Washing Machine Device Drivers

Real-time Software Engineering (ELC018)

**Jyri Kärpijoki – B212418**

**Jamroz Jamil – A911969**

**Saad Munir – A914620**

Contents

[Introduction: 3](#_Toc349283150)

[Review: 3](#_Toc349283151)

[Design Approach: 5](#_Toc349283152)

[Design Explanation: 6](#_Toc349283153)

[Test Software: 7](#_Toc349283154)

[Using Test Software: 7](#_Toc349283155)

[Future: 7](#_Toc349283156)

Introduction:

Based on the top level design of the washing machine system presented before, this report provides details and discussion on how the initial design was used to create functioning device drivers and their associated tests to demonstrate correct functionality. Please review the contents of this report in conjunction with the code and the executable files attached.

Review:

As highlighted in the earlier document, the features required for the washing machine simulator as follows:

|  |  |
| --- | --- |
| Physical Feature | Operational Feature |
| Accept Button | Pressing the accept button starts the execution of relevant program. |
| Cancel Button | Pressing the cancel button interrupts the running program. |
| Program Selector | 3 program selector buttons are used to choose one of the 6 programs. |
| Door (Switch) | The washing machine door is modelled as a switch containing a LED which lights up when the “door” is in the open position. |
| 7-Segment Display | The display indicates the stage the program has reached. |
| Buzzer | The buzzer sounds if the door is open when the accept button is pressed. |
| Timer | The timer is used to time various stages/cycles of the wash program. |
| LEDs | LEDs are used to indicate the “selected” program on the simulator. |
| DC Motor + Sensor | The DC motor mounted with a plastic disc represents the wash drum which rotates depending on the wash cycle, and its speed is measured using the IR Sensors. |

In addition to the above features:

* The **accept** button only results in the execution of the program if the selected program is legal and if the door is closed.
* Pressing the **accept** button whilst the program is executing causes program status to advance by a stage.
* Pressing the **accept** button when the program is interrupted (by pressing cancel button) results in resumption of the program.
* Pressing the **cancel** button when the program is already interrupted resets the program.
* Operating the **door** **switch** whilst the wash program is operational sounds the buzzer and suspends the program until the door is closed again.

Following a detailed analysis, the structure for the software was designed which was represented by the class diagram below:

|  |
| --- |
|  |
| Figure 1 : Class diagram for the proposed system operation |

Just by a visual inspection, it was evident that a device driver would be required for each component of the simulator. In the diagram above, these are the classes that are connected to the controller and the program class and are listed below:

* Display
* Timer
* Door
* Motor
* IRSensor
* Button
* Buzzer

All the device drivers for the above components except IR sensors are provided along with user friendly tests, to confirm their functionality. For this task, only running of motor in clockwise and anticlockwise direction was required, therefore IRSensor was removed for simplicity.

Design Approach:

Test-driven development (TDD) approach was employed to produce the device drivers. TDD is a software development process which starts by writing tests (usually automated) for the desired function of declared classes or the code. Of course, the tests initially fail as no or only default definitions of the classes are present. This is followed by writing minimum code to pass tests, followed by refactoring and improving of the working code.

This approach forces programmers to think about all the possible scenarios that the code (classes and functions) should be able to handle and their expected outcome along with the added benefit of testing it. Moreover, writing tests helps to improve code coverage, i.e. allows to programmer to run majority of the code in order to mitigate risk of discovering any bugs later on.

As a result, small tests for each class similar to unit tests were written prior to writing any meaningful code. These include tests to check what each function was expected to do and the expected outcome. For example, the getState() function in the Door class should return either “open” or “close”, depending on the state of the door. Unfortunately, due to the nature of the problem at hand, i.e. use of a washing machine simulator, meant most of the tests could not be automated as verification required visual confirmation. For example, for the above mentioned door state test, information about the actual state of the door needs to be provided to the program by visual inspection, before it can confirm that the output from the port matches the actual state. There is no other way to know for sure that the states match otherwise.

In terms of the structure of the program, since an object-oriented approach was being followed and team members needed to work in parallel, it was decided that the best approach would be to split the program into multiple files. In addition, classes were divided into a header and source(.cpp) file where the header contained the declaration for the class whereas the definitions were included in the .cpp file. This has the several advantages, including improved build time but more importantly it makes it unlikely for someone to accidentally modify the class. Moreover, someone unfamiliar with the classes can quickly look at the header file to find out all the available functions, without needing to go through the whole code.

Information common to all and used by many classes such as the memory address for the i8825A programmable peripheral interface (PPI) as well as definitions of various structures was included in the module.h file. This meant that all the classes could access this information simultaneously, and any changes in the future only need to be made to one file. It also helps to keep the program as a whole more coherent and provide consistency by using same variable names and structures.

Based on the unit tests, some changes were made to the classes due to set ‘structure of information transfer’ required for communication between the processor and the simulator, as well as to provide some essential functionality. Some additional functions and parameters were introduced, to increase ease of use of the device driver by a new user, such as getName() function was introduced to the button class so they could be easily identified whilst debugging. These changes are highlighted in the updated class diagram below:

|  |
| --- |
|  |
| Figure 2 : Updated Class diagram |

Design Explanation:

As mentioned earlier, a common design approach was to define addresses, structures and any other information that may be useful for the device drivers in a common module.h header file. This file was included in the declaration for every class. This makes it easier to see all the hardware specific variables in one place, and make changes if need be in the future and is therefore not specific to individual devices.

Another design approach common to all classes, was to create local copies of all the variables or parameters that are used by multiple devices. This specifically refers to the pointer address for the PPI, where a local copy was created to avoid the whole system failing if a single device driver accidentally the global pointer. In addition enums were used wherever appropriate, to limit what the user may enter for specific functions, again to avoid the drivers developing bugs due to unexpected inputs.

Overall, the approach was to provide enough functionality that the user may utilize all possible functions of a device without making any changes to the drivers or damaging it. One can easily learn about all the functions provided by the device driver, by simply looking at the declarations in the header files without ever having to see any definitions.

In order to further demonstrate the capability of the device drivers and familiarize the user with all that can be achieved from the devices, the unit tests were modified to allow the user to explore the functionality they may be interested in. The individual devices tests were then collected into one menu based system, allowing the users to easily test the drivers. These tests also serve the dual function of being example code, where the users can easily see how these drivers are to be used properly.

Test Software:

As mentioned before, test software was written before the class definitions were written. These were modified so that the user could interact with the tests and ask for the ones they are interested in. Tests were written specific to each device separately, in an object oriented manner. They were later collected for the demonstration software where different tests are called through a menu system.

The approach whilst designing these tests was to ensure all components of the code were run at least once, where effort was made to test all possible scenarios. Most tests require visual inspection to get result, so user entry is necessary to confirm some results. This means tests cannot be automated which is perhaps not very convenient, but the limited complexity of the device means that it’s still possible.

Individual tests and their purpose are described in the header of the respective tests.

Using Test Software:

Using the test software is quite straight forward, where a menu based system provides all possible options that the user may test/choose. The tests were all successful on 22nd of February, where a Baud rate of 19200 was used to communicate with the device. Tests can be run by loading the debugRel ARM image through the AXD Debugger onto the ARM Evaluation board. The entry point i.e. the main is in the **demonstrationTest.cpp** file.

Future:

The device drivers provided form the basic building blocks which will be used to produce working code for an operational washing machine simulator. Several functionalities were neglected in the device drivers which are specific to operation of the washing machine i.e. action that will be taken by the ‘accept’ and ‘cancel’ buttons will depend on the program. In addition, due to certain limitation of the hardware some desirable feature such as turning individual LED off cannot be performed without resetting all buttons. Nevertheless, the device drivers are sophisticated enough to program the washing machine simulator with all the required functionality.

The motivation whilst programming has always been to produce comprehensive and bug-free device drivers, where the unit tests ensure proper functionality and the logger implemented make debugging easier. Of course, future changes may be necessary depending on how the devices may be required to interact with controller and the program. For example, the clear function currently being performed by the cancel button should really be performed by controller.

In this way, the device drivers provided allow user to interact with the device, but specific functions for the washing machine simulation may either need to be added or modified or implemented outside the device drivers.

.