Embedded Debugger

Final Report

Schweitzer Engineering Laboratories

Mentor

Ananth A. Jillepalli
Washington State University

Subham Behera, Daniel Ochoa, Howard Potter

CptS 423 Software Design Project II

Spring 2024

TABLE OF CONTENTS

I. Introduction

Schweitzer Engineering Laboratories develops industrial-grade devices designed to protect critical infrastructure. These devices, like many embedded systems, occasionally encounter software faults that need to be resolved efficiently. To improve SEL's ability to diagnose and fix these issues, we want to implement an exception handler for their embedded platform. This handler should capture and store valuable system information related to the CPU and memory when a software issue is detected.

I.1. Mentor

Ananth A. Jillepalli is a professor at Washington State University who mentored us throughout the development of this project.

I.2. Contact information

Schweitzer Engineer Laboratories
Phone: 509.332.1890

I.3. Sponsor

Schweitzer Engineering Laboratories creates world class products that enhance the power grid in many ways such as preventing problems like blackouts and implementing cutting-edge solutions which add cybersecurity and automation. This makes the grid very efficient, however it is not impervious. To minimize downtime and quickly resolve issues, preventative measures must be implemented. This is where the embedded debugger comes into picture.

II. Team Members & Bios

Daniel Ochoa is a computer science student interested in software development. Daniel's skills include C/C++, C#, Python, Haskell, and SQL. For this project, his responsibilities include developing the exception handler and snapshot handler..

Subham Behera is a computer science student interested in game development. Subham's skills include C/C++, Python, Haskell, and SQL. He has had prior experience programming games such as space invaders and tetris. For this project, his responsibilities include developing an integration channel between the serial port and its decoder by making an interface.

Howard Potter is a computer science student interested in software development and software security. Howard's skills include C/C++, Python, Haskell, SQL and C#. For this project, his responsibilities include parsing and decoding the stack dump.

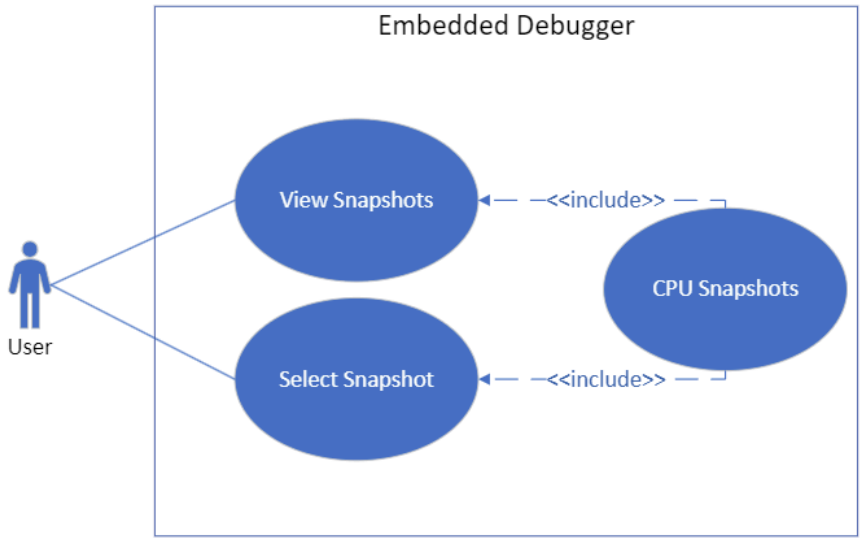
III.Project Requirements Specification

III.1. Project Stakeholders

The primary client is Schweitzer Engineering Laboratories. They prefer the main code to be in C/C++ and it should use ARM assembly to communicate with the hardware. It is given that they need this as a backup plan whenever a product fails so they can revert it to a previous state without much loss.

Other indirect stakeholders include consumers who use power that is supplied from power grids operated by Schweitzer Engineering Laboratories. With more advanced technology comes more responsibility and if the embedded debugger wouldn't have been considered to develop, newer products would be completely wasted if they were to fail. This means more chances of blackouts and unreliable energy, affecting consumers.

III.2. Use Cases



III.3. Functional Requirements

III.3.1. Exception Handler

Description	This application will be an embedded debugger that contains error and fault handling. The embedded debugger must create a system snapshot that retains system information related to the CPU core registers and associated memory.
Source	Internal requirements
Priority	Level 0: Essential and Required Functionality

III.3.2. System Snapshot Storage

Description	This application needs to store the system snapshot and CPU core and memory information in flash memory
Source	Internal requirements
Priority	Level 0: Essential and Required Functionality

III.3.3. Scripts and TCL Commands

Description	This application needs to contain scripts to read the snapshot information. It must also have TCL commands to restore the CPU to the snapshot state
Source	Internal requirements
Priority	Level 0: Essential and Required Functionality

III.3.4. Test Setup

Description	This application must contain a test setup that verifies required functionality.
Source	Internal requirements
Priority	Level 0: Essential and Required Functionality

III.3.5. User Interface

Description	This application will be an embedded debugger that contains error and fault handling. The embedded debugger must create a system snapshot that retains system information related to the CPU core registers and associated memory.
Source	Internal requirements
Priority	Level 0: Essential and Required Functionality

III.4. Non-Functional Requirements

Self-Contained

The application should be complete and specific enough to be understood and implemented without additional information. It will include all the necessary details for the user to be able to use all the functionality of the application.

System Compatibility

The application developed must be compatible with the SK-KV260-G development boards provided by Schweitzer Engineering Laboratories (SEL). Compatibility refers to the seamless integration and functioning of the software/system on these boards without any technical constraints or issues.

Programming language

The application must be primarily developed in C/C++ for core functionality. C/C++ are essential programming languages for embedded systems development due to their efficiency and direct hardware interaction capabilities. TCL scripting language should be used for specific tasks, such as automation, board communication, and debugging processes. TCL is a scripting language commonly used for hardware/software interaction, automation, and testing.

Efficient Performance

The application must demonstrate efficient performance under normal operating conditions and expected user loads. The average response time for critical user interactions should be within an acceptable range, defined based on user expectations and industry standards. The application must also be capable of efficiently storing and loading a system snapshot within acceptable timeframes.

IV. Software Design - From Solution Approach

IV.1. Architecture Design

Overview

As this project is made to handle exceptions automatically, as well as be accessible for debugging and system restoration purposes, a model in which different systems directly interacted with each other both with and without direct user input was required. With this specification in mind, a Model-View-Controller architecture will be used for this. While this model is generally used for GUI and web application development[1], it will also be suited for this project. See appendix 1 for the model.

Subsystem Decomposition

The Model in this instance will be the main logic handler. It will interpret data that is being sent from the controller for interpretation, dumping, and restoring. It will be connected to a storage medium of some form which it can read and write from. As this pertains to just reading and writing, it is well suited to being dedicated as a singular component.

The controller is almost self-explanatory. It will control the signals from the main components of the program, as well as the SOM that the debugger will be running on. It will detect any exception thrown, either by the SOM or manually triggered, which will then communicate with the Model to handle the logic of dumping data. A very similar case happens when the exception is being handled by restoring a previous snapshot, again by the SOM's exception or a manual trigger.

The view represents the user interactivity end of this application. From here, a user can send manual signals to the controller. This can range from loading snapshots manually, causing an exception, or viewing logs of the system. All of these requests will be sent to the Controller to handle.

1. Controller

a. Description

The primary responsibilities include processing logical commands received from the Controller Exception Detector/Handler and orchestrating the collection of system snapshots. This subsystem ensures efficient snapshot capture and coordinates with the Data Storage subsystem for the seamless storage and retrieval of captured data.

b. Concepts and Algorithms Generated

Our approach is to directly access CPU core registers and memory areas using low-level assembly instructions. This approach provides real-time access to the system state. We want to capture exceptions as soon as they occur so we will also be using ISRs to capture snapshots during specific software exceptions. ISRs are routines triggered by specific events, making them suitable for capturing system states during exceptions. Although managing multiple ISRs can be complex, careful design and modularity can simplify the implementation. Each ISR can focus on capturing specific types of exceptions, ensuring a modular and maintainable codebase. We also

want to minimize our interrupt handling overhead by using careful optimization and efficient code implementation. This will help ensure that the system's overall performance is not significantly impacted.

c. Interface Description

Services Provided:

Service Name	Service Provided To	Description
CaptureSnapshot	Data Storage	The Handler will Handler and store it in the Data Storage.
RetrieveSnapshot	Controller Exception Detector/Handler	The Handler will get CPU snapshot from Data Storage and provide it to Controller Exception Detector/Handler.

Services Required:

Service Name	Service Required	Description
SendData	Data Storage	The Handler receives data from the Data Storage.
SendSnapshot	Controller Exception Detector/Handler	The Handler receives data from Exception Detector

2. View

a. Description

While direct user input into this system is limited, it is an important consideration during the design process of this application. This component of the system will handle just that – the user's ability to interact with this debugger. This includes accepting user inputs to select different menus and send different signals, as well as printing information on screen.

b. Concepts and Algorithms Generated

This component will handle printing onto the screen and accepting input from the user. Entering inputs will be limited to text inputs from the user, as the user interface is planned to be a command line interface. This will also handle loading information required for the user to know important information about snapshots.

With the limited scope of this component, the only algorithms needed will be ones to read data being given to it by the controller through a request, as well as the logic for the menu selection to send requests as intended to the controller.

c. Interface Description

Services Provided:

Service Name	Service Provided To	Description
SaveSnapshot	Controller	Sends a signal to the controller to save the current information in the CPU as a snapshot.
LoadSnapshot	Controller	Sends a signal to the controller to load a selected snapshot into the CPU.
CauseException	Controller	Send a signal to the controller to manually trigger an exception
ViewLog	Controller	Sends a signal to the controller to send a list of snapshot information (names, dates, cause of exception, etc.)

3. Model

a. Description

This part of the system will be the backbone of the debugger. This means there will be no user interaction here as it will be in the backend performing tasks based on requests from other parts of the system. Its main job is to process requests and handle data depending on what type of request. Whenever there isn't a request, it passively collects data.

b. Concepts and Algorithms Generated

As this handles storage and retrieving data, we need to think about data structures. Perhaps a stack data structure to store snapshots is the best because it implements last in, first out (LIFO). The most recent snapshot will always be stored at the top and when the most recent snapshot is needed, it can be instantly popped from the top, providing efficiency.

c. Interface Description

Services Provided:

Service Name	Service Provided To	Description
SendData	Controller	Send the snapshot that was requested.
SendStatus	Controller	Send the status of an action.

Services Required:

Service Name	Service Provided From	Description
SendRequest	Controller	Controller sends what action to take.

IV.2. Data design

This project will require moving and storing data. The application will read data from the CPU core registers and store it in flash memory. It will also load data into the CPU core registers. The data that is being manipulated will be binary data. We will be using a System Snapshot structure to hold the data. As well as some internal and temporary data structures.

System Snapshot Structure:

- A struct or a collection of variables to store CPU core registers and associated memory states captured during a software issue.

Internal Data Structures:

- Data structures used internally within the software for processing and manipulation of captured data.
- Status Flags: Flags or enumerations can be used to indicate the status of various operations or conditions within the system. For instance, a flag might indicate whether a snapshot capture operation was successful or if an error occurred during the process.
- Arrays, linked lists, or queues for managing multiple snapshots. Data structures for handling stack decoding and exception handling logic.

Temporary Data Structures:

- Buffers: Temporary buffers can be used during the process of capturing system snapshots or during the restoration process. For instance, when reading data from CPU core registers or writing data to flash memory, temporary buffers are employed to hold the data before processing or storage.
- Stacks or Queues: Data structures like stacks or queues can be used temporarily for managing intermediate data during the snapshot capture or restoration process.

IV.3. User Interface Design

As the purpose of this debugger is to catch exceptions to dump and restore snapshots on an embedded system, the User Interface will be kept basic, as doing these actions manually should not be a common occurrence. A Command Line Interface (CLI) will be implemented for user interaction. This will still allow basic information to be printed on screen and allows the user to perform required functions. As this debugger is not meant for consumers or a typical user, it can be assumed that any individual interacting with this machine will be comfortable with a command line interface. This has the added benefit of being very simple, keeping it easy to implement and adapt to any requirements.

This interface will largely consist of a main menu giving the user several options for input. One of these options will be to view the log of saved snapshots. This will provide the user with contextual information on any snapshots available (names, date of origin, cause of exception if applicable, etc.). This will allow the user to view any available snapshot, as well as select one for further usage (loading, examining for further information). An example of this concept can be found in the appendix.

Additionally, from this main menu, the user will also be able to manually create a snapshot of the current CPU's data, for examination and backup reasons. The user will also be able to manually trigger an exception from this menu – this feature may not be included in the finished product, or if it is only when a debug mode is activated, as it is mainly for diagnostic purposes. This can also be decided by an end user, as these options would be configurable.

This menu could be made available whenever the SOM is running, and accessible to any connection made to the SOM (for example, a terminal), as a debugger like this should be otherwise left to run.

V. Test Case Specifications and Results

V.1. Testing Overview

Project testing will compose of both automated unit testing, as well as manual testing done during development. While components of this project can be tested using unit tests, such as the Exception Detector, there is an anticipation for manual testing for other components, such as reading data from the CPU registers. Due to this, not every test can be completed for every component, however this should still allow for testing through a CI pipeline. Any component that requires manual testing can be well tested before integrating, and all others can be automatically tested by CI. Because of this, CI may not be observed at the beginning aspects of the project due to these unique constraints, however this will not remain true for the project's lifetime. Generally speaking, the testing process will be:

i. Write code: A feature for a component will be developed, and will continue development until it is believed to fulfill requirements of the feature.
ii. Determine test case: For every functionality, it will be determined if this function will require manual testing, or automated testing.
    a. If manual testing: The developer will generate and justify a list of testing requirements for the function. This list will then be documented.
    b. If automated testing: The developer will generate test cases to test both expected values and edge cases, or other exceptional behavior.
iii. Run tests: The developer will run the tests for that function within the testing environment, whether they be automated or manual tests.
iv. Fix issues: If any test fails, the developer should address these failures. Repeat until all tests pass.
v. Push code to remote: The developer will push their code to a remote branch.
vi. Pull request to main: The developer will make a pull request against the main branch. At this point another developer will perform a code review and approve of the commit
vii. Merge branch: The branch will be merged into main.

V.2. Environment Requirements

The environment will require having access to the KV260 SOM, as this project uses this specific model for its purposes. The testing environment must also be capable of reading data through a serial connection with the SOM to transfer data between the SOM and the computer being used for testing and development.

Required software for the environment includes Python 3, as this is the language used for the scripts, as well as Xilinx Vitis for running applications on the SOM itself. Additionally, the computer being used for testing must also use Windows for the purpose of extracting the symbol table from an instance of the SOM being run, as the objdump provided by Vitis runs on Windows, however it is possible that other applications are available to do this same task that can work in other operating systems.

V.3. Test Results

V.3.1. Deadlock Detection

V.3.1.1. Test Case Requirement

This test case requires FreeRTOS to be enabled.

V.3.1.2. Details

Tests if a task can detect when two other tasks try to take each other's semaphore resulting in a deadlock.

V.3.1.3. Test Approach

Three tasks were created. Two of them initialized their own semaphores and after some time tried to take each other's semaphore. The third task kept on checking when this deadlock happened.

V.3.1.4. Expected Result

The third task should successfully be able to detect the deadlock.

V.3.1.5. Observed Result

The third task successfully detected the deadlock.

V.3.1.6. Test Result

Test passed.

V.3.2. Stack Trace - Single Thread

V.3.2.1. Test Case Requirement

This test case requires that the trace stack can be recreated from snapshot data.

V.3.2.2. Expected Result

The expected results of this test is that the recreated stack trace matches that of an identical instance of execution on the vitis debugger, and that which is reported by the compiler. In this case, a program with the following trace stack was used:

bar()

fooFooFoo()

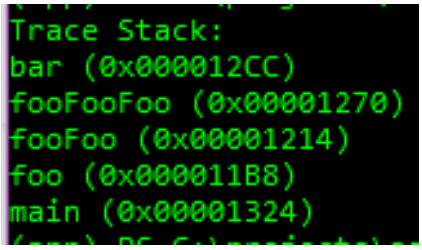
fooFoo()

foo()

Main()

V.3.2.3. Observed Result

The stack trace script produced the following result:



V.3.2.4. Test Result

Test passed.

V.3.3. Stack Trace - FreeRTOS

V.3.3.1. Test Case Requirement

This test case requires that the trace stack of non-active tasks can be traced from snapshot data. This test involves only the non-active tasks, as the active task is traced just as a single-threaded application would be traced

V.3.3.2. Expected Result

The expected results of this test is that the recreated stack trace of a given task matches that of an identical instance of execution on the vitis debugger, and that which is reported by the compiler. In this case, a program with the task “Tx” with the following trace stack was used:

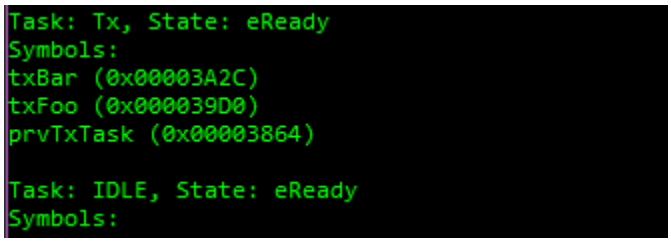
txBar()

txFoo()

prvTxTask()

V.3.3.3. Observed Result

The stack trace script produced the following result:



V.3.3.4. Test Result

Test passed.

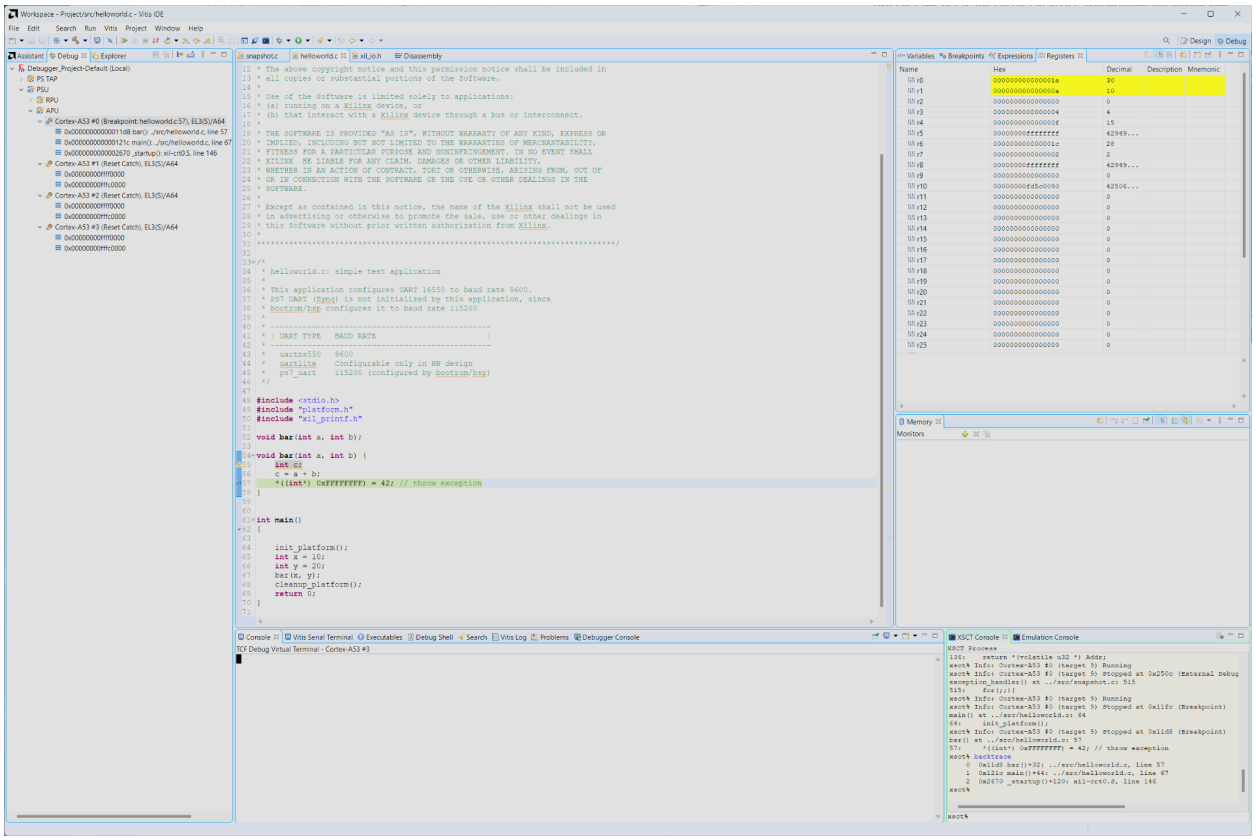
V.3.4. Snapshot Store and Writeback

V.3.4.1. Test Case Requirement

This test case requires that snapshot data can be written back to the program correctly.

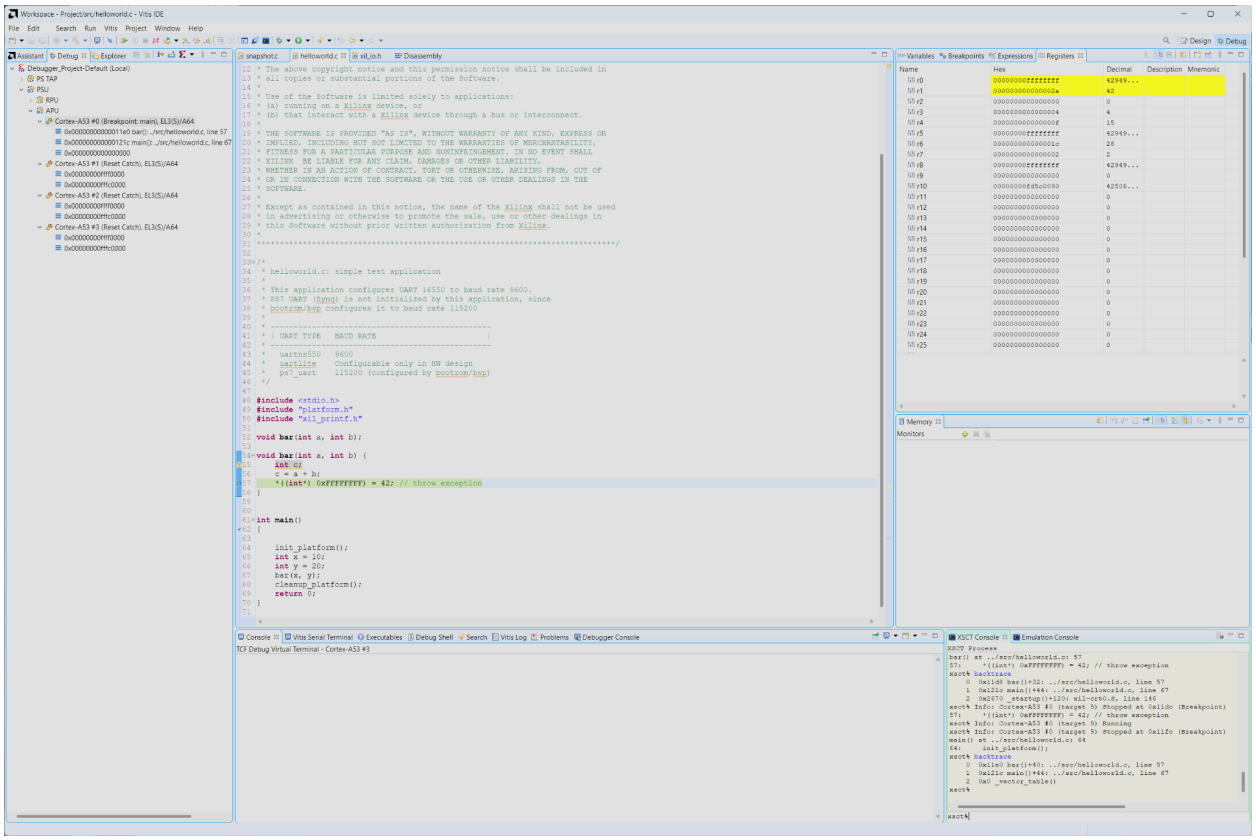
V.3.4.2. Expected Result

The expected result of this test is that the debugger goes back to the point of exception when the snapshot is loaded onto the board. At this point, the registers and stack should match. We expect the values of the registers to be the same and for the stack trace command to work properly. Below is a screenshot of the Vitis debugger at a breakpoint at the line of the exception. We can see the registers on the upper right box, and the backtrace on the lower right box. After the exception is caused, the board will automatically send the snapshot data to the snapshot handler. After the snapshot handler has received the data, we will reset the board and write back to it. We expect to see something similar to this. There can be slight variations as functions are called after the exception to retrieve the data.



V.3.4.3. Observed Result

We can see that the Vitis debugger has correctly stopped at the point of exception after writing the snapshot back to the board. The registers are the same as well as the backtrace. There is a slight difference on the third call but that is expected as it started off in the vector table instead of startup. This result shows that both storing the snapshot and writing back are working as intended.



V.3.4.4. Test Result

Test passed.

V.3.5. FreeRTOS Snapshot Store

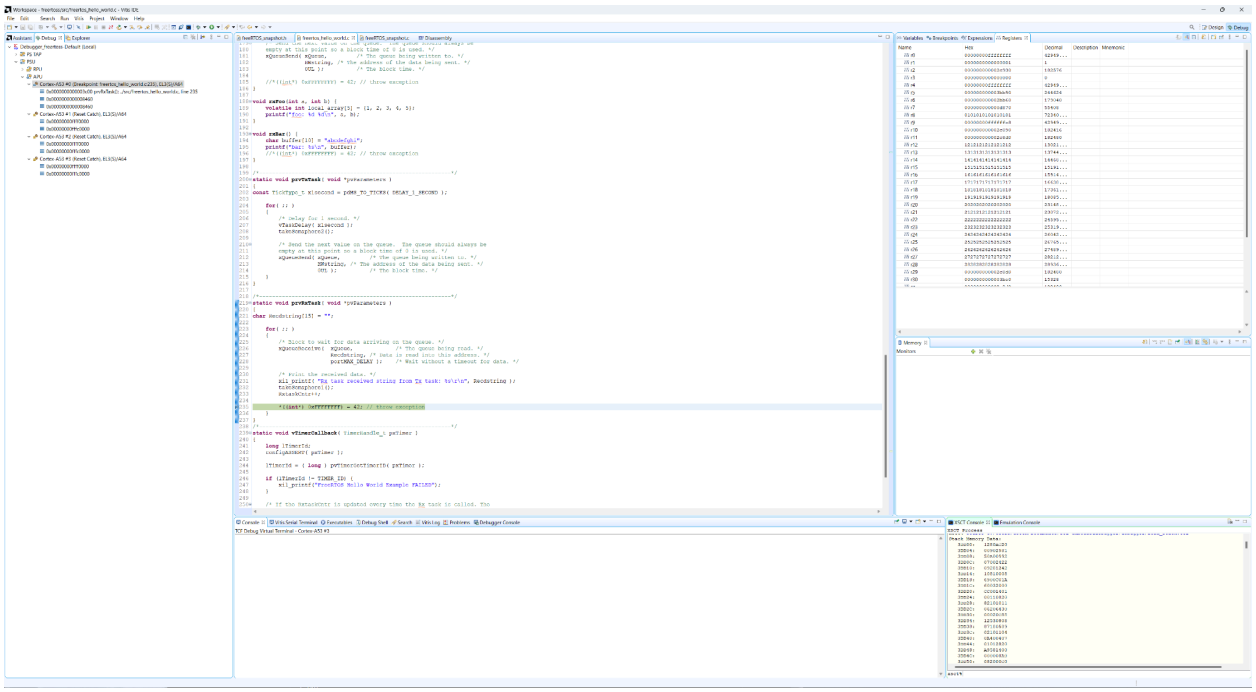
V.3.5.1. Test Case Requirement

This test case requires that snapshot data match the Vitis debugger.

V.3.5.2. Expected Result

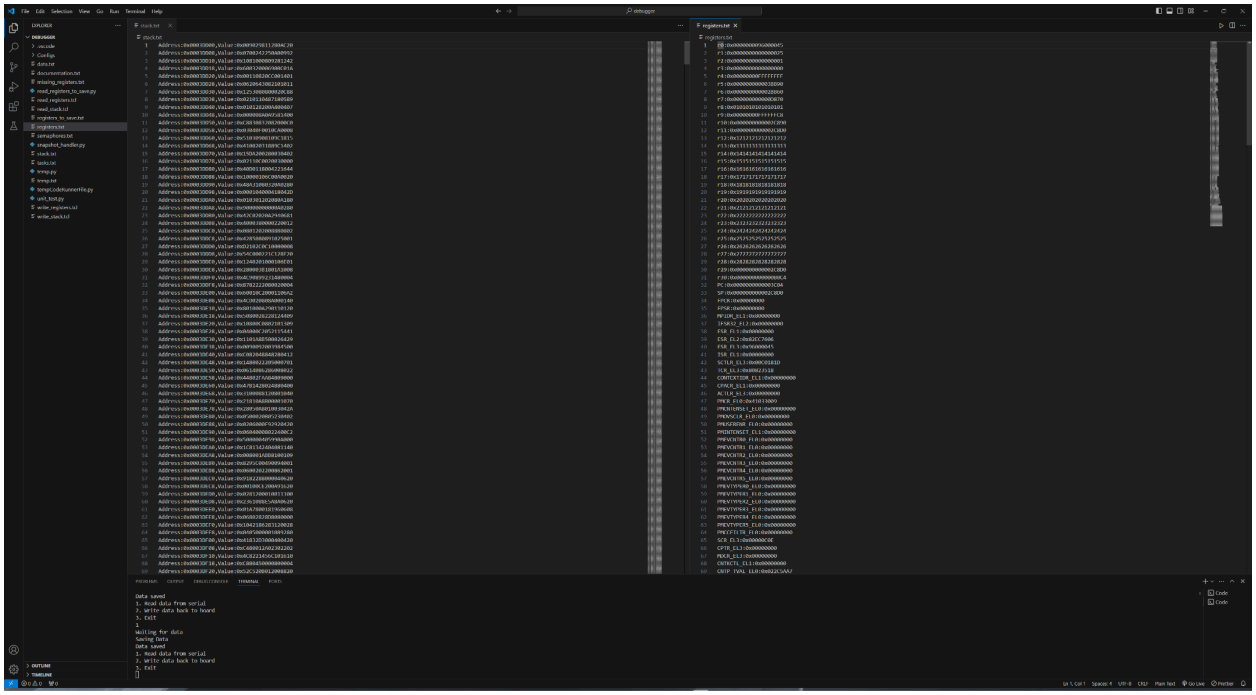
The expected result of this test is that the data generated by the exception_handler and written out into text files by the snapshot handler matches the data shown in the Vitis debugger. For the FreeRTOS, we are not writing the data back to board, just reading it. We expect the values of the registers to be the same. Below is a screenshot of the Vitis debugger at a breakpoint at the line of the exception. We can see the registers on the upper right box, and the stack on the

lower right box. After the exception is caused, the board will automatically send the snapshot data to the snapshot handler which generates the text files.



V.3.5.3. Observed Result

We compare the data in the text file to the data in the Vitis Debugger. The stack and registers are the same. There is a slight difference on the R0 register which happens as the return value can be lost between function calls. This result shows that both storing the snapshot and writing back are working as intended.



V.3.5.4. Test Result

Test passed.

VI. Projects and Tools used

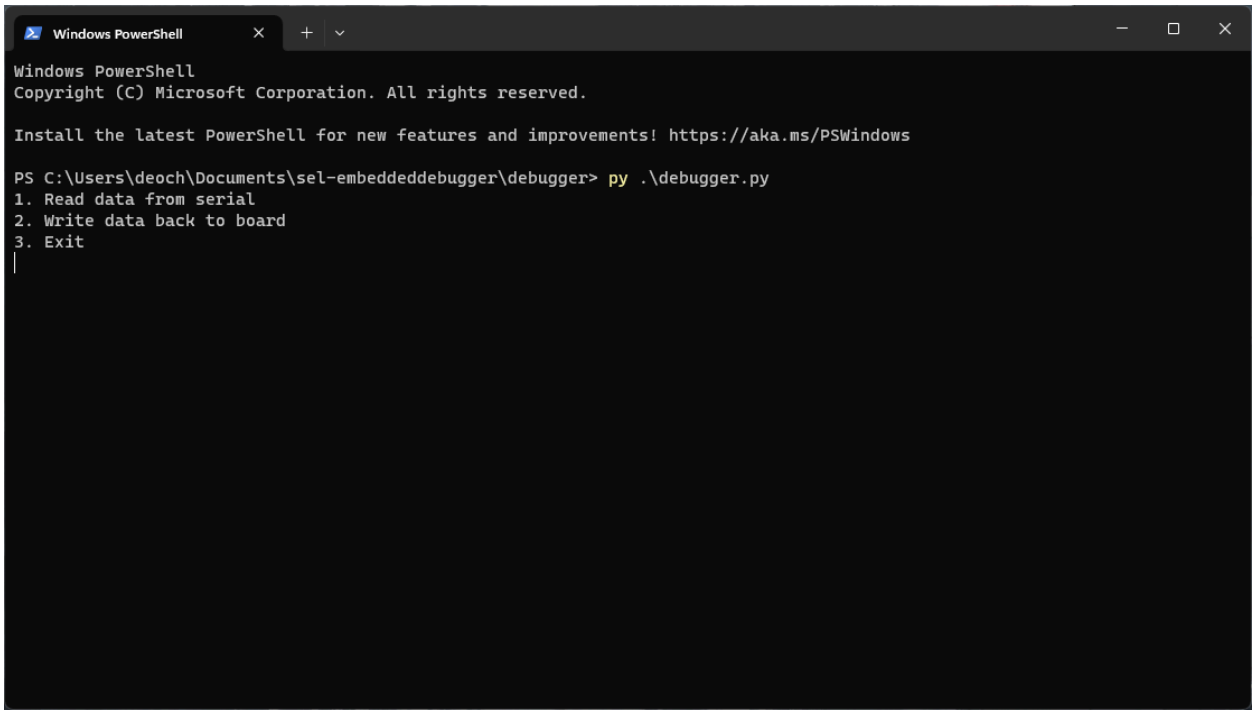
Tool/library/framework	Quick note on what it was for
Xilinx Vitis	IDE designed for use with KV260 SOM
pyserial	Used to communicate with SOM through COM port
pathlib	Python library for locating files within directories
subprocess	Python script to use multiple processes
FreeRTOS	Multithreaded library used in stretch goals for multithreaded capabilities

Languages Used in Project			
C	TCL	Python	Assembly

VII. Description of Final Prototype

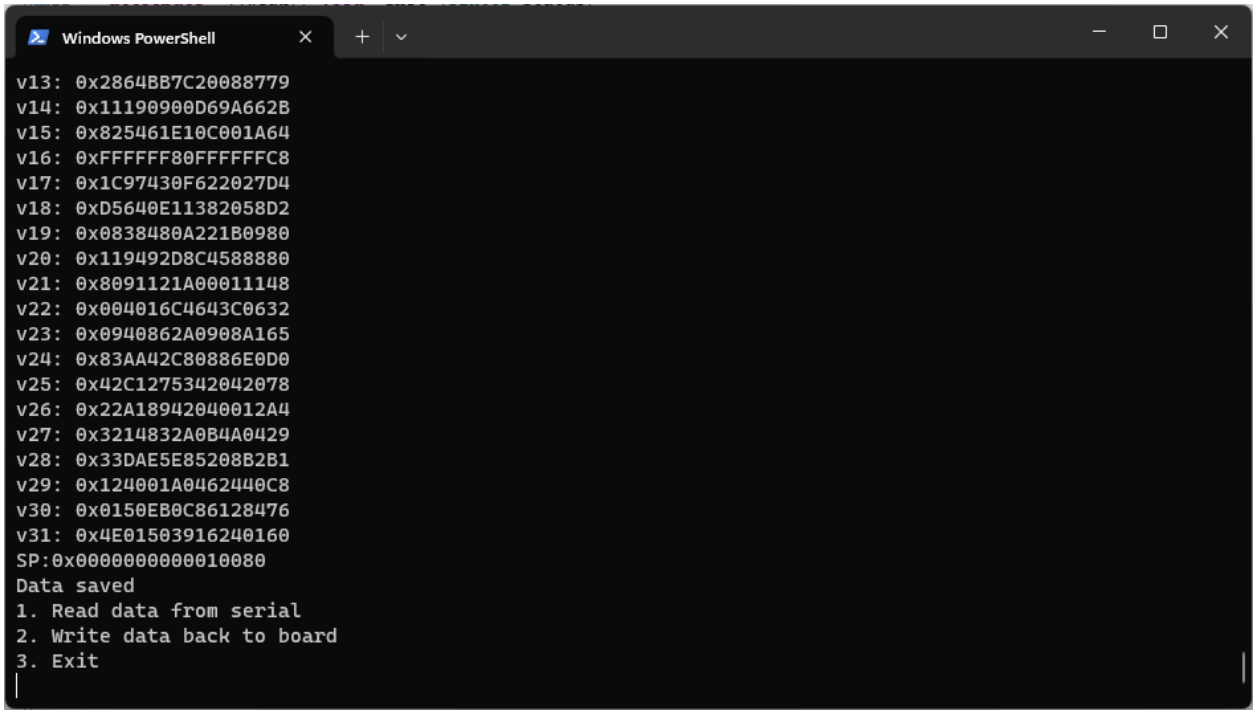
VII.1. Standard Debugger

The debugger is used to retrieve the snapshot from the SoC which contains the CPU's register and stack data. The debugger can also generate TCL scripts that will write back the snapshot to the SoC.



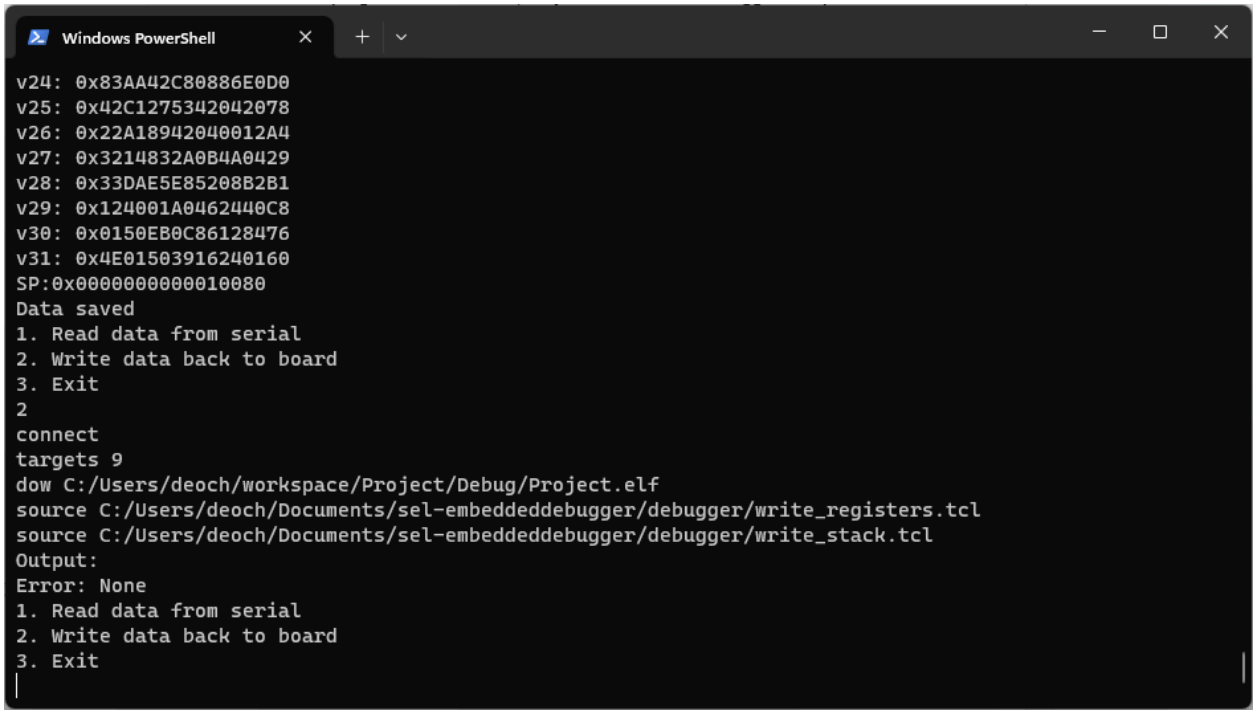
Debugger: Main Menu

When the user wants to run a program on the board, they can launch the debugger and press 1. This will put the debugger into listen mode where it will wait for the SoC to throw an exception. The snapshot.c file on the SoC will send a signal to the debugger when an exception is thrown and pass the snapshot information. When the debugger starts recording information, it will show what data is being recorded then announce once it is finished.



Debugger: Data Saved

If the user wants to load the snapshot back to the device after a snapshot has be stored, they can press 2. The debugger will first use the register and stack data to generate TCL scripts to write it back to the board. It will then start a subprocess which uses the XSCT console to run those scripts. It will first connect to hw_server/TCF agent then it will download ELF and binary file to target. Lastly it will run write_registers.tcl and write_stack.tcl commands. The debugger shows any output from the XSCT console as well as if any errors occurred. The SoC will now have the same register and stack memory as the snapshot that was stored by the debugger. The user can press 3 to exit the debugger.



Debugger: Data Wrote Back to Board

VII.2. Deadlock Detector

One of the tasks before implementing multiple threads with FreeRTOS was to check if deadlock detection between two tasks worked properly. This is important to test and document because should such a scenario happen when using multiple threads, now we know how to use another task to detect this so we can solve it and not cause further delay.

VIII. Product Delivery Status

Our project was demoed on the last week of April 2024, and has been handed off to SEL during the first week of May 2024. Our prototype works in both single core and multicore applications by FreeRTOS implementation, as demonstrated during our client demo.

The project is stored on our github repo, which SEL will take over. The Github repo, as well as installation instructions and usage guidelines can be found at:
https://github.com/WSUCptSCapstone-F23-S24/sel-embeddeddebugger

The equipment issued to us for development of this project has been handed back to Lihua at SEL.

IX. Conclusions and Future Work

IX.1. Limitations and Recommendations

Environment setup is very particular at the current moment. The Vitis IDE needs to be installed in its default directory. Because of this, refactoring the code to be able to use other directories has not been a priority, however it is a limitation that should be considered for addressing.

IX.2. Future Work

Our current exception handler depends on the snapshot handler running on the host PC to receive data transmitted through the serial port. However, future iterations of this project could explore storing this data on non-volatile memory instead. This modification would eliminate the dependency on the snapshot handler and enhance the practicality of the exception handler, especially for units deployed in the field where connections to monitoring computers may not be feasible.

In a FreeRTOS environment, our exception handler can efficiently capture crucial system information, including stack traces, register values, task details, and semaphore statuses during an exception. However currently the exception handler is limited to logging this information externally and is unable to directly write it back to the embedded system. Integrating functionality to write the data back to the embedded system is another area for future development.

X. Acknowledgements

Technical Mentors: Preston Stephens, Lihua Ran

Project Management Mentor: Ananth Jillepalli

XI. Glossary

Deadlock: When two tasks are indefinitely blocked due to each waiting for the other to release resources

Error Handling: The process of identifying, reporting, and managing errors or exceptions that may occur during the execution of a program. Error handling is primarily concerned with handling unexpected events, exceptions, or errors that arise as a result of incorrect or unexpected input, software bugs, or other exceptional conditions.

Fault Handling: The process of detecting, isolating, and responding to faults or failures that occur within a system or application. Fault handling deals with managing faults or failures that can occur due to a variety of reasons, including hardware failures, software bugs, environmental factors, or unexpected system behaviors.
GUI: Graphical User Interface. The application in which the user interacts with graphical components.
System Snapshot: A capture of the system's current state at a specific point in time. It represents an image of the system, including the state of memory, CPU core registers, and other relevant system components.
TCL: Tool Command Language. It is a scripting language that is commonly used for various purposes, including automation, scripting, and embedded systems.
ISR: Interrupt Service Routine. It is a function in embedded systems and operating systems that is invoked when a specific interrupt condition occurs. Interrupts are signals sent by hardware devices or software processes to the CPU, indicating that an event has occurred that needs immediate attention. When an interrupt is triggered, the CPU interrupts its current execution, saves its state, and transfers control to the appropriate ISR to handle the interrupt event.
System On a Chip (SOC): An integrated circuit that incorporates different components onto one chip, as an integrated system.
System-On-Module (SOM): An integrated circuit board that is modular by design and can accept many different components to suit a need, including SOCs

XII. References

““System-on-Modules (SOMs): How and Why to Use Them”, AMD XILINX, 2023, available at https://www.xilinx.com/products/som/what-is-a-som.html, accessed (accessed Sep 28, 2023)
“We Do Our Part So They Can Do Theirs, and Together We Power the Future,” selinc.com. https://selinc.com/company/our-part/
“Arm Architecture Reference Manual for A-profile architecture,” Arm Developer, April 2023, available at https://developer.arm.com/documentation/ddi0487/latest/
“Zynq UltraScale+ MPSoC Data Sheet”, AMD, November 2022, available at https://docs.xilinx.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview
“Kira K26 SOM Data Sheet”, AMD, available at https://docs.xilinx.com/r/en-US/ds987-k26-som/Overview (accessed Sep. 28, 2023)

XIII. Appendix A – Team Information

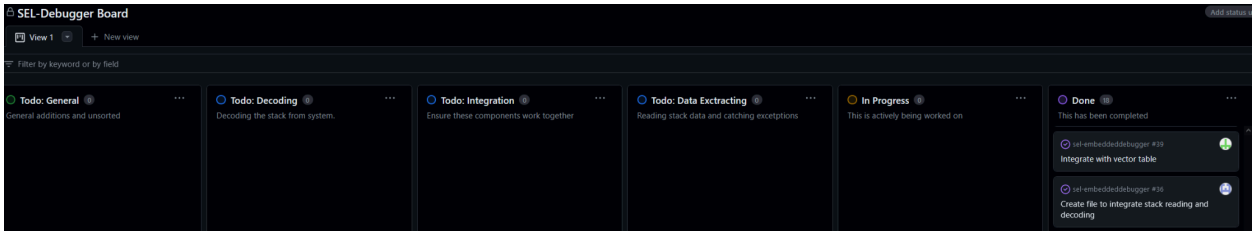
Left to right: Daniel Ochoa, Subham Behera, Howard Potter

XIV. Appendix B - Project Management

We have weekly meetings with our instructor every three weeks when available. We talk about recent progress and upcoming assignments relating to the course, and how they pertain to the project. More importantly, we have weekly meetings with our sponsor every Monday. This is the most beneficial meeting as we talk about what we did the past week and can get live feedback on different issues. It helps that they know the technology well so they can help a lot. We will also set up additional meetings for troubleshooting problems within the project, and for advice.

For planning documents, we mainly used email to communicate with our sponsor should we need quick help over the week. We used Discord to communicate with each other about our different parts of the project, and to organize meeting times when additional meetings are required, or scheduling conflicts arise.

For planning our project, we utilized Github's issues feature, and project's feature for formal planning. We would also communicate through our Discord for informal support and discussion, as well as emails and meetings on Teams with our sponsors for informal planning.



Example screenshot of Github Projects formatting.