plan the order of operations, so that your teammates would not wait for one person completing a task.

The competition is intended for teams of 4 people. For the best result, a good teamwork is needed. All members should contribute to the work and the tasks should be given to the most suitable person.

Once the time is over, **you hand over your device** and **USB flash drive.** You'll get your device back after the review. The USB drive shall contain your source code and answers to written tasks, the list is provided at the end of this document. Afterwards, all teams are individually asked to present your work in front of the judges.

All of the components used for the task have public datasheets on the web, copies of most of these shall be on the USB drive. These documents contain all the information about them – what they are, how they work, what the pinout is and how the chip is marked. Find the information and read it. We know that you haven't used these things before – now it's time to learn.

If you believe there is a mistake or something shall be explained better, ask the judges. If we find your question appropriate, we will answer it publicly for all teams.

This project is Open Source Hardware. The PCB design files are provided to you on the USB drive and will be published at Artec Design's Github repository after the competition.

Good luck! Design the best product!

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

Safety and soldering

The work for this task includes soldering. To avoid burning yourself, others or items, please read this paragraph.

- **Switch off** the soldering iron when you are not using it. Having unsupervised hot items around is a hazard. Also, the soldering iron tip oxydizes when hot, so switching off will preserve the life of the tip. A soldering iron with oxydized tip does not solder well.
- **A hot and a cold soldering iron look the same**. Never assume that the soldering iron is cool before you check it. Do not grab the hot end of the soldering iron even if you assume it is not hot. Do not try to catch a falling soldering iron mid-air, pick it up after it has landed on the floor.
- **Keep your workspace around the soldering iron clean.** Cables, tools and other items nearby may get accidentally burned.
- Soldering makes things hot. (That is the point of it) **Do not hold metallic items that you are soldering with your fingers**, you may burn your fingers. There are **tweezers** to hold things. (Or hold on to items further away or from non-metallic parts).
- Electronic components do not tolerate excessive heat. Things may melt, die from overtemperature internally or otherwise get broken. **Keep the heating to minimum.** Heat up, add solder and quickly remove soldering iron.
- **Do not solder when the device power is on!** Always unplug the USB cable before touching anything with soldering iron to avoid ESD damage!
- **Keep the tip of the soldering iron tin-coated.** Solder on the tip conducts heat; a dry soldering iron tip does not work well. There is a wet sponge or brass wool to clean the tip. In case of orange sponge, it must be moist (go and make it moist at the tap if it is not). When the soldering iron tip has contaminants on it (black burned stuff), wipe it on sponge or brass wool and immediately apply new solder on the tip.
- **Add solder to the component**, not to the soldering iron tip. The best practice is that you heat the location you want to solder (component lead and PCB pad) with the soldering iron and then add solder to the component lead or PCB pad with your other hand. Do not try to transport the solder using the soldering iron tip to the soldering location, it will oxydize while hot and may cause bad solder joints. Keep the tip thinly covered with solder, but do not have a large blob hanging on it.
- Check that you solder the components on the board **the right way around** and **on the correct side** of the board! Each item goes on that side of PCB where its designator is printed (for example, "J11"). On the PCB, pin 1 is indicated on the white silkscreen print. For larger components, **pad for pin 1 has rectangular shape**, others are round. All the chips should have marking near pin 1. If you are unsure, download the datasheet and check the documentation, how pin 1 is marked!
- There is **soldering flux** at your disposal. Flux is a gel, that keeps the things from oxydizing. This makes the solder flow better and smoother. The solder wire itself contains some flux (fluxcore), so the additional flux is usually not needed for larger items. For smaller surface mount things, adding extra flux makes soldering easier. Flux residue can be removed with cleaning spray.
- There is **soldering wick** at your disposal (the copper thing looking like braided wire). This can be used to remove excessive solder or to un-solder things (this may be quite tricky). First, cut off the already used part of the wick if it has been used. Put the end of the soldering wick on the pad with unneeded solder. Heat the wick with soldering iron and it will wick in the solder. Then remove the wick and soldering iron at the same time (otherwise the wick may stick to the piece).
- There is **wire** at your disposal. It is there for you to do repairs on the board. If you rip a pad or a track off the board, you can try to fix it by making the connection with wire.

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

Surface mount soldering

Tasks 5 and 6 contain surface mount soldering. Here are some hints to succeed:

- **Clean** the tip of the soldering iron and apply a very thin layer of tin on it.
- **Tin** one of the pads on the board. For bigger components, this shall be one corner pad. (If you have already messed around, remove solder from all of the pads but one)
- Add some **flux** on the pads on the board. For example using a toothpick or other sharp-tipped object. Having the solder flow under a layer of flux makes things so much easier.
- **Grab** the part with tweezers and locate it where it goes.
- Apply some **heat** with soldering iron on the pin which has the tinned pad. The pin shall sink into the solder and lie flat on the board.
- Now the part is supposed be fixed to the board. Put down tweezers, grab the solder wire to the other hand.
- **Solder** the opposite corner pad.
- Solder the rest of the pads.

If you have enough flux and not too much solder, you can touch multiple pins at once with soldering iron and drag away from the chip, the tip of the iron does not have to be needle-sharp. The pins get soldered and you do not get any shorts. The prototype was soldered using only a single 1.8mm chisel tip soldering iron.

If you get a blob of solder on the board, that short-circuits multiple pins or otherwise needs to be removed, use the soldering wick. First, add flux; this will help the wick to work. Then remove the blob with wick. Finally re-solder the area correctly.

The schematics of tasks 5,6 contain a ferrite bead (FB) at the power wire. Solder the ferrite bead after everything else; it can be left unplaced or removed if the soldering is so messed up that it would short the power supply.

There are extra SMD passive components. If you loose or break one, you have replacement.

The passive components can be distinguished by color: **resistors are black, capacitors are yellow and ferrite beads are gray**. We have only one value for each type, so you do not need to distinguish between values. The packaging tape is marked with color on reverse side by us: resistor packaging has **black stripe,** capacitor packaging has **red stripe** and ferrite bead packaging has **green stripe.**

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

1. Microcontroller module bring-up

This task of the project gets you familiarized with the hardware. You will install the tools and get some basic functionality running.

The kit comes supplied with **STM32F031K6** microcontroller module. The microcontroller on the module is a 32-bit ARM Cortex-M0 based controller having 4kB SRAM and 32kB flash memory, operating up to 48MHz.

For this project, it is required to use **Atollic TrueStudio**. For pin-muxing and peripheral setup you are required to use **STM32CubeMX**.

Additional information about the microcontroller can be found on STMicroelectronics web site. The information you may need is divided into multiple documents:

- **UM1727**: Module getting started guide.
- **UM1956**: Module user manual.
- **STM32F031K6 datasheet**: Microcontroller chip hardware-related documentation (pinout, voltages, timings etc).
- **RM0091 Reference Manual**: Programming-related documentation (peripheral descriptions, registers etc).

1.1. Install software

This project is designed to be developed using Atollic TrueStudio. TrueStudio is a professional IDE, that was recently released as free to use on ST microcontrollers. TrueStudio is used to develop the firmware, load it into the device and debug the running code.

In addition to that, you are using STM32CubeMX to configure the chip and generate project file for TrueStudio. It is a very useful tool to plan the usage of the microcontroller internal peripherals and the muxing of peripheral signals to microcontroller pins. By using this tool and STM32 HAL library, you do not need to write code to directly access hardware registers (although you may if you want to). Using intermediate libraries such as HAL in actual projects may be not always the best choice, but for quick prototyping, they spare you a lot of development time.

The STM32CubeMX requires Java to be installed on system. Install jre (Java Runtime Environment) in case you do not have it.

For this project, STM32CubeMX needs "STM32Cube MCU package for STM32F0 Series". If you have internet connection, it will be downloaded when needed. Alternatively, we have provided the zip to you. You can import it via Help → Manage Embedded Software Packages → click From Local.

To display the debug information sent via USB-serial port, you need a serial port terminal emulator program. Install your favorite if you already do not have one installed. For example, TeraTerm.

- Install **Java Runtime Environment** in case you do not have it already.
- Install a **terminal emulator** application.
- Install **Atollic TrueStudio**.
- Install **STM32CubeMX**.
- Import **STM32Cube MCU Package for STM32F0 series**.

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

1.2. CubeMX project

Create the STM32Cube project with correct module:

- Click "Access to Board Selector".
- Open "Type" submenu at left side and select "Nucleo32".
- Select NUCLEO-F031K6 at bottom menu.
- Click "Start Project" at top right corner.
- Click "No" on "Initialize default peripherals".

You get the project window. The MCU chip (STM32F031K6T6) view is shown on the screen. A couple of pins are already configured, these are the internal connections of nucleo-32 module. Do not change the configuration of these.

The STM32CubeMX tool will be used during this project to configure the pins and peripherals to suit your needs. The microcontroller has lots of functionality inside, but not so many pins exiting the chip. To overcome this limitation, the microcontroller has configurable multiplexers (switches) on the pins, the pin can be connected to one of many internal connections. Each internal connection can only be connected to a couple of pins, so care must be taken to be able to connect up all functions you need. This process of configuring the pin multiplexers is called pin planning.

The CubeMX tool shows the view of the microcontroller chip, that is on the Nucleo-32 board (U2). Most of the pins of the microcontroller are wired to the pin headers on the sides of the module and from there will connect to the project board at JS11 and JS12 (see board schematics PDF). The project board has these microcontroller pins connected only to pin headers J11 and J12. The project board comes supplied with jumper wires. It is your task to connect J12 and J12 pins to the rest of the schematics using the jumper wires. **Not all microcontroller pins are usable** and there are some parallel-connected pins, only one of these can be used! Check the board schematic for details! At the end, you'll probably use almost all of the pins, so plan them wisely.

The microcontroller contains a number of peripherals (SPI, UART, timers, etc.); there may be multiple ones of same type. Before using them, check for the differences. For example, the timers are not the same and you may need some specific features for some tasks. Also, multiple peripherals may conflict using the same pins on the chip, so you may have to shuffle things around to fit all your functionality.

1.3. Blink the LED

The Nucleo-32 module has an on-board LED connected to PB3 pin of the microcontroller. For this exercise, the module does not have to be inserted into the project board, just the bare module is needed.

- **Save** the CubeMX project using the "File" menu at the top. (Create a new folder for the project beforehand). Saving will trigger downloading of additional packages if they are not already present.
- Map the PB3 pin as GPIO output. To do that, click on PB3 of the chip and select **"GPIO_Output"** from the drop down menu. Right-click the pin and set up its **user label** as "LED".
- For further configuration options of that pin, click on **"GPIO"** peripheral located in "System Core" section of the left side menu. At the **"GPIO Mode and Configuration"** dialog at the top middle, click on "PB3". Now the bottom middle part of the screen can be used to perform further configuration. At this point, we leave everything to defaults (Output push-pull, no pull-up, no pull-down, low speed).
- Before we generate the project, there are some additional settings to set up. Click on **"Project Manager"** at the top. Select "TrueStudio" as "Toolchain/IDE".

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

- Click the **"Generate Code"** button at the top.
- Now that the project is generated, start up TrueStudio. Eclipse-based IDEs use a concept of workspaces. Each workspace can hold multiple projects and stores the configuration of the tool. On start, the user is prompted to select a **workspace**. Accept the dialog, it will create one. For other future projects, you can create additional workspaces to start at a clean page.
- Import the generated project to workspace. Click **"Open Projects from File System"** from "File" menu. Press the "Directory" button and browse the directory containing ".project" file, under the CubeMX project folder.
- Open the **"Src/main.c"** file. This file contains the main function, that is launced on power-up. There are a couple of initialization lines already auto-generated, there will be more as you get further on with the project. The main function must never exit, for that it has an infinite loop at the end. The user code shall only be added between "USER CODE BEGIN" and "USER CODE END" markers.
- The settings you configured in CubeMX are already set up. There is auto-generated code at the bottom of main.c and inside main.h. Also, there are HAL libraries copied over.
- For this exercise, we add a couple lines after the **"USER CODE BEGIN 3"** line (and inside "while(1)" block) to blink the LED. This may be edited or removed later.

```
HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
HAL_Delay(500);
```

  This code will change the state of PB3 pin and then wait for 500ms. This shall blink the LED at 1Hz.
- To explore around, you can Ctrl+click on function and variable names. The corresponding source code for that function is opened up. When Ctrl+clicking on function name at function implementation, you are sent to function declaration and vice versa. All of the HAL library and project code is provided as source and you can look inside to see what it is doing.
- Press the **"Build"** button on the toolbar (with hammer icon). The firmware will be built; build progress is displayed at the bottom of the screen in "Console" window. First build is a bit slower, later re-builds are very fast.
- Open the **"Build Analyzer"** tab at bottom right. It will show the statistics of the code and data memory used. If it does not show anything, you may have to open "Debug" in "Project explorer" and click on your .elf file. When developing you application, keep an eye on "Build Analyzer" to avoid running out of space.
- To upload the code to the microcontroller, first connect the board to your computer using a USB micro-B cable. Once the drivers are loaded and everything is ready, press **"Debug"** button on the toolbar (with bug icon). The firmware shall be uploaded to the board.
- At the first time you use the module, you may get a debugger FW update prompt. Follow the dialogs to get the debugger FW updated, then try again to start the debug session.
- Press the **green play button** to start the firmware. The LED shall start blinking.
- Press **red stop button** to go back to editing mode once you are done testing. The firmware keeps on running. In case the red stop button is inactive, first select your ".elf" in "Debug" window. It may get un-selected sometimes.

The board has a proper built in debugger, that can be used to monitor the code execution. At any time, the code can be stopped by pressing the **pause button** and the state of the microcontroller examined. There is the "SFRs" tab on the top right, where you can monitor the values of the peripheral registers. The execution can be performed manually line-by-line by single stepping using the "Step Into", "Step Over" and "Step Return" buttons. Breakpoints can be set too – if the code reaches a breakpoint, it will stop and you

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

can examine the situation. To set or clear a breakpoint, right click on the line number in front of the code line you want to set the breakpoint at, then click "Toggle breakpoint".

The initialization code from CubeMX can be always edited and re-generated. All of the user-added code between the "USER CODE" tags is preserved, do not edit the tags or anything outside of them. As you may have noticed, the settings you configured in the CubeMX are just exported as code into the code at the bottom of main.c. This would be overwritten each time the CubeMX project is re-generated, so don't change anything there.

Alternatively, you could move the project to another folder and not have the CubeMX change your source. Then you can elliminate all of the comments and re-organize things as you wish, but to update you should let the CubeMX generate the code to a different folder and copy-paste the needed changes into your project manually from there.

For accessing the peripherals, you need the HandleTypeDef's, declared at the top of main.c. These are generated by CubeMX for each enabled peripheral; they are initialized at the bottom of main.c with the settings you configured in CubeMX. The peripheral numbers (ie htim1, htim2) may differ from the examples in this document; substitute these with the exact peripheral handles that you chose.

You may add additional source code files (.c) and split the functionality between files. To make the existing device HandleTypeDef's located in main.c accessible from other source code files outside main.c, extern declare them in main.h and #include the main.h in the source code file where you need them. For example, this can be added to main.h to export UART1:

```
extern UART_HandleTypeDef huart1;
```

Working code on the microcontroller (at least a blinking LED if not anything else)	4 points

1.4. Clock Tree

The STM32 microcontroller has a runtime-configurable clock tree. The chip may be clocked from different sources and it has an internal PLL, that multiplies the input clock rate. Nucleo-32 boards having STM32F-series microcontrollers (as opposed to the ones having STM32L-series) have no external clock sources fitted, thus the only options are the internal RC oscillators. The only option to supply the clock signal to the CPU and peripherals is 8MHz HSI oscillator (High Speed Internal).

The RC oscillators are not so precise as crystal-stabilized clock sources. The HSI oscillator at room temperature has accuracy of 1%. This is enough for non timing-critical tasks. For this project, the initial development can be done using that oscillator. There is a precise crystal-stabilized battery-backed oscillator on the project board (task 5), that can later be used to gain more accuracy.

- For this project, we want to configure the CPU to work at maximum speed. Before we do that, fix up the RCC settings to avoid potential misconfiguration issues. Go back to "Pinout and Configuration" and click on "RCC" on left menu at "System core" submenu. Set the HSE to "Disable", because on STM32F series Nucleo-32, the HSE is not connected.
- Click on "Clock Configuration" at the top of the CubeMX window. Set PLLMul to "X12". This should speed up the CPU to run on 48MHz (FCLK on right side of the tree). The "System clock mux" should now be set to "PLLCLK" and "PLL Source Mux" as "HSI /2".
- Press "Generate Code" to update the TrueStudio project.
- Go to TrueStudio, it should automatically reload the files that were changed.
- Recompile and restart the firmware (press "Debug"). The LED shall still blink at 1Hz as the SysTick timer frequency, which the Hal_Delay is using, is automatically set up correctly to 1kHz.

Be aware, that setting up some invalid clock settings may make the system unprogrammable, unbootable or unstable. Do not play around the clock settings! If this happens, the CPU should be rebooted to bootloader so that the bad user code would not boot. Then you can overwrite the FW to get out of the mess.

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

To boot to bootloader, the BOOT0 pin must be shorted to VDD during power up. BOOT0 pin is accessible at SB12 jumper link. VDD is at pin 14 of CN4 of the module.

1.5. Debug UART

To debug software more easily, it is important to get a transmitting debug serial port working. This could be used to send debug log to PC. Serial port debugging can show the operation of your firmware and can be used to reporting values to host computer.

- Open your terminal emulator application. Configure it to listen at the usb-serial (COM) port provided by Nucleo module, configure the settings to speed 115200 baud, 8 bytes, no parity, 1 stop bit.
- In CubeMX, configure USART1 to "asynchronous" mode. In parameters of USART1, set baud rate to 115200.
- Configure the microcontroller pin multiplexing so that the USART1 signals are mapped to pins connected internally on the module to usb-serial converter on module – PA2 as TX and PA15 as RX.
- Re-generate the program code in CubeMX.
- In main.c add the following pieces of code to corresponding locations for printf support:

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */

/* USER CODE BEGIN 0 */
int _write(int file, char *ptr, int len) {
   HAL_UART_Transmit(&huart1, (uint8_t*)ptr, len, ~0);
   return len;
}
/* USER CODE END 0 */
```

- Add a printf statement somewhere in the main loop along the LED flashing code:

```
printf("Test\r\n");
```

  This is raw serial port, the data is sent byte-by-byte. Terminal emulators usually require "\r\n" to get a new line.

- Start debug of the code. Your terminal emulator should display the "Test" string each time the LED changes state.

Working debug serial port	4 points

1.6. Solder Task 1 components

This is the first task, that requires a soldering iron. You need to prepare the alarm clock project board to accept the microcontroller module.

It is time to look at the project board schematics. We are going to solder the "TASK 1" components. These are two 15-pin sockets J13 and J14 to accept the module and two headers J11 and J12 for connecting the module to other parts of the boards using jumper wires.

- Open the schematic and study the "Task 1" part.

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

- Find the sockets in the component bag. Find the pin header strip. The pin headers may be provided as long strips. In that case, cut off pieces with required amount of pins for J11 and J12.
- The locations for J11..J12 and JS11..JS12 are indicated with rectangles printed on the board. Insert the item from the board side, that has the rectangle and component name printed on and solder the pins from the other side. First solder 2 corners, then the rest.
- Plug in the module.

Tip: to hold the sockets J13..J14 upright to the board, take a long piece of pin header strip and stick it between them, into holes, connecting them. This keeps them aligned and vertical while you solder the first pads.

Properly soldered Task 1 components.	2 points

2. LED Matrix Display

A clock needs to have a display to show time. This clock project has a LED matrix display.

This is a dynamic display – LEDs are not lit all the time. At any moment, only a single line is lit up. By switching between the lines rapidly enough (>75Hz), the user gets an illusion of a stable image – called "peristance of vision". Thus – for this display to show an image, the microcontroller must work all the time by rapidly changing between the lines.

2.1. Display hardware

The hardware is composed of 8 row drivers and 32 column drivers. The row drivers connect the corresponding row line to positive supply, column drivers connect the column line to ground.

As the display is made of LEDs, they light up only when anode is has higher potential than cathode. For a pixel to light up, the anode (row) must be high and cathode (column) must be low. All other combinations do not light up. When we switch only one row line high, the LEDs on this line can light up. Only these leds light up on a column, which are switched low.

The row drivers are inverted – by setting the control bit to 1, the driver is enabled, output is pulled low and LED is lit; by setting the control bit to 0, the output is disconnected and the LED is not lit.

The column drivers are not inverted – by setting the control bit to 1, the output is enabled and pulled high and LED is lit; by setting the control bit to 0, the output is disconnected and LED is not lit.

Both row and column drivers are shift registers. They contain a chain of D-flipflops. On each rising clock edge (CLK going low to high), every flip-flop gets the value of previous one (the bits are "shifted" up the chain). The first one gets the value from DIN signal and the last value is output on DOUT signal. This reduces the number of connections needed to 2 wires – CLK and DIN. Even better – the shift registers can be cascaded to a longer chain by DOUT->DIN connections, so only 2 pins are needed for a large number of output bits.

Next to the shift registers, the row and column driver chips contain holding registers (latches). They are there so that the previous outputs can be shown on display during the time the next ones are uploaded. The holding registers have an another wire (LE), that can be used to shift all data from shift registers into holding registers, all at once. The column drivers update the latches as long as LE is high; row drivers update on rising edge of LE. When connecting all LE wires in parallel, a single low-high-low pulse will update both.

After the holding registers, there are output drivers. The output drivers are high-current amplifiers, that can drive high loads. Row drivers have an extra amplifier chip U24 because each line must be capable of all the LEDs in the line. Column drivers contain an constant current driver for each line, thus keeping the LED current at set level and removing the need of series resistors for each line. The output drivers are enabled by OE# wire. The suffix "#" denotes that it is an low active signal (sometimes also indicated by prefix "n",

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

suffix "_N" or overline) – the output is on when the signal is driver low and off when high. This signal may be used to blank the whole display or control the brightness by PWMing the column drivers (see task 7.3).

Now it is time to solder the Task 2 components to board.

- Solder the rear side of the board (U2x, C2x, R2x, J2x). Verify the direction of the chip (pin 1 mark) before soldering it in! The legs of the chips may be bent a bit too wide to fit into holes. Push the chip sideways on the table to bend the legs more perpendcular to have correct spacing.
- Solder the sockets JS21..28 for the LED matrices to the front side of the board.
- Plug the LED matrices LD21-24 into the sockets JS21-28. Verify the direction, a small key (notch) is on the case on the side closest to pins 1-8. The key is indicated on the silkscreen print on the board, align the two.

Properly soldered Task 2 components	6 points

2.2. Connection

The pin headers J21..J23 contain all the control signals of the row and column drivers. These shall be connected using jumper wires to eachother and/or microcontroller module headers J11..J12. There are 2 pins for each signal this gives the opportunity to connect multiple inputs to one microcontroller pin. All input signals must be connected; not needed outputs (DOUTs) may be left unconnected.

The type of interface used here is called synchronous serial bus. As the clock and data are carried on separate wires, it is synchronous (as opposed to asyncronous interface having timing embedded on the data wire along the data). As the bits are sent using one line one after another, it is serial (as opposed to parallel interface having one wire for each bit).

Although it is feasible to bit-bang all of the signals using GPIO outputs, there is dedicated HW peripheral in the microcontroller to send and receive synchronous serial data - the SPI block (alternatively, USART could be used in SPI mode, but the only one is in use for debugging serial port). SPI can be used to send out a stream of data bits and toggle clock line during that; program code just writes bytes of data and the SPI itself breaks it apart to bits. A reasonable introduction to SPI (serial peripheral interface) is in wikipedia, you should consider reading it.

The pin-muxing of NUCLEO32 modules is quite limited due to the small 32-pin chip and already made connections on the module (PA5,6 are in parallel to PB7,6). To use SPI, I2C and rotary encoder, you can use PA5,7 for SPI and PA10,PB6 for I2C.

Reasonable settings for SPI block are: Data size 8 bits, MSB first, prescaler 16, Clock polarity low (CPOL=0), Clock phase 1 edge (CPHA=0).

For the LE and OE# signals for this task, you are advised to use GPIO outputs to get things working quickly. Later on, as explained in task 7.3 Display brightness, the OE# can be driven by PWM to control display dimming.

- Open STM32F031K6 reference manual RM0091. Browse through the SPI section.
- Open CubeMX. Enable and configure SPI. Set up pin multiplexing for SPI and GPIO outputs.
- Connect the corresponding pins on board using jumper wires.
- Re-generate TrueStudio project.

Usable CubeMX setup and connection to display	2 points

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

2.3. Display update timer and its interrupt

To get even illumination across the display, all rows shall be iluminated the same duration (1/8 of time). This requires some attention on the firmware side. It is possible to generate the timing in main loop only, but it would be much simpler to use a timer interrupt handler for it.

To calculate the timer frequency, you must first check the clock rate in CubeMX "Clock Configuration" page – the "APB1 Timer clocks" field on the right side tells you the timer input frequency. This frequency is first divided by the internal clock divisor value, then the timer prescaler frequency, that is set up at timer configuration page. Finally the result is divided by the timer counter period value in autoreload register. Both the prescaler and autoreload registers can be set in CubeMX, but they can also be updated in user code as needed. Both the **prescaler and autoreload values shall be set to (value-1)**.

To test out the timer, first configure it to 1Hz and printf a debug message from timer handler. Once everything is fine, remove the debug printf and speed it up as needed.

- Open STM32F031K6 reference manual RM0091. Browse through the timer and NVIC sections.
- Open CubeMX. Go to "Clock configuration" and check out the timer clock rate. It should be 48MHz.
- Go back to "Pinout and Configuration". Select your timer at the left side.
- Activate the timer. Depending on timer type, either check "Activated" or set "Clock source" to "Internal".
- Set Prescaler to "4799" and counter period to "9999". (48000000Hz/4800/10000 = 1Hz).
- Click "NVIC settings". Enable the interrupt. Set "Preemption Priority" to 1 to have HAL_Delay working inside timer handler.
- Re-generate TrueStudio project.
- Add timer handler code (replace 1 to your timer number):

```
/* USER CODE BEGIN 4 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
	if (htim == &htim1) {
		// call here actual handler function
		printf("TIMER!\r\n");
	}
}
/* USER CODE END 4 */
```

- Add code to enable the timer interrupts in the main function (replace 16 your timer number):

```
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start_IT(&htim1);
/* USER CODE END 2 */
```

- Compile and start the firmware. The debug serial port should display "TIMER!" every second.

Once you have verified that the timer is generating interrupts, you can continue developing the functionality described in next section and remove the debug printf.

The actual display refresh rate shall be above 75Hz (ie. around 100Hz) for the whole display refresh (8 lines), so the timer interrupt shall fire at around 800Hz. Set the timer interrupt preemption priority to 1 to support HAL_Delay() calls inside timer handler.

Interrupt timer working	4 points

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

2.4. Display driver

Now it is time to write the display driver, that scans the display matrix and displays pixels from frame budffer memory.

The important part to handle is data storage. We need to store 8 lines of pixels, 32 bits each. The data must be accessible line-by-line. The most appropriate data type in 32-bit microcontroller would be 32-bit unsigned integer (uint32_t). Thus the video memory can be declared as:

```
uint32_t video_memory[8];
```

The Cortex-M0 microcontroller core contains a barrel shifter. All 32-bit shift instructions (<< and >> in C) are very fast. Modifying a pixel in video memory is simple and fast:

```
void pixel_set(int x, int y, int val) {
 if (x < 0 || x >= 32 || y < 0 || y >= 8) return;
 video_memory[y] &= ~(1<<x);
 video_memory[y] |= (val<<x);
}
```

Of cource, it is possible to write multiple pixels more efficiently than one-by-one, but for this project this simple solution works well enough.

When using SPI interface, you need to access the 32-bit data 8 bits at a time. This can also be performed using shift operations, for example:

```
uint16_t high_byte = (video_memory[y] >> 24) & 0xFF;
```

To verify the hardware, it is wise to start small. Do not start with scanning the whole display, instead write a known pattern to column drivers and a single high bit to row driver to activate one row. Check that the result is the same as you intended. If not, use a multimeter or oscilloscope to measure the voltages and check what is wrong.

After you are comfortable of setting up one line on the display, add the scanning part. At fixed intervals, load new data from shift registers to holding registers, this changes to next line. Then start pre-loading the next line after that to the shift registers. You have set up a timer in 2.3, now you can use it.

The display scanning interrupt handler shall look like:

```
void disp_scan() {
 // - set LE high
 // - set LE low
 // - transmit via SPI the row and column shift register data
}
```

Example code for sending data over SPI:

```
 // Example: send 1 byte over SPI:
 uint8_t t = 0x42;
 HAL_SPI_Transmit(&hspi1, &t, 1, ~0);
 // Example: send 5 bytes over SPI:
 uint8_t tmp[5] = {0x00, 0x01, 0x02, 0x03, 0x04};
 HAL_SPI_Transmit(&hspi1, tmp, 5, ~0);
```


Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

The timer is in stopped state on boot. First you must start it, like shown in section 2.3. The OE# signal can just be set to GPIO output low value to keep the display enabled at all times:

```
HAL_GPIO_WritePin(OE_GPIO_Port, OE_Pin, GPIO_PIN_RESET);
```

There are a lot of advanced tricks that can be done in display driver for effects. For example, multiple framebuffers, offset framebuffers to scroll output, display some pixels only in half frames to dim them etc.

- Define video memory as global array.
- Write function, that sends one row line data and the given row selection to shift registers.
- Write functions to operates the latches and OE# signal.
- Set some bits in video memory and test your code.
- Integrate the written code with timer interrupt to automatically scan all lines of the display.

Display pixel data from video memory to the display	4 points

2.5. Fonts

As we are able to display pixels, we need to create fonts and write a renderer, that converts glyphs (letters and numbers) into pixels. But first, we need to know what glyphs we need and what is the geometry - how many glyphs you need on the screen at a time, what is the width of one glyphs etc.

- Figure out what glyphs you need and what are the glyphs sizes used (linked to 3.1 User interface design).
- As the requirements are known, draw your font(s).

For a task as small as this, you can just do it by hand. In old-fashion way, you may use graph paper and pencil. But using a computer will make things simpler. Either write 0's and 1's with fixed-width font, use Excel/Libreoffice Calc or whatever tool that fits for you.

After the font has been drawn, you need to encode it somehow to encorporate it into the firmware. A good thing to note here is that the font is 8 pixels high. 8 bits are in a byte, so one column can be stored as one 8-bit unsigned integer (uint8_t), where every bit encodes the value of one pixel in that column. The whole letter can be stored as an constant array of such integers.

Be aware that the amount of SRAM on your microcontroller is very limited. **You want to keep the constant data in flash**, which is much larger. All variables declared in C are usually placed in SRAM. If they have initial value, these initial values will be in flash too. But if you declare the variable as const, it will not be placed in SRAM, the compiler performs the reads directly from flash. So for example for storing fonts, you may want to declare arrays as:

```
const uint8_t my_glyph_2[4] = {
        0x62, // .**...*.
        0x51, // .*.*...*
        0x51, // .*.*...*
        0x4e, // .*..***.
 };
```

- Encode the pixels of the drawn font to numeric values, incorporate them to you firmware.

Font created and embedded into firmware	6 points

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

2.6. Font renderer, numeric output

To tie everything together, you need a bit more code. First, you need functions to render text to bitmap. It shall choose the correct glyph from memory and blit the pixels to the video memory, to a location provided by the caller. This function will probably consist of some nested loops, that extract pixels from font and write them to video memory (see pixel_set above).

You will probably need a function, that renders binary numbers. This function shall take a number as input. It shall extract the decimal digits using division and modulus operators (/ and %). For each digit, have the font renderer output corresponding digit glyph.

- Write font glyph renderer function.
- Write number renderer function.

Character and binary number rendering capability	4 points

3. Buttons and user interface

In this task, you create the user interface for the clock. This will include support for buttons and clock counting functionality. In addition to the buttons in Task 3, you have rotary encoder in Task 8 available if you choose to use it.

3.1. User interface design

For this task, you need to design an user interface for your alarm clock. Keep in mind, that you are expected to implement it as well, so the implementation you present in 3.5 should correspond to this design.

The design shall incorporate at least:

- Display clock time.
- Display date.
- Display temperature.
- Set clock time, date.
- Set alarm time.
- Enable/disable alarm, alarm enabled status indication.
- Alarm, shutting it off.
- Alarm automatic shut-off after a timeout (ie turn it of if you are not at home, but alarm was left enabled).

The design can use the 4 buttons SW31..34 and the rotary encoder SW81 (having turn and press functions) as inputs.

The preferred representation for this task is UML state diagram. You may use whatever tools you like to draw that (even a photo of paper with pencil drawing is better than nothing). This diagram shall describe all of the states the system can be in. State transitions (arrows between) shall be either button or timeout triggers. Each state shall have an description of the UI.

- Design a state diagram for the UI. Provide a PDF file for scoring.

Usable UI design, that corresponds to requirements	6 points

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

3.2. Button hardware and GPIOs

For the buttons, the hardware is very simple. You need to solder 2 headers and 4 buttons on the board. Then you need 4 jumper cables to connect them to microcontroller.

As you may have noticed, the schematic does not include pull-up resistors. The buttons connect the wires to ground; to get a signal, you need a resistor to pull the voltage back up when button is released. The microcontroller contains such resistors for each pad.

- Solder the Task 3 hardware to board. **Cut the SW33 mounting bracket so that it does not touch U22**, U22 has part of cut off metal lead frame sticking out on the side.
- Open STM32F031K6 reference manual RM0091. Browse through the GPIO section.
- Perform pin planning in CubeMX, select "GPIO_Input" pins for buttons.
- Activate pull-ups at GPIO configuration page.
- Connect the buttons using jumper wires.
- Re-generate TrueStudio project.

Button hardware soldered, CubeMX configured, jumper wires connected	2 points

3.3. Firmware for buttons

The buttons are mechanical devices. They contain a spring contact, that bounces when it gets closed and provides multiple short pulses before it stabilizes. To detect a clean signal, you need to filter out these spurious events. This is called debouncing.

Buttons could be handled in software in multiple ways. For example, you could set up an external pin interrupt and get called on button press. In this project, we already have a periodic timer interrupt set up at 2.3. You can hook the debouncer to the end of the same interrupt and have it output debounced button state into global variables:

```
uint8_t button_state = 0; // 1 bit per button
// other variables to store debouncer state
void button_debounce_timer() {
 // read button states:
 // ie: uint8_t val = HAL_GPIO_ReadPin(SW31_GPIO_Port, SW31_Pin);
 // debounce, store outputs to button_state
}
```

Then another function can be used to get these values and react on the values:

```
uint8_t button_state_prev = 0; // 1 bit per button
uint8_t buttons_get() {
 uint8_t ret = button_state & ~button_state_prev;
 button_state_prev = button_state;
 return ret;
}
```

Button debouncing is a common operation in all of the embedded devices and there are a lot of different algorithms for it. The issue we are solving is that we do not want to react too fast on the noise generated when the contact closes or opens, getting multiple button press events. You can do up-down counting and

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

reacting when the counter reaches opposite edge. You may also just wait for a number of times the same value in sequence before reacting. Or you may react immediately and not accept next input before a set time.

- Write a debouncer for the buttons
- Write test code in main loop with printf for testing.
- Test the button events, then integrate to the rest of the system.

Working button debouncer; device reacts to button presses	4 points

3.4. Microcontroller real-time clock

The task 3.4 is an alterntive to task 5.4. Only one of these gets scored.

The appropriate way to keep time is using a crystal oscillator. But our microcontroller module does not have one. Instead it is using RC oscillator with 1% precision at room temperature. In addition, the microcontroller does not have battery backup, so it will loose time when power is cut. For demonstration purposes, this is still fine and you can complete the proof-of-concept clock project. There is a better alternative at Task 5, but it requires more soldering and firmware work. You can choose, which one you implement.

The microcontroller contains hardware peripheral to count real time. The RTC counts broken down time (hours-minutes-secons, days-months-years), as opposed to just a timestamp number (ie UNIX/POSIX timestamp of seconds since 1970-01-01 00:00 at Greenwich).

- Enable the RTC in CubeMX.
- Write functions to set time.
- Write functions to read time.

You may need additional functions to access alarm functionality (set alarm, enable alarm etc). Also consider the effects of timer being advanced while you read or write it. You should stop the timer when setting the clock (so hour number could not roll to next value between writing hours and minutes etc). You may also need to synchronize the reading of time components, to avoid reading invalid values (ie next hour number with minute number, 14:59→15:59→15:00). For synchronization, the simplest way is read the timer twice; if the readings differ then do it again until you get 2 same values – then numbers were stable.

Internal RTC is working, capability of setting time and reading time	4 points

3.5. Implement user interface

This task is to implement your user interface. You shall implement the design you created at 3.1 User interface design. For the information you do not yet have (ie when the RTC or temperature sensor does not yet work), insert placeholder data. If the alarm functionality is not present, create a manual trigger to start alarm (ie button).

The user interface has usually the lowest priority timings in the code. Thus it is usually implemented in main loop of the device.

There may be timeouts involved in UI flow. For those, you can use the system tick timer, that is always running. There is a running millisecond count available at all times using HAL_GetTick(). You have to store it to a variable at some event (ie last state change, button press etc). Then you can subtract the stored count from current value and if it is over a set limit, react on a timeout.

- Implement UI.

Working UI prototype, that implements the design created at 3.1	6 points

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

4. Buzzer

An alarm clock needs a sound output. For that we have a small buzzer on board. This is mainly intended to be used for alarm. It may be used for other UI purposes as well, ie button clicks and confirmation tones.

4.1. Hardware

The buzzer hardware contains a transistor amplifier. There is a diode, that limits the induced kick-back voltage from the sound coil when the transistor opens, otherwise you may kill the transistor. Verify the polarity of the diode, it shall be reverse biased normally (cathode is marked with a stripe on the body).

The buzzer system requires an AC signal to operate. Repeating signals can be generated using timer modules in the microcontroller. Most timers have output pins, that can be configured in PWM mode. By changing the frequency, you control the tone. By changing the PWM high/low ratio, you can control the loudness. Although the timers in the microcontroller have multiple outputs, they have only one counter and its auto-reload register, that is controlling the frequency. So cross-using a timer for multiple independent purposes is quite limited. Luckily, STM32 microcontrollers contain a lot of timers, ie 6 for our part.

Typical buzzer tones are around 1-4kHz. See the speaker datasheet, its loudness depends on the frequency. Figure out the timer settings for the buzzer timer.

For buzzer sequencing (turning it on and off periodically), you may want to use another timer and its interrupt (like the display scanning timer in 2.3), this interrupt shall fire at around 2Hz to provide 1Hz beeps.

For the sanity of your teammates and everyone else in the room, please stick some tape on top of the buzzer to keep the volume down! Disconnect the buzzer wire to mute it if you have issues with turning it off.

- Solder the Task 4 hardware.
- Open STM32F031K6 reference manual RM0091. Browse through the timer section, read about PWM.
- In CubeMX assign and configure a timer; configure output PWM pin for the buzzer.
- Assign another timer for buzzer sequencing and configure it.
- Install the jumper wire on the board.

Buzzer hardware soldered, CubeMX configured, jumper wires connected	2 points

4.2. Firmware

The firmware support for the buzzer consists of 2 timing components. One is generating the tone. The other is switching the first timer on and off, sequencing series of beeps.

The tone-generating part is handled by a timer. The PWM timer is configured in the CubeMX. You can update its pulse width and period in firmware if you wish, then start the timer to start buzzing:

```
__HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, pulse_width);
__HAL_TIM_SET_AUTORELOAD(&htim1, period);
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
```

The calculation of a timer frequency is explained in 2.3 Display update timer and its interrupt.

In addition to the frequency, you must set the channel pulse register value to set the pulse width. For a 50% pulse, this shall be half of period register value. Thus when updating period register value runtime, you must update pulse width as well to remain at 50%. When pulse width is reduced, the buzzer gets quieter. Around 50% is the loudest. You shall not go at higher values (it gets quieter again, but draws much more current and produces heat).

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

To disable the timer, you must stop the timer using HAL_TIM_PWM_Stop. By just stopping the counter (using registers or other APIs) and not disabling the output, the timer may stop so that the output if left high. This may damage the buzzer due to DC voltage left on coil.

The sequencing timer shall generate interrupts. In the interrupt handler, you just switch on/off the buzzer timer. Write upper level functions to start and stop the buzzer. For example:

```
void buzzer_enable() {
 HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1); // start PWM
 HAL_TIM_Base_Start_IT(&htim2);      // start sequencing
}

void buzzer_disable() {
 HAL_TIM_Base_Stop_IT(&htim2);       // stop sequencing
 HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_1);  // stop PWM
}
```

- Write handler function to sequence the start/stop of the buzzer.
- Write functions to start and stop the buzzer along the sequencing.
- You may also experiment with generating UI tones with buzzer.

Buzzer works, buzzer sequencing works	4 points

4.3. Alarm functionality

In this task we bring all the pieces together. For an alarm clock, we need the clock to sound an alarm at a predefined time. Once an alarm sounds, there must be a way to turn it off.

First, the system to start alarm. There are a lot of ways implementing it. You may just poll the time values and once they pass the alarm time, start the alarm. In that case – if you check the time being equal to the given value, you may happen to miss that exact value. If you check time being greater than the value, the alarm may sound after the value. There are different ways of overcoming it.

Alternatively, the RTC-s incorporate alarm functionality. You may configure the alarm at the RTC and then let it check for it by itself. Then you can either wait for an interrupt (for external RTC it requires another wire) or just poll regularly if the alarm status bit has been changed.

- Write code to get alarm event at preconfigured time.
- Write code to enable and disable that alarm event.
- Integrate the alarm system in the UI.

Alarm functionality works – at configured time, alarm happens and can be shut off	4 points

5. External precise Real Time Clock

As discussed earlier in 1.4 and 3.1, the given NUCLEO-32 microcontroller module does not have precise clock source. It also does not have connection for external backup battery, that drives only the RTC. To overcome these limitations, an RTC chip is included on the board.

The RTC chip connects to microcontroller by I2C bus, the same type of bus is also used by temperature sensor. The tasks 5.2 and 5.3 are common for both.

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

5.1. Hardware

The RTC section is designed using surface mount components. These do not have legs going through the holes in the board, instead they are soldered on the surface of the board. Most of modern day electronics is built in SMT (surface mount technology). Surface mount components are smaller and can have a lot more pins. They can be assembled on the board by machines.

The components in this section are chosen amongst the largest available. The chip U51 is in SO8 package, having pin spacing of 1.27mm (through-hole chips have spacing twice as large, 2.54mm). The passives (resistors, capacitors) have size code 1206, this is a standardized package of 3.2x1.6mm.

Before soldering the components, please read through the introduction to surface mount soldering at the beginning of this document.

- Solder the components of Task 5.
  Solder the J52 battery holder at a slight angle, not flat against the board (bottom edge should be around 1mm lifted off board) for the board to stay upright on its own and not fall over on the face.

You should leave the FB51 as last, so that in case you messed up, the board is still usable and does not have short circuit at power rail (other connections by wire can be disconnected).

RTC hardware properly soldered	4 points

5.2. I2C bus connections

The interface to RTC and temperature sensor chips is I2C bus. There is a good explanation about I2C bus in wikipedia. The bus is pulled up by resistors, devices only pull down the lines (this is called open drain connection). A single I2C bus may have multiple devices and each is identified by its address. Check the datasheets and find out the device addresses.

There is an I2C peripheral in the microcontroller. You can either use that or bit-bang the pins in GPIO mode. In either way, configure the CubeMX accordingly. The default settings for the I2C peripheral work well enough.

The pin-muxing of NUCLEO32 modules is quite limited due to the small 32-pin chip and already made connections on the module (PA5,6 are in parallel to PB7,6). To use SPI, I2C and rotary encoder, you can use PA5,7 for SPI and PA10, PB6 for I2C.

- Open STM32F031K6 reference manual RM0091. Browse through the I2C section.
- Perform pin planning and connect the jumper wires.

CubeMX configured for I2C bus, jumper wires connected	2 points

In addition to I2C, the RTC and temperature sensor have additional outputs. These may be connected in case you find them useful. As with other outputs, they may be left unconnected with no harm. The outputs are open-drain: they need a pull-up resistor. When setting them up, configure a pull-up resistor to the corresponding input of microcontroller in CubeMX.

5.3. RTC I2C communication

We need I2C communication for RTC and temperature sensor ICs. This task is to get I2C working, for RTC, temperature sensor or both.

The STM32 HAL has functions, that perform the I2C device addressing and register addressing layer. You can use these to access the registers inside the chips:

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

```
uint8_t value;
HAL_I2C_Mem_Read(&hi2c1, I2C_ADDR, reg_addr, 1, &value, 1, ~0);
HAL_I2C_Mem_Write(&hi2c1, I2C_ADDR, reg_addr, 1, &value, 1, ~0);
```

Open the datasheet for the chip(s) and find out what registers are there. Also find out the I2C address of the chip. The STM32 HAL functions require the I2C device address that includes the R/W bit (set to 0).

- Open PCF85263AT/AJ and LM75ADP,118 datasheets; learn about the communication and internal registers.
- Implement a function to read and a function to write given register in chip (for example: rtc_read_reg(reg), rtc_write_reg(reg, value), ts_read_reg(reg), ts_write_reg(reg, value)).
- If needed, write bulk access functions to read a lot of registers at one access.
- Write test code. Write some values and read them back to verify that the lower layer works.

Working I2C driver, functions to read/write RTC and temperature sensor registers	4 points

5.4. Firmware

The task 5.4 is alternative to task 3.4. Only one gets scored.

The RTC chips reports time in BCD (binary-coded decimal) format. In this format, the low 4 bits represent ones digit and high 4 bits tens digit in decimal. Use shift and binary bitwise (&, |) operators to convert the format.

After you have established communication with the RTC chip, it is time to write the driver code for it

- Write functions to set time
- Write functions to read time
- Write test code to test above using printf-s.

You may need additional functions to access alarm functionality (set alarm, enable alarm etc) if you choose to do it in hardware.

The RTC chip needs some configuration for best performance. The crystal used on the board has 50kohm series resistance and requires 6pF load capacitance. These settings are configurable in "oscillator register". You should program this register every time the time gets set.

The "RAM Byte" register may be used to store some kind of state information to persist it over power loss. A good device does not loose its state at power loss, it resumes where it was left off (there are exceptions, for example for safety reasons, machinery should not start by itself).

External RTC is working, there is code capable of setting time and reading time; battery back-up works	8 points

6. Temperature sensor

This task tests your soldering skills. We are using typical modern surface mount (SMT) components of size 0402 (1.0x0.5mm) and a chip in TSSOP case (lead spacing is half of the chip in previous task, 0.65mm). It requires a steady hand and a good eyesight (or microscope). Rest assured, that it is nowhere near impossible – in the electronics industry, soldering such components by hand is a normal thing and there are much more difficult components, that can be soldered by hand.

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

6.1. Hardware

The 0402 size components are normal size components being used in modern equipment (in space-constrained devices, even 0201 and 01005 are used). For good soldering experience, the use of flux is needed and the soldering iron tip shall be clean. As with previous task, there is a component FB61, that cuts power to the chip – it can be soldered last after you have measured that you do not have power rail short circuit that would bring down the whole board.

Before attempting this task, please read through one more time the SMD soldering advice at the beginning of this document.

- Solder the Task 6 hardware to board

Temperature sensor hardware properly soldered	8 points

The chip uses I2C communications. The connections are covered by the section 5.2. You may have to revise that section if you add temperature sensor later on to your project with RTC already set up.

6.2. Firmware

The lower layer of I2C communications are covered by task 5.3.

The printf-class functions do not contain floating-point functionality. The Cortex-M0 does not have a FPU and emulating the floating-point in software would be expensive both runtime and code size wise. You are reading the temperature as fixed-point value, having integer and fractional bits. It is up to you to split it and display both parts using integer arithmetics. Or just use the integer value and drop the fractional bits. The temperature is a signed value, write your code so that the negative values are shown appropriately as well.

- Write a function, that reads the temperature from the sensor.
- Integrate it into user interface, so that temperature can be shown on display.

Temperature is read and displayed	2 points

7. Light illumination sensor

The clock has a light sensor, that can be used to measure the light illumination in the room. As alarm clocks are used to wake people up and they are usually near the place you sleep at, you do not want a spotlight shining on your face at night. At the same time, it is not good if the clock is not visible in daylight, under direct sunlight. To overcome this, you should implement a dynamic brightness control, that dims down the clock in the dark.

The light sensor is a LDR – light dependant resistor. Its resistance will change depending on its illumination. In the schematics, it is connected along with R71 as a voltage divider. This makes the voltage at J71 change on the ratio of the resistances.

The microcontroller contains an ADC – analog to digital converter. ADC takes an input and converts its voltage to a digital value. The value is proportional to the input – having zero at GND and full-scale value (2^{12}-1) at VDDA.

7.1. Hardware

- Solder the Task 7 hardware on the board.
- Open STM32F031K6 reference manual RM0091. Browse through the ADC section.
- At CubeMX, configure a pin for the ADC; generate project.

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

- Connect the pin using jumper wire.

Task 7 hardware is properly soldered and connected	2 points

7.2. Connections and firmware

The HAL contains functions to handle ADC. In case you have only one channel, it is very easy:

```
HAL_ADC_Start(&hadc);
HAL_ADC_PollForConversion(&hadc, ~0); // wait
uint32_t val = HAL_ADC_GetValue(&hadc);
```

The PollForConversion function can be used to check if the conversion is done and not wait. To do that, provide 0 as second argument and read its result.

- Write code that starts the ADC and reads the result.
- Test the code using printf.

Firmware is capable of measuring illumination level	4 points

7.3. Display brightness

You should use PWM on OE# signal of row drivers to dim the display. To do that, you need another timer having an output pin.

You can decide, how you implement the brightness control. Shall it have some levels to switch between? Or fully stepless control? Should there be a hysteresis so it would not blink between two states or go into positive feedback when illuminating itself?

For 8-bit PWM, the counter period shall be set to 255 (for brightness control values of 0..255). The PWM frequency shall be around 5-10kHz to avoid flicker due to interference with display scanning frequency. Set the timer to low polarity as the OE# pin is low-active.

In firmware, you need to start the PWM timer. Afterwards, you can set its compare value to change the PWM duty cycle, that controls the display brightness:

```
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
__HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, brightness_value);
```

- Measure the ADC values for dark and bright room
- In CubeMX, set up the OE# signal as PWM timer output
- Implement LED panel brightness control
- Integrate light sensor to control LED panel brightness.

Light sensor controls LED panel brightness	4 points

8. Rotary encoder

In modern devices, the user interface may still need rotary buttons – in some situations, rotary knobs are much more intuitive to use than up-down buttons (ie volume controls etc). We have incorporated a rotary encoder on this board. You can use it for the benefit of your user interface.

The rotary encoder is nothing more than 2 switches (A and B). The turning of the button pulses the switches. The pulses are overlapped, but shifted in time (90° phase shift). This quadrature (90° shifted 2-

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

phase) signal contains the information about the amount of rotation and direction. When turned in one direction, one signal is leading and the other is following; in the other direction, the other signal is leading. When studying the timing diagram of such signals, you can observe that at the time of each rising edge of signal A, the signal B tells you the direction – it is high in one and low in the other direction.

In addition to the A and B signal, there is a switch on pin E, that is just a push button, actuated by pressing on the shaft of the encoder. For all of the switches (A, B, E), they connect the line down to ground. As with normal switches SW31..34, you shall configure a pull-up resistor at microcontroller input.

8.1. Hardware

- Solder the encoder and its pin header on the board.

Task 8 hardware is properly soldered	2 points

8.2. Connections and firmware

The rotary encoder could be handled by the GPIO inputs and interrupts, but it is a tricky thing to get right. You still need debouncing and on fast rotations, the pulse rate can be reasonably high. Just by polling it in main loop may not be enough, you would need pin interrupts. But the STM32 has special hardware built in, that can handle quadrature encoders in hardware – timer 1 has such feature. To set it up, you must select in "Combined Channels" drop-down box the option "Encoder Mode" (restart CubeMX if the "Combined Channels" box is not visible, it has happened...). This setting takes over and enables timer inputs 1 and 2. The timer is put into a special mode, where it is incremented or decremented only on encoder pulses. Other parameters still apply, ie auto-reload value for limiting the period etc. The inputs get a "input filter" parameter, that can be used to filter out contact bounce of the encoder switches.

Reasonable settings for rotary encoder are: prescaler 0, count up, period 65535, no clock division, Combined channels in Encoder mode, encoder mode TI1 and TI2, both channel input filters set to 15.

- Open STM32F031K6 reference manual RM0091. Browse through the Timer1 section, learn about quadrature encoder mode.
- Set up the pins in CubeMX and re-generate project.
- Connect the jumper wires.
- Write driver code to initialize handle the encoder hardware.
- Write function, that reads current encoder value.
- Integrate the encoder in the UI.

The encoder timer is a timer in a special mode. Instead of counting periodically as timers do, this timer counts only on the encoder button rotation. Also the direction is considered – it counts both up or down. The wrap-around location is set by autoreload period value. This can be set by CubeMX, but also updated via software. The value of the timer can be also set by the firmware.

Before using the encoder, you shall start the timer by:

```
HAL_TIM_Encoder_Start(&htim1, TIM_CHANNEL_ALL);
```

The simplest way to use an encoder is to set up its autoreload and counter value depending on the task you are using:

```
__HAL_TIM_SET_AUTORELOAD(&htim1, 60-1);
__HAL_TIM_SET_COUNTER(&htim1, 14);
```

The encoder value can be read using:

```
uint32_t val = __HAL_TIM_GET_COUNTER(&htim1);
```


Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

You may observe, that the encoder has physical detents (clicks) only on every 4th count. Dialing in a value between clicks is not too comfortable. You may want to consider 4 counts as one output count and scale your reads and writes accordingly. When doing that, you may want to offset the reading value so that the number boundary is not in the click, but between two clicks.

Encoder is working	8 points

9. Documentation

Write/draw/create a simple user manual for your device. Do not write a book – nobody is going to read it. The user should be able to use your product after quickly looking over the manual. Provide PDF file for scoring.

User manual	8 points

10. Extra points

These 10 points may be given to you by Artec Design representative. We like good products and will reward the best work with extra points.

For soldering quality	2 points
For reasonable pin planning of microcontroller pins	2 points
For source code quality	2 points
For the user interface styling	2 points
For overall look-and-feel of the device	2 points

List of items to be returned on USB flash drive

Copy these files to flash drive and hand over for scoring. Only the files on USB drive are used review and scoring, we do not care what is on your computer.

- CubeMX project (having the same pin muxing as the real thing)
- Atollic project, source code
- UI state diagram with display mockups (.pdf)
- User manual (.pdf)

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee

Scoring

The maximum score for this project is 128 points.

The tasks 3.4 and 5.4 are exclusive, only one is scored.

Task	Description	Points
1.3	Working code on the microcontroller (at least blinking LED if not anything else)	4
1.5	Working debug serial port	4
1.6	Properly soldered Task 1 components.	2
2.1	Properly soldered Task 2 components	6
2.2	Usable CubeMX setup and connection to display	2
2.3	Interrupt timer working	4
2.4	Display pixel data from video memory to the display	4
2.5	Font created and embedded into firmware	6
2.6	Character and binary number rendering capability	4
3.1	Usable UI design, that corresponds to requirements	6
3.2	Button hardware soldered, CubeMX configured, jumper wires connected	2
3.3	Debounced button events working	4
3.4	Internal RTC is working, capability of setting time and reading time	(4)
3.5	Working UI prototype, that implements the design created at 3.1	6
4.1	Buzzer hardware soldered, CubeMX configured, jumper wires connected	2
4.2	Buzzer works, buzzer sequencing works	4
4.3	Alarm functionality works – at configured time, alarm happens and can be shut off	4
5.1	RTC hardware properly soldered	4
5.2	CubeMX configured for I2C bus, jumper wires connected	2
5.3	Working I2C driver, functions to read/write RTC and temperature sensor registers	4
5.4	External RTC is working, there is code capable of setting time and reading time; battery back-up works	(6)
6.1	Temperature sensor hardware properly soldered	8
6.2	Temperature is read and displayed	2
7.1	Task 7 hardware is properly soldered and connected	2
7.2	Firmware is capable of measuring illumination level	4
7.3	Light sensor controls LED panel brightness	4
8.1	Task 8 hardware is properly soldered	2
8.2	Encoder is working	8
9	User manual	8
10	For soldering quality	2
10	For reasonable pin planning of microcontroller pins	2
10	For source code quality	2
10	For the user interface styling	2
10	For overall look-and-feel of the device	2
	Total:	**128**

Akadeemia tee 21/1, 12618, Tallinn, Estonia • +372 671 8550
info@artecdesign.ee • www.artecdesign.ee