University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

Optimizing Interprocess Communication Between MATLAB and C++ MEX Libraries

Infinera Corporation
555 Legget Drive Suite 222
Kanata, Ontario, Canada

Prepared by
Thomas Dedinsky
ID XXX
userid XXX
2B Computer Engineering
16 April 2020

XXX
XXX
XXX

16 April 2020

Vincent Gaudet, chair
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario
N2L 3G1

Dear Sir:

This report, entitled ‘‘Optimizing Interprocess Communication Between MATLAB and C++ MEX Libraries’’ was prepared as my 2B Work Report for the University of Waterloo. This report is in fulfillment of the course WKRPT 301. The purpose of this report is to analyze the existing behaviour of the current device simulations and to compare and contrast new proposals which aim to fix some of the most inhibiting flaws of the simulations of today.

Infinera Corporation is a frontrunner in the network solutions industry which allows customers of their products ‘‘to scale network bandwidth, accelerate service innovation and automate optical network operations’’ [1]. They create a wide variety of network solutions for different markets with different issues, and aim to provide choice ranging from low power to high performance products.

I was employed with the Firmware Engineering team, which is a group focused on creating the logic which surrounds and powers these optical networking innovations. Specifically the team who helped create the ICE4 optical engine. This team’s work is also used in simulations which help assess and improve the performance of future versions of these engines. This report was written for both the Department of Electrical and Computer Engineering of the University of Waterloo and my employer and team at Infinera.

I received assistance from my employer with helping determine what the current problems of the simulation are and with verifying the integrity of the analysis of the proposed solutions. Additionally, I received assistance from the Department of Electrical and Computer Engineering and Xin (Golson) Xie by using the provided Work Term Report template. I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Thomas Dedinsky
ID XXX

Contributions

During my co-op work placement, I was employed at Infinera Corporation in the position of Firmware Design Engineer - Intern. The team I worked with was relatively small, around a dozen people, however I interacted with other teams on a semi-regular basis. I was specifically in the Firmware Engineering team, however I interacted with the Systems ASIC Engineering team frequently and spent a fair amount of time working with them.

The team's main goal was to create the logic which actually powered the optical network devices being produced by Infinera. This involves an extensive knowledge in the field of digital signal processing as well as co-operation with different teams, such as the Systems ASIC Engineering team and Hardware Engineering team, in order to accomplish this with a level of accuracy and performance that meets design expectations. My work fit into these goals specifically as I helped with the simulations that help assess this logic.

I had three main tasks during my co-op term. The first one is the subject of this report, and the one I spent the most time on, which was the conversion of these device simulations. This is outside of the scope of the report, but there was a lot of time spent recently on optimizing what simulations were run at all. Limiting test cases to not cover every single scenario but merely ''important'' normal cases and edge cases was debated and eventually fine-tuned into a reduced list. These simulations take a lot of time to run cumulatively, and the MEX setup left a lot to be desired regarding performance.

So my biggest main task of the term was to help optimize this MEX setup. This ended up being a very expansive job to not only research but to implement. I had to research and understand new areas of programming that I hadn't touched, such as function programming with regards to the C++ preprocessor, as well as obtaining a slight understanding of how the simulation works. Without spoiling the results of this report, I can say that I felt I made a significant impact with my work, creating a solution which not only improved performance but was still usable and adaptable.

I had other tasks as well which I worked on throughout the term. One of these involved automated testing for our simulation using Microsoft Azure. Azure, as a service, offers a variety of tools suited for a full production environment. Three main features are of use to us, which were Azure DevOps, Jenkins, and Windows VM. Azure DevOps contains repos for code storage, pipelines for automation upon build, and other features. Jenkins, can deploy git repositories and perform automated jobs while running on an Azure VM. Windows VM allows us to configure a testing environment and then duplicate them on other hardware.

I was tasked with looking into their services and determining if it was feasible to use them for our codebase. This involved configuring two virtual machines, a Windows Slave and a Linux Master.

The Linux Master ran Jenkins, which would control the slave via a Java Web Agent. In order to create an automation test which could deploy and run successfully from Jenkins, I created a post code-deploy batch script which ran the MATLAB code and used a lot of convoluted methods to work around the proprietary nature of MATLAB, which discourages this type of testing, and the connectivity issues of the slave. Then I set up an Azure DevOps repository and made it so a push to it would cause the Jenkins test to run without any user input. After everything was connected, I ran some speed comparison tests and researched some pricing comparisons, including batch processing, for my employer.

I also worked on a simulation regarding reads and writes to our registers in the hardware. We have a Linux testing server that can simulate the low-level logic of our hardware. My task was to implement a system so when our firmware does a read/write, it will communicate with that server instead of having to connect to a local implementation. This is extended further for a read/write of multiple values in a block, where it outputs a variable amount of response messages.

This is a multi-use solution, so as a result, I made three cases that can use it. The first is a basic C++ function where you can pass in the text command and get the output. The second is integration into an existing function, which interacts with registers, where it will send the write commands directly to the server when it simulates those. Finally, through a MEX call, you can put your read and write commands through MATLAB and it will call the server and return the output.

The relationship between this report and my job is that the first main task of optimizing the solutions is the subject of my report. I was able to implement engineering decision-making into my work and build a solution that improved the overall quality of the simulation code. The recommendations were performed by me, although the developer was referred to in a general sense in order to comply with report guidelines, and I overall enjoyed this task.

In the broader scheme of things, I now have more competency with important languages and concepts related to Computer Engineering. This job gave me much more experience with C++ and MATLAB, languages that I had only briefly touched in the curriculum beforehand. C++ especially is very useful for lower-level jobs with embedded systems and digital hardware, and I actually learned and implemented modern C++ conventions as opposed to just ‘‘C with objects’’. Ironically, I used no objects in C++, only structures, in order to maintain some MATLAB-compatibility.

I also feel like this job gave me just enough insight into digital signal processing that I am aware of the general overview of how optic fibre communication works and the relevancy of Fourier transforms, a concept we learn several times in classes but never feel like we really learn it, to the whole process. However, it did not encourage me to want to take upper-year signal processing courses or pursue the associated specialization for my degree. Rather, it just slightly discouraged me. This may be for the best, just due to the amount of topics I can explore in my final year that has been agonizingly difficult for me to narrow and choose.

Finally, this job had a higher-level of expectation than my previous co-ops through WaterlooWorks and outside networking. I was given greater responsibility in my role and in the work I do, given time to research a problem in order to implement an ingenious solution, and given a healthy amount of oversight regarding the decisions I made in my code, all of which was greatly appreciated. I shouldn't have to say this, but it's refreshing to be given ownership of a task that has direct impact to the work of your team, be given some semblance of direction where you can get started and explore the problem on your own, and be given the ability to communicate with the people who are going to be using it about how to tailor it properly instead of them just redoing it if they dislike it. I have no major complaints about my time at Infinera.

Summary

The main purpose of the report is to evaluate the current state and potential future statues of our device simulations. This is done by taking into account the current state, strengths, and weaknesses of the simulations, and looking at remedies or solutions which retain these strengths but also overcome these weaknesses. Both quantitative and qualitative criteria is used in analysis with proposed designs in an engineering design decision.

The major points documented/covered in this report are the issues with the current simulation, what criteria new solutions should be assessed with, and the proposal and analysis of said solutions. The issues with the current simulation include variable ownership and duplication, intrafunction and interfunction communication utilization, and consistent practices between developers of the same codebase. The criteria determined to be important for analysis were Performance, Usability, and Adaptability. The solutions proposed were the Unchanged, MEX-Owned, MATLAB-Owned, and C++-Owned versions of the simulation.

The major conclusions in this report are that all four solutions ranked differently in different categories, and that there were no clear winners until values were assigned to the criteria. The best performing was C++-Owned, the most usable was MATLAB-Owned, along with Unchanged which was also the most adaptable. However, the MEX-Owned scored the highest despite not being the highest performer in any single category.

The major recommendations in this report are that this MEX-Owned version of the simulation be implemented. A developer who is not involved in the active development of the logic can set up the framework for unifying and testing these MEX libraries. Then, they can implement automation of the opcode system and boilerplate code. Finally, they can showcase it to the current developers and help them get started in integrating this new system into the Firmware Engineering team's workflow.

Table of Contents

Contributions . . . iii
Summary . . . vi
List of Figures . . . ix
List of Tables . . . x
1 Introduction . . . 1
1.1 Simulations . . . 1
1.2 Interteam Communication . . . 1
2 Background . . . 2
2.1 MEX . . . 2
2.2 Simulation History . . . 2
2.3 Benefits of MATLAB . . . 2
2.4 Limitations of MEX . . . 3
2.5 C++ Preprocessor . . . 4
2.5.1 Boost . . . 4
3 Current Problems . . . 4
3.1 Variable Ownership . . . 5
3.2 Intrafunction Communication . . . 5
3.3 Interfunction Communication . . . 5
3.4 Consistent Practices . . . 6
4 Assessment Criteria . . . 6
4.1 Performance . . . 6
4.2 Usability . . . 6
4.3 Adaptability . . . 7
5 Possible Solutions . . . 7
5.1 MEX-Owned, MATLAB-Maintained Interfunction Communication . . . 7
5.1.1 Preprocessor Programming . . . 8
5.2 MATLAB-Owned, MATLAB-Maintained Intrafunction Communication . . . 8
5.3 C++-Owned, C++-Maintained Intrafunction Communication . . . 9
6 Engineering Analysis . . . 9
6.1 Performance . . . 10
6.1.1 MEX-owned . . . 10
6.1.2 MATLAB-owned . . . 11
6.1.3 C++-owned . . . 11
6.2 Usability . . . 12
6.3 Adaptability . . . 12
6.4 Final Comparison . . . 13
Conclusions . . . 14
Recommendations . . . 15
Glossary . . . 16

References 17
Appendix A MEX Example 18
Appendix B Preprocessing Example 20
Appendix C Testing Example 23

List of Figures

Figure 6-1 A profiler summary of a MATLAB simulation called goRun. It shows where time is spent between MATLAB and MEX functions through the self-time column. . . 10
Figure 6-2 A profiler summary of a MATLAB simulation for a MEX-Owned mockup. . . . 10
Figure 6-3 A profiler summary of a MATLAB simulation for a MATLAB-Owned mockup. 11
Figure A-1 goRunMexDemo.m; this would be the MATLAB-facing code. 18
Figure A-2 mexMexDemo.cpp; this would be the MEX-facing code. 19
Figure B-1 The preprocessor of the following file outputs this to the compiler. 20
Figure B-2 mexMacroDemo.cpp; this would be the MEX-facing code. (1/2) 21
Figure B-3 mexMacroDemo.cpp; this would be the MEX-facing code. (2/2) 22
Figure C-1 goRunTestingDemo.m; this would be the MATLAB-facing code. 23
Figure C-2 TestFunction_Setup.m; this would be the MATLAB-facing code. 24

List of Tables

Table 6-1 Comparison between proposed solutions using the defined criteria. 13

1 Introduction

Infinera Corporation creates various network solutions by producing both the hardware that consumers purchase and utilize and the firmware that powers its communication abilities. There are significant barriers for testing the firmware directly on the hardware. Primarily, there is a significant gap of time between the hardware design being finalized and the manufactured hardware being produced. Another factor is that the development of the firmware for the next release happens partially in parallel with the hardware development, meaning the firmware team will not be able to test on the final version of the hardware until they have finished finalizing the design. On top of this, there are a wide variety of tests that testers would like to run, testing many configurations regarding high performance and low power mode, which would be less practical on a limited amount of physical devices.

1.1 Simulations

Thus, simulations of the hardware were created and are maintained. These simulations are updated using design specifications that both the developers working on hardware and firmware adhere to. This helps solve the issues listed earlier. Since this causes less conflict between the hardware and firmware development schedules, there is little delay regarding when testing can begin once hardware is finalized, and vigorous testing can occur on various computational machines.

However, simulations are designed to account for a wide variety of test cases based on practical applications of the device. This includes various amounts of noise, different testing modes, and a large amount of input combinations. While simulations need to be accurate regarding their process, they do not have to match the speed of the produced hardware. Thus, there is an ongoing effort to make these simulations faster while maintaining their accuracy.

1.2 Interteam Communication

There are various development teams at Infinera. The Hardware Engineering team is responsible for designing the hardware and the accompanying Verilog code. The Firmware Engineering team is responsible for creating the C++ code which is the main logic of the device. The Systems ASIC Engineering team is responsible for managing the MATLAB code which runs these simulations and for assessing the results of these simulations. These three teams need to work synergistically in order to accomplish their goals. The Firmware team needs the hardware to run their logic, the Systems ASIC team needs the logic of the device done in order to run simulations, and the Hardware team needs feedback from the simulations in order to detect underlying faults in the actual usage of

the hardware.

2 Background

The simulations are design to run without any involvement of the Hardware team. The main logic produced the Firmware team can function without any communication to the hardware device. However, there needs to be a way for the Firmware team to let the Systems ASIC team use their code for simulation purposes.

2.1 MEX

MATLAB allows the communication between C++ and MATLAB through the use of MEX libraries [2]. Normally in C++, one has to define a main function in which, when it runs, the user controls what are the inputs and when it stops. However, C++ projects can be compiled as a dynamic link library, which means that programs which load and use this library can dictate its execution and its inputs. MEX functions act as a management system for these libraries for MATLAB. Using this, MATLAB can communicate directly with the C++ code, allowing it to be called as a function. An example of this functionality can be found in Appendix A.

2.2 Simulation History

The Systems ASIC team designed the MATLAB simulations so that they called these libraries, but these simulations also have additional code which helped managed these MEX calls. The simulation would create and manage variables which were specific to one or two libraries so it could utilize libraries when needed, but also had logic in-between these MEX calls in order to affect the inputs being created for these calls and how the outputs were analyzed.

2.3 Benefits of MATLAB

MATLAB was chosen because it has three main benefits for the Systems ASIC team. Firstly, it is very flexible in its ability to manage data, allowing scripts with no defined inputs and outputs to be evaluated upon run-time. It also allows polymorphic data, the ability for a variable to change type or an object to modify what fields it owns without prior notice. This is very useful for interacting with data which may gain complexity during a MEX call if it obtains a non-zero imaginary value, the ability to reshape data before and after a MEX call, and modifying the size of matrices.

The second advantage extends this flexibility to being able to manually manipulate data during runtime. Someone who is running a MATLAB simulation can pause the script at any time, assess what the current value of any variable is, modify it (or create new ones), and then continue the execution of the script. This removes the need to store intermediate variables to assess the execution of a program at any particular point.

One thing to note is a library can only be debugged if it is built in Debug mode, however the C++ code is generally much, much slower this way. In practice, this is only used when trying to debug a function, with the MEX call being placed almost at the beginning of the function in order to save time. Additionally, data manipulation cannot occur when viewing these variables inside of a MEX call.

Finally, MATLAB has a very intuitive and expansive plotting feature. This allows users to take any data, open up a new graph window, and populate it through the user of simple commands in the code. These graphs can then be updated as the program continues, allowing for a visual representation of the current state and progress of the simulation.

2.4 Limitations of MEX

Due to the proprietary nature of MATLAB, MathWorks exclusively controls MEX and how MATLAB interacts with C++. They have made some design decisions which have limited the effectiveness of these simulations.

The first issue derives from how MATLAB actually invokes a MEX call. ‘‘The MEX file contains only one function or subroutine, and its name is the MEX file name. To call a MEX file, use the name of the file, without the file extension.’’ [2] This means that each function has to be separated into a different library to be able to use a distinct MEX call to run it. This limits the ability for variables to be shared between functions, as this can only be done through communicating these through the MATLAB overhead.

Additionally, extracting these variables is very cumbersome. MATLAB will provide information to the C++ code about how many inputs and outputs it expect, since C++ main functions allow for a variable number of inputs and MATLAB functions allow for a variable number of outputs, as well as data arrays containing these inputs and outputs. From there, MATLAB allows access to information regarding the size of inputs (but not outputs), and not much else, not even the data type.

This means that MEX calls need to be co-ordinated between MATLAB and C++ code during development. Structures which are effortlessly created in MATLAB now need to be carefully dissected in C++, making sure to correlate each field of each structure to the proper MATLAB equivalent. MATLAB also only permits certain functions to be used to convert from and to these MATLAB

variables, meaning experienced C++ developers will need to relearn how to manage data types.

Finally, MATLAB is really slow. What it makes up for usability regarding matrix operations, it loses in performance for computing most complex logic. Regarding computations with mid-size matrices, C++ is upwards of 500 times faster than MATLAB [3]. C++ is a much more efficient language, hence the main logic of the hardware being written in it.

2.5 C++ Preprocessor

C++ code compiles so that the functions can execute properly at run-time, but it also allows for functions to be made to execute at compile-time. Code that is designed for this latter purpose is deemed as preprocessor programming, also known as metaprogramming. This is a functional programming style C++ supports that can be used to generate code at compile-time that is then compiled and executed at run-time, or ‘‘functions that create functions’’. There are many benefits to this, mainly automatic function generation or definitions which are abstracted and consistent throughout the codebase. This can extend to some high-level concepts, such as run-time polymorphism, which can lead to a consistent function call declaration pattern for various input data types. An example of this functionality can be found in Appendix B.

2.5.1 Boost

Boost is a set of libraries designed to enhance the performance of C++ while keeping inline with current programming standards. Many of the libraries they have designed have made it into C++ in recent years, and generally they are widely used [4]. Boost has a preprocessor library which allows for very sophisticated metaprogramming. More detail on this subject can be found at [5] if desired. The functionality that Boost provides will be lumped into the regular C++ preprocessor for any preprocessing conversation without explicit mention.

3 Current Problems

The task of optimizing these simulations while maintaining proper functionality and adding new functionality to correspond with the latest hardware and firmware changes is a daunting problem. There are a few barriers regarding this task, all related to how MATLAB to C++ communication is handled.

3.1 Variable Ownership

One advantage of the C++ code being stored in a library is the ability to persist data between calls. Global variables can be made in order to capture data between MEX calls, so they do not have to be passed in again if they have not been modified. This can reduce the amount of data transmitted and thus the speed at which these functions operate at. However, this requires the ability to be able to set a starting value, which with only one function cannot be done in as clean of a manner. This problem alone is relatively easy to solve through the use of an opcode system and getter/setter functions.

However, data persistence comes with it the topic of variable ownership. Currently all of the variables in the simulation are owned by MATLAB. If the libraries are also retaining copies of these variables, this means that there are two separate variables representing the same purpose. This increases the amount of memory consumption, processing time, and overall maintenance. However, by shifting the ownership of the variables to solely C++, the memory and time costs reduce significantly. However, maintenance is an issue if the MATLAB code touches the variables too often with specific setters and getters, and this may negate the performance gains entirely.

3.2 Intrafunction Communication

As mentioned previously, libraries are limited to one main function, so MEX calls cannot directly communicate between each other. However, by combining multiple functions into one, this barrier disappears. Combining functions by having one top-level function can reduce the number of MEX calls and the amount of data transferred between if inputs are common or outputs of one function become inputs of another. This means that the MATLAB code between these functions would have to be ported to C++, as well as losing some of the ability to view and modify these intermediate values.

3.3 Interfunction Communication

The concepts of data persistence and intrafunction communication can be combined to implement interfunction communication. This would allow different functions to access these captured variables simply through a C++ function call. This can be very useful for the reasons described above, however it encounters more trouble. A limitation is that these calls are being separated by some distance, meaning a careless MATLAB programmer could change the variable between MEX calls while the C++ code is unaware of this change, thus producing an incorrect result. Thus, interfunction communication would need to be treated with more care, with or without data persistence tools.

3.4 Consistent Practices

Much of the hassle when dealing with MATLAB and MEX calls is the proprietary nature of these. There exists no easy-to-use open source libraries for dealing with data conversion. Data conversion patterns are hard to establish, especially with structures, as MATLAB only provides conversion functions for primitive data types. This means that every non-primitive data type which is defined in MATLAB needs a corresponding data type as well as conversion functions for the MEX call in order to function properly. Historically, this has resulted in competing patterns for structure conversion function declarations with some structures having specific macros dedicated to a field of theirs despite it already being defined elsewhere in the codebase. Unifying macros and function declarations is a good start to establishing these practices with little resistance.

However, this can be extended further if we introduce preprocessor programming and define some macros for structures. With this, we can pass in these definitions into functions which will create the code to handle structure declaration as well as structure conversion. However, this creates a level of abstraction which may be unintuitive to future users using the code, and can run into the same issue the current macro system has. This does not negatively affect the performance, however the time saved from using macros may less than the time lost from developers having to understand these macros. This also applies for using any of this preprocessor programming with the previous issues.

4 Assessment Criteria

The following quantiative and qualitative criteria will be looked at in this report.

4.1 Performance

The main focus regarding this is the ability to speed up the simulation. Most of these problems are based around the premise that this has a negative impact to the simulation speed and solutions to these problems can speed this up. This can be assessed quantitatively, using time measurements and estimates based on research and experiments done.

4.2 Usability

Performance of simulations is very important, however if it is very hard to understand how to use the changes that speed up the simulations, then the time cost saved from performance may be less

than the time cost due to developers having to learn a new system. Thus, any changes done need to be usable, having some sort of intuitive nature, as well as being able to be worked on and maintained in a relatively easy manner. This includes striking a balance between abstraction and readability, and retaining functionality of the simulations that people using the simulations desire.

4.3 Adaptability

There is a cost regarding how much time and effort it will take for this to be implemented and for developers to switch over their code and practices to the newly implemented standards which take advantage of these changes. If this is too high, then the system will not get fully implemented in the first place, and the improvements fail to serve their purpose. A small amount of changes needed to existing code and changes which can be implemented incrementally while development continues in an intuitive manner are preferred.

5 Possible Solutions

All of these problems have solutions, however a common solution for each is doing nothing. This will be assessed in the analysis, but will not be explicitly stated here.

5.1 MEX-Owned, MATLAB-Maintained Interfunction Communication

One solution is to only transfer ownership of the data that the MEX functions absolutely need. This includes ownership of parameters, variables which are set once and never changed again, and states, which are variables which are set once and do change but only within the MEX function. There would be the ability to get and set these parameters and states through the opcode system, if needed by the MATLAB code for logic in MATLAB and unconverted MEX functions. Additionally, these would be able to be directly called between functions, as all of the functions would be hosted in one library, as well as be called independently through the use of an opcode system. Once fully converted, the only thing MATLAB would be responsible for is creating the inputs for these functions and analyzing/plotting the outputs. However, this approach is also effective if only partially converted.

An advantage of this approach is that it reduces the amount of redundant data that is passed in. These MEX functions are called many times, often with variables not being modified by MATLAB in-between calls. This extends further, as often variables are shared between function, and they can now be shared without having to convert any MATLAB logic like intrafunction communication. Additionally, the Systems ASIC team would still be able to view and modify parameters and states

at any time with a simple function call, allowing them to work without imposed restrictions of data manipulation.

A disadvantage is that this would require an active effort by developers to maintain an opcode system for these on the MATLAB-side and the C++-side. There's no way that C++ can communicate these opcodes at run-time to MATLAB without copious amounts of data-transfer. So, boilerplate code in C++ would need to be maintained for each setter and getter of parameter, states, and potentially outputs, as well as a massive enumeration list on MATLAB's side which compiles the smaller enumerations each individual function has. This both increases the likelihood of errors that are hard to track and limits the granularity of this implementation.

5.1.1 Preprocessor Programming

These are modifiers which can work in parallel with the proposed solutions. Specifically with preprocessor programming, this can help with the Consistent Practices criteria found in Section 3.4. The ability for preprocessor output allows for complex logic to produce code without taking a performance hit during run-time. Thus, this can solve issues such as boilerplate code, which has plagued the structure conversion functions currently in-place, but can also be used for opcode generation. This will be explored more in the analysis.

This is useful in particular for this solution due to the integration available with the boilerplate code and opcode system. By creating wrappers for the parameters, states, and outputs, they can be invoked in the preprocessor to create variable declarations and getter/setter functions. This then can be tied into the opcode system, by having a post-build step which generates these opcodes for MATLAB. This allows for only one version of the variable opcodes to have to be maintained.

5.2 MATLAB-Owned, MATLAB-Maintained Intrafunction Communication

One possible solution is to make any constant variable, which are not intended to change throughout the simulation, owned by C++. This would allow less redundant variable passing with little additional cost. However, this prevents developers from changing these constant variables between functions without recompiling, or accessing these variables directly without additional work done to the MEX calls. These constants are used somewhat frequently in the MATLAB code, but in specific configurations, so these files can be generated ahead of time for both C++ and MATLAB in specific files for each. As these would be owned by one library, all of the functions would be moved under one main library, with an opcode system in place which is much simpler than the previous option. Functions would not be able to hold any variables, instead being only able to share variables if two functions combined into one.

An immediate benefit is that it requires little work for some gain, as these constants are already organized into variables at the beginning. They do not need to be changed upon run-time, and not transferring them results in a moderate performance boost. Additionally, the intrafunction communication conversion can be done in steps without halting the development process of the simulation.

The disadvantage is that the process is very gradual, resulting in very little improvement unless intelligent conversion occurs. This requires stuffing more and more of the code into the MEX calls, which will slowly reduce the Systems ASIC team ability to view and modify intermediate values.

5.3 C++-Owned, C++-Maintained Intrafunction Communication

This solution requires the most amount of work, however it would yield the most performance benefits. This would give ownership of everything to C++. The code would no longer need to work in a library, but instead could be run directly. Data would be outputted at the end of the simulation for the purposes of plotting in MATLAB. Combining everything into one main function would remove the need to create any sort of opcode system or any complicated data management.

The advantage is much better performance. Simulations can run faster, and more of them can be run in a shorter time-frame.

However, this has several disadvantages. This would require the most amount of work, as every single line of MATLAB would need to be converted. This means that all developers have to buy into this, including the Systems ASIC team. They would lose the ability to modify values at intermediate points, instead only being able to view them if a debug mode was enabled output data at regular intervals, as well as having to adapt to totally new syntax for them. There are also no intermediate stages of this, so this conversion would halt development temporarily.

6 Engineering Analysis

The analysis of the solutions based on the requirements and criteria listed above based on the metrics listed in Section 4 on page 6. Each of these criteria will use a point scale from 1 to 10 when comparing solutions. For the sake of brevity, these solutions will be referred to as MEX-Owned, MATLAB-Owned, and C++-Owned.

6.1 Performance

For analyzing performance, we can use simple quantitative analysis. Figure 6-1 shows a snippet of the profiler summary of this simulation showcasing the top 10 contributors to its length.

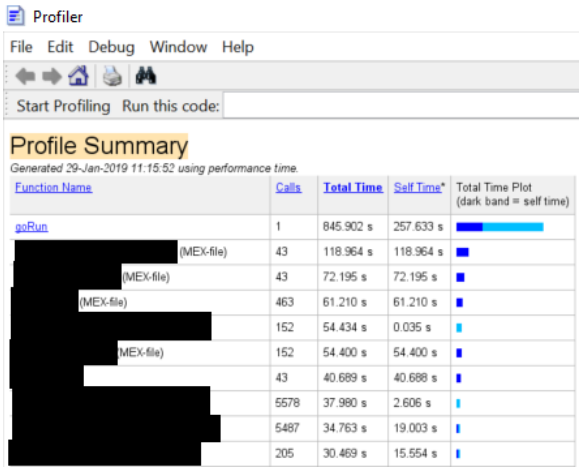

Figure 6-1. A profiler summary of a MATLAB simulation called goRun. It shows where time is spent between MATLAB and MEX functions through the self-time column.

6.1.1 MEX-owned

For the MEX-owned variables, we can compare this to a profiler summary made using a mockup which estimated the time savings if the few longest MEX functions were combined into one library in Figure 6-2.

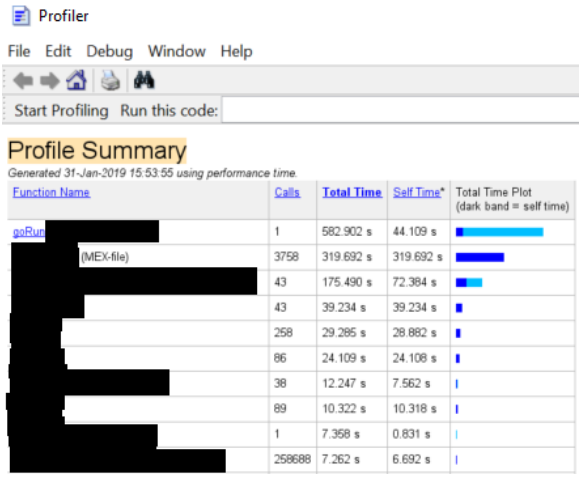

Figure 6-2. A profiler summary of a MATLAB simulation for a MEX-Owned mockup.

This results in an over 30% time save according to equation (1).

$$\frac{Normal-time-MEX-time}{Normal-time} = \frac{845.902-582.902}{845.902} = 0.311 \tag{1}$$

6.1.2 MATLAB-owned

A similar mockup was done in Figure 6-3 with an estimate if the constants were converted and a few somewhat adjacent MEX libraries were merged.

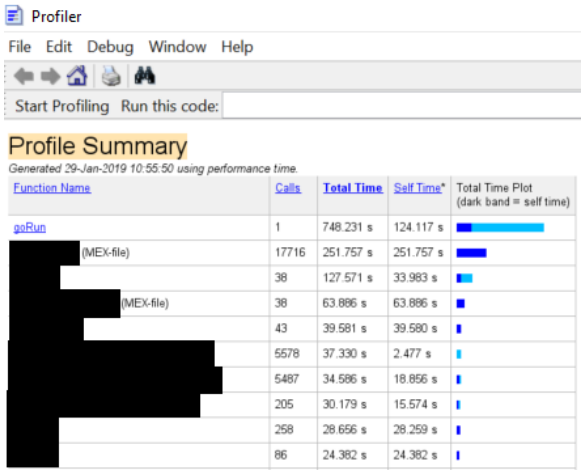

Figure 6-3. A profiler summary of a MATLAB simulation for a MATLAB-Owned mockup.

$$\frac{Normal-time-MATLAB-time}{Normal-time} = \frac{845.902-748.231}{845.902} = 0.115 \tag{2}$$

This results in an over 10% time save according to equation (2).

6.1.3 C++-owned

Using Figure 6-1, we are able to derive an approximate ratio of time spent in MATLAB versus in MEX calls from the top 10 contributors in length using the Self Time.

$$\frac{MATLAB-time}{MEX-time} = \frac{257.633+0.0035+40.688+2.606+19.003+15.554}{118.964+72.195+61.210+54.400} = 1.094 \tag{3}$$

This results in an over 50% time save according to equation (3).

6.2 Usability

The ability to learn and maintain existing code is important. Luckily, the macros themselves will allow for little extra work for the MEX-owned solution. The logic of the functions can be changed with no additional charge. If parameters/states need to be added or removed, then this requires a one-line change in C++ in order to use it appropriately in MATLAB with getters and setters throughout the simulation.

For MATLAB-owned, it is very usable. Having to define constants in a separate file which is swapped out is also very usable, as they can be defined by functions with a simple name change to obtain different constants.

The C++-owned solution will have the most resistance, as the Systems ASIC team will have to learn how to interact with an entirely different language. This will require the most time to relearn out of all of the solutions.

6.3 Adaptability

The MEX-owned solution will require some time-consuming and sophisticated development work to execute, as preprocessor programming is especially difficult to implement and debug. These variable macros need to be able to produce variable declarations, getter/setter functions, and accompanying enumerations as well as be used for creating the opcodes, with lots of little steps. However, due to the nature of this, once it is setup for one function, all other functions can use the same boilerplate template. Additionally, there needs to be strict verification measures in place to make sure the parameters and states are updated in the MEX library anytime they're modified in MATLAB, however this again requires only a boilerplate template. An example of this functionality can be found in Appendix C.

The MATLAB-owned solution is once again the simplest, providing little work to convert the constants into a common set of variables for both MATLAB and the MEX library. Developers will be able to intuitively modify constants and are familiar with the dynamic of intrafunction communication if they need to modify existing MEX libraries.

For C++-owned, much time is needed to adapt this, as it's a roadblock to other development if not done all at once. This needs to be done in co-operation between the Firmware and Systems ASIC teams. Each part of the MATLAB code needs to be verified when converted to C++ in order to maintain accuracy. Additionally, a debug mode will need to be created to allow for the retrieval of intermediate values for plotting purposes.

6.4 Final Comparison

The final scores out of 10 for each category are relative to the solution of doing nothing, which has also been included in Table 6-1 as an option.

Table 6-1. Comparison between proposed solutions using the defined criteria.

Solution	Performance	Usability	Adaptability	Total
Unchanged	0	10	10	20
MEX-Owned	6	8	8	22
MATLAB-Owned	2	10	9	21
C++-Owned	10	4	2	16

As you can see, the Unchanged solution had fixed values of 0, 10, and 10. Since it is the current version, it is the most usable and adaptable as no additional work needs to be done, but it is the worst on performance. For performance specifically, the scale of 0 to 10 was based off of the performance increase it had versus the unchanged in percentage divided by 5. The other two criteria had numbers assigned to them in a qualitative manner based on the reasoning given.

Conclusions

From the analysis in the report body, it was concluded that there were three important criteria to compare, which were Performance, Usability, and Adaptability. Performance was important due to the desire to run many different variations of the same simulation in a reasonable timespan. Usability was important as developers who may have never touched the previous versions of the simulations will have to work with and maintain this code. Adaptability was important as there is a desire to transition into this new simulation version seemlessly and without stopping other code production.

There were four solutions compared. First, the Unchanged version (Unchanged), next the MEX-Owned, MATLAB-Maintained Interfunction Communication version (MEX-Owned), afterwards the MATLAB-Owned, MATLAB-Maintained Intrafunction Communication (MATLAB-Owned), and finally the C++-Owned, C++-Maintained Intrafunction Communication (C++-Owned). The best performing was C++-Owned, the most usable was MATLAB-Owned, along with Unchanged which was also the most adaptable. However, they each scored 16/30, 21/30, and 20/30 respectively, with the MEX-Owned scoring the highest of 22/30 despite not being the highest performer in any single category.

Recommendations

Based on the analysis and conclusions in this report, it is recommended that the MEX-Owned, MATLAB-Maintained Interfunction Communication version of the simulation be implemented. This would involve one developer, who does not have to have previous experience with this project, going through the code and initially merging some of the biggest MEX libraries into one unified library. From there, they should implement a basic opcode system which allows them to distinguish between MEX functions both in MATLAB and C++. Afterwards, they can set up a testing framework in which the old MEX libraries are compared with the new MEX library upon each call, specifically the parameters, state, and outputs. This will make converting them from being input and output arguments with each call to being set once and updated/called when necessary easier to test. The developer should implement a couple of these setters and getters in order to test and perfect the format.

Then, they need to implement the preprocessor programming practices. Convert the opcode system to generate automatically by defining the parameters, states, and outputs of each function in macros. This will allow them to develop an automated post-build step of compilation which can create a file for an enumeration class for MATLAB to use in its communication. After that, they can convert the setters and getters to be generated automatically by macros. This will allow the conversion of variables from inputs and outputs for all remaining MEX functions within the library. The developer can then add any additional optimizations that they would like, keeping in mind to run the testing functions to confirm the logic and output of the functions is still the same.

After this, it should be introduced to current developers of this logic on the Firmware Engineering team. They can try to merge a smaller MEX library into this unified library, encountering any misunderstandings with the framework while being able to communicate with the developer for clarification. This will help with both adaptability and usability of this new simulation version. From there, the goal is that the Firmware Engineering team will slowly convert over the remaining MEX libraries at a time most suitable and convenient for them, with the end goal that there will only be one remaining MEX library and that every C++-Owned variable can be communicated between MEX functions.

Glossary

C: A programming language designed to be as low-level as possible while still maintaining readability. Unlike C++, it does not have objects, or many other complex features. All C code should run properly using C++.

C++: A programming language designed as a balance between performance and usability. Notable features of the language include its ability to define custom objects to use complex data structures and the mandate that code must be compiled to run.

functional programming: Functional programming is the technique of programming that does not allow for data manipulation, rather it operates on function evaluation. Variables are not stored, rather the outputs of these functions are based entirely on the inputs.

getter: A colloquial term used to describe a function which sets a specific variable or value.

MATLAB: A programming language mainly designed for numerical computation purposes, such as data science and signal processing. Notable features of the language include its flexible variable management and file execution and its proprietary nature allowing for limited community-expansion.

MEX: The types of files MATLAB uses in order to communicate with C++ code. This involves compiling the C++ code into a dynamic linked library that is then loaded and utilized by MATLAB.

object-oriented programming: Object-oriented programming is the technique of programming that operates on data evaluation and manipulation, this data stored in objects, which are assisted by functions. This is the normalized programming technique to most readers, it is what C++ and MATLAB both operate in most use cases, and thus functional programming will be treated with more explanation.

preprocessor: The part of a compiler which does actions before the compiler assesses the code and creates a compiled version.

run-time polymorphism: Run-time polymorphism is a process in which a call to an overridden method is resolved at runtime rather than at compile-time.

setter: A colloquial term used to describe a function which sets a specific variable or value.

Verilog: A hardware description language designed to model physical hardware.

WKRPT: Work-term report; the acronym used by the University of Waterloo Undergraduate Calendar.

References

[1] I. Corporation, *Telstra Deploys Major Upgrade to Network Services in Asia Pacific Using Infinera Technologies*, Jan. 2019. [Online]. Available: https://www.infinera.com/telstra-deploys-major-upgrade-to-network-services-asia-pacific-using-infinera-technologies/.

[2] MATLAB, *Introducing MEX Files*, Mar. 2019. [Online]. Available: https://www.mathworks.com/help/matlab/matlab_external/introducing-mex-files.html.

[3] T. Andrews, ''Computation time comparison between Matlab and C++ using launch windows,'' Jun. 2012. [Online]. Available: https://digitalcommons.calpoly.edu/aerosp/78/.

[4] *Boost C++ Libraries*. [Online]. Available: https://www.boost.org/.

[5] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond*. Pearson Education, Inc., 2005, pp. 281–306.

Appendix A MEX Example

Below are some of the files would be included in a project for a MEX library, but for the purposes of demonstrating the MATLAB-MEX communication capabilities, these two should be sufficient in showcasing that.

```
restoredefaultpath
warning('off','MATLAB:structOnObject') % Include this to prevent warning spam

ImportantClass.paramA = -1;
ImportantClass.paramB = -2;
inputA = 1;
inputB = 2;
stateA = 10;
state0 = 20;

mexMexDemo(struct(MexOpcodes.TestFunction_SetParams), ImportantClass.paramA,
    ImportantClass.paramB);
mexMexDemo(struct(MexOpcodes.TestFunction_SetStates), stateA);
mexMexDemo(struct(MexOpcodes.OtherFunction_SetStates_State0), state0);

for ndx = 1:99
    mexMexDemo(struct(MexOpcodes.TestFunction_Main), inputA, inputB);
    inputB = inputB - inputA;
    inputA = inputA + 2*inputB;
end

[outputZ] = mexMexDemo(struct(MexOpcodes.TestFunction_GetOutputs));
[stateA] = mexMexDemo(struct(MexOpcodes.TestFunction_GetStates));
state0 = mexMexDemo(struct(MexOpcodes.OtherFunction_GetStates_State0));

```

Figure A-1. goRunMexDemo.m; this would be the MATLAB-facing code.

```cpp
#include <mex.h>
#include "mexOpcode.h"
#include "mexTestFunction.h"
#include "topOtherFunction.h"

namespace {
    // You would normally use Opcode and not tOpcode
    enum tOpcode {
        Help = 0,
        TestFunction = 1,
        OtherFunction = 2
    };
}

// Don't forget to clear mex on MATLAB when compilations fail due to locked mexw64 files
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) { // standard mex main signature
    try {
        mexOpcode mo;
        mxToStruct(mo, prhs[0], 0);
        const tOpcode opcode_in = static_cast<tOpcode>(mo.Block);
        //The reason that we have to convert the opcode to a struct when we pass it in the mex is so we can deconstruct it here

        switch (opcode_in) {
        case tOpcode::Help:
            printf("Read the comments for help.\n");
            break;
        case tOpcode::TestFunction:
            mexTestFunction(nlhs, plhs, nrhs - 1, &(prhs[1]), mo); //standard mex block call signature
            break;
        case tOpcode::OtherFunction:
            if (nlhs == 1) { //This is poor practice, it's so I don't have to define two extra files in this example
                plhs[0] = mxCreateDoubleMatrix(1, 1, mxREAL);
                mxGetPr(plhs[0])[0] = (double)(topTestFunction_GetStates_State0());
            }
            else {
                topTestFunction_SetStates_State0(((double*)mxGetPr(prhs[0]))[0]);
            }
            break;
        default:
            mexErrMsgIdAndTxt("Mex:BADARG", "The first input parameter must be a valid opcode. It was %d\n", opcode_in);
            break;
        }
    }
    catch (std::exception const &e)
    {
        mexErrMsgIdAndTxt("Mex:UNKNOWN", e.what());
    }
    catch (...)
    {
        {
            mexErrMsgIdAndTxt("Mex:UNKNOWN", "Unknown exception");
        }
    }
}

```

Figure A-2. mexMexDemo.cpp; this would be the MEX-facing code.

Appendix B Preprocessing Example

Below is the code for a normal C++ function, as well as the file containing preprocessor code that actually creates this. This file can be run on its own in Visual Studio 2017.

```
struct structureA    {  int nameA; double nameB1; std::array<int, 4> nameB2; };;

int main() {
    structureA tmp;     tmp.nameA = { 1 }; tmp.nameB1 = { (double)2.5 }; tmp.nameB2 =
     { (int)2.5 }; int sum = 0  +1 +1 +1;
    tmp.nameB2[2] = 5;
    std::string name = "structureA"   ;
    printf("Stats for %s\nNumber of Elements: %d\nTotal Value of Each Element: %d, %f
    , %d\n",
        name, sum, tmp.nameA, tmp.nameB1, std::accumulate(std::begin(tmp.nameB2), std
    ::end(tmp.nameB2), 0, std::plus<int>()));
    printf("yay %d\n", 5);;
    std::this_thread::sleep_for(std::chrono::milliseconds(10000));
    return 0;
}

```

Figure B-1. The preprocessor of the following file outputs this to the compiler.

```cpp
#include <array>
#include <numeric>
#include <string>
#include <chrono>
#include <thread>

/*Let's define the structure first. As you can see, there are a few different inputs. TYPE_MACRO will be used when declaring the name of the structure and when instantiating it. FIELD_MACRO will be used when manipulating each field of the structure. Separating the field macros into different types, A and B, will be useful.*/
#define STRUCTURE_A(TYPE_MACRO, FIELD_MACRO_A, FIELD_MACRO_B) \
    TYPE_MACRO(structureA) \
    FIELD_MACRO_A(structureA, int, nameA) /*comment*/\
    FIELD_MACRO_B(structureA, double, nameB1) \
    FIELD_MACRO_B(structureA, int, nameB2, 4)

//Util functions, some output nothing, some require another function call in order to properly function
#define UTIL_ECHO(...) __VA_ARGS__
#define UTIL_DO_NOTHING(...)
#define UTIL_CONC_(A, B) A##B
#define UTIL_CONC(A, B) UTIL_CONC_(A, B)
#define UTIL_TO_STRING_(x) #x
#define UTIL_TO_STRING(x) UTIL_TO_STRING_(x)

/*Function I helpers - notice how there are different functions for each macro with different inputs. The field macro b accepts both doubles and ints as types, so by casting the double automatically, we can account for assigning it to different types of scalars (assuming c++ didn't automatically convert)*/
#define FUNCTION_I_TYPE(m_struct) m_struct tmp;
#define FUNCTION_I_FIELD_A(m_struct, m_type, m_name) UTIL_CONC(tmp., m_name) = { 1 };
#define FUNCTION_I_FIELD_B(m_struct, m_type, m_name, ...) UTIL_CONC(tmp., m_name) = { (m_type)2.5 };
#define FUNCTION_I_GET_COUNT(...) +1

/*Function I definition - This really shows why the structure was defined in "wrappers" previously. A structure contains these wrappers, so when it is passed in a function, the function can call other functions on these wrappers! Thus the structure type is only outputed once while each field outputs accordingly, based on what type of field it is, and you can chain these together for a total count!*/
#define FUNCTION_I(STRUCTURE) \
STRUCTURE(FUNCTION_I_TYPE, UTIL_DO_NOTHING, UTIL_DO_NOTHING) \
STRUCTURE(UTIL_DO_NOTHING, FUNCTION_I_FIELD_A, FUNCTION_I_FIELD_B) \
int sum = 0 STRUCTURE(UTIL_DO_NOTHING, FUNCTION_I_GET_COUNT, FUNCTION_I_GET_COUNT);

//Function II definition - Notice here that the fields have different declarations based on
#define FUNCTION_II_TYPE(m_struct) m_struct
#define FUNCTION_II_FIELD_3(m_struct, m_type, m_name) m_type m_name;
#define FUNCTION_II_FIELD_4(m_struct, m_type, m_name, m_size) std::array<m_type, m_size> m_name;

/*Due to allowing different number of arguments for C and standard arrays, this hackjob exists. It allows you to make a chooser which will choose an appropriate function based on the number of inputs. The extra field we allow optionally is number of columns, as the default is 1*/

```

Figure B-2. mexMacroDemo.cpp; this would be the MEX-facing code. (1/2)

```
#define GET_5TH_ARG(arg1, arg2, arg3, arg4, arg5, ...) arg5
#define FUNC_RECOMPOSER_5(argsWithParentheses) GET_5TH_ARG argsWithParentheses
#define NO_ARG_EXPANDER_5() ,,,,UTIL_DO_NOTHING

//This is more of the hackjob
#define FUNCTION_II_FIELD_CHOOSE_(...) FUNC_RECOMPOSER_5((__VA_ARGS__,
    FUNCTION_II_FIELD_4, FUNCTION_II_FIELD_3, \
    UTIL_DO_NOTHING, UTIL_DO_NOTHING))
#define FUNCTION_II_FIELD_CHOOSE(...) FUNCTION_II_FIELD_CHOOSE_(NO_ARG_EXPANDER_5
    __VA_ARGS__ ())
#define FUNCTION_II_FIELD(...) FUNCTION_II_FIELD_CHOOSE(__VA_ARGS__)(__VA_ARGS__)

//Finally, the definition. It doesn't feature anything particularly new, remember
    backslashes for multi-line
#define FUNCTION_II(STRUCTURE) \
struct STRUCTURE(FUNCTION_II_TYPE, UTIL_DO_NOTHING, UTIL_DO_NOTHING) { \
    STRUCTURE(UTIL_DO_NOTHING, FUNCTION_II_FIELD, FUNCTION_II_FIELD) \
};

/*This yay/nay bit is a showcase of function name concatenation with variables.
    Change STRUCTURE_B's variable from YAY to NAY to see the outcome.*/
#define PRINT_NAY(num) printf("nay %d\n", num);
#define PRINT_YAY(num) printf("yay %d\n", num);

#define PRINT_YAY_OR_NAY__(AY, num) UTIL_ECHO(PRINT_ ## AY)(num)
#define PRINT_YAY_OR_NAY_(AY, num) PRINT_YAY_OR_NAY__(AY, num)

#define PRINT_YAY_OR_NAY(AY, num) \
PRINT_YAY_OR_NAY_(UTIL_ECHO(AY), num)

#define FUNCTION_III(STRUCT) STRUCT(PRINT_YAY_OR_NAY)

#define STRUCTURE_B(MACRO) \
MACRO(YAY, 5)

/*If you would like to view the preprocessor output, please do the following in
    Visual Studio. Project Properties -> C/C++ -> Command Line -> Additional Options
    -> Add /E. Then when it outputs, scroll to the bottom of the output to see what
    the macros expand to when used. Note that you need to remove the option in order
    to have a successful build due to .obj file issues*/

/*Note that the additional semi-colons, while not visually the best, do not affect
    the correctness of the code and are in-fact not even visible as duplicates as the
     output is not normally visible to the developer.*/
FUNCTION_II(STRUCTURE_A);

int main() {
    FUNCTION_I(STRUCTURE_A)
    tmp.nameB2[2] = 5;
    std::string name = STRUCTURE_A(UTIL_TO_STRING, UTIL_DO_NOTHING, UTIL_DO_NOTHING);
    printf("Stats for %s\nNumber of Elements: %d\nTotal Value of Each Element: %d, %f
    , %d\n",
        name, sum, tmp.nameA, tmp.nameB1, std::accumulate(std::begin(tmp.nameB2), std
    ::end(tmp.nameB2), 0, std::plus<int>()));
    FUNCTION_III(STRUCTURE_B);
    std::this_thread::sleep_for(std::chrono::milliseconds(10000));
    return 0;
}
```

Figure B-3. mexMacroDemo.cpp; this would be the MEX-facing code. (2/2)

Appendix C Testing Example

Below are some of the files would be included in a project for a MEX library, but for the purposes of demonstrating the MEX-version testing capabilities, these two should be sufficient in showcasing that.

```
restoredefaultpath
warning('off','MATLAB:structOnObject') % Include this to prevent warning spam

import matlab.unittest.constraints.IsEqualTo % Fun stuff for testing
testCase = matlab.unittest.TestCase.forInteractiveUse;
testing = true;

ImportantClass.paramA = -1; % Would be in ImportantClass system class
ImportantClass.paramB = -2; % Other classes may contain important variables

inputA = 1;  % With all of these, keep in mind that much of the infrastructure
inputB = 2;  % is built around the ability to compare the old mex with new mex
stateA = 10; % Thus, many of these variables don't need to be maintained or
stateO = 20; % kept up to date on the MATLAB side in an ideal situation

% Setting the params and states would happen before the main loop
mexTestingDemo(struct(MexOpcodes.TestFunction_SetParams), ImportantClass.paramA,
    ImportantClass.paramB);
mexTestingDemo(struct(MexOpcodes.TestFunction_SetStates), stateA);
mexTestingDemo(struct(MexOpcodes.OtherFunction_SetStates_StateO), stateO);

for ndx = 3:6
    activeAssertionRange = ndx < 5; % If a function is called lots, limit the tests

    % This function sets up the testing and passes back a verification function
    handle
    TestFunction_Verify = TestFunction_Setup(testing, testing && activeAssertionRange
    , testCase, mexTestingDemo, ImportantClass, {inputA, inputB}, {stateA, stateO});

    % Without this test setup, this would be the only line we need
    mexTestingDemo(struct(MexOpcodes.TestFunction_Main), inputA, inputB);

    % Call anything part of the old mex's output in order to compare
    [outputZ] = mexTestingDemo(struct(MexOpcodes.TestFunction_GetOutputs));
    [stateA] = mexTestingDemo(struct(MexOpcodes.TestFunction_GetStates));
    stateO = mexTestingDemo(struct(MexOpcodes.OtherFunction_GetStates_StateO));

    % Verify that the params, states, and outputs match the old mex and the MATLAB
    versions
    TestFunction_Verify({stateA, stateO, outputZ});
end

```

Figure C-1. goRunTestingDemo.m; this would be the MATLAB-facing code.

```
function TestFunction_Verify = TestFunction_Setup(testing, assertsOn, testCase,
    ImportantClass, tf_save_inputs_current, tf_save_states_current)

    if testing % If we're not testing, the only penalty here is an if statement check
        TestFunction_Verify = @(x) TestFunction_Verify_Test(x); % Return the function
     handle

        % Retrieve the parameters and states in order to compare with the MATLAB
    versions
        % The point is that we've tracked where they have changed in the code
        % and updated the mex accordingly so there aren't any discrepancies
        [paramA, paramB] = mexTestingDemo(struct(MexOpcodes.TestFunction_GetParams));
        [stateA] = mexTestingDemo(struct(MexOpcodes.TestFunction_GetStates));
        state0 = mexTestingDemo(struct(MexOpcodes.OtherFunction_GetStates_State0));
    else
        TestFunction_Verify = @(x) TestFunction_Verify_Null(x);
    end

    function TestFunction_Verify_Null(~)
    end

    function TestFunction_Verify_Test(tf_output_new)
        import matlab.unittest.constraints.IsEqualTo

        if assertsOn % Limit the amount of assert spam but still run the old function
            verifyParams(testCase, ImportantClass, {paramA, paramB}); % All of the
    params should be stored in the ImportantClass or similar
            assertThat(testCase, CellArrayGeneralize({stateA, state0}), IsEqualTo(
    CellArrayGeneralize(tf_save_states_current)), 'tf_save_states')
        end

        [inputA, inputB] = feval(@(x) x{:}, tf_save_inputs_current); %#ok<ASGLU>
    Unpack the inputs so they can be used

        [stateA, state0, outputZ] = TestFunction_Old(paramA, paramB, stateA, state0,
    inputA, inputB);

        tf_output_old = {stateA, state0, outputZ};

        if assertsOn % If you don't want to stop the whole process on a failure, use
    verifyThat
            assertThat(testCase, tf_output_new, IsEqualTo(tf_output_old), 'tf_output'
    )
        end
    end
end

function verifyParams(testCase, ImportantClass, CPPParams)
    import matlab.unittest.constraints.IsEqualTo
    paramA = ImportantClass.paramA;
    paramB = ImportantClass.paramB;
    % The purpose of the generalize function is that we don't want to
    % compare shapes of inputs to old mex, just data. Half the time on a
    % first run it just changes the shape, and there are weird mechanics
    % where complex 0s are treated differently after passed through functions
    ICParms = CellArrayGeneralize({paramA, paramB});
    CPPParams = CellArrayGeneralize(CPPParams);
    assertThat(testCase, CPPParams, IsEqualTo(ICParms), 'tf_save_params')
end

```

Figure C-2. TestFunction_Setup.m; this would be the MATLAB-facing code.