1.

TDDI11

Embedded Software

Lab Manual

March 2024.1

1.

Table of Contents

1.

1.

Introduction

Often embedded systems are designed to be small, low weight, low power, low energy, low cost, real-time, distributed, reliable, durable, safe, and secure. In general, this means the simplest systems are usually preferred and whatever hardware and software that is unnecessary is taken away from the system and its development process. Therefore, embedded engineers frequently face limited resources and deal with low-level programming as well as bit-operations on small processors and microcontrollers. Many embedded systems are bare-machine, meaning there is no operating system. Such systems are often programmed in assembly and C.

The concepts that do not need to be practiced on a bare-machine, are performed on the host-machine.

In the following we start with a hello-world program on the bare-machine in chapter 0. We will see how a simple C code is cross compiled to run on a machine without OS. We use Qemu virtualization for chapters 0, 2, 3, and 4. Chapter 1 only involves a host machine. Chapter 5 uses the STM32L562 board.

Chapter 1 focuses on bit operations in C on the host machine. Chapter 2 explains how to mix C code with assembly. This may be needed to design a superior system by writing the crucial segments of the code directly in assembly. Chapter 3 explores polling- and interrupt-based I/O operations as well as serial communication between devices (virtually on Qemu). Chapter 4 deals with preemptive multi-threading as well as handling mechanisms for shared resources, on the bare-machine. Chapter 5 focuses on Finite-State Machine (FSM) representation of systems by implementing a simple game on on the STM32L562 board.

1.

Notes

The distribution of the deadlines does not exactly correspond to the time required for each chapter to complete. The deadlines are displayed on the lab's web page.

Please read the related chapter in the lab manual and the required resources, early on, so that you can estimate the required time and plan ahead of the deadlines.

Please do not jump over the chapters. The chapters are arranged in such a way that the earlier ones provide information and training that is required for the later chapters.

Please read each chapter completely to the end, before you start working on the assignment. Different chapters might be organized differently.

The number of pages in a chapter does not correspond to the time required for completing the assignments. Some small chapters may require you to study a lot from other resources.

Please arrive on-time in the lab sessions.

The lab assistants' time (for questions and demo) is prioritized for the students that are working in sync with the deadlines, as well as the students that arrive on-time in the lab session.

Qemu: Use Ctrl+Alt (sometimes Ctrl+Alt+g) to release the mouse (pointing device). Qemu grabs the mouse when one clicks inside its window.

Working Remotely: You can work remotely on your own using a ThinLinc client or an SSH connection for chapters 0-4 (ssh -X userID@ssh.edu.liu.se). For chapter 5, you can download the STM32CubeIDE to your private machine as explained in the corresponding chapter.

Help and demo are on campus during the lab sessions!

1.

Chapter 0 Bare-Machine Hello World

0.1 Host and Target Machine

Typical embedded devices do not provide software development friendly environments such as editors, compilers, or debuggers. They might not even have a console or an operating system. Therefore, most of the time, the software development process is carried out on ordinary desktop workstations with full-fledged integrated development environments. The development workstations are called host machines, while the embedded device is called the target machine.

0.2 Cross-Compilers

The host machines are equipped with cross-compilers, i.e. compilers that do not produce code for the machine/architecture they are running on but for a different machine/architecture. For example, if the host computer is an Intel x86 with Windows, a cross-compiler would be a compiler that creates code for an Intel x86 with Linux. A different example would be the case when the host is an Intel x86 with Linux and the target is a Sun Ultra SPARC with Linux. In the first example, the hardware architecture is the same but the interfaces to the hardware (the operating systems running on the host and target respectively) differ. In the second example, the hardware architectures differ but the operating systems are the same. A third example illustrates the case in which both hardware architecture and operating system differ from host to target: the host is an Intel x86 with Windows, while the target is an ARM processor with no operating system.

0.3 Emulators

Code developed on the host for the target cannot be run natively on the host for testing purposes. An alternative would be to upload the code on the target computer/board and to test it there. This approach may have limitations. For example, the target could not be present at the developing site, or the target (or the uploading process) is very slow which would render the testing process very time consuming. Therefore, it is not rare for the software to be debugged and tested on the host machine by making use of an emulator, i.e. a program that mimics the behavior of the target machine.

Depending on the host and target platforms, the emulation can be slower or faster than the actual execution on the embedded platform. The emulated execution can be slower due to the emulator overheads. On the other hand, the emulator may be running on a much faster host machine. For example, the host can be an Intel x64 at 2.6GHz while the target is an 8-bit microcontroller at 4MHz.

0.4 Target Machine in This Lab

In this lab, the target machine is an Intel x86 PC with no operating system (bare-machine). The embedded software will run directly on the hardware (Qemu virtualization) without any help from an operating system. The software will be loaded to the target from a standard boot device (floppy disk).

1.

0.5 Work Flow

Copy “chap1” from the “skeleton” directory to your local directory. You need to work on a lab computer or connect via ssh as described in the introduction. Assuming you have created a TDDI11 directory under your home directory:

```
cp -r /courses/TDDI11/lab/skel/chap0 /home/userID/TDDI11/chap0
```

Please note that userID is the user name that you use to login to the lab computers. Now let us change to the new directory and check it:

```
cd /home/userID/TDDI11/chap0

ls
```

Check to see if “hello_world” directory contains the following:

- main.c
- Makefile
- floppy.img
- mtools.conf
- makeNrun.sh

The source code is in “main.c”. Compiler and linker commands are listed in “Makefile”. The binary file that will be generated by this process must be placed in the floppy image file “floppy.img”. Placing the binary file into the floppy image must be done with “mcopy” command. The necessary settings for “mcopy” are defined in the configuration file “mtools.conf”. The floppy image is used to start the virtual machine “Qemu”. The overall process is described in the script “makeNrun.sh”. The details are discussed in the following.

0.5.1 The Source Code

Open the “main.c” file (for example with “cat” if you only want to check the content. We can use for example “nano” or “gedit” if we want to edit the content):

```
cat main.c
```

Or

```
gedit main.c &
```

The code is as follows:

```
#include <libepc.h>

int main(int argc, char *argv[])

{

ClearScreen(0x07);

SetCursorPosition(0, 0);

PutString(">>>>>>> Empty … Skeleton <<<<<<<\r\n");

return 0;

}
```

1.

This application will run without using an OS. A library is used to provide some basic functionality. In most C programs strings are displayed by means of the “printf” function. “Printf” is part of the standard C library “libc”. “Printf” typically is implemented by the operating system call “write”. As there is no OS on the target platform here, we cannot use “write” and “printf”. Therefore, we use the function “PutString” from the “libepc” library, which communicates useing functions such as “inport” and “outport”. These will directly communicate with the bare-machine.

0.5.2 Cross Compilation

The target machine architecture is “i386-elf”. The “Makefile” includes instructions for cross compiling the main.c file and linking it with the libepc. The compiler is installed in the following directory:

```
/courses/TDDI11/sw/i386-elf-gcc-7.3.0/bin/i386-elf-gcc
```

The linker is installed in the following directory:

```
/courses/TDDI11/sw/i386-elf-gcc-7.3.0/bin/i386-elf-ld
```

Usual compilers/linkers will generate an executable program suitable for a specific operating system. In such an implementation, the library codes are not integrated in the generated program file (shared libraries are typically linked in by the OS when the program is running). However, for this bare-machine implementation, a “raw” application with all needed codes integrated in one binary application file must be generated. The object files and libraries are combined in a single binary file, “embedded.bin”. To generate this file, type:

```
make
```

Then, do

```
ls
```

Check if “embedded.bin” file is generated.

0.5.3 Copying to Floppy

The binary file must be placed in the floppy (we use a virtual image). A prepared floppy image, “floppy.img”, is provided in the skeleton that we copied at the beginning. There is a skeleton binary file in the floppy image that must be replaced. Since the desired binary file must be placed to the floppy under specific considerations “mcopy” command is used. The configuration file “mtools.conf” defines “floppy.img” as drive “a:”. To point the “mcopy” to the proper configuration file, do:

```
export MTOOLSRC=/home/userID/TDDI11/chap0/mtools.conf
```

Place the newly generated “embedded.bin” into drive “a:”

```
/courses/TDDI11/sw/bin/mcopy embedded.bin a:
```

A question will be asked by “mcopy” about what to do with the existing binary file. This is the skeleton binary that you must overwrite with your newly generated binary file. Select “o”.

1.

0.5.4 Starting Qemu

The final step is to load the application in the emulator (Qemu). The file that represents the floppy disk must be specified for Qemu to boot the system:

```
qemu-system-i386 -drive format=raw,file=floppy.img,if=floppy
```

A new window for Qemu will open. The application will be executed printing:

```
>>>>>>> Empty Floppy Used as Skeleton <<<<<<<
```

Now you can close Qemu window.

If you click inside Qemu window, it grabs the mouse. Use Ctrl+Alt to release the mouse. This may not work in ThinLinc

0.5.5 The Script to Make and Run

There is a script with required commands to simplify the process that we did manually in the previous sections (1.5.2 to 1.5.4). In order to run the script type:

```
makeNrun.sh
```

0.6 Evaluation

0.6.1 Assignments

Modify the program (main.c) to write “VT24”, your names, and your student IDs instead of “Skeleton” message. Start from 1.5.1 and edit “main.c”. Then you can either take step by step from 1.5.2 to 1.5.4, or use the script as described in 1.5.5.

0.6.2 Demonstrations

This is a “warm up lab”. You do not need to demonstrate or send your code to the lab assistant.

1.

Chapter 1 Bit Manipulation and Data Representation

In many embedded systems, the usual variable and data types may not directly make sense. Instead, one must figure out what each individual bit represents. The purpose of this chapter is to get familiar with such scenarios which often require bitwise operations. We will see an example about a room controller on the host platform.

Note that a number of functionalities are already implemented in the source code, but a few are left for you to fix and complete (marked with "`// Assignment`" in the source code). You can start by reading the following explanations quickly by skipping the details. Then, study the source code together with the information given below to understand how it works.

1.1 Room Controller

Let us assume that there is a room with a lockable door, a red/green signal light, a fan for ventilation, a temperature sensor, a heater, a cooler, a humidity sensor, a warning light for low humidity, and a warning light for high humidity. Assume that there is a room controller which reads status of the lock, the fan speed switch, the air conditioning (AC) switch, the temperature, and the humidity. Depending on these, the embedded system that works as room controller sends proper commands to the signal light, fan, heater, cooler, and humidity warning lights. It also logs the readout values and the outputs.

1.1.1 Data Acquisition and Format

All the input and sensor data are connected to pins (on the Integrated Circuit (IC)) that can be read collectively from a single input register. Such input registers are often physically read-only. It will appear as these data are concatenated in one unsigned integer variable:

31								23	22																						0
-	-	-	-	-	-	-	-	-																							

–	i_{22}	i_{21}	i_{20}	i_{19}	i_{18}	i_{17}	i_{16}	i_{15}	i_{14}	i_{13}	i_{12}	i_{11}	i_{10}	i_9	i_8	i_7	i_6	i_5	i_4	i_3	i_2	i_1	i_0

Letter "-" implies that these other bits are don't-care and i_n represents $n-th$ bit in input register, I, which is shown above. Although the input data bits come in a variable, their meaning and values must be interpreted as follows:

Status of lock is captured by bit i_0 as:

Table [1]

i_0	Status
0	Door is unlock
1	Door is locked

1.

Fan speed switch affects bits i_3, i_2, i_1 and should be interpreted as:

$i_3\ i_2\ i_1$	Value set by room user
1 1 1	Off
0 1 1	Low Speed
1 0 1	Medium Speed
0 0 0	Full Speed

Table [2]

AC switch affects bits i_9, …, i_4 and should be interpreted as:

i_4	AC on/off
0	Off
1	On

Table [3]

In this system we will have a desired temperature that room user decides and communicate it to the system through a temperature selector switch. Moreover, the actual room temperature is measured by a temperature system and read by the system.

This particular temperature selector accepts temperatures from 15.0 ℃ to 25.0 ℃ with a resolution of 0.5 ℃. The binary representation starts at "`00001`":

$i_9\ i_8\ i_7\ i_6\ i_5$	Temperature set by room user
0 0 0 0 1	15.0 ℃
0 0 0 1 0	15.5 ℃
…	…
1 0 1 0 1	25.0 ℃

Table [4]

The above table suggests that the desired temperature can be calculated based on selector value as:

$$temperaturDesired = 14.5 + (i_9 i_8 i_7 i_6 i_5) \times 0.5\ ℃ \qquad (1)$$

The room temperature measured by the sensor is captured by i_{17}, …, i_{10} as follows:

$i_{17}\ i_{16}\ i_{15}\ i_{14}\ i_{13}\ .\ i_{12}\ i_{11}\ i_{10}$	Room temperature
0 0 0 0 0 . 0 0 0	0.0 ℃
0 0 0 0 0 . 0 0 1	0.2 ℃
…	…
1 0 0 1 1 . 0 1 1	19.6 ℃
…	…
1 1 1 1 0 . 0 0 0	30.0 ℃
1 1 1 1 1 . 0 0 0	$<$ 0.0 ℃ (for convenience show -1)
1 1 1 1 1 . 1 1 1	$>$ 30.0 ℃ (for convenience show 31)

Table [5]

This particular temperature sensor, reports temperatures in a range between 0.0 ℃ to 30.0 ℃ with a resolution of 0.2 ℃. If the temperature is lower than 0.0 ℃, the sensor reports "`11111000`" and if higher

1.

than 30.0 °C the sensor reports "`11111111`". The above table suggests that the measured temperature (assuming in-range values) can be calculated as:

$$temperaturMeasured = (i_{17}i_{16}i_{15}i_{14}i_{13}) \times 1.0\ ℃ + (i_{12}i_{11}i_{10}) \times 0.2\ ℃ \qquad (2)$$

Note that if the temperature is outside the valid range, the above equation will not work correctly.

The room humidity percentage measured by the sensor is captured by i_{23}, …, i_{18} as follows:

$i_{22}\ i_{21}\ i_{20}\ i_{19}\ i_{18}$	Room humidity	$i_{22}\ i_{21}\ i_{20}\ i_{19}\ i_{18}$	Room humidity
0 0 0 0 0	0 %	…	…
0 0 0 0 1	4 %	1 1 0 1 0	80 %
0 0 0 1 0	8 %	1 1 0 1 1	83 %
0 0 0 1 1	11 %	1 1 1 0 0	87 %
0 0 1 0 0	14 %	1 1 1 0 1	91 %
0 0 1 0 1	17 %	1 1 1 1 0	95 %
…	…	1 1 1 1 1	100 %

Table [6]

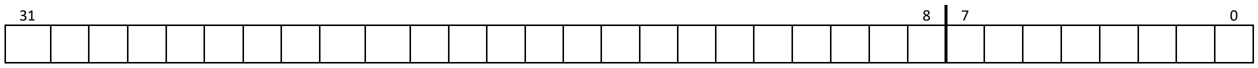
This sensor, reports humidity percentage in a range between 0 % to 100 % in a nonlinear manner. The resolution is 3 % in the middle of the range and 4 % for low and high values, except that it takes a 5 % step from 95 % to 100 %. The above table suggests that the measured humidity can be calculated as:

$$humidity = \begin{cases} (i_{22}i_{21}i_{20}i_{19}i_{18}) \times 4, & (i_{22}i_{21}i_{20}i_{19}i_{18}) < 3 \\ 8 + (i_{22}i_{21}i_{20}i_{19}i_{18} - 2) \times 3, & 3 \le (i_{22}i_{21}i_{20}i_{19}i_{18}) \le 27 \\ 83 + (i_{22}i_{21}i_{20}i_{19}i_{18} - 27) \times 4, & 28 \le (i_{22}i_{21}i_{20}i_{19}i_{18}) \le 30 \\ 100, & (i_{22}i_{21}i_{20}i_{19}i_{18}) = 31 \end{cases} \qquad (3)$$

As implied by letter "X", the rest of the bits (i_{31} … i_{23}) are don't-care. We can ignore them.

1.1.2 Command and Actuation

Depending on the acquired data and their interpretation and information that they provide, a number of commands must be sent to actuators. All these are put together in an integer variable which is then enforced on an output register which is connected to the output pins of the IC.



Y	o_7	o_6	o_5	o_4	o_3	o_2	o_1	o_0

Letter "Y" implies that these other bits must not be touched and o_n represents $n-th$ bit in the output register, O, which is shown above. Their meaning and values are interpreted by the actuators as follows:

The signal lights' inputs are defined as:

o_1	Red light
0	Off
1	On

o_0	Green light
0	Off
1	On

Table [7]

1.

The fan inputs are defined as:

$o_3\ o_2$	Fan speed
0 0	Off
0 1	Low Speed
1 0	Medium Speed
1 1	Full Speed

Table [8]

The cooler input is defined as:

o_4	Cooler
0	Off
1	On

Table [9]

The heater input is defined as:

o_5	Heater
0	Off
1	On

Table [10]

The humidity warning lights are defined as (note that they are active low):

o_7	Low humidity warning
1	Off
0	On

o_6	High humidity warning
1	Off
0	On

Table [11]

As implied by letter "Y", the rest of the bits (o_{31} … o_8) are don't-touch. These must not be changed, as they might be connected to other devices that we do not want to influence. We can ignore them.

1.1.3 System Operations

Let us use a polling scheme to acquire input data. The main application runs in an infinite loop continuously reading the input register "I" (see section 1.1.1). Based on these inputs and laws described in this section, the outputs are determined and continuously updated by writing into output register "O" (see section 1.1.1). Apart from "O" related information must be logged into a file.

If the door is locked the red light must be on. If the door is unlocked the green light must be on.

The fan speed must be set according to the fan switch. For example, if the fan switch indicates the off status ($i_3 i_2 i_1 = 111$), the fan must be turned off by writing the output $o_3 o_2 = 00$. Refer to tables 2 and 8.

The cooler can be turned on only if the AC is turned on by user. If AC is on, the condition for switching the cooler on from an off state is

$$temperaturMeasured > temperaturDesired + 0.3°\mathrm{C}\,. \qquad (4)$$

1.

The heater can be turned on only if the AC is turned on by user. If AC is on, the condition for switching the heater on from an off state is

$$temperaturMeasured < temperaturDesired - 0.3℃ . \qquad (5)$$

The heater continues to work until it is warmer than or equal to the specified desired temperature. The cooler continues to work until it is colder than or equal to the specified desired temperature. If the AC switch is turned off by the user, both heater and cooler must be turned off.

If the humidity is less than 25% low humidity warning light must be on. If the humidity is more than 60% then the high humidity warning light must be turned on.

1.1.4 Hazardous System Behavior

Embedded systems are often used to govern a system consisting of a number of subsystems. These subsystems must work in coordination in order to assure a safe and trouble-free operation. Therefore, it must be assured that some hazardous situations will never happen. Considering the room controller, what do you think about the following?

- Keeping both cooler and heater on at the same time!
- Keeping both red and green light on!
- Keeping both red and green light off!
- The room temperature is outside the sensor's operational range!
- Keeping both low and high humidity lights on at the same time!

1.1.5 Temperature Control Approach

One of common roles of embedded systems is to work as a controller in a "control theory" sense. In this chapter we used a closed-loop approach to control the temperature. The feedback is provided using the temperature sensor. The controller scheme that we used is a Bang-bang controller, also known as hysteresis controller. A Bang-bang controller is particularly useful here since the only change that we can make is to switch on or off the cooler or the heater.

An open-loop approach would be to switch the heater/cooler on and off regularly without reading the actual temperature. The period and the duty cycle (borrowing terminology from Pulse-Width Modulation) must be calculated and fixed beforehand and will not change when the system is working. For example, for keeping the temperature at 24℃ the heater is turned on for 1 second and then turned off for 3 seconds. This repeats over and over. This open-loop approach is useful if we assume that almost everything is fixed. For example, the following must not change: (1) room's thermal characteristics, (2) the outside ambient temperature, and (3) heating/cooling output power of the heater/cooler. However, in reality these will change and therefore an open-loop controller will not be able to keep the temperature at the desired level. Examples for factors that affect room's thermal characteristics include number of people in the room as well as the windows being open or closed.

1.2 Work Flow

Copy "chap2" from "skeleton" (/courses/TDDI11/lab/skel) directory to your local directory (we assume you have a userID/TDDI11 directory in your account)

```
cp -r /courses/TDDI11/lab/skel/chap1 /home/userID/TDDI11
```

1.

Please note that userID is the user name that you use to login to the lab computers. Now let us change to the new directory and check it:

```
cd /home/userID/TDDI11/chap1
ls
```

Check to see if "chap1" directory contains the following:

- main.c
- inputData.txt
- outputLog_Correct.txt
- displayLog_Correct.txt

The source code is in "main.c". Some inputs for simulation are provided in "inputData.txt". The correct outputs for these inputs are logged in "outputLog_Correct.txt". The corresponding display log is in "displayLog_Correct.txt". In the assignment, you will use "inputData.txt" for simulation and compare your results with correct results that are provided in "outputLog_Correct.txt" and "displayLog_Correct.txt".

1.2.1 The Source Code

Please note that the area in the source code marked with "**`// Assignment`**" are not correct. These are areas left to be completed as part of the assignment.

The acquisition of inputs is simulated by a function called "readInput". In an embedded system such a function will read the values from an input register. In this lab, we read from a file, instead. Each line in "inputData.txt" contains a sample which is a simulated readout of the input register. Please ignore this function:

```
unsigned int readInput(unsigned int sampleIndex)
```

The enforcing of output is simulated by a function called "writeOutput". In an embedded system such a function will write the outputs determined by the program to the output register. In this lab, we write to a file, instead. Each line in "outputLog.txt" contains an output value that is determined based on a corresponding input value. Please ignore this function:

```
int writeOutput(unsigned int outputCurrent, unsigned int sampleIndex)
```

The main part of the code is in an infinite loop. This loops has three main sections: (1) Read inputs and sensor data; then convert them to meaningful information. (2) Decide on outputs. (3) Write outputs.

Extracting Inputs

For extracting and isolating the related bits out of the input register, we use shift and mask operations. Masking means that we change the bits that we do not need to zero while keeping the important bits unchanged.

For example, humidity sensor data in the input variable can be visualized as "`XXXX XXXX XIII IIXX XXXX XXXX XXXX XXXX`" assuming that "`I`" is an input bit from the humidity sensor and "`X`" is a don't-care bit. To extract humidity value, we use a mask that is all 0 but at bits 18 to 22. The mask is "`0000 0000 0111 1100 0000 0000 0000 0000`" in binary. The hex equivalent is `0x007C0000`. We

1.

use bitwise AND operation to filter out all irrelevant bits in the "inputRegister". Bitwise AND operation with 0 (false) results in 0, wiping out the unrelated bits. For example:

```
inputRegister:  "XXXX XXXX XIII IIXX XXXX XXXX XXXX XXXX" &
mask:     "0000 0000 0111 1100 0000 0000 0000 0000"
Results in:  "0000 0000 0III II00 0000 0000 0000 0000"
```

Bitwise AND operation with 1 will not change the value of the other bit, keeping the needed data intact.

For example:

```
inputRegister:  "XXXX XXXX X101 01XX XXXX XXXX XXXX XXXX" &
Mask:     "0000 0000 0111 1100 0000 0000 0000 0000"
Results in:  "0000 0000 0101 0100 0000 0000 0000 0000"
```

Then we shift the remaining bits to end up with sensor data.

```
humiditySensor = (inputRegister & 0x007C0000) >> 18

For example: "0000 0000 0101 0100 0000 0000 0000 0000" >> 18
Results in:  "0000 0000 0000 0000 0000 0000 0001 0101"
```

The isolated bits must be then interpreted using tables and equations given in section 2.1. For example "1 0101" is the humidity sensor data value which must be converted to humidity percentage using table 6 and equation 3.

Based on information gained in section (1) of the code, the outputs are determined. For example if it is colder than desired, the heater must be turned on, if it is too dry, the warning light for low humidity must be lit, and so on.

Combining Outputs

Now that the outputs are determined, they must be combined into a single variable. Section (3) in the code combines these to be written to the output register. To combine these, shift and bitwise OR operations are used. For example low humidity warning light control bit (o_7) can be visualized as:

"YYYY YYYY YYYY YYYY YYYY YYYY OYYY YYYY" , assuming that "O" is the output bit and "Y" is a don't-touch bit that its value must not change. The output value "O" must be first shifted:

```
    "0000 0000 0000 0000 0000 0000 0000 000O" << 7
Results in:  "0000 0000 0000 0000 0000 0000 O000 0000"
```

Bitwise OR operation with 0 (false) will not change the value of the other bit, keeping the outputs written by other parts of the program intact. For example:

```
outputRegister:"1001 1001 1001 1001 1001 1001 O101 1001" |
Shifted bit: "0000 0000 0000 0000 0000 0000 O000 0000"
Result in:"1001 1001 1001 1001 1001 1001 O101 1001"
```

Since before writing anything else in the output variable, we clear it and since the other parts of the program do not change bits that are not for their use, the bits related to the this part of the program are 0. Therefore, a bitwise OR operation with the determined value will correctly place them in the output variable.

```
outputRegister:"YYYY YYYY YYYY YYYY YYYY YYYY 0YYY YYYY" |
Shifted bit: "0000 0000 0000 0000 0000 0000 O000 0000"
Result in:"YYYY YYYY YYYY YYYY YYYY YYYY OYYY YYYY"
```

Pay attention that in this case, as define in table 11, writing 1 actually turns off the light.

1.

Another aspect of bit operations is implementing simple multiplications with shift and summations. On some platforms, this approach might be faster or consumes less power. We utilize the fact that a single shift left is multiplication by 2. For examples: $(a \times 2) = (a << 1)$; $(a \times 3) = ((a << 1) + a)$; $(a \times 4) = (a << 2)$; and so on.

Compile and Run

Compile and run the code with:

```
gcc main.c -std=c99
./a.out > displayLog.txt
```

For debugging purposes, we have the possibility to stop the loop by "getchar()" and see the output on the terminal. Alternatively, we can redirect the display output to "displayLog.txt" by "./a.out > displayLog.txt". To be rigorous you can use "Wall", "Wextra", and "pedantic" while compiling.

Please note that the c code can be understood only by referring to tables and equations given in section 1.1.

In each of the loop, a new sample from the input file is read and processed. When we reach to the end of file, the program ceases execution. In reality, it will run as long as the system is powered and "on". There will be no end of file in real life.

1.2.2 Outputs

The input and output register value are not human readable. We can compare the output values with the correct ones in "outputLog_Correct.txt". The terminal log, "displayLog_Correct.txt", is however human readable. The following is displayed for a single input

```
inputCurrent=246192
inputs: Lock=0, Fan=0, AC=1, Desired=21.000000, Measured=30.000000, Humidity=0
outputs: Green=1, Red=0, Fan=3, Cooler=1, Heater=0, Humidity2H_n=1, Humidity2L_n=0
```

first line, "inputCurrent", shows the input sample as a decimal value. The interpreted inputs are displayed in the next line. The open locks means green signal as shown in the outputs printed in the third line. It is warmer than desired, therefore, the cooler is on. It is too dry, therefore, dryness warning is on (pay attention that the signal is active low and "`0`" means "`true`").

1.3 Evaluation

1.3.1 Assignments

Compile and run the source code and then compare the results (displayLog) with the correct one. Read and understand the source code. Pay attention that the area marked by "**`// Assignments`**" indicates missing code or incorrect code that you need to update, change, or add new lines to it, in order to correct the program. It is needed to go back and forth between the lab manual section 1.1, the source code, its output, and the correct output.

Identify the correct behavior and correct the source code by adding missing code and correcting the wrong existing code. The corrected code must generate correct outputs similar to the "_Correct" files.

1.

Discussions

Please briefly discuss the answers to the following questions. Write just a few sentences.

1. We discussed combining outputs with shift and bitwise `OR` in section 1.2.1. Can we "add" (+) different outputs instead of performing bitwise `OR` operations (section 3 of code inside the main loop)? Why?

3. Do a brief research on Bang-bang (Hysteresis) controllers in relation to equations 4 and 5. Answer the following questions in a few words:

4. Which part of equations 4 and 5 represents "hysteresis" in our system? How much is the value of "hysteresis"?

5. Why do we need hysteresis in our system? What will happen if its value is zero?

1.3.2 Demonstrations

Run the program and show the code to the lab assistant. Make sure that you have used the given inputData.txt and compare your output files against the "_Correct" files. Briefly discuss the answers to the above questions.

1.3.3 Deliverables

- The corrected source code
- outputLog.txt
- displayLog.txt
- Answers to the above questions

After demonstrations and reception of an "ok" from your lab assistant, send her/him an email with the deliverables. Write in the subject: TDDI11 Chapter 1.

1.

Chapter 2 Mixing Assembly and C[1]

The goal is to learn how to use C and assembly in the same program and become aware of performance issues. We will implement a 64-bit multiplication. The related files are in the following directory:

/courses/TDDI11/lab/skel/chap2

Please copy them as usual to your own account. Your modifications go to "llmultiply.asm" and "main.c". The rest of the files are the usual files that we used before.

2.1 Assignment 1, Assembly Implementation

Processors used in a small (embedded) part in a mass-produced product have to be very cheap. Therefore, they are quite rudimentary. For example, they seldom have floating point arithmetic capability, or they may be limited to 8-bit integer operations. Software must compensate for the limited hardware capability. We will look at an example in which we want to multiply two numbers larger than the hardware supports natively. On an 8-bit processor this could involve multiplication of two 16-bit integers. As our target already can do that we will look at multiplying two 64-bit operands on our target machine that natively supports only 32-bit operands.

Our compiler supports 64-bit integers with a data type called "long long int". Since the registers of the Intel processor are only 32-bits wide, how does the compiler generate code to implement $a \times b$ when a and b are long long int's? This is discussed later on in this chapter.

Write a function in assembly that has the following C signature:

```
void llmultiply(unsigned long long int a, unsigned long long int b, unsigned char *result);
```

The function multiplies the two 64-bit parameters a and b. The result of the multiplication is a 128-bit number. It has to be copied to the array of 16 bytes that is pointed to by result. Note that the x86 machines are little-endian, meaning that the least significant byte of a multi-byte number is placed at the lower address, and the most significant byte of a multi-byte number is placed at the higher address.

Based on the explanations in section 2.4, you will implement your multiplication function and test it with tests given in section 4.5. These tests are also given in "main.c". Test your function with at least the given test cases. The given test cases are tailored to generate carries in all possible steps of the calculation.

[1] Note that the structure of the chapters are not identical. Please read them from beginning to the end, before you get to work. Some concepts may be explained at the end of a chapter.

1.

2.2 Assignment 2, C Implementation

Implement the same function, but this time in C. Here you must make sure to use appropriate data type for the multiplication and addition in order to be able to store the entire result in a sufficiently large type.

- Compile it **without** any optimization and verify that it gives correct results.
- Compile it **with** optimization turned on and verify that it gives correct results.

The compiler optimization options are described in section 2.7.

2.3 Assignment 3, Performance Comparison

Call "llmultiply" from a C program in which you test the function. Put the invocation of the function in a loop in order to invoke it many times. Obtain the time or the contents of the CPU cycle register before entering and after exiting the loop. Print the difference of the values before and after the loop. When emulating on Qemu, depending on the host machine, the results might not make sense. The important thing is that you write the program correctly. No worries if Qemu returns wrong values.

Test both your C-version (optimized and non-optimized) and your Assembly-version in the same way (the same number of iterations). Which version is most effective? How big improvement does compiler optimization give?

Note that depending on the computer-architecture and the compiler, the optimization benefits might be partly resulted from optimized loop implementation that affects the testing-loop execution not necessarily the actual instruction inside the loop.

2.4 Multiplication theory

Consider that each 64-bit operand can be split in two 32-bit parts, one contains the high order bits and the other one contains the low order bits.

$$a = a_h \times 2^{32} + a_l$$

$$b = b_h \times 2^{32} + b_l$$

If we break the operands to two parts as suggested above and perform the multiplication with these separated parts we get:

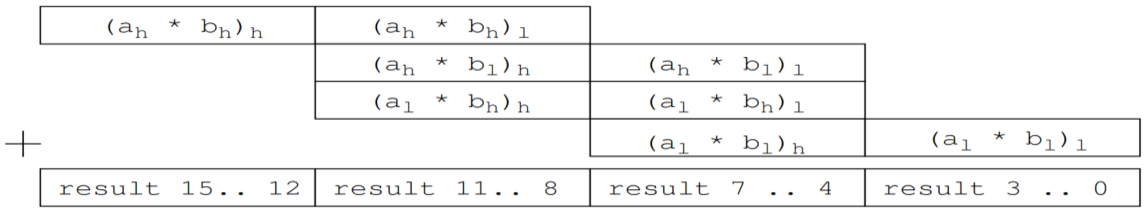
$$a \times b = (a_h \times 2^{32} + a_l) \times (b_h \times 2^{32} + b_l) =$$

$$\begin{aligned} &= a_h \times b_h \times 2^{64} + \\ &+ (a_h \times b_l + a_l \times b_h) \times 2^{32} + \\ &+ a_l \times b_l \end{aligned}$$

All multiplications that appear in the above expressions are now 32-bit multiplications and therefore they can be implemented on an Intel x86 processor. Your only concern is to handle the additions and carries that may appear after the additions. Note that each 32-bit multiplication yields a 64-bit result, but you can only add 32-bits in each addition. A graphical view of the procedure is given below. Think carefully of

1.

where carries can occur. In the figure below, in the lower box that shows the results, the numbers represents a numeration of the bytes. For example “result 3 .. 0” represents 4 bytes that have the lowest significant bits.



Be careful of the carries when performing the additions. You should whether there will be carry outs in C. in assembler, you might want to use the ADC instruction. For multiplication, you might want to use the MUL instruction. Observe that in C, if a_l and b_l are unsigned long ints, then $(a_l * b_l)$ will also be an unsigned long int, even if you store it in an unsigned long long int. You should cast a_l and b_l into two unsigned long long ints in order to obtain an unsigned long long int as a result of the multiplication.

2.5 Test Cases

We provide the following test cases. You should of course add your own. All digits are hexadecimal.

```
0000111122223333 * 0000555566667777 = 0000000005B061D958BF0ECA7C0481B5

3456FEDCAAAA1000 * EDBA00112233FF01 = 309A912AF7188C57E62072DD409A1000

FFFFEEEEDDDDCCCC * BBBBAAAA99998888 = BBBB9E2692C5DDDCC28F7531048D2C60

FFFFFFFFFFFFFFFF * FFFFFFFFFFFFFFFF = FFFFFFFFFFFFFFFE0000000000000001

00000001FFFFFFFF * 00000001FFFFFFFF = 0000000000000003FFFFFFFC00000001

FFFEFFFFFFFFFFFF * FFFF0001FFFFFFFF = FFFE0002FFFDFFFE0001FFFE00000001
```

2.6 C Function Call Interface

When writing the assembly code that cooperates with C-code you have to follow the compilers idea of how to pass parameters to a function. The parameters are passed on the stack. The stack grows from large addresses to small ones. The last parameter to a function is pushed first on the stack. Hence, the last parameter will be at a larger address than the first parameter. The stack pointer points to the top of the stack and not to the first free location! That means that pushing a value on the stack first decrements the stack pointer and then writes the value.

To understand better, look at the following snapshot of a stack frame just after entering the function:

```
byte 0 of return address | 0x3fffffe8 <-- stack top (esp)
byte 1 of return address | 0x3fffffe9
byte 2 of return address | 0x3fffffea
byte 3 of return address | 0x3fffffeb
;; The first parameter (a) start here.
;; Notice the 32-bit little endianess:
;; The least significant byte come first (byte 0)
;; The most significant byte come last (byte 3)
byte 0 of a, byte 0 of al| 0x3fffffec
byte 1 of a, byte 1 of al| 0x3fffffed
```

1.

```
byte 2 of a, byte 2 of al| 0x3fffffee
byte 3 of a, byte 3 of al| 0x3fffffef
byte 4 of a, byte 0 of ah| 0x3ffffff0
byte 5 of a, byte 1 of ah| 0x3ffffff1
byte 6 of a, byte 2 of ah| 0x3ffffff2
byte 7 of a, byte 3 of ah| 0x3ffffff3
;; The second parameter (b) start here.
byte 0 of b, byte 0 of bl| 0x3ffffff4
byte 1 of b, byte 1 of bl| 0x3ffffff5
byte 2 of b, byte 2 of bl| 0x3ffffff6
byte 3 of b, byte 3 of bl| 0x3ffffff7
byte 4 of b, byte 0 of bh| 0x3ffffff8
byte 5 of b, byte 1 of bh| 0x3ffffff9
byte 6 of b, byte 2 of bh| 0x3ffffffa
byte 7 of b, byte 3 of bh| 0x3ffffffb
;; The third parameter (c) start here.
;; Notice that only the address to the array is passed.
byte 0 of result array address| 0x3ffffffc
byte 1 of result array address| 0x3ffffffd
byte 2 of result array address| 0x3ffffffe
byte 3 of result array address| 0x3fffffff <-- stack bottom
```

Typically, a function has the following prologue:

```
push ebp   ;; save the value of ebp register on the stack

mov ebp, esp ;; save the address of the stack frame

               ;; (the value of esp after entering the function)
```

The reason for the prologue is to get a fix base pointer for convenient access to the function parameters (the stack pointer esp may be changed in the function to store local variables). It also makes printing of a stack trace easy, which is an important debug feature. After the prologue ebp will contain the value `0x3fffffe4`, i.e. the value of the stack pointer before pushing ebp, that is `0x3fffffe8`, minus the four locations occupied by ebp. The stack will now look like this:

```
byte 0 of previous stack frame (ebp) | 0x3fffffe4 <-- top (ebp, esp)
byte 1 of previous stack frame (ebp) | 0x3fffffe5
byte 2 of previous stack frame (ebp) | 0x3fffffe6
byte 3 of previous stack frame (ebp) | 0x3fffffe7
byte 0 of return address    | 0x3fffffe8 <-- previous top
byte 1 of return address    | 0x3fffffe9
byte 2 of return address    | 0x3fffffea
byte 3 of return address    | 0x3fffffeb
byte 0 of a, byte 0 of al   | 0x3fffffec <-- parameter 1
byte 1 of a, byte 1 of al   | 0x3fffffed

…
byte 4 of b, byte 0 of bh   | 0x3ffffff8
byte 5 of b, byte 1 of bh   | 0x3ffffff9
byte 6 of b, byte 2 of bh   | 0x3ffffffa
byte 7 of b, byte 3 of bh   | 0x3ffffffb
;; The third parameter (c) start here.
;; Notice that only the address to the array is passed.
byte 0 of result array address | 0x3ffffffc
byte 1 of result array address | 0x3ffffffd
```

1.

```
byte 2 of result array address | 0x3ffffffe
byte 3 of result array address | 0x3fffffff <-- stack bottom
```

To illustrate how you can get that said convenient access to the parameters, let's add the values of b_h and a_l as an example (this addition is of course irrelevant for the assignment). You will find b_h 20 bytes from the address in ebp (count in the picture). To get the value of b_h to register eax we write:

```
mov eax, [ebp + 20]
```

Then we fetch a_l, but as eax now is occupied we have to use another destination register:

```
mov ebx, [ebp + 8]
```

And to add the two (with the result in eax) we do:

```
add eax, ebx
```

To make your code more readable it is good to use symbolic names for the offsets:

```
mov eax, [ebp + BH_OFFSET]

mov ebx, [ebp + AL_OFFSET]

add eax, ebx
```

As an exercise, think of how to load the address of the first byte of the result array in ebx.

Since we push ebp in the prologue inside the function we should restore the previous value before we return. We also have to make sure the stack pointer point to the same address it had when we entered the function, in order to use the correct return address. This is the function epilogue that is to pop ebp from the stack before leaving the function.

```
mov esp, ebp ;; perhaps needed...

pop ebp

ret
```

2.7 Compiler Optimizations

In order to compile without any optimization, pass the -O0 switch (minus capital O -- as in "Origami" -- followed by zero) to the gcc compiler. In order to compile with full optimization, pass -O3 to the compiler. You can do it by modifying the CFLAGS in your Makefile.

2.8 Deliverables

Demo the application for the lab assistant. Present your conclusions with respect to the three run times. Send the lab assistant an email with the assembly and C implementations of your "llmultiply" function. The subject should be "TDDI11 chapter 2".

2.9 Resources

NASM tutorials: https://cs.lmu.edu/~ray/notes/nasmtutorial/ , https://asmtutor.com/

Instruction set: https://docs.oracle.com/cd/E19455-01/806-3773/index.html

1.

Chapter 3 Polling and Interrupt-Driven I/O

In this chapter we work with serial-port read and write operations in Assembly. We become aware of the various ways to communicate with the peripherals. Understand their advantages and disadvantages. The related files are in the following directory:

/courses/TDDI11/lab/skel/chap3

Please copy them as usual to your own account. Your modifications go to “serial.asm”.

3.1 Assignment

You use a program that reads characters typed on the keyboard and sends the read characters on the serial interface. Another computer will be connected to this serial communication line through its serial port. This second computer reads the port and prints the received characters on the screen. The second computer can read the keyboard and send the read characters serially to the first computer. At the same time, both computers display characters read from the serial interface. Both computers, display in a separate window, the typed character read from their own keyboards. In principle the programs for the first and second computer are identical.

You will develop assembly code to write the serial port by polling. With polling you continuously test a bit of a register. If the bit is not set, the serial interface is busy sending another character, and therefore you should not write the new character to the serial interface. If the bit is set, the serial interface is accepting new characters to be sent (transmitted).

Write the assembly code to read characters from the serial interface by interrupts. Upon receiving an interrupt, the serial interface controller notifies the CPU by means of an interrupt. The serial port share interrupt signal with other devices, so you must still check that a byte really arrived at the serial port before reading it.

Edit the file “serial.asm” to fill-in the missing details of the polled waiting-loop output function (SerialPut) and the serial input ISR (SerialISR).

In order to be able to test the program, you need to emulate two computers connected by a serial line. You may check the “makeNrun.sh” to see how this is done. You will only need to implement the methods “SerialPut” and the “SerialISR” in “serial.asm” file.

3.2 Deliverables

Demo for the lab assistant. Email (subject “TDDI11 chapter 3”) the file “serial.asm” with the code written in “SerialPut” and “SerialISR” sections.

3.3 Background

We discuss two ways of communicating with peripherals: polling and interrupts.

1.

Polling means continuously checking the peripheral to detect if it has changed state. For example, if the keyboard is expected to place the ASCII code of a pressed key in a certain register of the keyboard controller, polling would imply continuously reading that register to see if the value changes. We can see immediately the disadvantage: instead of using the processor to do something useful, we waste resources checking something that happens with a low rate.

An alternative to polling is interrupt-driven communication. Instead of the processor checking the peripheral each time, the peripheral notifies the processor if its state has changed. The peripheral sends a signal that interrupts the execution of the processor. The execution jumps to a predefined address where the Interrupt Service Routine (ISR) must reside. The ISR implements the response to the interrupt. In this way, the processor does not need to frequently check the peripherals.

Nevertheless, polling may have advantages too. For example, there might be intervals when we want to ignore the peripheral and to run uninterrupted. If we used interrupt-driven communication the processor would be interrupted even if it decides not to service the interrupt. Temporarily disabling the interrupts may not be a good idea if we want not to be interrupted, because we could be interested in other interrupts.

Polling can also have advantage when communicating with a device that operate almost as fast as the processor. If we use interrupts many cycles will go to waste in the overhead of doing the interrupt, while polling may be able to read one data every iteration in the poll loop.

In such cases DMA is another solution, in which a large block of data is transferred to memory without CPU intervention, and an interrupt signal delivered when the transfer is done, and the CPU need to provide a new empty buffer. While the new buffer is filled, the CPU can process the previous buffer content in parallel. This approach requires extra hardware and must be supported in the platform. Many embedded platforms might not support this. The main part of the extra hardware is sometimes called a DMA controller.

3.4 The IBM-PC Serial port

The IBM-PC typically supports two (or more) serial ports, called COM1 and COM2. These devices are implemented using an I/O chip called a Universal Asynchronous Receiver-Transmitter (UART). UARTs convert the 8-bit parallel format used inside the PC to a serial formation in which information is transmitted one bit at a time, and vice versa. This is sometimes called serial to parallel and parallel to serial conversion.

Each serial port is assigned a block of eight I/O port addresses, starting with a “base” address of 0x2f8, 0x3F8, 0x2E8, or 0x3E8. These addresses typically correspond to COM1, COM2, COM3, and COM4 respectively. These port addresses are used with I/O instructions to write data or commands into the internal registers of the UART, or to read data or status information. The three most important registers are:

1.

1. LSR (Line Status Register) I/O Port

Base+5 (Read-Only)

7	6	5	4	3	2	1	0
FIFO Error	Transmitter Empty	**THRE**	Break Detected	Framing Error	Parity Error	Overrun Error	**RBF**

For this lab, only two status bits are important:

Bit **5**, THRE: (Transmitter Holding Register Empty): This bit is 1 when the UART is ready to transmit another byte of data out on the serial line. Wait for this bit to become 1 before writing to THR.

Bit **0**, RBF: (Receiver Buffer Full): If this bit is 1, input data is available in RBR.

2. THR (Transmitter Holding Register) I/O Port

Base+0 (Write-Only) Data to be transmitted serially to a remote computer may be written to this port when bit THRE in the LSR is 1.

3. RBR (Receiver Buffer Register) I/O Port

Base+0 (Read-Only) Data received serially from a remote computer may be read from this port when bit RBF in the LSR is 1.

These two registers (THR and RBR), albeit being physically distinct, have the same address. System can tell them apart since THR is write-only and RBR is read-only. When writing to this address, data is copied to THR and when reading from this address, data is copied from RBR.

To write a byte to the serial port you must first make sure the device is ready. This is done by reading the Line Status Register (LSR) and extracting bit 5 which is called Transmitter Holding Register Empty (THRE). If it is “1” it means the previous value in Transmitter Holding Register (THR) has been transmitted, and you can safely proceed by writing a new value to the THR. If bit 5 is not set you must first wait for it to become set. Check/Poll it until the previous data is transmitted.

To read a byte from the serial port by using interrupts, the address to an interrupt handler must be placed in the processors interrupt service vector. (In the lab this is already done for you.) In the interrupt handler you must then determine what caused the interrupt. Besides data arrival (getting a character on the serial line), other events such as “break detected”, some errors, and so on may cause interrupts. To see if a new character is available on the serial port read the Line Status Register (LSR) and extract bit 0 which is called Receive Buffer Full (RBF). If it is set, new data has arrived, and you can safely proceed by fetching it from the base port and storing it in the byte at address data in serial.asm before passing the address data and the address [inbound_queue] to QueueInsert (see libepc.h under the includes used by the Makefile). If RBF is not set it means the interrupt was for something else and you should do nothing (as we only care about the serial port). In both cases you have to send the End Of Interrupt (EOI) to the interrupt controller. This is done by writing 0x20 to port 0x20 and returning to interrupted code (see iret instruction). Do not forget to save modified registers!

1.

3.5 The x86 I/O Instructions

The 80x86 supports two I/O instructions: “in” and “out”. They are like very limited versions of the “mov” instruction, with the major difference that they do not access the normal memory bus (or the usual register bank), but instead a special I/O bus, with 65536 I/O ports (a port is an address on the I/O bus, but named port to distinguish it from a memory bus address). The I/O instructions take the forms:

```
in eax/ax/al, port

in eax/ax/al, dx

out port, eax/ax/al

out dx, eax/ax/al
```

The “in” instruction must always have (part of) eax as destination, and can read the port address either as a directly specified 8-bit port number (0 - 0xFF) or use the full 16-bit port number in dx. In most cases you must thus load the port number to edx first. The out instruction is exactly the same, but reverses the source and destination. Neither instruction modifies any flags in the flags register.

3.6 Resources

NASM tutorials: https://cs.lmu.edu/~ray/notes/nasmtutorial/ , https://asmtutor.com/

Instruction set: https://docs.oracle.com/cd/E19455-01/806-3773/index.html

To better understand I/O with polling and interrupts the structure of the x86 system you can read chapters CH03.5, CH17-3, and CH22-1 in http://flint.cs.yale.edu/cs422/doc/art-of-asm/pdf/

1.

Chapter 4 Preemptive Multi-Threading (uC/OS-II)

In this chapter we will work with preemptive kernels and understand the critical region concept. The related files are in the following directory:

/courses/TDDI11/lab/skel/chap4

Please copy them as usual to your own account. Your modifications go to “packet.c”.

4.1 Background

uC/OS-II is a preemptive real-time multitasking kernel. It is used to build “event-driven” applications in which each thread sleeps (in a “suspended” state) until an external event triggers an associated interrupt. With this kind of kernel (preemptive) the threads do not have to cooperate to get execution time. The kernel makes sure that all threads get a chance to execute.

Since the kernel is preemptive, a higher priority thread can suspend an active thread at any point during its execution, forcing it to release the processor. This may significantly reduce response time compared to a non-preemptive system (such as Multi-C in previous lab) in which the active thread must explicitly call a kernel function to yield to other threads. In a preemptive system, external events can trigger a context switch by manipulating the registers and stack within an ISR so that it returns to the higher priority thread instead of the one that was interrupted.

As a consequence, data inconsistency or corruption may occur. For example, if a thread is interrupted in the middle of sending a packet and someone else uses the serial line before it get a chance to complete the packet. To prevent corruption, data is often communicated between threads and ISRs using kernel resources such as mailboxes and queues. Shared data or other resources are protected by using kernel mutex-es or semaphores.

4.2 Assignment 1

This program is similar to the previous, except that in addition to establishing a two-way chat with a display of local elapsed time, each application also displays the remote elapsed time. This requires that each application send and receive both chat text and time text over a single serial line. To distinguish arriving chat text from arriving time text, information is sent as packets in the following format:

0xff	Type	N	Byte 1	Byte 2	…	Byte N
Start Flag	0x01=Chat 0x02=Time	Byte Count	Actual Data Bytes			

One thread assembles incoming packets from a queue filled with data received by the serial ISR. As each packet is assembled, its data is posted to one of two queues according to the type code. Data is removed from these queues by two more threads that update the display.

Implement “SendPacket” function in “packet.c”. Function prototype is:

```
void SendPacket(int type, BYTE8 *bfr, int bytes);
```

1.

Send each byte of the packet by calling the function “SerialPut” in “serial.c”. Compile, run, and test your application. You should be able to send and receive text and both time displays (local and remote) should be running at the same time.

4.3 Assignment 2

Fix packet corruption problem in “SendPacket”. If you type very fast, the person at the other computer should notice intermittent corruption of the time displayed in their remote time window (especially when you hit enter or introduce delay intervals with “OSTimeDly(10)” between the SerialPut calls for sending bytes). That’s because your keyboard interrupts can initiate transmission of a chat packet in the middle of transmitting a (14-byte) time packet from your computer to theirs. To understand, consider that SendPacket(2, “12:34:56.7”, 11) is called and start sending:

```
0xff 02 11 ’1’ ’2’ ’:’ ’3’ ’4’ ’:’
```

Imagine now that the user start to type on the keyboard and SendPacket(1, “a”, 1) is invoked, and the running “SendPacket” preempted. The following is sent:

```
0xff 01 01 ’a’
```

Now SendPacket(1, “a”, 1) is completed and SendPachet(2, ..., 11) continues sending:

```
’5’ ’6’ ’.’ ’7’ 00
```

The receiving side will now receive the following timer package:

```
0xff 02 11 ’1’ ’2’ ’:’ ’3’ ’4’ ’:’ 0xff 01 01 ’a’ ’5’
```

Followed by the following bytes that have invalid packet header and are discarded:

```
’6’ ’.’ ’7’ 00
```

To solve this problem, you must consider the code in “SendPacket” to be a critical section and protect it against preemption by a semaphore. To do this, you’ll need three mCOS kernel routines:

```
OS_EVENT *OSSemCreate(int count);
```

This function allocates and initializes a semaphore data structure and returns a pointer to it. The parameter “count” is set to 1 for a mutex lock (what you need).

```
OSSemPend(OS_EVENT *semaphore, int timeout, BYTE8 *err);
```

This function returns when it acquires the semaphore; if the semaphore is currently owned by another thread, this function causes the current thread to be suspended while it waits for the semaphore to be released. The parameter “semaphore” is the pointer returned by “OSSemCreate”. Set parameter “time-out” to 0 for no time-out. Parameter “err” is a pointer to a byte that is set to an error code if a time-out occurs.

```
OSSemPost(OS_EVENT *semaphore);
```

This function causes the current thread to relinquish ownership of the semaphore. The parameter “semaphore” is the pointer returned by OSSemCreate.

1.

4.4 (Optional): Assignment 3

Display a count of chat text characters received by the remote computer:

1. Count the number of chat text characters received.

2. Convert the integer count to an ASCII string using the libepc function FormatUnsigned (see example in elapsed.c).

3. Send this string to the remote computer using a packet type code of 3:

0xff	Type	N	Byte 1	Byte 2	…	Byte N
Start Flag	0x01=Chat 0x02=Time 0x03=CharCount	Byte Count	Actual Data Bytes			

4. Modify ReceivePackets to handle the new packet type (0x03). Use the code for displaying the elapsed time of the remote computer as a guide. This will require creating another queue and another thread.

5. Display the count in a new window located in the middle of the screen.

4.5 Deliverables

Demonstrate for the lab assistant. Then, send an email with subject “TDDI11 chapter 4” to your assistant.

1.

Chapter 5 State Machines

State machines (more precisely finite-state machines) provide a methodology to design, analyze, and formally communicate the functionalities of embedded systems. We will implement the classical Tic-Tac-Toe 2-players game as a finite state machine on the STM32L562 Discovery Kit.

First, we briefly describe how to download a skeleton for the lab and how to run and debug it on the STM32L562 board. Then, we describe the state machine to be implemented in the skeleton.

5.1 Getting started

In a lab room[2] using you LiU account:

- copy the project:

  ```
  cp -r /courses/TDDI11/lab/skel/chap5 /home/userID/TDDI11
  ```

- Load the "prog/stm32cubeide/1.14.0" module to access the stm32cubeide and the stlink-server for interfacing with the STM32L562 board:
  - Each time you start a terminal where you want to use stm32cubeide or stlink-server:

    ```
    module add prog/stm32cubeide/1.14.0
    ```

  - Or, add the module the the shell init file to avoid having to that for each new terminal (see `module -help`)

    ```
    module initadd prog/stm32cubeide/1.14.0
    ```

- Run the stlink-server from one terminal

  ```
  stlink-server
  ```

- Run the ide from another terminal (or run the stlink-server in the background)

  ```
  stm32cubeide
  ```

- In the IDE, choose "`File > import > General > Projects from Folder or Archive`". Press "`Next`" and "`Directory`" and choose "`/home/userID/TDDI11/chap5/STM32CubeIDE`" assuming you copied "`chap5`" under "`/home/userID/TDDI11`" as described above.

- Connect the board to the board to the computer via the "`CN17 STLK`" micro USB port. In the IDE, got to "`TicTacToe > Example > User > main.c`". You do not need to modify any other file for this lab. Except for you liuIDs at line 37 of "`main.c` " (important!), the areas you need to modify are delimited by the two comment lines:

  ```
  // *********************** LAB: Do not modify before this line *********************
  ```

and

```
// *********************** LAB: Do not modify after this line *********************
```

[2] You can download the IDE to your own laptop from "https://www.st.com/en/development-tools/stm32cubeide.html". Make sure to select the "`stlink-server`" during installation. The remaining steps are similar.

1.

- Click in the `main.c` file and run or debug the skeleton code. It already implements part of the targeted finite state machine. During debugging, we recommend to place breakpoints at one, or both of, the switch statements.

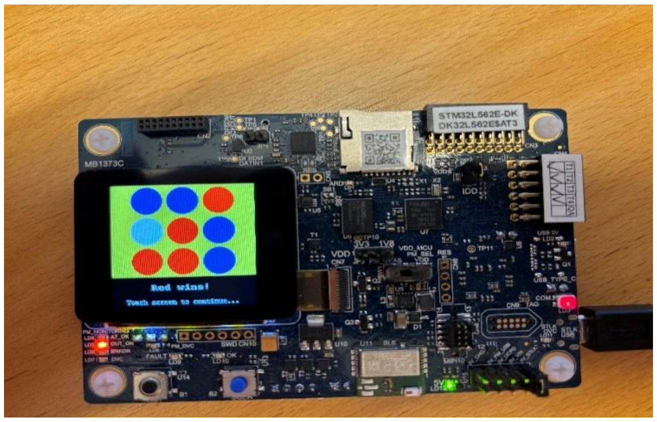


Figure 1. TicTacToe on the STM32L562. You need to implement the game as described in the state machine of Section 5.2

5.2 Tic-Tac-Toe as a state machine

The state machine you need to implement is captured in Figure 2 below. In your implementation, the infinite loop in main should have the "read touch and button", "update state on entry" and "transitions" parts as exemplified in the skeleton:

```
while (1){
        {//Read touch and Button. Do not modify.
          if (TouchDetected == SET)
           …
        }//End read touch and Button. Do not modify.

        // ********** LAB: Do not modify before this line **********************
        //Update State on entry
        if(State != PreviousState){
          switch(State)
          {
             case WELCOME:

          } //update state
        }//on entry

        // transitions
        switch(State)
        {
          case WELCOME:
           …

        }// transitions
        // ********** LAB: Do not modify after this line **********************
}//infinite loop
```

Again, you should only change the "update state on entry" and "transitions" parts.

1.

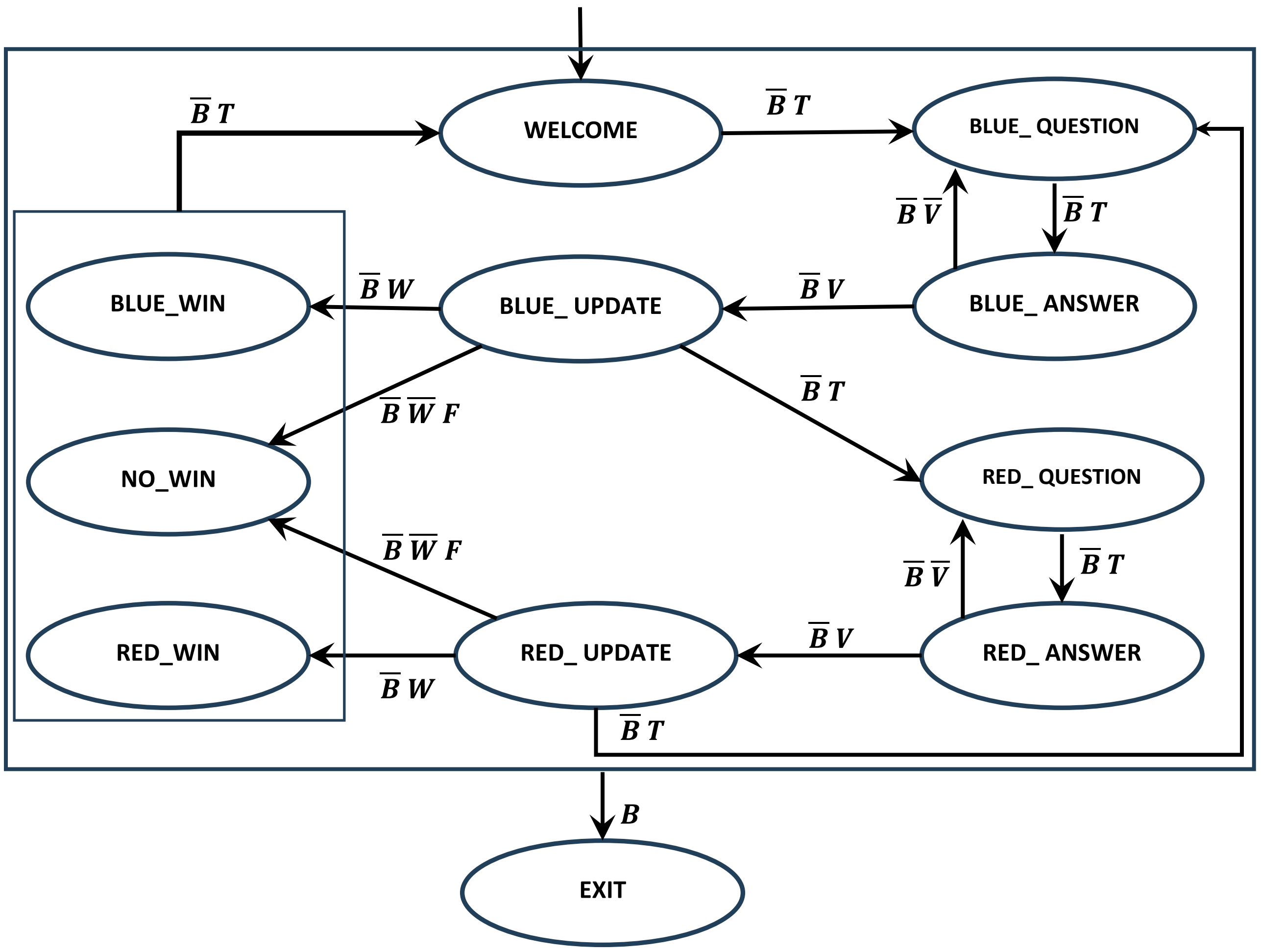


Figure 2 State Machine for TicTacToe. B stands for (pressed) Button, V for Valid (move), T for Touch (screen detected), W for Win, F for Full. Overlines correspond to negations.

As the game proceeds, the previous choices made by the players are stored in a 3x3 matrix. Initially, all cells are free. The matrix is represented by the `board_state` array in column major, i.e., `cell[row][col]` is stored in `board_state[row+3*col]`.

Here comes a brief description of the involved states:

WELCOME	Initial state. Prints name of game, liuIDs of authors and info about effect of touching the screen or pressing the user push button.
BLUE_QUESTION	Prints board state and informs it is the turn of the blue player.
BLUE_ANSWER	Checks if move corresponds to a free cell and computes its row and column
BLUE_UPDATE	Updates board state and updates values of Win and Full variables
RED_QUESTION	Prints board state and informs it is the turn of the red player.
RED_ANSWER	Checks if move corresponds to a free cell and computes its row and column
RED_UPDATE	Updates board state and updates values of Win and Full variables
BLUE_WIN	Prints Blue player won and info about effect touching screen
RED_WIN	Prints Red player won and info about effect touching screen
NO_WIN	Prints the game was a draw and info about effect touching screen
EXIT	Prints name of game, authors' LiUIDs and "Good bye".

1.

For its transitions, the machine will only use values of the variables:

`Button`	"Remembers" if the user button was pressed. Always result in moving to EXIT.
`Touch`	"Remembers" if the screen was touched. Need to be reset to 0 for detecting a touch.
`Valid`	"captures" if move corresponds to a free cell with legitimate row and column
`Win`	"captures" if current player has three cells aligned (horizontally, vertically, diagonally or anti-diagonally).
`Full`	"captures" if no cell is free.

5.3 Evaluation

5.3.1 Assignments

Implement the state machine described above in the `main.c` file. Run the code and test it.

5.3.2 Demonstrations

Demonstrate the state machine before sending the deliverables.

5.3.3 Deliverables

After you demonstrate your code to your assistant, send an email with subject "TDDI11 chapter 5" to your assistant with your `main.c`.