

Dynamic Command Line Based Debugger

A Project Report
Presented to
The Faculty of the Computer Engineering Department
San Jose State University
in Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Engineering

By
Benake Preeti Popat
Borad Rishit
Chandra Bhargava Leepeng
Chaturvedi Nikhil

May 10, 2016

Copyright © 2016

Benake Preeti Popat, Borad Rishit, Chandra Bhargava Leepeng, Chaturvedi Nikhil

ALL RIGHTS RESERVED

APPROVED FOR COMPUTER ENGINEERING DEPARTMENT

Prof. Preetpal Kang, Project Committee Member

Dr. Jerry Gao, Project Committee Member

Dr. Lee Chang, Project Committee Member

Keith Perry, Project Committee Member

Dr. Xiao Su, Chair, Computer Engineering Department

ABSTRACT

Dynamic, command-line based Debugger

by

Benake Preeti Popat, Borad Rishit, Chandra Bhargava Leepeng, Chaturvedi Nikhil

Programming an embedded device can be challenging, especially when debugging, as many products do not include a debugger as costs for one is relatively quite high. A programmer has to make use of logs onto a console for all debugging purposes, which becomes tedious as the program complexity increases. Also, many a times remote debugging has to be done with systems not physically present with the programmer, making debugging/corrections even harder.

For a program written for, say a PC or a mobile phone, an extensive number of tools for code development and debugging is presented to the programmer. But this is not a case for programmers on embedded systems. The most common way of knowing what the code is actually doing is to dump the program information as logs through a serial port between the board and the development environment, besides using emulators, in-circuit debuggers and so on. These add cost to product development and makes debugging tedious.

The motivation behind the idea stems from our experiences programming embedded boards and equally wishing there were cheaper tools to debug the code. The tool would work towards having a system in place where, for example, a function can be tested readily through command-line or an expression can be verified using the existing list of variables in the program or the value of those variables can be verified. Also, as an extension of this idea, we also intend to test the feasibility of debugging code running on boards not physically connected to the development environment.

Acknowledgements

We are deeply indebted to Prof. Preetpal Kang, for his invaluable motivation, guidance and support he has provided in the preparation of this study.

We would also like to thank Prof. Donald Hung, for helping us out with reviews and comments on our presentations.

Table of Contents

| | |
|---|---------------|
| Chapter 1: Introduction | 1 |
| 1.1 Project goals and objectives | 1 |
| 1.2 Problem and motivation | 1 |
| 1.3 Project application and impact | 3 |
| 1.4 Project results and deliverables | 4 |
| 1.5 Market research | 5 |
| 1.6 Project report structure | 6 |
| Chapter 2: Project Background and Related Work | 8 |
| 2.1 Background and used technologies | 8 |
| 2.2 State-of-the-art technologies | 9 |
| 2.3 Literature survey | 10 |
| Chapter 3: System Requirements and Analysis | 11 |
| 3.1 Domain and business requirements | 11 |
| 3.2 Customer-oriented requirements | 13 |
| 3.3 System function requirements | 15 |
| 3.4 System performance and non-function requirements | 17 |
| 3.5 System behavior requirements | 17 |
| 3.6 System context and interface requirements | 21 |
| 3.7 Technology and resource requirements | 23 |
| Chapter 4: System Design | 25 |
| 4.1 System architecture design | 25 |
| 4.2 System interface and connectivity design | 26 |
| 4.3 System component API and logic design | 30 |
| 4.4 System design problems, solutions, and patterns | 32 |
| Chapter 5: System Implementation | 34 |
| 5.1 System implementation summary | 34 |
| 5.2 System implementation issues and resolutions | 43 |
| 5.3 Used technologies and tools | 46 |
| Chapter 6: System Testing and Experiment | 48 |
| 6.1 Testing and experiment scope | 48 |
| 6.2 Testing and experiment approaches | 50 |
| 6.3 Testing report (or case study results) | 52 |

| | |
|--|-----------|
| Chapter 7: Conclusion and Future Work | 54 |
| 7.1 Project summary | 54 |
| 7.2 Future work | 54 |
| References | 55 |

List of Figures

| | |
|---|------------|
| Figure 1. Block Diagram For Makefile..... | Page No.11 |
| Figure 2. Block Diagram For Terminal | Page No.12 |
| Figure 3. Block Diagram for Target Platform | Page No.12 |
| Figure 4. Terminal with Commands | Page No.21 |
| Figure 5. System Block Diagram..... | Page No.25 |
| Figure 6. Make file..... | Page No.27 |
| Figure 7. Parser Block Diagram..... | Page No.28 |
| Figure 8. Appender Block Diagram..... | Page No.28 |
| Figure 9. Memory Map | Page No.28 |
| Figure 10. Symbol Execution | Page No.29 |
| Figure 11. Class Diagram | Page No.31 |
| Figure 12. Primitive Function Handler | Page No.32 |
| Figure 13. Makefile snippet | Page No.35 |
| Figure 14. Primary ELF file sections | Page No.36 |
| Figure 15. Secondary ELF file sections..... | Page No.36 |
| Figure 16. 1st file Parser | Page No.37 |
| Figure 17. Input 1st File..... | Page No.38 |
| Figure 18. Output LUT file from 1st file | Page No.38 |
| Figure 19. map file Parser | Page No.39 |
| Figure 20. Input map file | Page No.39 |
| Figure 21. Output LUT file from map file..... | Page No.40 |
| Figure 22. Symbol Search Logic | Page No.42 |
| Figure 23. Symbol execution code snipped | Page No.43 |
| Figure 24. Makefile wildcards | Page No.44 |
| Figure 25. 1st file before parsing | Page No.45 |
| Figure 26. LUT file after Parsing..... | Page No.45 |
| Figure 27. GUI to select input file | Page No.46 |
| Figure 28. Parser Layer 1 | Page No.48 |

| | |
|-------------------------------------|------------|
| Figure 29. Parser Layer 2..... | Page No.49 |
| Figure 30. Parser Layer 3..... | Page No.49 |
| Figure 31. Parser Layer 4..... | Page No.49 |
| Figure 32. Sample Output Demo | Page No.53 |

List of Tables

| | |
|---|------------|
| Table 1. GPL family Lincense usage statistics | Page No.05 |
| Table 2. GPL license usage according to the Black Duck Software | Page No.05 |
| Table 3. System Function Requirements | Page No.16 |
| Table 4. Setup Requirements and Interfaces..... | Page No.20 |
| Table 5. Terminal Application Setup..... | Page No.21 |
| Table 6. Debugger Application Setup..... | Page No.23 |

Chapter 1. Introduction

1.1 Project goals and objectives

As Embedded Engineers, we understand the importance of the debugger. Debugging is equally important as programming the devices. One of the ways that we as student practice to debug the programs is adding logs to the console. This is tedious and is not a feasible as the complexity of the program increases. Also, the debuggers available in market for the embedded device are pretty expensive and not affordable to the students and small companies.

Our main goal is to develop a cheap and inexpensive debugger, which helps debug functions and variables from the command line. As students at San Jose State University, we did work on some complex projects and realized the need of a debugger that can ease our debugging process. The target platform we choose is SJOne board (i.e. LPC1758), which is used by the students of engineering for subjects like CMPE243 and CMPE244 at San Jose State University. We are designing a debugger that can be used by the students to debug their programs and simplify the debugging session by giving them the ease to debug variables and Functions from command line. We hope to reduce the error logging, which is currently practiced as one of the ways of debugging complex code.

1.2 Problem and motivation

Programming an embedded device can be challenging, especially when debugging, as many products do not include a debugger as costs for one is relatively quite high. A programmer has to make use of logs onto a console for all debugging purposes, which becomes tedious as the program complexity increases. Also, many a times remote debugging has to be done with systems not physically present with the programmer, making debugging/corrections even harder. For a program written for, say a PC or a mobile phone, an extensive number of tools for code development and debugging is

presented to the programmer. But this is not a case for programmers on embedded systems. The most common way of knowing what the code is actually doing is to dump the program information as logs through a serial port between the board and the development environment, besides using emulators, in-circuit debuggers and so on. These add cost to product development and makes debugging tedious.

The motivation behind the idea stems from our experiences programming embedded boards and finding a lack of cheaper tools to debug the code. The tool would work towards having a system in place where, for example, a function can be tested readily through command-line or an expression can be verified using the existing list of variables in the program or the value of those variables can be verified. One thing behind emulators that cut into programmer time is the fact that, while porting the code from one board to another with platforms or porting code from one version of the board to a higher version, the emulators used may change adding two kinds of dependencies. One, the cost of the emulators and the next being the time to setup each emulator with the development environment. Now, one of the most widely used debuggers is the GDB which is mainly used on Linux based embedded systems. There are two main reasons we do not think of GDB as a viable tool is, first, embedded systems have scarce CPU cycles and memory. GDB makes for a heavy tool for embedded systems making it quite difficult to use. Second, we now have many platforms on embedded systems apart from Linux-based ones which make up for a large part in the ecosystem. We would like to provide easier debugging capabilities for a few boards that we think are very well used in SJSU College of Engineering and outside.

One other justification for this idea is to give back to the College of Engineering which has given us many opportunities to learn and develop. We do see students generally indulge in tougher practices of sorting through many log files of considerable sizes to check, for example, if the device connected through I2C has been initialized correctly. This is not to mention the logs that deal with the state machine associated with I2C and the complexity it introduces when multiple devices are interfaced. The Embedded Software course

(CMPE244) has been one of our main motivations, where the class dealt with LPC1768 MCU without a debugger due to high cost involved. As stated earlier, we wish to enable students debug in an easier way and spend more time programming and correcting errors than read log files.

1.3 Project application and impact

This project is implemented to address some of the difficulties that embedded professionals are facing when writing programs for embedded platforms. This project is intended to enhance the learning experience and transform writing programs for embedded real time systems an interactive process. With the project completed successfully and all the options provided, this tool finds applications with professionals and students in academic and commercial space in the following areas.

1. Vendors can supply the debugger tool along with the **Board Support Package**.
2. Students or Enthusiasts can use this to improve the learning programming on cross platforms.
3. Professionals can use this tool to supplement for the expensive hardware debugger tools and reduce the overall cost of the project.

The impact of this project, we believe will be major, particularly in the academic space. Students will be able to get the tool and incorporate it into their projects and start off by writing programs for their intended target platform and test their functionalities using this debugger tool during run time. This effectively eliminates the need for some hardware debugger which may not come that easy for academic applications. By keeping this a simple interface, we avoid any overhead, and the user should be able to use it on minimalistic system configurations. With this being part of a BSP, the vendor can help the end user to achieve faster and robust prototypes which is the primary goal with the evaluation platforms.

1.4 Project results and expected deliverables

The primary objective for the project is to design and implement a software debugger tool that can debug symbols/ functions in programs written for an embedded target platform during run-time. Next to this, is a parser script that can extract the symbol table from multiple input files to provide flexibility. And, the application layer is to provide a user interface via a custom designed serial terminal for the user to input commands from the list of options provided.

The various parts of the systems provide intermediate results for the successive modules. At the initialization, we have the Makefile that invokes and co-ordinates among the different steps until the final image with the required symbol table linked to primary image is generated. This process generated multiple files, however the file we are most concerned about is the executable in hex format.

The parser script, which is another module within the make module, will generate the required symbol table from LST and MAP files when the input path is provided by the user through the GUI interface.

After the target is programmed with the hex++ image (the secondary executable with the symbol table linked), it is time to access the target using the COM ports via a custom designed terminal that support the required user inputs for debugging.

The front-end user interface is implemented in C & C++. The back-end parser script is written in Python. And the lower level make module is using the ELF file configuration from the UNIX system design.

1.5 Market research

GDB Debugger: GDB is open-source free software released under GNU General Public License. Table below shows the statistics from 2009 – 2013 of usage of GPL family license.

| 2009 | 2010 | 2011 | 2012 | 2013 | 2014 |
|------|------|------|------|------|------|
| 72% | 63% | 61% | 59% | 58% | ~54% |

Table 1: GPL family usage statistics

The GPL license family is used by 54% open-source projects in 2013 and according to GitHub repositories GPL license usage is around 25%. Below table shows breakdown of different GPL license usage according to Black Duck Software.

| License | 2008 | 2009 | 2011 | 2013 | 2015 |
|------------|------|------|------|------|------|
| GPL v2 | 59% | 52% | 43% | 33% | 23% |
| GPL v3 | 2% | 4% | 7% | 12% | 9% |
| LGPL 2.1 | 11% | 10% | - | 6% | 5% |
| LGPL 3.0 | 1% | .5% | - | 3% | 2% |
| GPL Family | 72% | 67% | - | 54% | 39% |

Table 2: GPL License Usage according to Black Duck Software

MinGW: MinGW debugger is developed from the Cygwin, to port the Unix software to Windows focusing mainly on performance and simplicity and compromising compatibility. MinGW debugger could not keep up with the code base update promptly and was declared open source. MinGW is also part of GPL license family, statistics above are also presumed valid for MinGW.

Cygwin: Cygwin is also an open source library currently run by RedHat. It is used to port UNIX software to windows focusing mainly on compatibility and compromising

performance. Complete installation takes 36GB of hard-disk space and usable configuration requires around 1-2 GB.

ARM toolchains: is a GNU toolchain with a GCC source branch targeted at embedded ARM processors. Additionally, ARM releases binaries pre-built and tested from ARM embedded branch. It also allows for integration into 3rd party toolchains.

1.6 Project report Structure

This report explains the entire project progress and status in detail. The report is divided in 7 chapters and each chapter explains the project schedules and the design and implementation. The chapters are as follows,

1. Chapter 1 introduces the project in brief and explains the goals and objectives we had planned to achieve and how far did we achieve it. It also covers the motivation behind this idea and its academic contribution. It also includes a market research, which explains the project related companies and their market shares. We also describe the results and the pitfall in achieving the expected deliverables.
2. Chapter 2 covers the related background study we did in order to understand the project better. This study helped us to initially begin the project and guide us in the right path. State of Art that explains related tools and technologies study, which gave us in depth knowledge if we need to use it in our project or no. And finally the literature survey that includes some related technical papers and useful information.
3. Chapter 3 explains all the preliminary and interfacing requirements. This explains our software extensions and costumer interfaces. It also includes deployment plan, module uses cases and testing environment setup.
4. Chapter 4 gives an overview of the system architecture and explains the system block diagram. Each sub module is explained in detailed including block diagram

or related figures to explain the concepts in a better way. It also covers user interface design, API design and problems encountered.

5. Chapter 5 explains the Project Implementation. The technologies used and issues and problems encountered during the implementation of each module. Here every module is illustrating the software design and implementation. Also, includes the screenshots of code snippets and the results/output.
6. Chapter 6 the testing modules, test cases implemented, bugs report and approaches to overcome the errors. Again, this chapter is divided in different sub modules and the testing done for every module and related results are described. It also includes the screenshots and bugs report.
7. Chapter 7 covers the conclusion and future scope of the project.

Chapter 2. Background and Related Work

2.1 Background and used technologies

Makefile: This is a script file containing the Unix commands that will perform certain operations such as compile, link etc., on the respective source files written in C or C++ programming languages. The script uses the binary utilities from the Unix/ Linux system libraries. Makefile is primarily used to automate the build process in most of the projects to, so that it can automatically detect if there are changes to the source that are to be included in further builds. This relieves the user from having to keep track of all the modifications done in the source code making it the development process a much better experience. Makefile is used in the industry to automate the organization wide build processes and perform checks on the builds from time to time.

Symbol table: It is an important data structure created and maintained by compiler. It is used to keep a track about the identifiers (variable and function names, objects, classes etc.) used in the source program. It is a compile-time data structure. It stores the information of occurrence variables, functions, objects, classes etc. It is used to serve the purposes like store all variables, to verify if variables are declared, implement type checking. We are using symbol table in executables file to recover entities name while debugging.

Interpreter: Programmers code in high level programming language, which humans can understand, referred as the source code. The interpreter is a translator that translates and runs the program at the same time. It converts the program written in high level languages into machine language which the computer understands and executes it instruction by instruction at the time the program is being run. The interpreter continues to translate the code until the first error is encountered. Hence debugging is easy.

Parsing: A parser is a program that takes each program statement that a developer has written and divides it into parts (for example, the main command, options, target objects, their attributes, and so forth) that can then be used for developing further actions or for creating the instructions that form an executable program. Parsing is used to generate the look up table.

Search Algorithms: The search algorithms are used to search a needed entity by different available algorithms. We are implementing the search algorithm to search variables and functions from the look up table created by the look up table generator using a python parsing tool. This helps to load the required global variable or function and debug it dynamically.

2.2 State-of-the-art technologies

GDB Debugger: it is a GNU Project Debugger. It allows the user to look inside a program while it executes or crashed. It allows users to set breakpoints enabling the user to debug only the required part of the program. It supports many programming languages. It has an improved support for accessing shared libraries during remote debugging. GDB can run on UNIX and Microsoft Windows variants.

MinGW: Minimalistic GNU for Windows, is a minimalistic development environment for Microsoft windows application. It is an open sourced programming tool set without a 3rd party runtime DLL's. MinGW compilers provide access to the functionality of the Microsoft C runtime and some language specific runtime.

Cywin: is UNIX like environment and command-line interface for Microsoft windows. It is a large collection of GNU and open source tools which provide functionality similar to a Linux distribution on Windows. It is also a dynamin link library (DLL) which provides substantial POSIX API functionality.

ARM toolchains: is a GNU toolchain with a GCC source branch targeted at embedded ARM processors. Additionally, ARM releases binaries pre-built and tested from ARM embedded branch. It also allows for integration into 3rd party toolchains.

2.3 Literature survey

The project references a lot of work done in the area of debuggers, compilers, linkers, programming languages and the association of these with embedded systems. We have tried as much to focus most of our research on the respectable sources. The primary way we began our research was by talking to advisor which proved very useful in establishing the general direction the members of the team had to follow. This idea was backed by studies done via sources from Google Scholars, IEEE sources, ACM along with the materials on the public domain published by the manufacturer of the tools which we intend to use. The following section gives a list of the publications that were mainly referenced, besides the other materials not mentioned here.

Chapter 3. System Requirements and Analysis

3.1 Domain and business Requirements

With the evolution of Internet of Things, embedded system is exposed widely among programmer's community. There are more programmers now using embedded systems than ever before. With this project not only we are extending helping hand to those who are embedded programmer but to those who are getting out of their usual way and dive into embedded programming. With the evolution of IOT, scope of our project becomes wider as we are giving capabilities to programmers to debug programs without using any hardware, which helps keep the hardware cost of the project low.

On the other hand, for the programmer's community that deals with low level programming, this project empowers them to debug their programs without using any external debugger, just a light weight chunk of software that deals with board's memory system would help debug the original software.

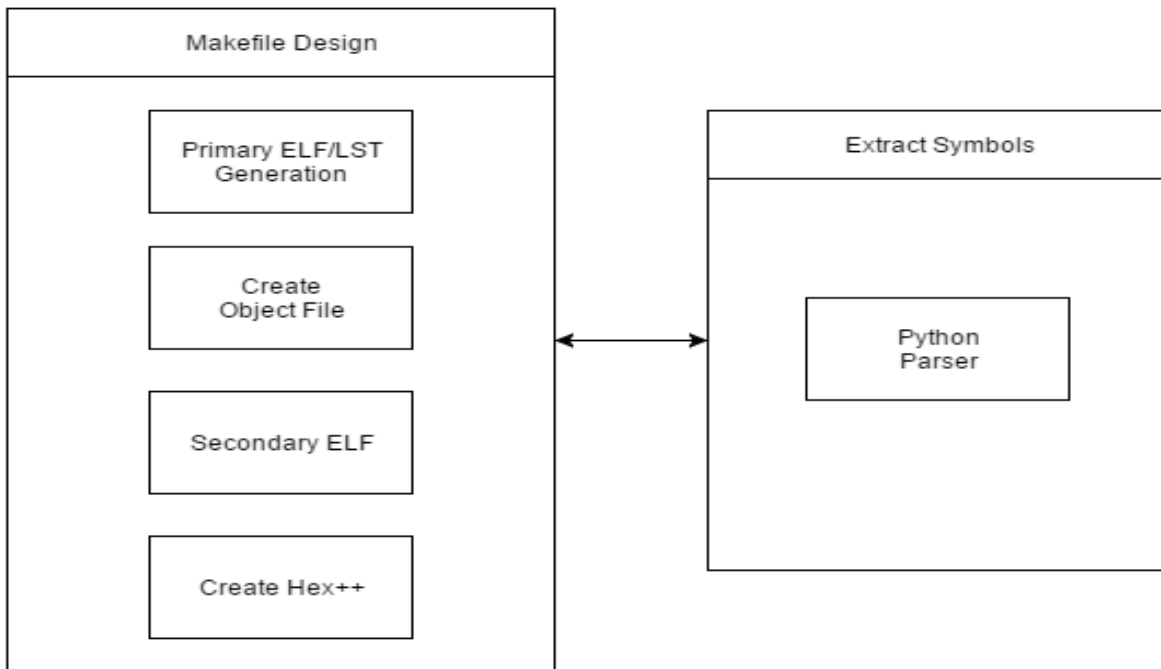


Figure 1: Block Diagram for Makefile

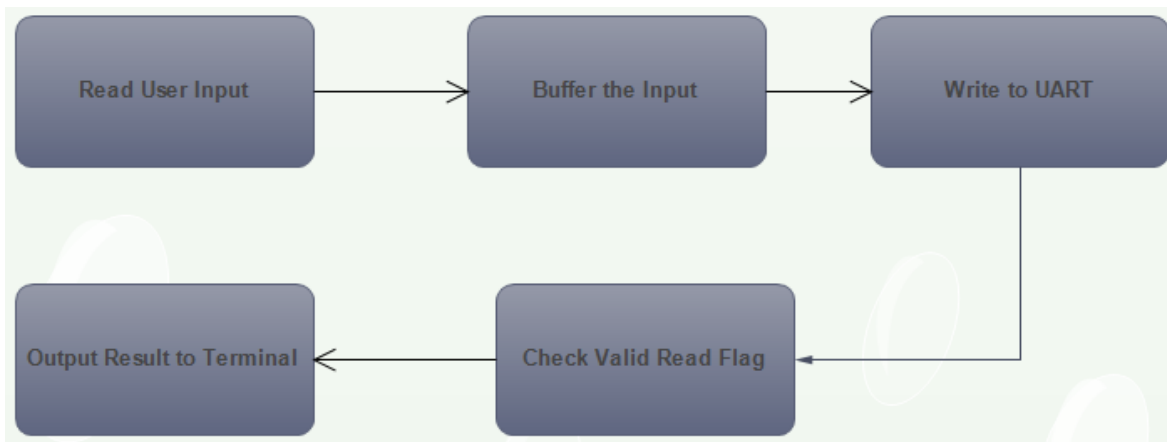


Figure 2: Block Diagram for Terminal

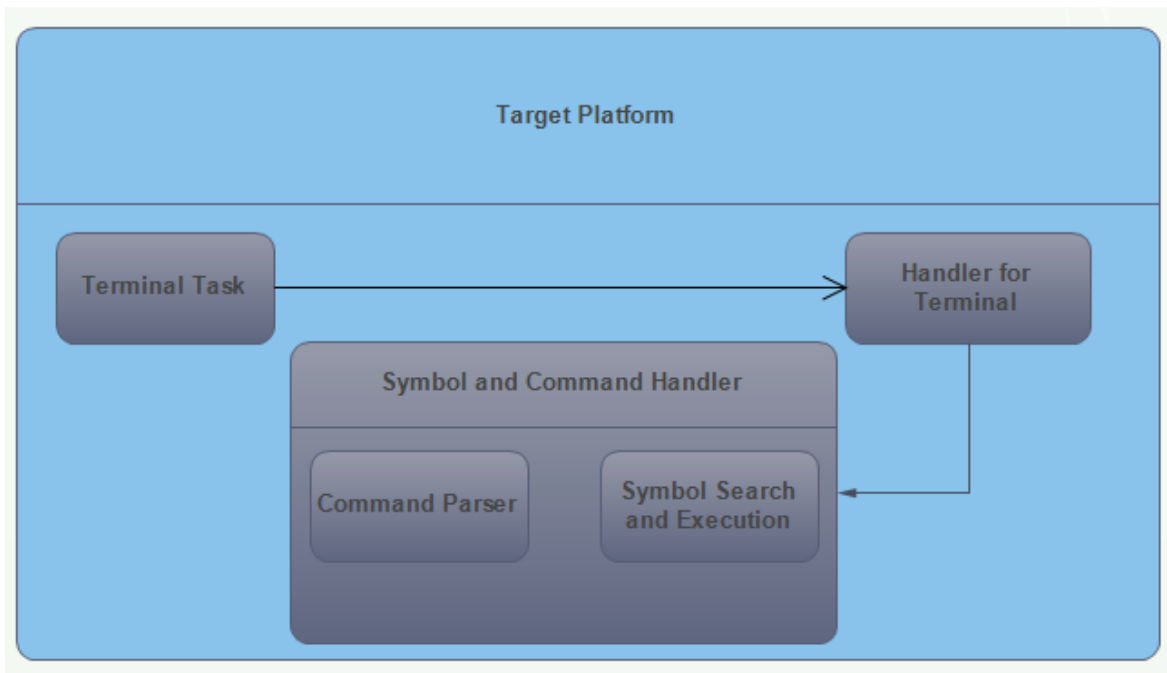


Figure 3: Block for Target Platform

3.2 Customer-oriented requirements

Dynamic command-line debugger, as the name suggests, can be ported to any ARM Cortex-M based embedded development platform for the current release with planned support to other platforms in the upcoming releases. The debugger uses GCC tool-chains which are used by most embedded platform. We are also implementing platform specific post-build methods that vary based on the hardware platforms. Most important of all, this project eliminates need of external hardware.

Based on the project description in the paragraph above, customers are not required to change their original development pattern. However, to add the debugging capability to their project customers are required to include a light weight software extension. The software extension includes following modules:

1. **Post-build Makefile:** Post-build makefile is second makefile that connects customer's project with the debugger. This file defines the rule for compiler tool-chains to include symbol table to hex file generated during pre-build. This file integrates all the debugger modules and connect them to the user's project.
2. **Python parser:** Python parser is an executable that extracts symbol table from the .lst or .map file, removes the unwanted spaces from the symbol table and generates a formatted symbol table to include with the executable.
3. **Loader file:** Loader file defines the start and end address of the symbol table in hex file. This file needs to modify based on the hardware platform.
4. **Terminal Handler:** This file contains source code for terminal that reads function name from the user executes the function if it finds it in the symbol table.
5. **UART terminal:** Adds IO capabilities to the user program.

Functionality and implementation pulls the boundaries of the targeted audience of this project to a broader level. All the ARM users, irrespective of operating system, can use Dynamic command-line debugger with little or some user intervention at this level. Future scope of the debugger is to minimize the user intervention.

Use cases

1. **Read global variable values:** It is handy for a developer to know values of the variables used during the program execution. Arbitrary values of even a single variable might cause serious trouble in overall execution. User can enter the name of global variable that needed to be monitored through command-line and debugger returns the value.

In another use case, user can actually monitor the variable if it the variable is being set to a specific value after executing an instruction. Especially for embedded system programs, it is necessary for a programmer to determine if the register values are changing as expected while debugging software. Monitoring a variable is the primary task of any debugger available in the market.

2. **Assign values to global variables:** This tool also provides the user with the option to modify the value of a global variable during run-time without the need to modify the source code and reprogram. This effective saves a lot of time for the user and makes the development process more convenient and fun. The variables will retain the new values until the platform is reset or re-programmed. Thus user has to ability to tap into the global variables and test the functionality for certain inputs he/she intends to test.
3. **Execute void functions:** This feature helps user to know execution flow of the program. User can enter void function name and determine the flow of the execution is correct or not. This feature is useful for block testing of the software.

That way user does not need to worry about the internals of the function and abstract this part of the software during debugging. This feature, along with other features explained above and below, can collaboratively emulate the debuggers currently available in the market.

4. **Execute functions with up to four parameters and return type:** This feature, along with the feature described above, gives flexibility to the programmers to access almost any function of the program. Use case of this is same as the use case of executing a void function in debug mode. It is handy for a programmer to block test the whole program and determine if the execution flow is correct.

3.3 System (or component) function requirements

This system is developed as a package of various tools integrated to work together. The user should use this package as a whole with suggested modifications for the tool to be able to communicate with his development platform and access the system resources. This tool is an add-on to the user's development platform rather than a standalone application.

1. At the very start, the user should have a computer running windows operating system and capable to load eclipse IDE.
2. The user should use one of the supported target platforms, as this tool supports multiple platforms but is not a universal tool.

Note: The number of supported platforms will increase with every new release.

3. User must download the software package in order to make use of the debugger tool provided.
4. Then he will need to perform the modifications (say directory/path) as directed. This will link the tool with his development setup on eclipse. Or he can use one of the BSP that is supplied with the debugger tool.

5. After the setup has been completed, the user has to compile and build the source code with him specific settings, that should generate at least one of the files (LST/MAP/ELF) for the parser to be able to work.
6. The secondary build is invoked upon successful completion of the primary build. The first step in the secondary build is to provide the parser with the path to the input file.
7. Once the file type is determined, the parser will extract the symbol table and pass it onto the next steps automatically. And the final flash image is generated.
8. Once the HEX++ is generated, it has to be programmed to the target platform and reset the board.
9. The user has to initialize the custom serial terminal from the software package and establish a connection to the target platform over the COM port.
10. After a successful connection is established, the user can input a symbol (global variable or function) available is the list of functions in the source code.
11. The command line interface parses the user input and if the function is found in the symbol table, then it will be executed and the result will be displayed back on the terminal.

| Requirement Type | Requirement Category | Description |
|------------------|----------------------|----------------------------------|
| FR-1 | Setup | Download the Tool/ BSP |
| FR-2 | Setup | Configure the Dev. Platform |
| FR-3 | Build | Source code compile and build |
| FR-4 | Build | Provide the input file to parser |
| FR-5 | Build | HEX++ generated successfully |
| FR-6 | Build | Flash the target platform |
| FR-7 | Debug | Setup serial terminal connection |
| FR-8 | Debug | Input the command and symbol |
| FR-9 | Debug | Verify the result |

Table 3: System Function Requirements

3.4 System performance and non-function requirements

The system performance tests are performed during the project implementation to verify the usual average execution time and performance overhead.

NF-1: Performance

The speed is not the critical component for this system. However, to work in real-time, the address fetching and code execution should occur at designated speeds to obtain better efficiency of the entire system.

NF-2: System Requirements

Since the tool is working on the embedded real time systems, hardware resources are critical and limited. So the symbol table should be limited on memory by getting rid of the redundant special characters.

NF-3: Reliability

Obtaining correct data that the user can rely upon is the primary and critical moto for this system as the user will rely on tool to verify the functionalities and based on these results the user may deploy/ release the application.

3.5 System behavior requirements

The system is divided into two sub-systems, the client and the target platform. The client is the computer that runs the terminal which in turn connects to the target platform through USB to serial connection.

The terminal tool is written in C++ on Windows. It supports various commands like, “open”, “help”, “exit”, “clean” and “debug”. The terminal writes to and reads from a designated COM port, which is handled by the terminal handler on the target platform.

The target has multiple layers of handlers that interact with the terminal. The first layer consists of the command handler that receives that bytes of data from the terminal through UART0. Second layers parse the user command and proceeds to symbol execution. The system requirements can be summarized in the table below,

| Requirement Type | Requirement Category | Description |
|------------------|----------------------|--|
| UI-1 | | Generate a look up table for symbols that define the functions and global variables available in the source code. |
| UI-2 | | Debugger Terminal, this tool allows the user to enter any of the function's or global variable's names in order to test and debug their respective values. |
| HI-1 | | This is connected to the user's PC by some common communication medium like USB cable connection. |
| SI-1 | | The Look-up Table Generator tool and the Debugger Terminal tool |
| SI-2 | | Debugger terminal connected to target and ready to accept commands |
| CI-1 | | The debugger terminal communicates with the board to access the symbol |

| | | |
|------|--|---|
| | | table stored in the on-board flash. |
| CI-2 | | The debugger terminal communicates with the board to run the function. |
| CI-3 | | The debugger terminal communicates with the board to retrieve the values of the executed function. |
| FR-1 | | The user has to download this debugger software package to make use of the debugger tool provided as part of the package. |
| FR-2 | | The user has to compile his source code using a compiler of his choice to generate the machine executables. |
| FR-3 | | The user has to open the generator tool and select the specific target platform from the list supported platforms. |
| FR-4 | | Provide input file to generator tool |
| FR-5 | | Program the target board |
| FR-6 | | Deploy the debugger terminal |

| | | |
|-------|--|--|
| FR-7 | | The user has to enter the name of the function which is a valid function that is available in the source code |
| FR-8 | | The user has to enter the name of the global variable for which he wants to find the value in the code |
| FR-9 | | The tool then executes the function from the symbol table by fetching its location in flash memory and move that to RAM to execute on target board |
| FR-10 | | The tool retrieves the result of the function or a global variable from the target board and display the result on the terminal. |

Table 4: System Requirements and Interface

Legend for the table above,

| Color | Requirement Type |
|-------|-------------------------|
| | User Interface |
| | Hardware Interface |
| | Software Interface |
| | Functional Requirement |
| | Communication Interface |

3.6 System context and interface requirements

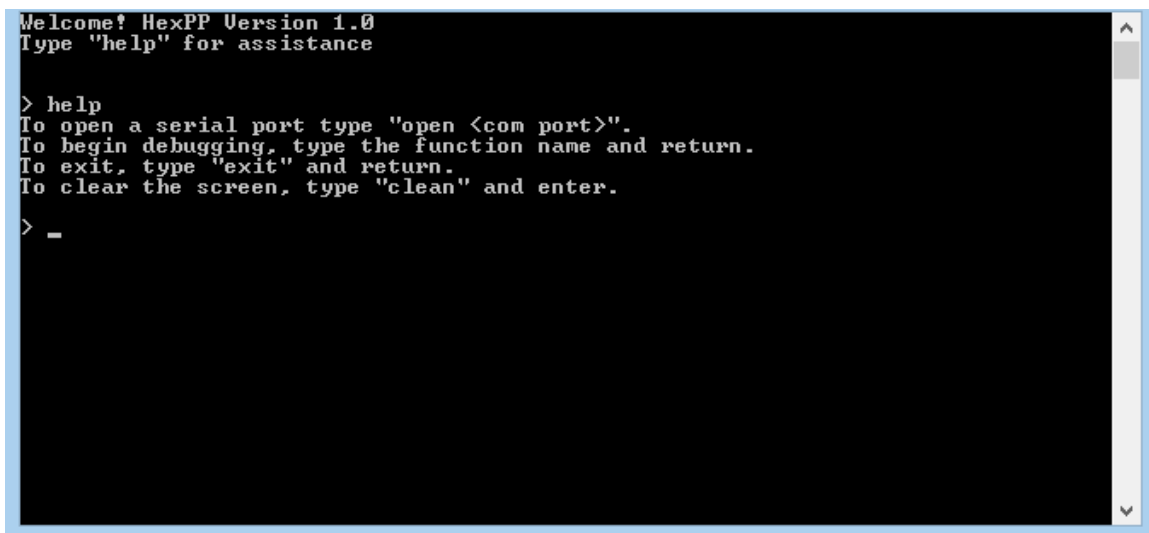
The debugger tool is split in two; one, the terminal, used to send the symbol to be debugged and two, the target platform, which has the code that the user would debug.

The terminal is built on Windows, with future plans to implement on Linux as well. The table below shows the environment setup for the terminal application,

| | |
|------------------------|---|
| Programming Language | C++ |
| IDE Used | Visual Studio, Code Blocks |
| Platform | Windows |
| Commands Supported | Debug, help, exit, clean, open |
| Baud rate of operation | 38400 (in sync with the serial write from the target) |

Table 5: Terminal Application Setup

The figure below illustrates the terminal, with a command typed in and result displayed.



```
Welcome! HexPP Version 1.0
Type "help" for assistance

> help
To open a serial port type "open <com port>".
To begin debugging, type the function name and return.
To exit, type "exit" and return.
To clear the screen, type "clean" and enter.
> _
```

Figure 4: Terminal with Commands

An interface is set up with the target by using the “open” command which takes the name

of the COM port as a parameter in the form of a string and tries to make a connection. The user would have to know the COM port to which the target is connected to and use it with this command. After the connection is established, the symbols to be executed are used with the “debug” command, like “exec foo()”.

To ensure that the connection happens successfully, the user must make sure to disconnect from any other connection using this particular COM port.

The terminal was tested for a number of use cases, most of which involved the read and write operations. The code was tested for different read and write buffer sizes, baud rates and byte size, on machines running different Windows versions.

The design of the terminal has been kept simple to maintain ease of use and clarity in reading the output from the target board. After a command has been handled by the terminal, it waits for the user to enter another command until the tool is closed. An option for help has been included to guide the user in using the tool, which would prove necessary for users who use the debugger for the first time. This facilitates fast acclimation to the tool.

The debugger code that is written to the target board has been coded in C++ for ARM Cortex-M3 platform running FreeRTOS. This being the initial version, further plans include extending the code to include many other platforms as well. The table below explains the setup of the debugger application,

| | |
|----------------------|----------------------------|
| Programming Language | C++ |
| IDE | Eclipse Luna, Eclipse Mars |
| Toolchain | Cross ARM GCC |
| Builder | GNU Make |
| Platform | ARM Cortex-M3 |
| Device | LPC1758 |

| | |
|------------------|----------|
| Operating System | FreeRTOS |
|------------------|----------|

Table 6: Debugger Application Setup

The application code is included in the hex file which is written to the target. This includes the debugger code and the symbol table along with the user application code and other low-level code.

To achieve an interface between the terminal and the target, USB to serial communication is used. UART0 on the target is dedicated for all reading and writing operations to and from the terminal. This is handled by the terminal handler and the command handler which parse the command and invoke the search and execute function.

3.7 Technology and resource requirements

The project mainly focuses on ARM based microcontrollers. It is also the uniqueness of the project that technology to develop the project revolves around the technology used by targeted audience. Hence, users do not need to out of the way to integrate the debugger with their system. Technologies used in the project are as follow:

- 1) **Python:** Project uses python parser to extract the symbol table from the lst or map file.
- 2) **C:** Terminal is programmed using UART which uses C as a low-level programming. Symbol table is traversed to look for the user requested function, switch the program counter value to the address of the found function and execute the function block.
- 3) **GCC compiler tool chains:** Compiler tool-chains generate executable output file along with MAP and LST files. Information in these files are used to integrate the debugger with the user project.

- 4) **Eclipse:** An IDE that uses compiler tool-chains to develop an embedded project in C. Eclipse generates ELF file after compilation that is used by python parser to extract the symbol table. Loader instructions are set in eclipse that embeds the symbol table to hex file during post-build.
- 5) **ARM based development board:** Development board is the target platform on which the user program is executed.

Chapter 4. System Design

4.1 System architecture design

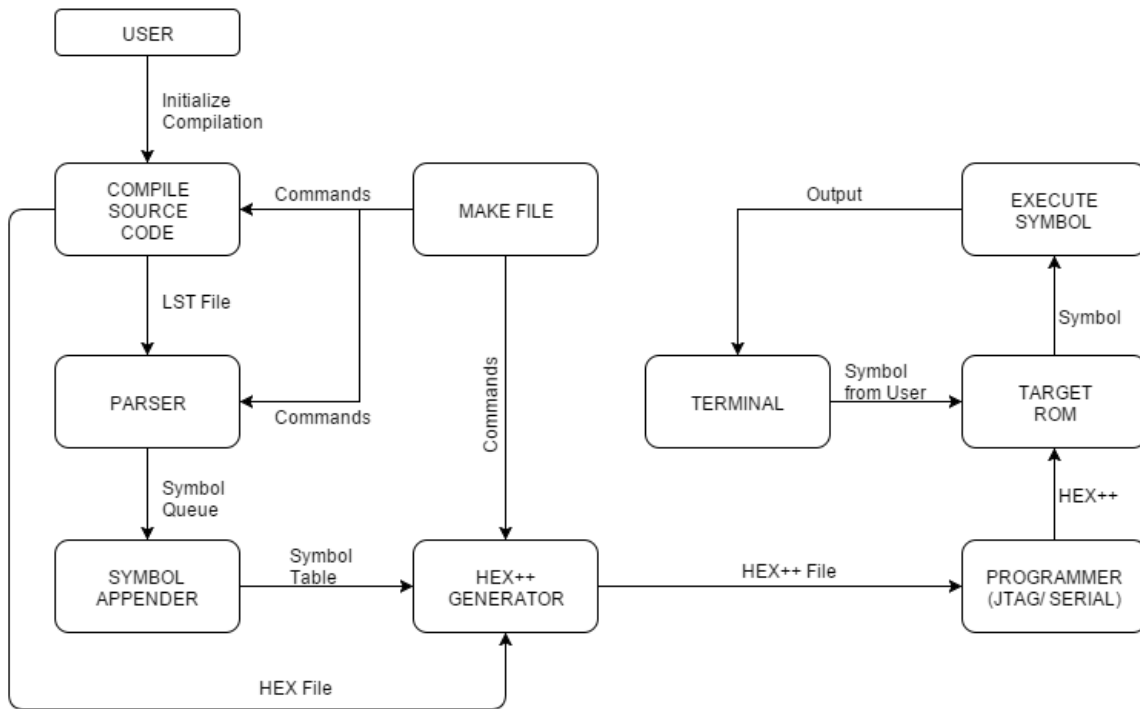


Figure 5: System Block Diagram

1. Symbol table plays a very crucial when it comes to debugging a code. A Symbol Table which is like a Look-Up table for various text, data and bss section in a source code. If it is an embedded platform then the symbol table will be the prime source for getting the addresses of various global variables, functions and any other information like type of a particular entity.
2. Often programmers depend on some compiler generated look-up tables, to understand various issues in a software. But, it can only be a probability of retaining that file even after the executable is generated. This is not a reliable way for a programmer to expect that the look-up table is available.

3. In this project, our aim is to generate this Symbol table from certain files generated by the compiler during the compilation process. Files like '.map' or '.lst' can be appropriate to use for creating a reliable symbol table file.
4. After the source code is compiled by the software developer, the secondary Makefile takes the primary ELF file as input to generate the '.lst' file which is taken as an input by a Parser script developed in 'Python' language. This will parse and extract all the appropriate symbols; variables, functions etc.
5. Once a symbol table is extracted, the makefile appends it to the object files initially generated during primary build process and a new ELF file is generated and in turn this ELF file is used to generate the final '.hex++' image.
6. This newly generated hybrid file that contains the hex image and symbol table, will be flashed by a programmer to the ROM of the target platform connected via JTAG/Serial connection.
7. The flashing is performed in such a way that the symbol table can be accessed in a specific process, so that when the user wants to execute a function, we can obtain the memory address of that particular symbol, which can be a variable or a function.
8. The user will input the symbol that is of interest to him via a terminal, which is then picked up by the debugger tool, execute it on RAM and project the output back on to the terminal for the user to verify his expected output and actual output.

4.2 System interface and connectivity design

Internal Connectivity:

- 1. Makefile:** Make file initializes the source code compiler to compile the code to generate the necessary files required for parsing operation. This in-turn also contains necessary

commands to direct the parser to take in the compiled files and generate the symbol table. Then it finalizes this operation by taking in the hex file and symbol table to generate the required executable.

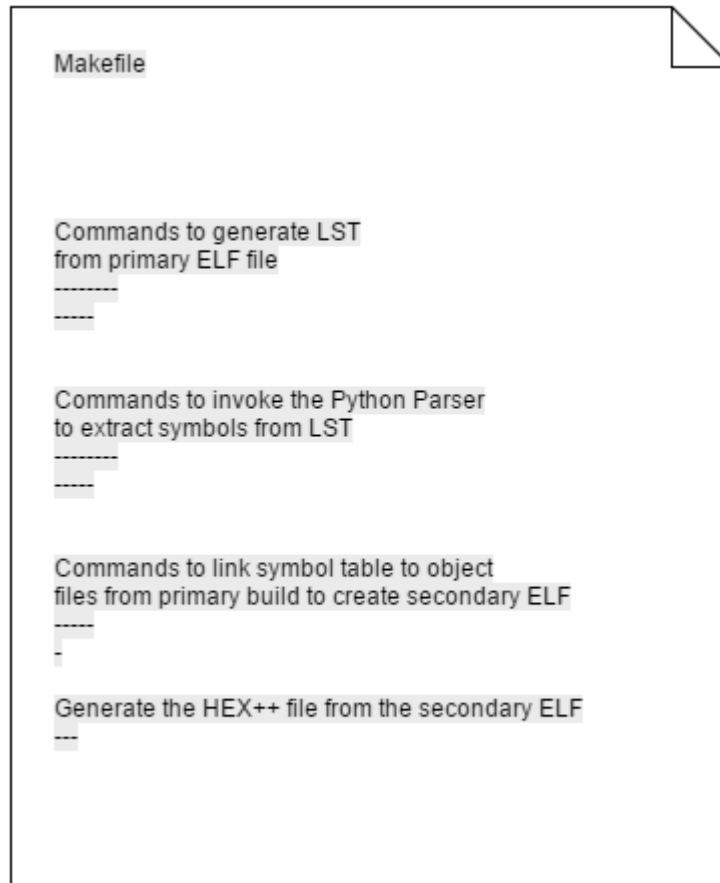


Figure 6: Make file

2. Parser: This block of the project creates a symbol table. The lst or map file is given as the input file to the parsing algorithm, which has two parts, lexical and semantic. The parser reads the input file and sorts the “.data”, “.text” and “.bss” section and writes it to the LUT file, which acts as the symbol lookup table. The LUT file consist of the address of the variable or the functions, the code segment it belongs to, the variable or function name and the identifier flags (‘a’ for variables, ‘v’ for void functions, ‘i’ for functions with arguments).

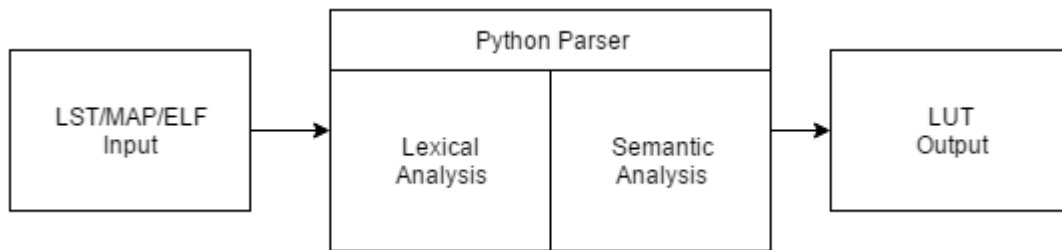


Figure 7: Parser block diagram

3. Appender: The appender in this project is used to link the symbol table against the original executable which is essentially a part of the makefile design. This performs the linking of the LUT with the rest of the files generated during the primary build step. This is also responsible for generating the hex++ images to program the target platform.

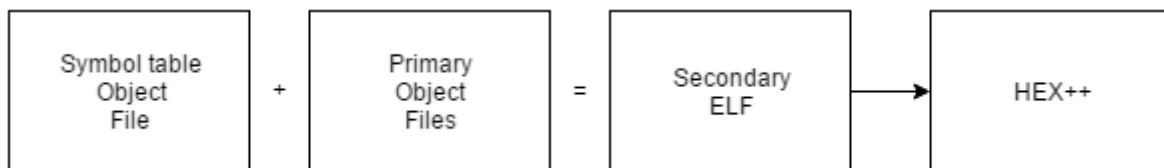


Figure 8: Appender block diagram

4. HEX++: Hex file along with symbol table is embed to the ROM (HEX++). Writing HEX++ on ROM gives users flexibility to access the symbol table. Having access to the symbol table enables functionality to find the address of functions declared in source code. Functions can be executed independent to the source code, with limitations, after the address is determined from the symbol table.

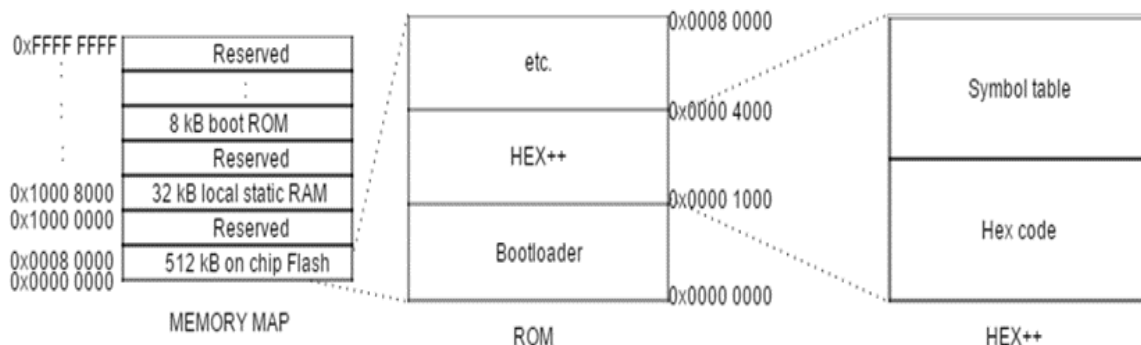


Figure 9: Memory Map

External/User Connectivity:

5. Symbol Execution: The terminal acts as the main point of entry into the debugger. A choice between UART1, UART2, UART3 can be made for establishing a serial communication with the development board through USB to Serial protocol. The function that has to be debugged is entered via the terminal. The HEX++ file which contains the lookup table (LUT file), is accessed for the address of the said function. The function is executed for its output and sent via USB to Serial communication back to the terminal for display.

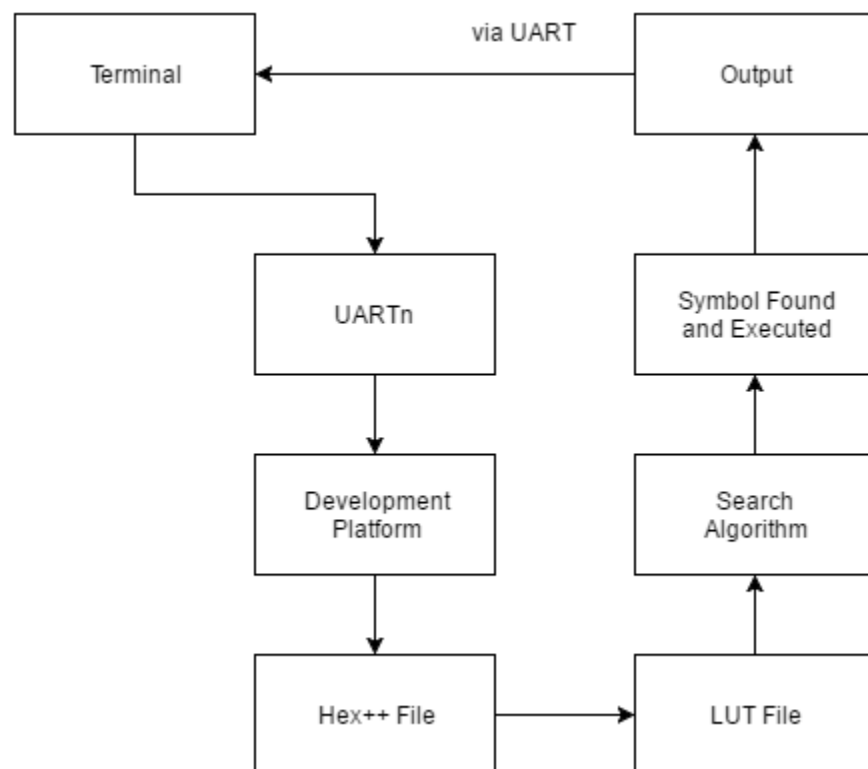


Figure 10: Symbol Execution

4.3 System component API and logic design

The software for the debugger is written in C++. It is divided into mainly two parts; one, the function handlers and the other, search and execute.

The terminal task which accepts the terminal commands, holds a list of commands on the target which the user would invoke. This command list is used by the command handler which invokes the search and execute algorithm.

The lookup table written with the hex file is read when a debug request is made from the terminal. The format of the lookup table is as follows,

- The type of the symbol, a variable or a function
- The address of the symbol
- The symbol name

When the string is entered in the terminal, the command is parsed to retrieve the parameters (in case of a function) or the variable name. This is then passed onto the command handler, which makes a search for the symbol name. A success here means that the symbol to be debugged exists and further the address of this symbol is retrieved and passed to the execution handler.

The execution handler consists of various generic handler classes which handle cases with respect to the return type and the parameters a function accepts. Currently, the types of handlers are,

- Void return with void function parameters
- Void return with Integer parameters
- Integer return with Integer parameters
- Any other complex data type parameters

These handlers are invoked with their respective public functions belonging to the respective class. These take in the pointer of the type of class whose function has to be invokes. That is, the aforementioned classes are abstract classes with implementations that handle the symbol execution. The fourth type of handler mentioned above handles any

type of complex data type apart from the primitive types. This is achieved by using a buffer which is used to hold a dynamic number of parameters. This buffer holds the serialized variables that are packed with a padding between each other. The buffer is wrapped using a public wrapper function and unwrapped by the execution handler using a public unwrapping function. The buffer has been designed with a custom protocol. The first 8 bits hold the type of data and the number of bytes of data to be read. The next bytes contain the actual data and the last 8 bits contain the padding. This sequence continues for the number of variables the user would debug.

The class diagrams of the handlers are illustrated as follows,

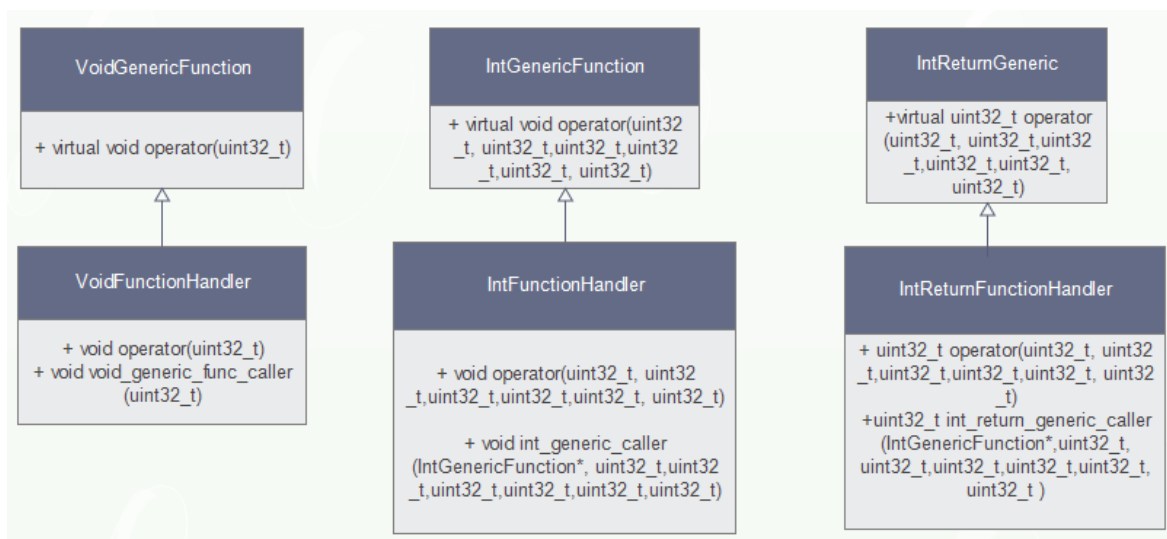


Figure 11: Class Diagram

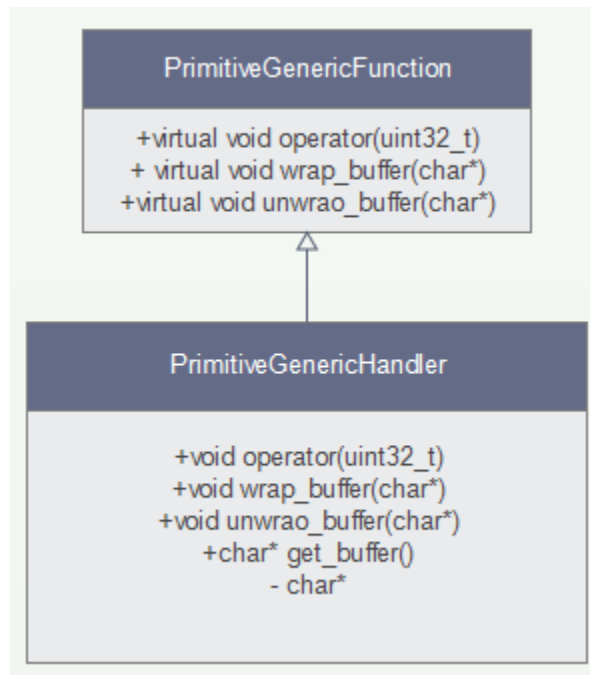


Figure 12: Primitive Function Handler

4.4 Design problems, solutions, and patterns

- When a script is being invoked from a makefile, the makefile was not able to execute the script since there was no proper reference to the executable that the script need in order to run as a standalone application. To resolve this issues, we implemented a shebang in the script, which can help the makefile search and located the executable by accessing the PATH in the system environment variables. Later, this decision came to advantage in making the software package compact and setup is much easier for the user.
- The functions with void parameter and void return and other primitive types was handled with little trouble. But the challenge was in debugging functions that accepted complex data types as parameters. This problem was solved by serializing the parameters into a buffer with our custom protocol.
- To design the symbol execution in a way that adheres to good design guidelines, we divided the various types of symbols (only functions) based on their return type and parameters into classes.

- The classes mentioned above implemented functors which accept the address of the function to be debugged along with the parameters. This proved helpful by allowing us to have abstract classes which would be implemented by the handler classes.
- There are many platforms which support C++. It becomes difficult to debug C++ based functions as we need a handle to the object over which the functions are called or any other members are accessed. This problem has been included in the list of work that this project needs in the future. As such, the parser was updated to exclude the C++ functions in the lookup table.

Chapter 5. System Implementation

5.1 System implementation summary

This section describes the implementation of each sub module of the project. It includes the code snippets, algorithms or pseudocodes, test cases and the input and output files.

The Project is divided is four parts,

1. Makefile,
2. Parser,
3. Hex++ generator,
4. Symbol Search
5. Symbol Execution

1. Makefile

The makefile is a script file, which initializes the process. There are to responsibilities to the makefile. The primary one is to generate multiple files that are necessary for the debugger while the secondary responsibility is to co-ordinate between the build steps until the final image 'hex++' file is created for the programmer to take off. In order to perform the build, the makefile uses the Unix/Linux commands to invoke several binary utilities defines in system libraries. This script goes through each source file during the build process to generate respective object files. The build is divided into 2 steps. The first one being the primary build is done by the auto-generated makefile to generate all the object files from the source files and the second is the post-build that takes the ELF/LST file from the primary build and invoke the Python based parser to extract the symbol table as explained in the next section. After getting back the symbol table file, which is a normal test file, the makefile generates the object file for this symbol table and links it against all other object files generated during the primary build and created the secondary ELF that we use to generate the HEX++ images. This final image has the symbol table embedded in it so that we can access the symbols during the run-time.

```

99 # Macros for each step in the debug_build process
100 SYMBOL_TABLE := $(DEBUG_TARGET).sym
101 SYMBOL_OBJ += $(DEBUG_TARGET).o
102 ELFAGAIN += $(DEBUG_TARGET).elfagain
103 HEXPLUSPLUS += $(HEX_EXE).hex
104
105 # Make all command
106 all: $(HEX_EXE).hex
107
108 # Generating lpc1758_freertos.lst
109 # Note: We need to generate lst here because, all the seco
110      on the post-build make. So we need it here to input
111
112 $(BUILD_TARGET).lst: $(BUILD_TARGET).elf
113     @echo 'Debugger: Generating listing file'
114     arm-none-eabi-objdump --source --all-headers --demangle
115     @echo 'Finished building: $@'
116     @echo ' '
117
118 # Rules to generate the symbol table and embed it and gene
119 # Generating lpc1758_freertos.sym
120 $(DEBUG_TARGET).sym: $(BUILD_TARGET).lst
121     @echo 'Debugger: Extracting symbol table from LST/MAP'
122     $(PYTDIR) parse_symbol.py $@
123     @rm map.txt
124     @rm LUT.txt
125     @echo 'Finished building: $@'
126     @echo ' '
127
128 # Generating lpc1758_freertos.o
129 $(DEBUG_TARGET).o : $(DEBUG_TARGET).sym

```

Figure 13: Makefile snippet

```

ELF Header

  Class:           ELF32
  Encoding:        Little endian
  ELFVersion:      Current
  Type:            Executable file
  Machine:         ARM
  Version:         Current
  Entry:           0x10193
  Flags:           0x5000202

Section Headers:
[  Nr ] Type           Addr           Size          ES Flg Lk  Inf Al  Name
[  0 ] NULL            00000000      00000000      00 00 00 00 00
[  1 ] PROGBITS        00010000      00040c98      00 AX 00 00 08 .text
[  2 ] PROGBITS        2007c000      000002b0      00 WA 00 00 08 .data
[  3 ] NOBITS          2007c2b0      00001cbc      00 WA 00 00 08 .bss
[  4 ] PROGBITS        00000000      00000070      01 00 00 01 .comment
[  5 ] ? (0x70000003)  00000000      00000031      00 00 00 01 .ARM.attributes
[  6 ] PROGBITS        00050c98      00000018      00 A 00 00 04 .ARM.extab
[  7 ] PROGBITS        00050cb0      00000028      00 A 00 00 04 .eh_frame
[  8 ] ? (0x70000001)  00050cd8      000000b8      00 A 01 00 04 .ARM.exidx
[  9 ] PROGBITS        00000000      0000fb74      00 00 00 04 .debug_frame
[ 10 ] STRTAB          00000000      00000072      00 00 00 01 .shstrtab
[ 11 ] SYMTAB          00000000      00027c20      10 0c 17d4 04 .symtab
[ 12 ] STRTAB          00000000      00028b8e      00 00 00 01 .strtab
Key to Flags: W (write), A (alloc), X (execute)

```

Figure 14: Primary ELF file sections

```

ELF Header

  Class:           ELF32
  Encoding:        Little endian
  ELFVersion:      Current
  Type:            Executable file
  Machine:         ARM
  Version:         Current
  Entry:           0x10193
  Flags:           0x5000202

Section Headers:
[  Nr ] Type           Addr           Size          ES Flg Lk  Inf Al  Name
[  0 ] NULL            00000000      00000000      00 00 00 00 00
[  1 ] PROGBITS        00010000      00040c98      00 AX 00 00 08 .text
[  2 ] PROGBITS        2007c000      000002b0      00 WA 00 00 08 .data
[  3 ] NOBITS          2007c2b0      00001cbc      00 WA 00 00 08 .bss
[  4 ] PROGBITS        00000000      00000070      01 00 00 01 .comment
[  5 ] ? (0x70000003)  00000000      00000031      00 00 00 01 .ARM.attributes
[  6 ] PROGBITS        00050c98      00000018      00 A 00 00 04 .ARM.extab
[  7 ] ? (0x70000001)  00050cb0      000000b8      00 A 01 00 04 .ARM.exidx
[  8 ] PROGBITS        00050d68      00002443      00 A 00 00 04 .symbol_table
[  9 ] PROGBITS        000531ac      00000028      00 A 00 00 04 .eh_frame
[ 10 ] PROGBITS        00000000      0000fb74      00 00 00 04 .debug_frame
[ 11 ] STRTAB          00000000      00000080      00 00 00 01 .shstrtab
[ 12 ] SYMTAB          00000000      00027c70      10 0d 17d6 04 .symtab
[ 13 ] STRTAB          00000000      00028bc2      00 00 00 01 .strtab
Key to Flags: W (write), A (alloc), X (execute)

```

Figure 15: Secondary ELF file sections

2. Parser

The Parser was built in Python. The task of the parser was to parse the input files and generate the LUT [Look Up Table] with the required data. We used the “TKinter” the standard GUI library for Python. The GUI gives the user an option to choose between the lst or map file as input. The “.data”, “.text” and “.bss” section from the input file are read. The Parser is split in two different sections depending on the input file selected by the user. Below the figures represent the input lst file code snippet for the parser and the generated LUT file.

```
#!/usr/bin/python

# GUI to open a file required
from Tkinter import *
from tkFileDialog import askopenfilename
# we don't want a full GUI, so keep the root window from appearing
Tk().withdraw()
# show an "Open" dialog box and return the path to the selected file
filename = askopenfilename()

# Open a file
fo = open(filename, "r+")
myarray = fo.read().split('\n')
# print "Name of the file: ", fo.name

del myarray[0]
fu = open("map.txt", "w+")

for line in myarray:
    if (".data" in line and "::" not in line) or (".bss" in line and "::" not in line) or (".text" in line and "(" in line and "::" not in line):
        # delete the F.L.O
        fu.write(line)
        fu.write('\n')
        # break

fu.write('\0')

fo.close()
fu.close()

# delete the address offset column as we do not need it
f = open("LUT.txt", "r")
g = open("map.txt", "w")

for line in f:
    if line.strip():
        g.write(" ".join(line.split()[0:2])+" "+ " ")
    if line.strip():
        g.write(" ".join(line.split()[3:1]) + "\n")

f.close()
g.close()
```

Figure 16: Lst file Parser

```

000135dc 1      F .text  00000110 nordic_transfer
000136ec 1      F .text  00000020 nordic_readRegister
0001370c 1      F .text  00000020 nordic_writeRegister
00000000 1      df *ABS*  00000000 wireless.c
00013a90 1      F .text  00000030 nrf_irq_callback
00013ac0 1      F .text  00000070 nrf_driver_init
00013b30 1      F .text  00000050 nrf_driver_app_recv
00013b80 1      F .text  0000005c nrf_driver_send
00013bdc 1      F .text  00000022 nrf_driver_receive
00013bfe 1      F .text  00000064 wireless_get_queued_pkt
00013c62 1      F .text  00000028 nrf_driver_get_timer
2007c410 1      O .bss   00000004 g_ack_queue
2007c414 1      O .bss   00000004 g_rx_queue
2007c418 1      O .bss   00000004 g_nrf_activity_sem

```

Figure 17: Input lst file

```

i 00011a28 .text wsFileTxHandler(str&, CharDev&, void*)
v 00011d70 .text hl_print_line()
i 00011da8 .text hl_mount_storage(FileSystemObject&, char const*)
a 2007c000 .data  g_pkt_hist_wptr
a 2007c3a9 .bss   g_pkt_hist
a 2007c2d8 .bss   g_driver
a 2007c2ec .bss   g_mesh_pnd_pkts
a 2007c37c .bss   g_our_name
a 2007c394 .bss   g_mesh_stats
a 2007c3a0 .bss   g_locked

```

Figure 18: Output LUT file from lst file

The following figures represent the input map file, map file parser and the generated LUT file. Both the output files differ in the address.


```

a .bss 0x2007cd50 g_rit_callback
a .bss 0x2007cf20 g_freertos_runtime_timer_start
a .bss 0x2007cf3c gp_timer_ptr
v .text 0x000106f4 high_level_init()
i .text 0x0001159c taskListHandler(str&, CharDev&, void*)
i .text 0x00011688 memInfoHandler(str&, CharDev&, void*)
i .text 0x000116ac timeHandler(str&, CharDev&, void*)
i .text 0x00011770 logHandler(str&, CharDev&, void*)
i .text 0x00011900 cpHandler(str&, CharDev&, void*)
i .text 0x00011980 catHandler(str&, CharDev&, void*)

```

Figure 21: Output Lut file from map file

3. Hex++ Generator:

To generate a final image (Hex++) that contains the symbol table linked against the primary image, we have used Makefile that contains the shell commands. This is the secondary Makefile as the eclipse IDE is auto-generating a Makefile for the primary compile and build process.

Inside the makefile, we are executing the following steps in the form of commands to generate several files in order to generate the final image.

1. During the primary build process, the makefile is generating one or more of these files, say LST or MAP or ELF, that will be required by the secondary makefile for extracting the symbol table.
2. Once the primary build is successful, the secondary build process is started automatically but the setup configurations in the eclipse IDE.
3. The first step in the secondary makefile will invoke the python parser, that will take in one of the specified files as input to extract the symbol table.
4. The symbol table generated from the parser is then taken by the next target. The symbol table initially is just a text file. This is converted to an Object file by using the 'objcopy' commands specific to the operation. During this process, we are also appending the start and end of this object file with certain definitions to facilitate the access of the symbol table during the run time.

5. After the Object file is generated, it will be linked against the original ELF image generated during the primary build process. This is achieved by creating a separate section in the flash memory on the target processor by modifying the linked script of that target. This section in the memory will contain the symbol table object file data.
6. With the final ELF generated, we use this to generate other supported executables like the HEX image, named HEX++ file that will contain the primary image and the symbol tables linked to the primary image.

4. Symbol Search

When user enters a string, a function name or variable name, in the debug terminal, handler function, that works as a symbol table parser, in debug software activates. Handler function reads the string from the terminal and matches the string with the symbol names from the symbol table that is included with the application code in executable file. Because the starting address and size of the symbol table is known, Program starts from the starting address of the symbol table to match the user entered string with the symbol names in the symbol table up to the end of the symbol table.

When the parser finds the symbol name, it gets the associated address. Depending on the symbol type, variable name or function name, handler function moves the PC to the address. If the symbol is a variable name, it returns the value of the variable to the terminal. In case of symbol name being a function, the handler function moves the PC to the address given by the parser and start executing from there until the end of the function.

Functions with parameters and return types are handled same as it handles function with void type and no parameter, except, the terminal asks user to enter the parameters. After user enters parameters, handler function reads the parameters and packs them in a stream of data bytes and sends the data to the 'generic function' handler.

```

699 /*
700  * Print the contents from symbol table
701  * @Sample:
702  * list sym:    Print the symbols
703  * list range:  Print the range
704  * list var:    Print all variables
705  * list fun:    Print all functions
706  */
707 CMD_HANDLER_FUNC(listHandler)
708 {
709     char *trace = table_start;
710     char func_name[256] = { '\0' };
711     char type[10] = { '\0' };
712     char para_type[2] = { '\0' };
713     char addr[9] = { '\0' };
714
715     if(cmdParams.compareTo("sym"))
716     {
717         printf("**** START OF SYMBOL LIST ****\n");
718         while (trace < table_end)
719         {
720             putchar(*trace);
721             trace++;
722         }
723         printf("**** END OF SYMBOL LIST ****\n");
724         return true;
725     }
726     else if(cmdParams.compareTo("range"))
727     {
728         printf("****Start Address: %p\n****End Address:  %p\n", table_start, table_end);
729         return true;
730     }
731     else if(cmdParams.compareTo("var"))
732     {

```

Figure 22: Symbol Search Logic

During the packing of the parameters in the stream of data bytes, padding of one byte is added before each parameter that represents the type of parameter that follows after that byte. For example, there are three parameters of type char, int and long. Size of char, int and long is one, four and eight respectively. In this case, handler function adds 3 more bytes each in front of three parameters. The data stream looks something like, padding - data char – padding – data int – padding – data long.

5. Symbol Execution

Symbol execution deals with executing functions and variables. In our project we have dealt with global variables and function that have

- Void return and void parameters
- Void return and Integer parameters
- Integer return and Integer Parameters
- Any other data type with void return type

After the symbol search returns a valid address of the symbol, it is checked for the type of symbol. It is then categorized as either a variable or a function of a type mentioned above by parsing the string returned by the search algorithm. Accordingly, the symbol handler corresponding to the type is invoked.

```
#define LITTLE_ENDIAN 1
#define BIG_ENDIAN 0
//define aliases for existing data types for ease of accessing them from the debugger
typedef float float_d;
typedef long long_d; //long type not required since we already have uint32
typedef double double_d;

struct data_types
{
    char c;
    int i;
    float f;
    double d;
};

class VoidGenericFunction
{
public:
    VoidGenericFunction(){};
    virtual ~VoidGenericFunction(){};
    virtual void operator()(uint32_t addr) = 0;
};

//Generic function handler for four int params and void return type
class IntGenericFunction
{
public:
    IntGenericFunction(){};
    virtual ~IntGenericFunction(){};
    virtual void operator()(uint32_t addr, uint32_t param1, uint32_t param2, uint32_t param3, uint32_t param4,
```

Figure 23: Symbol execution code snipped

5.2. System implementation issues and resolutions

This section targets mainly in explaining the issues encountered in every sub module of the project. This will follow the same structure as above, explained section wise.

1. Makefile

During the makefile design and implementation, we have faced few problems and these led to a better design.

- The initial design plan was to include a single makefile for the whole build process. But, that seemed to be less practical as the user have to do several modifications to his build environment. So we have come up with 2 step building by adding the secondary makefile to the post-build options leaving the primary makefile as it is.
- Another issue we faced is to access the primary build files, which cannot be done dynamically in makefile. So instead we have included directory paths instead of providing the files manually using the concept of wildcards in makefile.

```
OBJS      := $(wildcard $(BLDDIR)/newlib/*.o) \
$(wildcard $(BLDDIR)/debugger/src/*.o) \
$(wildcard $(BLDDIR)/L5_Application/source/cmd_handlers/*.o) \
$(wildcard $(BLDDIR)/L5_Application/source/*.o) \
$(wildcard $(BLDDIR)/L5_Application/examples/*.o) \
$(wildcard $(BLDDIR)/L5_Application/*.o)
```

Figure 24: Makefile wildcards

The major issue in the makefile design was, during the linking process the symbol table section is being ignored from the secondary ELF file. This was due to the flag to ignore sections that weren't referenced. To overcome this, we have disabled the setting and included all the unused sections, but the catch here was we have to do it for both the steps in the build process, the primary and the secondary builds, for this to work.

2. Parser

There were many issues with the initial parser design and required a lot of design improvements. The issues with the parser are listed below.

- One of the major concern was that the lst and the map file had different addresses and so we came up with the map file parser.

- The size of the LUT file generated had to be as low as possible, which required eliminating the unwanted data.

```
000135dc 1      F .text  00000110 nordic_transfer
000136ec 1      F .text  00000020 nordic_readRegister
0001370c 1      F .text  00000020 nordic_writeRegister
00000000 1      df *ABS* 00000000 wireless.c
00013a90 1      F .text  00000030 nrf_irq_callback
00013ac0 1      F .text  00000070 nrf_driver_init
00013b30 1      F .text  00000050 nrf_driver_app_recv
00013b80 1      F .text  0000005c nrf_driver_send
00013bdc 1      F .text  00000022 nrf_driver_receive
00013bfe 1      F .text  00000064 wireless_get_queued_pkt
00013c62 1      F .text  00000028 nrf_driver_get_timer
2007c410 1      O .bss   00000004 g_ack_queue
```

Figure 25: Lst file before parsing

This file required several stages to remove unwanted data and reduce the file size to generate the required output file.

```
i 00011828 .text  wsRxHandler(str&, CharDev&, void*)
i 00011c54 .text  wsStreamHandler(str&, CharDev&, void*)
i 00011a28 .text  wsFileTxHandler(str&, CharDev&, void*)
v 00011d70 .text  hl_print_line()
i 00011da8 .text  hl_mount_storage(FileSystemObject&, char const*)
a 2007c000 .data  g_pkt_hist_wptr
a 2007c3a9 .bss   g_pkt_hist
a 2007c2d8 .bss   g_driver
a 2007c2ec .bss   g_mesh_pnd_pkts
```

Figure 26: LUT file after parsing

- In order to expedite the symbol search, the flags were added. The flags a, v, i were added for variables, voids Functions and function with arguments respectively.
- The Python GUI was implemented to give the users the flexibility to choose between map or lst files as input.

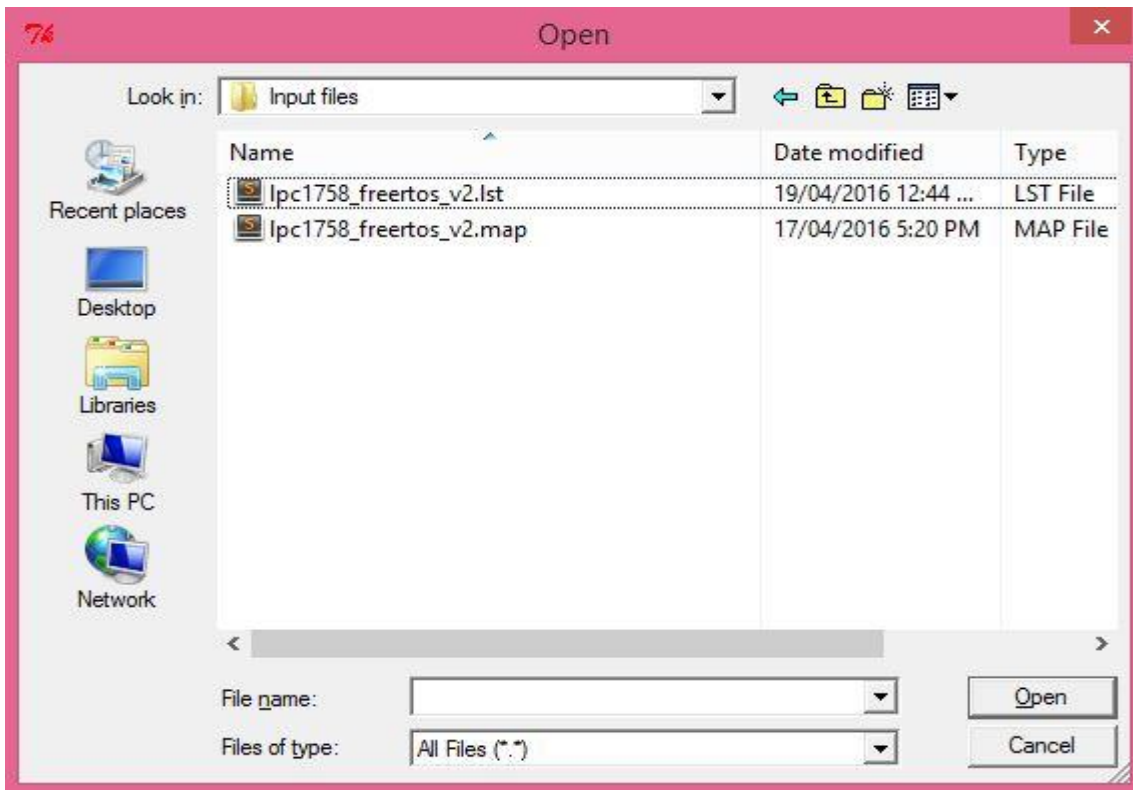


Figure 27: GUI to select input file

5.3. Used technologies and tools

This sections describes the technology and tools used to achieve the above mentioned design. It justifies the reasons to choose that technologies and tools, where did we use it and how it helped the whole system implementation.

1. ELF: This **E**xecutable and **L**inkable **F**ormat is the standard file format for executables, object code, libraries etc., The whole project works around this file format as it is the primary executable format being generated and being used to extract other types of files like the HEX image to program the target platform. It is one of the widely accepted file format on UNIX and UNIX-like systems on x86 machines.

2. Makefile: Makefile is a special file containing the shell commands that will be executed. These commands are executed upon calling make; which is a UNIX utility designed to start the execution of a Makefile. In this project, Makefile initialized the secondary build and coordinates several steps to generate a final image with symbol table embedded into it.

3. Python: We used Python language for designing the parser. It is currently being used in many application and also it reduces the source code size, as it gets the work done using minimum instructions.

4. C & C++: The application source code for this project is written in C and C++ programming languages.

5. Eclipse: This is an open source IDE (Integrated Development Environment) we have used as our primary program editor. This tool supports multiple platforms and target platforms through add-on applications.

Chapter 6. System Testing and Experiment

6.1 Testing and experiment scope

1. Makefile

The primary testing for makefile is to verify if the primary ELF and the secondary ELF agree on the sections and the addresses in the symbol table. Through the tests, we have checked several instances of the symbols for address verification among the primary and secondary files. So we have determined the symbol_table section is added in the right location with respect to other sections as defined in the linker script to accommodate the symbol_table sections. The addresses of all the symbols are verified to be as per the expectations agreeing to the addresses populated in the lst/ map files.

2. Parser

The Parser testing involved a lot of data to be filtered to generate the necessary output. The generated symbol table is then attached to the HEX image to create HEX++. Following screenshots illustrates the parser different layers of filtering the input file.

```
2007c000 1 d .data 00000000 .data
2007c270 1 d .bss 00000000 .bss
2007c04c 1 0 .data 0000008c g_isr_array
2007c270 1 0 .bss 00000004 g_last_sbrk_ptr
2007c274 1 0 .bss 00000004 g_next_heap_ptr
2007c278 1 0 .bss 00000004 g_last_sbrk_size
2007c27c 1 0 .bss 00000004 g_sbrk_calls
2007c280 1 0 .bss 00000004 g_input_dev_fptr
2007c288 1 0 .bss 00000004 g_output_dev_fptr
0001160c 1 F .text 0000003c wirelessHandlerPrintStats(CharDev&, mesh_stats_t*, unsigned char)
00011648 1 F .text 0000001e wsStatsHandler(str&, CharDev&, void*)
00011668 1 F .text 00000118 wsTxHandler(str&, CharDev&, void*)
00011780 1 F .text 00000034 wsAddrHandler(str&, CharDev&, void*)
```

Figure 28: Parser layer 1

```

00011c54 .text      0000011c wsStreamHandler(str&, CharDev&, void*)
00011a28 .text      0000022c wsFileTxHandler(str&, CharDev&, void*)
00011d70 .text      0000000c hl_print_line()
00011da8 .text      00000094 hl_mount_storage(FileSystemObject&, char const*)
2007c000 .data      00000004 g_pkt_hist_wptr
2007c3a9 .bss      0000000c g_pkt_hist
2007c2d8 .bss      00000014 g_driver
2007c2ec .bss      00000090 g_mesh_pnd_pkts

```

Figure 29: Parser layer 2

```

00011c54 .text      wsStreamHandler(str&, CharDev&, void*)
00011a28 .text      wsFileTxHandler(str&, CharDev&, void*)
00011d70 .text      hl_print_line()
00011da8 .text      hl_mount_storage(FileSystemObject&, char const*)
2007c000 .data      g_pkt_hist_wptr
2007c3a9 .bss      g_pkt_hist
2007c2d8 .bss      g_driver
2007c2ec .bss      g_mesh_pnd_pkts
2007c37c .bss      g_our_name

```

Figure 30: Parser layer 3

```

i 00011828 .text      wsRxHandler(str&, CharDev&, void*)
i 00011c54 .text      wsStreamHandler(str&, CharDev&, void*)
i 00011a28 .text      wsFileTxHandler(str&, CharDev&, void*)
v 00011d70 .text      hl_print_line()
i 00011da8 .text      hl_mount_storage(FileSystemObject&, char const*)
a 2007c000 .data      g_pkt_hist_wptr
a 2007c3a9 .bss      g_pkt_hist
a 2007c2d8 .bss      g_driver
a 2007c2ec .bss      g_mesh_pnd_pkts

```

Figure 31: Parser layer 4

3. Symbol Search

Below are the test cases and experiment we performed for symbol search:

- Determine if the symbol table is placed along with application executable.
- Symbol table is being printed correctly.
- Handler function reads user entered string from the terminal.
- Handler function can match user entered string with symbol names in symbol table.

- Handler function reads correct address for requested symbol name.
- Handler function reads correct parameter types from the symbol table for requested symbol name.
- Handler function generates data byte sequence correctly in case of functions with parameters.
- Handler function can return the symbol address to the generic function handler.

6.2 Testing and experiment approaches

Symbol Search

- To test the symbol table position, use the system variable defined as an ‘extern’ in the program that was used in loader file to load the symbol table.
 - Print the data from start address to the end address of the symbol table as defined in the loader file.
- Check if the printed symbol table in the terminal matches with the symbol table in ELF file generated after pre-build.
- Print the scanned string from the terminal to verify correctness of UART and terminal.
- Compare string scanned from the terminal with each of the symbol name in the symbol table. Verify if the comparison works correctly and returns correct function and its address. Also verify if match is not found, handler does not return arbitrary function name and address and throws exception to the user.
- Verify the address of the symbol name with the address in ELF file. Both the address should be same to execute the function.
- Enter a function name that has parameters in the terminal and verify if the parser can extract all the parameters correctly with correct data type. Print the parser output and match it with the ELF file symbol table.
- Pack all the parameters in a stream of data byte along with padded bytes. Print the stream of byte to output and check the data correctness.

- Verify if only the address associated with requested symbol name is passed to the generic function handler. This can be verified by printing the return address and matching it with the respective address in the ELF file.

Algorithms

Parser Algorithm

- Use the 1st file generated after the code is compiled as the input to the parsing algorithm.
- Run the lexical analysis on the files to break it into small part generating tokens having predefined definitions.
- Run the semantic analysis on the tokens to generate strings termed as symbol.
- Use symbol appender to generate the symbol table consisting of function names, variables and addresses.
- This symbol table is passed as one of the input to the hex generator.

Search Algorithm

- The user enters the input string which is to be debugged, it can be a function name or variable.
- This input string is passed to the memory map unit which invokes a function that returns the starting address of the lookup table.
- This address is passed as the input to the search algorithm to find the function or variable.
- The whole lookup table is scanned and the function address is returned.

- The function is then located on the memory and the value or parameters of the function are returned.
- The returned data is displayed on the terminal through UART.

6.3 Testing and experiment reports

Test Cases

- Check for generation of .lst file and correctness of the file.
- Check for sequence for unwanted strings in the parser.
- If queue is full and symbol appender does not dequeue, verify the parser operation.
- If queue is empty and parser is not queuing verify operation of appender.
- Verify queuing and dequeuing from the same position of the queue.
- Verify the generate HEX++ file.
- Verify correct flashing of HEX++ file.
- Check if enough space is available to accommodate the HEX++ file in the ROM.
- Check for validity of the function name entered on the terminal.
- Check for function name entry in the LUT.

Reports:

A sample execution is showing in the following output below. The demo consists of simple commands from the tool. Each command will result in an appropriate output. If the symbol (variable or function) is found, the value of that variable or the executed output of that function will be presented to the user. In case, the entered symbol is not

in the system or if the user entered something that is not valid, the user will be warned appropriately.

```
LPC: list sym
**** START OF SYMBOL LIST ****
a 0x1 .data 2007c000 00050c00 0004c000 2**3 CONTENTS, ALLOC, LOAD,
a 0x2 .bss 2007c2b0 00050eb0 0004c2b0 2**3 ALLOC
a 0x2007c000 d 00000000 .data
a 0x2007c2b0 d 00000000 .bss
a 0x2007c064 .data g_isr_array
a 0x2007c2b0 .bss g_last_sbrk_ptr
a 0x2007c2b4 .bss g_next_heap_ptr
a 0x2007c2b8 .bss g_last_sbrk_size

LPC: read val_int
Variable found at: a 0x2007c000 .data val_int
val_int = 345
Finished in 144571 us
LPC:
LPC: exec foo
Found the function inside symbol table
v 0x00010cb4 .text foo()
**** Started Execution ****
The square of val_int 345= 119025
**** Done Execution ****
Finished in 118383 us
LPC:
LPC: assign val_int -12
Variable found at: a 0x2007c000 .data val_int
val_int = 345 (old)
val_int = -12 (new)
Finished in 145044 us
LPC:
LPC: exec foo
Found the function inside symbol table
v 0x00010cb4 .text foo()
**** Started Execution ****
The square of val_int -12= 144
**** Done Execution ****
Finished in 118340 us

LPC: exec multif 10 23.4
Got Float: 23.400000
Found the function inside symbol table
i 0x00010d5c .text multif(int,
**** Started Execution ****
Enum[0] value: 1
Enum[1] value: 2
Enum[2] value: 4
Enum[3] value: 10
mul(10, 23.400000): 234.000000
**** Done Execution ****
Finished in 121547 us
LPC:
```

Figure 32: Sample Output Demo

Chapter 7. Conclusion and Future Work

7.1 Project summary

As a team of embedded engineers, we got the motivation to implement this system through our personal experiences during the embedded development. Through research, we have noticed that the need for such a tool is ever increasing. So we set to develop the tool for us or for the embedded community. With the ability to execute functions during run time or to find out values stored in a variable dynamically, the development process is much more convenient and effective. With the current status of the system being able to execute global variables and functions with arguments of primitive data types, it has reached the primary objective. The results are promising through our testing process and working on the project was a whole new experience. It feels good supplying the community with a useful tool that engineers will use in their development. Thereby leading to better technology and innovation.

7.2 Future work

This project has been developed with the intention to help embedded software developers easily debug their code. The target platform chosen to test this project is LPC1758 ARM Cortex-M3 microcontroller, running FreeRTOS. While this platform is widely used, we intend to expand our project to support multiple platforms. This requires us to separate our search and symbol execution code from the FreeRTOS code and couple it loosely with any other target platform. We also intend to develop the project to debug code running on bare-metal devices. There is also scope to introduce remote debugging and code patching capabilities which would help debug embedded devices not physically connected to the programming device. This would require us to develop the system to dynamically update the lookup table which was programmed as part of the hex file. This is challenging since dynamic updates to files in the executable is difficult.

References

1. Tankul Akgul, Pramote Kuacharoen, Vincent Mooney, Vijay K. Madiseti. Debugger operating System for embedded systems (2002). Retrived by Oct 2003, from <https://www.google.com/patents/US20030074650>

[Configuration of a debugger that saves resources and debugs on a target system rather than from a host system. The target system stores a table with global variables and addresses, and a module table with system-wide functions and addresses. In response to a trigger, a debugger module is loaded from the host system and linked to the target system by causing the debugger module to exchange information with the tables. The debugger module uses the table to find a variable address and sets a pointer to the address. The debugger module provides the table with a local name and address of a variable local to the debugger module. The debugger module uses the module table to find an MT address of a function and sets a pointer to the MT address. The debugger module provides the module table with an MT address of a function local to the debugger module.]

2. Andrew Blasciak, Greg Parets. System of debugging software through use of code makers inserted into spaces in the source code during and after compilation. (1991). Published by November 1993,
From <https://www.google.com/patents/US5265254>

[A system for inserting code markers for observing indications (external to the microprocessor upon which the software operates) of the occurrence of an event in the execution of the software. Additional instructions or markers are added to the software to be debugged to produce simple, encoded, memory references to otherwise unused memory or I/O locations that will always be visible to a logic analyzer as bus cycles. Although the code markers cause a minimal intrusion in the underlying software, they make tracing events by a conventional logic analyzer much simpler and allow for performance evaluations in manners not heretofore possible. In particular, the inserted code markers

provide a method of dynamically extracting information from a running software system under test using simple low intrusion print statements, encoded I/O writes on procedure entries and exits and the like. Generally, the code markers are inserted at compile time or interactively during the debug session to make visible critical points in the code execution, such as function calls, task creation, semaphore operations and other resource usage so as to speed isolation of problems at test points during debugging.]

3. Timmerman, M. Dept. of Computer and Inf. Sci., R. Mil. Acad., Brussels, Belgium.
Gielen, F.J.A. (1992) the design of DARTS: Dynamic debugger for multiprocessor real time applications.

From http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=277471

[The authors propose a dynamic debugging technique, which presents the possibility of trapping errors specific to real-time applications. The DARTS (Debug Add on for Real-Time System) monitors, which use the dynamic assertion method, can be used as activity monitors. In that case the DARTS gives the user crucial information on the temporal logic of the system (direct control flow). DARTS monitors can also perform watch functions on variables and control the invariant relations which may exist between application and system level objects (indirect data flow)]

4. Kwangyong Lee, Chaedeok Lim, Kisok Kong, and Heung-Nam Kim. A Design and Implementation of a Remote Debugging Environment for Embedded Internet Software.

From Languages, Compilers and tools for embedded systems book

[It is necessary to use development tools in developing embedded real time application software for Internet appliances. In this paper, we describe an integrated remote debugging environment for Q+ (QPlus) real-time kernel which has been built for an embedded internet application. The remote development toolset called Q+Esto consists of several independent support tools: an interactive shell, a remote debugger, a resource monitor, a target manager and a debug agent. Using the remote debugger on the host, the developer

can spawn and debug tasks on the target run-time system. It can also be attached to already running tasks spawned from the application or from interactive shell. Application code can be viewed as C source, or as assembly-level code. It incorporates a variety of display windows for source, registers, local/global variables, stack frame, memory, event traces and so on. The target manager implements common functions that are shared by Esto tools, e.g., the host target communication, object file loading, and management of target-resident host tool's memory pool and of target system symbol-table, and so on. These functions are called OPEN C APIs and they greatly improve the extensibility of the Esto Toolset. Debug agent is a daemon task on real-time operating systems in the target system. It gets requests from the host tools including debugger, interprets the requests, executes them and sends the results to the host.]