TDD In Embedded Systems: A Case Study

Sayali A. Kulkarni, Charudatta V. Kulkarni, Rakesh Mehta
MIT College of Engineering, Pune

Abstract

The complexity and quality of the soft ware's required is increasing day by day, and hence the requirement of various new techniques or methods to tackle and handle them. One such method is the TDD – Test Driven Development in software development. TDD is a technique to build software incrementally. The same method and the process can be effectively used in the development of the embedded system software development. This paper provides a case study of an application specific embedded system, the problems faced in the development and the solutions found using the TDD. The TDD has become very popular and plays a significant role in increasing the efficiency of the programmers and developing a bug free code. This technique not only very easy to implement for those who are used to programming the DLP (Debug Later Programming), hence the efficiency of the programmers and the quality and maintainability of the software is at stake.

This paper provides an example for using TDD in Embedded Software Design and Development. TDD also increases the modularity of the program to a great extent. Hence, the use of TDD can be made a practice in the field of embedded software development.

1. Introduction to TDD

TDD that is the Test Driven Development is an advanced software development technique used for software development. With in the field of software development embedded software is a unique and special. The Higher level of software developments those made in the IT sector generally have very less contact with the physical/ actual world and are run in clean sophisticated environments.

Bugs when present on the IT sector software, enterprise software PC the applications are patched and ready to use. This is relatively easier in contrast with the embedded software which is used to such a great extent in the public domain applications and is in contact with the real world. For example in the embedded application of the automobile sector if a flaw / bug is present in the fuel injection system then this can cause massive explosion which can also result in humane loss.

The result of this flaw can lead to severe financial damage as well. Hence, a small flaw in the software of an embedded system can result in the loss of human life. As the complexities of embedded systems grow and the ubiquity of embedded systems continues to grow, hence grows the probability and possibility of the software flaws resulting into really expensive losses. The use of TDD can bring these flaws to narrow down extremely well.

The TDD flow or process is as given in the below diagram:

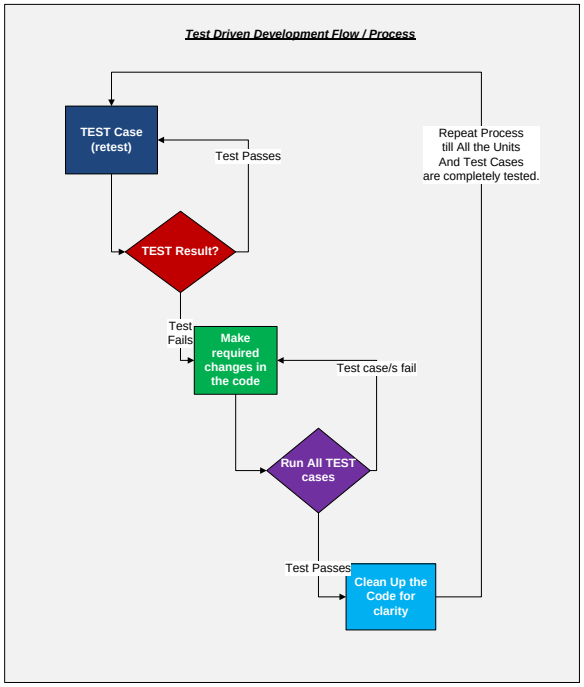

Figure 1: Test Driven Development Flow / Process

The TDD follows the following essential steps:

1. Identifying a part of system module or functionality to be implemented (a single function or method).
2. Simple test program to verify the identified functionality
3. Compile the code under test and terminate the functional code
4. Compile; run the test, the test fails
5. Terminate the non required functional code.
6. Compile; run the test.
7. Refactor the functional code to clean the code and remove duplication
8. Repeat the tasks 6 & 7 until the test passes and the functional code is cleanly implemented.

The tasks above will be repeated until all the modules and functionalities are cleanly and precisely implemented.

2. About the System, Problems Faced during development and solutions

2.1 The System:

The goal is to develop a system using, ARM processor which will act as the Front Panel for the complete system for automated navigation. The panel should be able to communicate with external UART (8 channels), External Flash, Audio Amplifier, Display for Displaying, and also taking inputs through a 5x4 self-illuminated keypad for decision making of the system. The hardware must be supported with the APIs and Device drivers for all the interfaces. The hardware should be made with an ARM processor and the interfaces such that can be used as a front panel for any system.

The main objective of the project is to develop a low power, efficient flexible and low cost hardware which can implement and be used as a User interface for the system and can be effectively used in applications related to vehicle automation and defence sectors.

The system should have the following architecture:

- Processor: The processor needs to be an advanced fast processor which can meet the requirements like Ethernet, sufficient internal memory, inbuilt display driver, etc.
- Ethernet: The system must have an Ethernet chip which can enable it to be controlled over the LAN.
- External UART: It needs to have at-least 8 UART (serial) channels for communication, hence and external UART is a requirement as most processors cannot satisfy this demand.
- Flash Memory: The external ROM is required to store the application code (firmware) if the internal memory is insufficient.

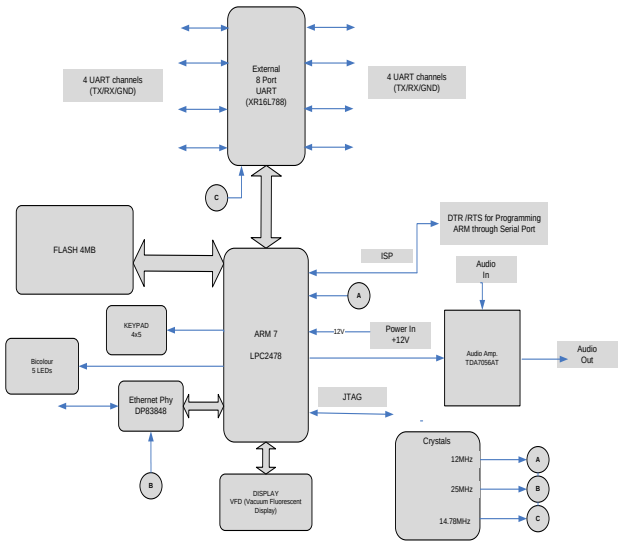

Figure 2: Block Diagram of the Application Specific system

2.2 ARM Processor Overview:-

The ARM processor is chosen as a modular, configurable and versatile hardware platform for the working and interfacing of the various interfaces used for the front panel module. The processor must satisfy the requirement of interfacing with the various interfaces simultaneously.

Apart from the above requirements the selection is also dependent on the aspects like Number of Input Output Pins required, the operating Speed at which the system, the power consumption, the memory requirements and the firmware or the size requirement for the system .

2.3 List of Problems faced and solutions:-

1. Flash: Addressing of the Flash - wrong offset calculated for the access, Write and Read cycles not proper

Solution: Observations of the W/R pin on the Oscilloscope: Modifications required in the code for address access.

2. Ethernet Boot loader: - booting from external flash: JTAG connections were not accessible outside the enclosure. Hence it was not possible to program the external flash through JTAG. Thus, the Ethernet boot loader was required.

Solution: Ethernet Boot Loader was used.

3. External UART and External FLASH: Mapping of external UART was done on CS0 and that of the Flash memory was done on CS1. For an external boot loader to work, to store the executable in the external flash and to execute from external flash, the ARM processor has an option open, only if the Flash is connected to CS0. But in our case the case was opposite.

Solution: Flexibility in the selection of CS0 and CS1 for both flash and UART was provided so that the mapping of the Flash on CS0 can be made as and when required. Changes need to be made in the UART production code.

4. Generation of HEX file for the external flash

Solution:

A. In the "Options for Target 'Target1'" window go to the "Target" tab. Select the default "Read/Only Memory areas" as the off-chip ROM1 and enter the appropriate start address and size.

B. In the "Options for File 'LPC2400.s'" window click on "ASM" tab.

C. In the "Conditional Assembly Control Symbols" enter the define symbols "REMAP, EXTMEM_MODE". This alters the "MEMMAP" register of ARM core for code execution from external flash. (Refer to "Memory Mapping" section in LPC24xx user manual for more details) NXP Ethernet utility did not accept the hex file for external flash. (It is designed to be used for internal flash only)

Solution: Keil Command sequence from "elf --bin temp\Ext_flash_UDP.axf –o temp\firmware.bin" was used to convert the hex file into binary file. The TFTP protocol was used to download this generated bin file into the external flash. The new boot loader which accepts the TFTP protocol was downloaded into the internal flash using the Flash Magic utility.

2.4 Testing with the Hardware:-

Using the various simulation and emulation techniques along with the interfaces and isolation of the various peripherals to be testes allows us to know and develop a test significant system behavior and n in which the hardware would react to the developed production code without access to the real

target hardware. The simulation, emulation and automated tests applied on the hardware add value as they perform tests independent of the hardware target subsystems. This also helps in avoiding the loss in terms of finances and also loss of the target subsystem in case the production code goes too wrong.

Example of a UART code for our system using TDD is as given below: uart.c

```
#include <stdio.h>
#include "LPC24XX.h"
#include "7_CHANNEL_UART.h"
#include "uart_RT.h"

//UART receiving string
s32 Uart_RcvString(Uart
driver,s8* strgToRcv, u32
StringLength)
{
  int i;
  s8 string_char;
 if ( strgToRcv == 0)
    return 0;
 if ( driver == (Uart) 0 )
    return 0;
for (i=0; i<StringLength; i++)
  {
    string_char=
getByteFromHardware(driver);
    stringToRcv[i] =
string_char;
    if (string_char ==
UART_RX_NOT_READY)
      break;
  }
  return i;
}


//Getting a Byte from the
Hardware
s8 getByteFrmHW( Uart driver )
{
//checking the Timeout for the
UART to //respond and calling
the //getByteFrmHWWhenRdy
function
  if(driver-
>timeout==UART_DRIVER_TIMEOUT_I
NFINITE )
    return getByteFrmHWWhenRdy(
driver->hardware );
  else
    return
getByteFrmHWRetImmediately(
driver->hardware );
}

//getting Byte from the ready
hardware
s8 getByteFrmHWWhenRdy( UartHW
hardware )
{
  volatile s8 theStringChar;
   do
  {
  theStringChar =
getByteFrmHWRetImmediately(
hardware );
  } while (theStringChar ==
UART_RX_NOT_READY );
}

//called when the timeout
occurs
s8 getByteFrmHWRetImmediately(
UartHW hardware )
{
  return UartHW_RcvChar(
hardware );
}
```

Test Case for testing the above Code:

```
void test_UartRcvStrg (void)
{
  s8 strToRcv[5];

UartHW_RcvChar_ExpectAndReturn(
mockHardware,UART_RX_NOT_READY)
;

Uart_SetRcvTimeout(mockDriver,
UART _TIMEOUT_NO_WAIT);

// when fails test returns a -1
TEST_ASSERT_EQUAL(0,
Uart_RcvStr(mockDriver,strToRec
eive,)); TEST_ASSERT_EQUAL(-1,
strgToRcv[0]);
}
```

The production code and the test code were written for both external as well as internal UART. The test cases were made and all the 8 channels of the external UART were tested

successfully using them. Hence making changes just in the header file would solve the problem of making the major changes in the code.

Using the TDD in the similar form for all the other modules the problems were effectively handled. The mocks for all the required modules were created so that the real hardware can be isolated from the testing phase. The production codes were then tested on the actual hardware after passing the test cases using TDD on each module.

3. Benefits of Using TDD in Embedded Systems

The results of all the unit tests provide a constant feedback that each component is working.

- The unit tests when well written including the comments present along with the code, act as updated documentation, unlike the separate documentations which need to be frequently updated.
- Verifying production code using the test cases reduces the risk, as the tests are independent of hardware, can be completed even before hardware is ready or when hardware is expensive and scarce.
- Reduce the number of long target compile, link, and upload cycles that are executed by removing bugs on the development system.
- Isolation of hardware/software interaction issues by modeling hardware using simulation and emulation techniques.
- When the test passes and the production code is "refactored" to remove duplication, the code is then cleaned and made easy to maintain. Hence, the developer can move to the development of the next module.
- For creating the production code and the test cases the developer needs to truly understand the requirement and must know what the expected / desired result to be and what should be the suitable test case for the same. The TDD forces the developer to analyze critically and design after understanding the requirements of the module.
- The software is better designed and easily maintainable. It is also loosely coupled and readable hence makes it easier to manage.
- Using TDD increases the efficiency of the developer as the debugging time is reduced to a great extent.
- The developer gains confidence to make decisions about the design and can "refactor" simultaneously knowing that the software would still be working.
- The set of test cases act as a protection layer from the bugs.

If any bug us found in any part of the software the developer must write a test to reveal the bug without disturbing the other part of the code. So that the other tests still remain in the passed state and the developer does not have to go through the entire length of the code to remove the bug. Thus every time the tests are run all the fixed bugs are verified all over again. Hence, a double check takes place each time the tests are run.

Debugging time is reduced to a great extent.

The software becomes more maintainable and robust.

Conclusion

The Test Driven Development plays a very significant role in the field of software development in terms of its value addition to the process of software development. All the benefits stated above prove that the TDD need to be used widely so as to develop better software which are maintainable and robust. It also allows the developer to take a free hand in designing the system software as the developer is more confident about the software and production codes because of the results of the tests that they undergo.

The TDD must be widely used in the field of embedded system development as well so that the process of software development for embedded applications also benefits from it.

Thus, this makes the embedded softwares more reliable, maintainable and robust.

References

[1] John Peatman, "*Microcomputer based Design*", published by Tata McGraw Hill, 2005 edition,pg. 365.
[2] Michael Barr, "*Programming Embedded Systems in C and C++*", Oreilly Publication, edition January 1999.
[3] Byte Craft, "*First Steps with Embedded Systems*", ByteCraft Limited, first edition.
[4] Andrew Sloss, Dominic Symes and Chris Wright "*ARM System Developers Guide – Designing and Optimizing System Software*" , published by ELSEVIER, 2009 edition.
[5] Ted Van Sickle, "*Programming Microcontrollers in C*", LLH Technology Publishing, 2001 edition.
[6] James Grenning, "*Test Driven Development for Embedded C*", The Pragmatic Programmers Publication, 2009 edition.
[7] Michael J. Karlesky, William I. Bereza, and Carl B. Erickson, "*Effective Test Driven Development for Embedded Software*", http://www.atomicobject.com/files/EIT2006EmbeddedTDD.pdf
[8] Wolfgang Schmitt. "*Automated Unit Testing of Embedded ARM Applications*" Information Quarterly, Volume 3, Number 4, p. 29, 2004.
[9] David Astels, "*Test Driven Development: A Practical Guide*", Upper Saddle River, NJ: Prentice Hall PTR, 2003.
[10] James Grenning, "*TDD with Embedded C*" , http://leandog.com/tdd-embedded/ (URL)
[11] Guidelines for Test Driven Development, MSDN library, http://msdn.microsoft.com/en-us/library/aa730844(v=vs.80).aspx
[12] Introduction to Test Driven Development, by Agile org. http://www.agiledata.org/essays/tdd.html (URL)
[13] "Unit Test." http://en.wikipedia.org/wiki/Unit_test
[14] http://www.atomicobject.com/embeddedtesting.page
[15] Many tutorial papers from URL http://www.cs.chalmers.se/~rjmh/tutorials.html