DIGITAL SIGNAL PROCESSING FOR COMMUNICATIONS AND RADAR LAB

Reinhard Feger, Lukas Furtmüller

Summer Semester 2024

Institute for Communications Engineering and RF-Systems

JOHANNES KEPLER UNIVERSITY LINZ

Today's Contents

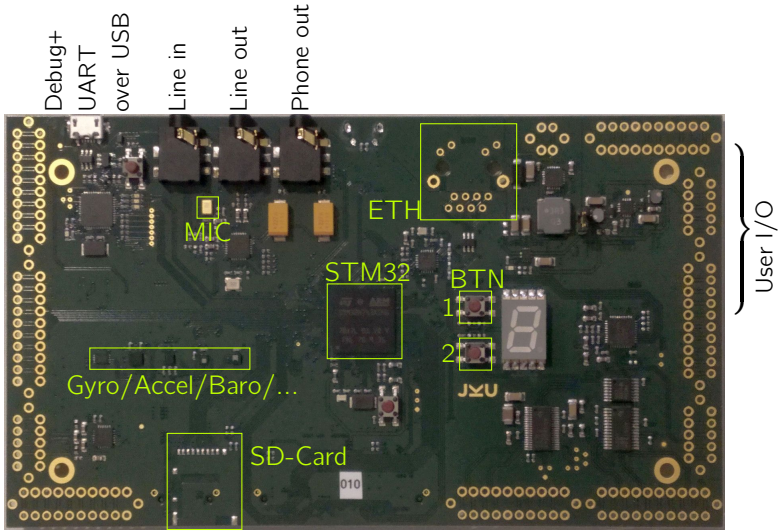
Introduction to the Development Environment

Exercises

Hardware Overview

- In-house developed board based around an STM32H743 processor (Arm Cortex-M7)
- PCB includes audio codec, microphone, ADCs, DACs, accelerometers, gyroscope, ...
- Includes debugger hardware and power supply over USB

Hardware Details (Micro-USB Version)

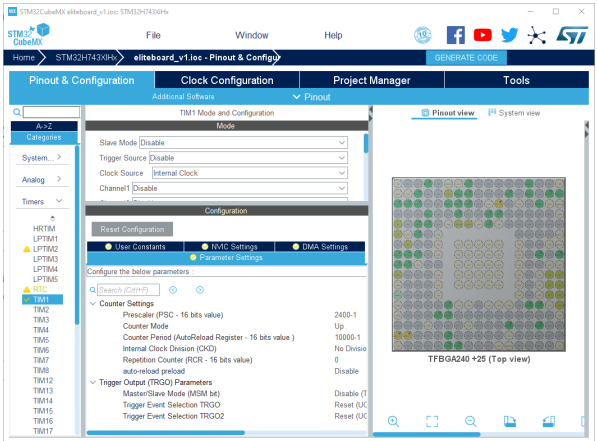


Software Overview

- Software is installed on lab computers, but can also be installed on private computers (see Moodle)
- Visual Studio Code (VSCode) inluding the following extensions
  - C/C++
  - Cortex-Debug
- GNU toolchain (Compiler, debugger, standard libraries, ...)
  - Compiler from Arm can be used for a multitude of Cortex-M and Cortex-R processors (“bare-metal”: no underlying operating system required)
- Makefile-based development
- Libraries+configuration tool provided by STMicroelectronics for their STM32 processors (HAL+STM32CubeMX)

CubeMX Details

- STM32CubeMX: Graphical tool to configure components of the STM32H743 processor
- For example: Timer configuration



Software Details

- Check also “Clock Configuration” to find out at which rate the timers are running
- When configuration is done “Generate Code” creates/updates C-Files, Linker script, ...
  - Make sure that your code is not overwritten (description follows in the first exercise)
- Settings in “Project Manager” are predefined in our templates. Check the configuration there if you want to create your own projects.

Software Details

- Important general VSCode shortcuts (in Windows):
  - Open Explorer View (or click on the two sheets top left) with CTRL+SHIFT+E
  - Build using CTRL+SHIFT+B
  - Start debugging with F5
- Debugging shortcuts
  - Step over: F10
  - Step into: F11
  - Step out: Shift+F11

Introduction

- Common tasks in microcontroller programming
  - Run periodic commands
  - React to input
  - Provide response
- Very often event based: Use interrupts (outside of linear program flow)

Introduction to Data-Types

- The C language is used on many different platforms
- “Conventional” data types like `int`, `char`, ... can have different meanings
- It is more clear to use:

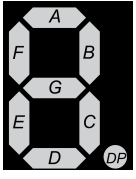
Name	Type	Range
`int8_t`	1 byte signed	-128 to 127
`uint8_t`	1 byte unsigned	0 to 255
`int16_t`	2 byte signed	-32,768 to 32,767
`uint16_t`	2 byte unsigned	0 to 65,535
`int32_t`	4 byte signed	-2,147,483,648 to 2,147,483,647
`uint32_t`	4 byte unsigned	0 to 4,294,967,295
`int64_t`	8 byte signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`uint64_t`	8 byte unsigned	0 to 18,446,744,073,709,551,615

Caveats

- Take care when you use built-in code snippets (suggestions)
  - Example: For loops are constructed with `size_t`, which does not support negative numbers and is again different on different platforms. Select the correct data type for your application
- Also floating-point data types can be defined using a fixed number of bytes
  - The definition is tricky, because it is not (always) standardized how the bits are used (mantissa and exponent)
  - `float32_t` and `float64_t` are not necessarily single and double precision, but in our case they are

Exercise 1: Timer, GPIO, Debugging

- Exercise 1: Write a program that performs the following functions:
  - Blinking segment A of 7-segment display with a period of $500\,\mathrm{ms}$
  - Blinking segment D of 7-segment display with a period of $1\,\mathrm{s}$
  - Perform multiple snake movements after a single press of button 1:
    - Consecutively switch on segments A to F in steps of $100\,\mathrm{ms}$. Only one LED should be on at a time
    - The number of repetitions should be configurable
  - Make sure that the snake movement has priority over the single blinking LEDs and blinking of A and D returns to normal after the snake movement



Exercise 1: Details

- Write your code in lines between `/* USER CODE BEGIN` and `/* USER CODE END`. STM32CubeMX does not change these parts of the code. All other code will be deleted when the project is regenerated from STM32CubeMX!
- Use the provided STM32CubeMX template and configure a single timer to suitable values
  - Note that dividing the timer clock of $240\,\mathrm{MHz}$ (check the clock configuration in CubeMX) has to be done using two 16-bit values. Setting the divide counter to N leads to a division of $N + 1$
  - STM32CubeMX allows to enter (simple) equations like 1000-1
- Activate the timer and its interrupts by calling `HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim)` in the main function. CubeMX already defines a global variable `htim1` which can be used in the call of `HAL_TIM_Base_Start_IT` by its address: `&htim1`

Exercise 1: Details

- Buttons and LEDs are defined in `main.h`
- Use `SEGX_Pin` and `SEGX_GPIO_Port` for the pin control functions:

```
1 void HAL_GPIO_WritePin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState)
```

```
1 void HAL_GPIO_TogglePin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)
```

  - Here X is either A to G or DP for the decimal point, `PinState` can be `GPIO_PIN_SET` or `GPIO_PIN_RESET`

Exercise 1: Details

- The hardware abstraction library (HAL) hides the interrupt service routines. Users work with so-called callback functions
  - For the timer this is

```
1 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
```

  - The callback for a button press is

```
1 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
```

- The callback functions have to be defined by the user and can be put into the main.c file without declaration

Exercise 1: Details

- Checking the source of the callback is possible by

```
1 if(GPIO_Pin == BTN1_Pin)
```

and

```
1 if(htim->Instance == htim1.Instance)
```

- Build (and fix all compilation errors)
- Start debugging (a rebuild is triggered when required)
- Debugging allows to set breakpoints (also conditional ones), step through the code, inspect variables etc.

Exercise 2: Signal generation

- Exercise 2: Write a sinusoidal signal generator based on the following approaches using
  - a (non-optimized) sin-function (`#include <math.h>`)
  - a lookup table
  - an IIR filter
  - complex multiplication/rotating phasors
- The generator should be used to output audio signals on the headphone
  - Realize two different frequencies: $800\,\mathrm{Hz}$ on channel 1 and $2000\,\mathrm{Hz}$ on channel 2 (left/right)
  - Storage of left and right data is at even and odd indices (if you use an array)

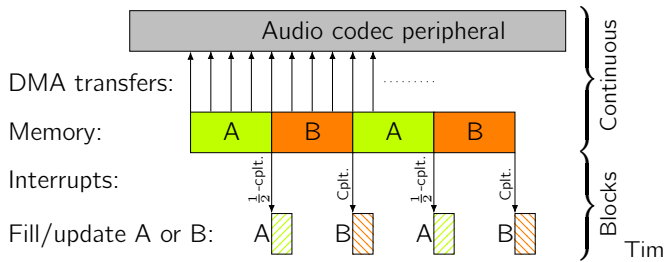
Exercise 2: Double Buffering

- In the provided template a driver for the audio codec (WM8731) is already included
- In the standard setting a sampling rate of $48\,\mathrm{kHz}$ is used
- Copying of data from memory to the audio codec is done using Direct Memory Access (DMA) without using the CPU. Continuously updating the output data without causing gaps in the audio output can be realized using the so-called double buffering approach
- To output a signal memory is continuously read (circularly)

Exercise 2: Double Buffering

- After the first half of the memory was transferred to the codec an interrupt (half complete) is triggered. From that time on, the first half of the memory can be updated, since the DMA controller is accessing the second half

- After the second half of the memory was read, the circular read starts again at the first half. At the same time another interrupt (complete) is triggered and the second half of the memory can be updated without corrupting the output signal



Exercise 2: Details

- The double buffering is hidden from the user. Instead the following functions to wait for availability of the output buffer and writing to the correct buffer are available:

```
1 void wm8731_waitOutBuf(struct wm8731_dev_s *self)
```

```
1 void wm8731_putOutBuf(struct wm8731_dev_s *self, int16_t *data)
```

  - The function `wm8731_putOutBuf` automatically selects the correct half of the memory to update
- The device handle `wm8731_dev_s` as well as initialization and start of the codec DMA are already included in the template
- Define your own `dac_buffer` array with 256 samples whose address can be passed to `wm8731_putOutBuf`

Exercise 2, Part 1: sin-Function Approach

- Important topics for the signal generation using the sin-function:
- How to make sure the program can run forever?
  - Consider data type and resetting of the time index
- Translate between frequency in Hz and frequency normalized to the sampling rate

Exercise 2, Part 2: Lookup Table Approach

- Store samples of a $\sin$ (or $\cos$) function and output them periodically
- Use only one table for both frequencies
- Determine the number of required samples such that the lowest frequency of the generated sinusoid is $100\,\mathrm{Hz}$
- Start with storing a full sine period, but finally implement storing only a quarter of the period to save RAM
- Use Python or Matlab to generate C-code for initializing the lookup table (for Python `numpy.array2string`, for Matlab `sprintf`)

Exercise 2, Part 3: IIR Filter Approach

- Design a digital biquad filter/a second order IIR filter with poles that lead to an oscillation at the filter output
  - To start the oscillation an input impulse, an input step, or preinitialized storage elements are required
- Calculate filter coefficients and simulate the filter response in Python/Matlab
- The filter can be described by the difference equation

$$x[n] = y[n] - 2r\cos(\theta)\, y[n-1] + r^2 y[n-2] \tag{1}$$

with x being the input and y the output of the filter

Exercise 2, Part 3: IIR Filter Approach

- The transfer function of such a system is

$$H(z) = \frac{1}{1 - 2r\cos(\theta)\, z^{-1} + r^2 z^{-2}} = \frac{A}{1 - re^{j\theta} z^{-1}} + \frac{B}{1 - re^{-j\theta} z^{-1}}$$

- Partial fraction decomposition and inverse z-transform leads to the impulse response

$$h[n] = \frac{r^n \sin(\theta(n+1))}{\sin(\theta)} \sigma[n] \tag{2}$$

Exercise 2, Part 3: IIR Filter Approach

- For $r = 1$ the step response is

$$y_{\text{step}}[n] = \frac{1 - \cos(\theta(n+1))}{\theta \sin(\theta)} \sigma[n] \tag{3}$$

- Using (1) with a suitable selection of r and θ and proper amplitude scaling (see (2) and (3)) allows to implement an oscillator without trigonometric functions in the calculation loop
- Maximize the output amplitude by proper scaling to a 16-bit integer and let the oscillator run for several (up to 30) seconds. What do you observe?
- If the previous experiment reveals some flaws: Propose (not implement) a possible fix (comment in the code)

Exercise 2, Part 4: Phasor Rotation Approach

- Complex multiplications allow to rotate a phasor from one time step to the other

$$\mathrm{e}^{\mathrm{j}\varphi[n+1]} = \mathrm{e}^{\mathrm{j}\varphi[n]}\mathrm{e}^{\mathrm{j}\delta\varphi}$$

- Taking the real or imaginary part of the phasor at each time step and using a proper $\delta\varphi$ allows to generate a $\cos$ or $\sin$ signal without using trigonometric functions in the calculation loop
- The microcontroller does not support complex-valued operations out of the box. Implement your own operations using real- and imaginary parts of the signals
- Hint:

$$\mathrm{Re}\left\{(\mathrm{Re}\left(x\right) + \mathrm{jIm}\left(x\right)) \cdot (\mathrm{Re}\left(y\right) + \mathrm{jIm}\left(y\right))\right\} = ?$$

Exercise 3: Chirp Signal

- Write a program that generates a linear chirp signal from 200 Hz to 1800 Hz with a duration of 10 s
- The output should be periodic ($T = 10$ s)
- Note, that a linear frequency ramp

$$f(t) = f_{\text{start}} + k_{\text{ramp}} t$$

leads to a quadratic phase term

$$\phi(t) = 2\pi \int_0^t f_{\text{start}} + k_{\text{ramp}} t' \mathrm{d}t' = 2\pi \left(f_{\text{start}} t + k_{\text{ramp}} \frac{t^2}{2} \right)$$

- Select the normalized frequencies and the normalized ramp slope to fulfill the given start and stop frequencies