

2007

Hardware garbage collection for embedded systems with integration with jrate java environment

Brett Matthew McNerney
University of Dayton

Follow this and additional works at: https://ecommons.udayton.edu/graduate_theses

Recommended Citation

McNerney, Brett Matthew, "Hardware garbage collection for embedded systems with integration with jrate java environment" (2007). *Graduate Theses and Dissertations*. 4353.
https://ecommons.udayton.edu/graduate_theses/4353

This Thesis is brought to you for free and open access by the Theses and Dissertations at eCommons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of eCommons. For more information, please contact mschlange1@udayton.edu, ecommons@udayton.edu.

**HARDWARE GARBAGE COLLECTION FOR
EMBEDDED SYSTEMS WITH INTEGRATION WITH
JRATE JAVA ENVIRONMENT**

Thesis

Submitted to

The School of Engineering of the

UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for

The Degree

Master of Science in Electrical Engineering

By

Brett Matthew McNerney

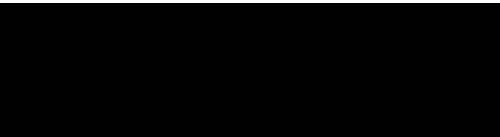
UNIVERSITY OF DAYTON

Dayton, Ohio, USA

May 2007

HARDWARE GARBAGE COLLECTION FOR EMBEDDED SYSTEMS WITH INTEGRATION WITH JRATE JAVA ENVIRONMENT

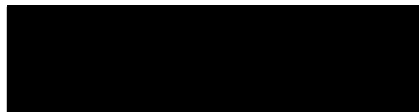
APPROVED BY:



John G. Weber, Ph.D
Advisory Committee Chairman
Professor, Electrical and
Computer Engineering



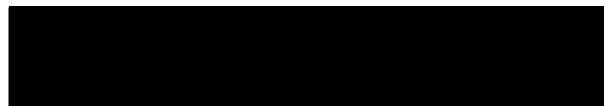
Frank A. Scarpino, Ph.D
Committee Member
Professor, Electrical and
Computer Engineering



John S. Loomis, Ph.D
Committee Member
Associate Professor, Electrical and
Computer Engineering



Donald L. Moon, Ph.D
Associate Dean
Graduate Engineering Programs and Research
School of Engineering



Joseph E. Saliba, Ph.D. P.E.
Dean, School of Engineering

Abstract

TITLE: Hardware Garbage Collection for Embedded Systems with Integration in the JRate Java Environment

NAME: McNerney, Brett, Matthew
University of Dayton

Advisor: Dr. John G. Weber

JAVA's success in the internet application area has sparked an interest in applying the language in other areas. One of these is real-time systems. While JAVA has many attractive features, there are several areas of concern. Standard JAVA executes interpretively severely limiting its use in real-time applications. Compiled versions of the language have been developed which overcome this difficulty.

A major JAVA feature is an automatic technique to free memory that is no longer needed. This technique is called "garbage collection. In Java, objects are allocated dynamically and are considered "live" as long as they are in use. Once an object is no longer needed, it is considered "dead" and the Java system will automatically deallocate it. This deallocation, called "garbage collection," identifies and removes dead objects stored in memory, thereby freeing the space for new objects. This process can be very time consuming and does not lend itself to real-time systems. With the use of dedicated hardware to decrease execution time of the garbage collector, JAVA can be made to work in real-time applications.

Over the years, hardware has evolved and memory access has become faster and faster. However, memory management has been left to the software programmer. With the implementation of larger memory sizes and with program size growing, this has

become more and more challenging. The consequences of poor memory management are programs that waste processor power, memory leaks, data fragmentation, and complete system failure. This most often occurs in situations where there are many programmers working on a single software program, since each programmer may implement memory usage in different ways that can conflict with one another.

The use of current automatic memory management algorithms solves some of these issues. However, they are often time intensive and consume significant processor time and power. In a real-time environment this can become a major issue by causing critical data to be missed and critical processes to not execute when required.

The design proposed in this paper uses modern hardware with classical real-time garbage collection algorithms. Making a system that is both easy to implement and able to be integrated into a real-time environment. Also, the system will allow a programmer to focus on making the software program as efficient as possible without having to focus on how to manually manage memory. The smart memory module (SMM) handles the memory management in dedicated hardware freeing up the processor to focus the real-time tasks. This is done by connecting the memory directly to the SMM hardware and off loading the software intensive memory management routines into dedicated hardware. The SMM gives a programmer a fast and easy way to access memory and allows garbage collection in real-time systems. Such systems would normally not contain memory management due to the overhead associated with it when implemented using software routines.

In this paper, current automatic memory management methods are discussed in order to provide a background of current Garbage Collector algorithms (Chapter 1). Then the hybrid smart memory module is discussed and how the classical garbage collection designs implemented the final design (Chapter 2). The hardware interface and then the actual hardware collection system are shown (Chapter 3 and 4). Next, the software interface functions to access the smart memory module laid out (Chapter 5). This is then followed by the results showing the working functionality and performance of the smart memory module (Chapter 6). Lastly, the paper finishes with future work to be done and conclusions (Chapter 7).

Acknowledgements

I would like to thank everyone who supported me and stuck by me during the numerous delays in completing this paper. But I would especially like to thank the following people:

- **Dr. John Weber, PhD:**

I would like to thank you for the support during all the ups and down of grad school. As well as the support during the rough times we had during the ARTEC program. Without the backing and support during the research done not only for this thesis but for the ARTEC program I don't think I would have made all the way through

- **Dr. Frank Scarpino, PhD:**

I would like to thank you for being on my committee and for all the valuable life lessons taught during the past year working at the Wright Patterson Air Force Base. As well as the vast amount of skills you have taught everyone on the team.

- **Dr. John Loomis**

I would like to thank you for being on my committee and for the knowledge obtained during various classes.

- **Mike Mills:**

I would like to thank you for the opportunity, financial support, and the patience of the ups and downs associated with the project.

- **Kerry Hill:**

I would like to thank you for the support during the ARTEC program during the ups and downs. For the financial support on current and future research projects allowing me to complete my education and continue on towards a PhD.

- **Ken and Patricia McNerney:**

I would like to thank both of you for all the support throughout my life and being great parents. For the motivation and financial support to continue on with my education. I would especially like to thank my mother for her support and motivation during the long days and nights when there seemed to be too much work to get done. I would especially like to thank my father for his support in driving me to succeed and making me the best I can be.

- **David McNerney**

I would like to thank my brother for always being there no matter what the situation was. For helping me out whenever possible and being a role model I have always looked up to.

- **Samantha St. Louis:**

Thank you for your patience over the last 3 years and standing by my side to make sure I succeeded. For the drive and push to make me do my best in both school and sports. And for being there when I needed you the most.

Table of Contents

ABSTRACT	III
ACKNOWLEDGEMENTS.....	V
TABLE OF CONTENTS.....	VII
LIST OF ILLUSTRATIONS	IX
LIST OF TABLES	XI
CHAPTER 1	1
INTRODUCTION.....	1
1.1 SIGNIFICANT TYPES OF COLLECTORS	4
1.1.1 <i>Mark-Sweep Collector</i>	4
1.1.2 <i>Reference Counting</i>	6
1.1.3 <i>Copying Collector</i>	9
1.2 SUMMARY	13
CHAPTER 2	14
HARDWARE, DESIGN, AND TESTING APPROACH OVERVIEW	14
2.1 VIRTEX 4 FPGA AND XILINX ML403 DEVELOPMENT BOARD	15
2.2 HYBRID APPROACH	16
2.3 TESTING ENVIRONMENT	20
CHAPTER 3	22
INTERFACE DESIGN.....	22
3.1 IMPLEMENTED FUNCTIONS	22
3.1.1 <i>New Structure Definition</i>	22
3.1.2 <i>New Object</i>	24
3.1.3 <i>New Array</i>	26
3.1.4 <i>Clone</i>	28
3.1.5 <i>Put Object Data</i>	30
3.1.6 <i>Get Object Data</i>	32
3.1.7 <i>Bump Reference Count</i>	34
3.1.8 <i>Decrement Reference Count</i>	35
3.1.9 <i>Get Reference Count</i>	37
3.1.10 <i>Reset</i>	38
3.2 IMPLEMENTATION	39
3.3 SUMMARY	40
CHAPTER 4	42
GARBAGE COLLECTOR HARDWARE DESIGN.....	42
4.1 GARBAGE COLLECTOR	42
4.2 HANDLE TABLE UPDATING	46
4.3 INTERFACE/COLLECTOR HANDSHAKE	47
4.4 SUMMARY	48
CHAPTER 5	50

LINUX TO HARDWARE JAVA INTERFACE	50
5.1 LINUX CUSTOM DRIVER	50
5.1.1 <i>ioremap Linux Function</i>	51
5.1.2 <i>iounmap Linux Function</i>	52
5.1.3 <i>in_le32 Linux Function</i>	53
5.1.4 <i>out_le32 Linux Function</i>	53
5.1.5 <i>New Structure Definition</i>	54
5.1.7 <i>New Object</i>	55
5.1.8 <i>New Array</i>	55
5.1.9 <i>Clone</i>	56
5.1.10 <i>Put Object Data</i>	57
5.1.11 <i>Get Object Data</i>	58
5.1.12 <i>Bump and Decrement Reference Count</i>	59
5.1.13 <i>Get Reference Count</i>	61
5.1.14 <i>Get Garbage Collection Count</i>	61
5.1.15 <i>Reset</i>	62
5.2 CNI INTERFACE CODE	62
5.3 SUMMARY	63
CHAPTER 6	64
IMPLEMENTATION RESULTS	64
6.1 TEST SUITE DESCRIPTION AND RESULTS	64
6.1.1 <i>Test Case 1 – Interface and Initial Garbage Collector Testing</i>	64
6.1.2 <i>Test Case 2 – Pointer Mask and Data Offset Testing</i>	66
6.1.3 <i>Test Case 3 – Multi-Thread Testing</i>	68
6.1.4 <i>Test Case 4 – Basic Garbage Collector and Memory Bounds Testing</i>	69
6.1.5 <i>Test Case 5 – Null Pointer and Out of Range Pointer Testing</i>	70
6.1.6 <i>Test Case 6 – Timing Statistics</i>	72
6.1.7 <i>Test Case 7 – Error Condition Testing</i>	74
6.1.8 <i>Test Case 8 – More Garbage Collector Testing</i>	75
6.1.9 <i>Test Case 9 – FACET Example</i>	77
6.1.10 <i>Test Case 10 – Extended Interface Testing</i>	78
6.1.11 <i>Test Case 11 – Extended System Testing</i>	78
6.1.12 <i>Test Case 12 – Extended Pointer Reference Testing</i>	79
6.2 HARDWARE VS. SOFTWARE COMPARISON	81
6.3 FULL DEMO ENVIRONMENT RESULTS	84
CHAPTER 7	87
FUTURE IMPROVEMENTS AND CONCLUSIONS	87
7.1 FUTURE IMPROVEMENTS	87
7.1.1 <i>Hardware Modifications</i>	87
7.1.2 <i>Embedded Kernel Module Driver</i>	87
7.1.3 <i>Garbage Collector Hardware Plug-in Module</i>	88
7.2 CONCLUSIONS	90
APPENDICES	91
APPENDIX A: SMM HARDWARE VHDL CODE	91
APPENDIX B: JAVA TEST SUITE CODE	91
REFERENCES	92

LIST OF ILLUSTRATIONS

Figure 1.1 Heap Memory Before and After Garbage Collection	3
Figure 1.2 Object Pointer Structures	6
Figure 1.3 Potential Hazards with Reference Counting	8
Figure 1.4 Cheney Copy Collector Example.....	10
Figure 1.5 Baker Copy Collector Example	12
Figure 2.1 Xilinx ML403 Development Board.....	16
Figure 2.2 SMM System Block Diagram.....	18
Figure 2.3 Complete System Bus Connections	19
Figure 2.4 SMM Internal Module Connections.....	20
Figure 3.1: Simple Block Diagram of New Structure Definition Function	23
Figure 3.2: New Structure Definition State Diagram.....	23
Figure 3.3: Simple Block Diagram of New Object Function	24
Figure 3.4: New Object State Diagram	25
Figure 3.5: Simple Block Diagram of New Array Function	26
Figure 3.6: New Array State Diagram	27
Figure 3.7: Simple Block Diagram of Clone Function	28
Figure 3.8: Clone State Diagram	29
Figure 3.9: Step 1: Loading Data to Internal PowerPC Registers.....	30
Figure 3.10: Step 2: Sending Data to the SMM from Internal PowerPC Registers	31
Figure 3.11: Put Object Data State Diagram.....	31
Figure 3.12: Step 1: Returned Data from the SMM to Internal PowerPC Registers.....	32
Figure 3.13: Step 2: Sending Data from Internal PowerPC Registers to SDRAM.....	33
Figure 3.14: Get Object Data State Diagram.....	33
Figure 3.15: Incrementing the Reference Count Stored in Internal SMM Memory.....	34
Figure 3.16: Bump Reference Count State Diagram	35
Figure 3.17: Decrementing the Reference Count Stored in Internal SMM Memory.....	36
Figure 3.18: Decrement Reference Count State Diagram	36
Figure 3.19: Returning of Reference Count.....	37
Figure 3.20: Get Reference Count State Diagram.....	38
Figure 3.21: Reset State Diagram	38
Figure 3.22: PLB IPIF Block Diagram [9]	39
Figure 3.23: Hardware System Block Diagram.....	40
Figure 4.1 Basic Copy Collection Scheme	43

Figure 4.2 ToSpace Free Memory Calculation.....	44
Figure 4.3 Data Object and Array Storage Structure.....	45
Figure 4.4 Comparing the to-copy Memory to the is-copied Memory	46
Figure 4.5 Updating Free Objects and Handle Table Memory.....	47
Figure 4.6 Handshake Basic System Design	48
Figure 5.1 Linux Custom Driver Interface to SMM System	51
Figure 5.2 ioremap Function Code.....	52
Figure 5.3 iounmap Function Code	53
Figure 5.4 in_le32 Function Code.....	53
Figure 5.5 out_le32 Function Code.....	54
Figure 5.6 New Structure Definition Function Code	54
Figure 5.7 New Object Function Code	55
Figure 5.8 New Array Function Code	56
Figure 5.9 Clone Function Code	57
Figure 5.10 Put Object Data Function Code.....	58
Figure 5.11 Get Object Data Function Code	59
Figure 5.12 Bump Reference Count Function Code	60
Figure 5.13 Decrement Reference Count Function Code	60
Figure 5.14 Get Reference Count Function Code.....	61
Figure 5.15 Linux Custom Driver with CNI Interface to SMM System	63
Figure 6.1: Screen Capture for Test Case 1 Outputs.....	66
Figure 6.2: Screen Capture for Test Case 2 Outputs.....	68
Figure 6.3: Screen Capture for Test Case 3 Outputs.....	69
Figure 6.4: Screen Capture for Test Case 4 Outputs.....	70
Figure 6.5: Screen Capture for Test Case 5 Outputs.....	72
Figure 6.6: Screen Capture for Test Case 6 Outputs (Time Shown in Seconds)	73
Figure 6.7: Screen Capture for Test Case 7 Outputs.....	75
Figure 6.8: Screen Capture for Test Case 8 Outputs for Hardware.....	76
Figure 6.9: Screen Capture for Test Case 8 Outputs for Software.....	76
Figure 6.10: Screen Capture for Test Case 9 Outputs.....	77
Figure 6.11: Screen Capture for Test Case 10 Outputs	78
Figure 6.12: Screen Capture for Test Case 11 Outputs	79
Figure 6.13: Screen Capture for Test Case 12 Outputs	81
Figure 6.14: Comparing Execution Times Between Hardware and Software.....	82
Figure 6.15 Prototype Software Collector Timing Results for Each Rate [7].....	84
Figure 6.16 FPGA Hardware Collector Timing Results for Each Rate [7].....	85
Figure 7.1 Proposed SMM Plug-In Module.....	89

LIST OF TABLES

Table 6.7: List of Supported Error Codes	74
Table 6.14: Timing Comparison Between Hardware and Software.....	82

Chapter 1

Introduction

Garbage Collection is a technique for automatically freeing allocated memory. It is considered a form of automatic memory management. Memory management defines how the programmer and system allocate and deallocate memory words [4]. When memory is allocated, it is placing data into a memory location. Deallocation of memory is the removing of data from memory, thereby freeing the space for reuse. There are two types of memory allocation schemes: static and dynamic. Various constructs are used to manage memory based on the type of the allocation being used. Some of the common ones are stack, static, and heap memory types [5]. Deciding which method is best suited for a particular program is left to the programmer.

When all the memory required by the executing program is completely allocated prior to execution, the method is called static. This is effective when speed is a concern, since the memory must have sufficient space to hold all of the objects created and, hence, there is no need to deallocate and reallocate memory. The object is placed in the already allocated memory space saving a step and time. The issue here is that a large amount of memory can be wasted if too large of a static memory space is allocated for the program. The other issue is that the system will fail if not enough memory is allocated. Therefore, static allocation of memory is useful when speed is a major concern, but it needs to be handled carefully.

When memory is allocated as needed, the method is termed dynamic. This allows the amount of memory space a program needs to grow and shrink. This results in a better utilization of memory, since objects that are no longer needed are removed from memory and the memory space is freed for other use. Dynamic memory allocation is slower than static memory allocation, since allocation must occur whenever an object is created.

The stack memory construct can be used with either static or dynamic memory allocation. Stacks are useful for managing function calls, preserving register values when an interrupt occurs, and supporting multiple tasking operations.

The static memory construct is similar to using static memory allocation, that is, static memory is allocated prior to execution. Since the memory is statically allocated, object sizes must be predetermined and cannot be modified during run time. This allows fast access to objects and the storing and retrieving of data. However the programmer is limited to only those already created objects. This results in a very inflexible system which forces the program to be modified each time an object must be modified. This can also be a waste of memory space since if all the created objects are not used they are just wasting space that could be used by other processes.

The last memory management construct is heap memory. Heap memory allows objects and arrays to be created dynamically. Objects and arrays grow and shrink in size during run time. This causes memory to not be wasted and can be reused when objects are deallocated. Only the required memory is allocated and, when the memory is no longer required, it is freed. A typical system using software-based memory management will suffer a time penalty due to the allocation and deallocation processes. In order to manage heap memory, either manual memory management or automatic memory management can be used. This can be beneficial in a system with limited memory.

Manual memory management is the process in which the allocation and the deallocation of memory are handled by the programmer. The program must allocate memory when an object or array is created. It must also deallocate memory when the object or array is considered to be dead. There are various issues with using manual memory management. The programmer must be careful to not allocate more memory than is physically available in the system. They must also be careful because the memory can be fragmented if the allocation and deallocation is not done correctly. This can cause the system to slow dramatically. When allocating new objects or arrays, the system must search for an enough contiguous memory space to accommodate the object. This can be a very difficult task for a programmer to

manage, especially in a program that handles large amounts of data. This process may be more difficult to properly implement. This method is used in programming languages such as C and C++ and uses predefined library functions to handle the allocation memory, creation of objects, deallocation of memory, and the deleting of objects.

An alternative to using manual memory management is to use automatic memory management. Automatic memory management was first introduced in the LISP programming language and has found its way into more modern programming languages such as C# and JAVA. The term 'garbage collection' is often associated with the method of automatic memory management. The garbage collector handles the deallocation of memory. It is activated when the system is running out of memory and needs to free memory for new objects. When activated, the garbage collector will scan the heap memory for dead objects, remove the dead objects, and free that memory for reuse. At the same time, the garbage collector will defragment memory placing all objects at the start of the heap. Figure 1.1 shows the heap before and after garbage collection.

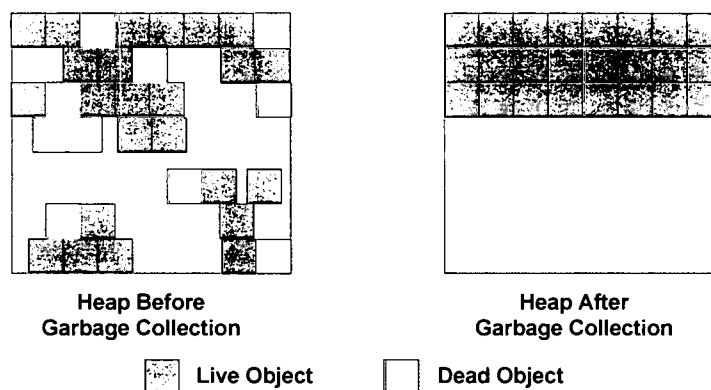


Figure 1.1 Heap Memory Before and After Garbage Collection

Figure 1.1 shows that prior to garbage collection, the memory can contain a large number of dead objects and can also be fragmented. After garbage collection, the dead objects are removed and the memory is defragmented. Before garbage collection, new large objects may be allocated as long as there is sufficient free memory to handle them. If there is not sufficient memory, then garbage collection is

activated and, after completion, sufficient free memory may be available to allocate the large object. The garbage collector is only in charge of deallocation and defragmentation of memory and not the allocation of memory. Allocation of memory is still done by the programmer, but the programmer does not have as great a concern in managing the number of objects created. The programmer still must be aware of the size of the physical memory and must be careful not to over allocate the number of live objects to full the memory.

1.1 Significant Types of Collectors

There are a number of different garbage collector designs that will do automatic memory management. Three popular collectors that are available are the Mark/Sweep Collector, Reference Counting Collector, and the Copy Collector. Each of these collectors is in charge of the same task, to manage memory by freeing up memory by removing dead objects so the program has enough memory to execute successfully.

1.1.1 Mark-Sweep Collector

The Mark-Sweep collector uses a two step method to complete garbage collection. The first step in the mark-sweep collector is the marking step in which memory is scanned and it changes a bit in the root of an object to mark if it is alive or dead. The root is the first address location in an object or array. Next, the sweep step removes the dead objects freeing up memory space. A moving scheme or a non-moving scheme can be implemented. In a moving scheme, the live objects are moved to defragment memory and in a non-moving scheme the live objects remain in their current locations and dead objects are just removed and memory locations marked as free.

An advantage to the moving scheme is that memory is defragmented making space for large objects to be allocated. This also is more time consuming and can slow the system considerably. In a non-moving scheme the collector executes in less time, due to not having to move objects around. But, this can lead to memory being fragmented and potentially an issue where there is not an enough memory space for a

large object to be allocated. Based on the design of the program being executed the programmer would have to choose which scheme to use.

There are two classes of the mark-sweep collector designs, recursive and non-recursive. The recursive collectors use repeated subroutines to trace live objects, but this can be very time consuming and can overflow the system stack, and are not suitable for hardware implementation [4]. In order to overcome this issue, non-recursive designs have been developed. These non-recursive mark-sweep collectors are based on an additional auxiliary stack [5].

The non-recursive mark-sweep collector uses the auxiliary stack to place objects which are referenced inside the current object being scanned into for further analysis when free time is available. First in an object is detected to contain references to other objects, the object is marked as alive and all the reference contained in it are pushed onto the stack. If the object has no references, but is determined to be alive it is also marked to be alive. The collector then pulls object pointers off the stack one at a time and locates the objects marks it as alive and checks if it contains any references and if it does it places them on the stack. This process is repeated until all objects in the stack have been checked and marked as alive or dead. The idea in marking objects as alive is so that the collector does not have to visit them again and will only need to visit dead objects removing them from the heap memory. The issue though with this type of design is that it can require a large amount of memory. Memory is not only needed for the heap memory but depending on the size of the heap memory a large auxiliary stack memory may be needed to accommodate for a large number of objects, sometimes on the order of millions of objects in larger programs. There are collectors that attempt to reduce the size of the auxiliary stack and one such collector design is the Boehm-Demars-Weisera design [2]. It attempts to uses a modified mark-sweep non-recursive algorithm to reduce the amount of memory required for the auxiliary stack. By doing so, it also adds additional overhead by having to mark objects on more regular bases to free stack space.

The mark-sweep algorithm is not designed to work in parallel with the program. When the mark-sweep collector is activated it executes till it completely

collects all dead objects in the heap and then returns control to the program. In a real-time system this could be cause problems, since there is no control over when the collector activates. This could lead to the missing of critical data due to the non-deterministic behavior.

1.1.2 Reference Counting

The reference counting collector is one of the simplest collector designs. It is based on the use of a reference count stored in the object which contains the number of other objects that reference to it. Due to this simple design the reference counting collector allows the collector to work in parallel with the program. When the collector is activated it can pause and allow the program to access the heap and when complete it continues. But, there are some catches to this in which the collector will not pause and release control. These will be described later in this section.

There are various combinations in which objects can point to one another. There is a sequential structure, a cyclic structure, and a cascading structure. Figure 1.2 shows each of these structures and the reference count associated with each object.

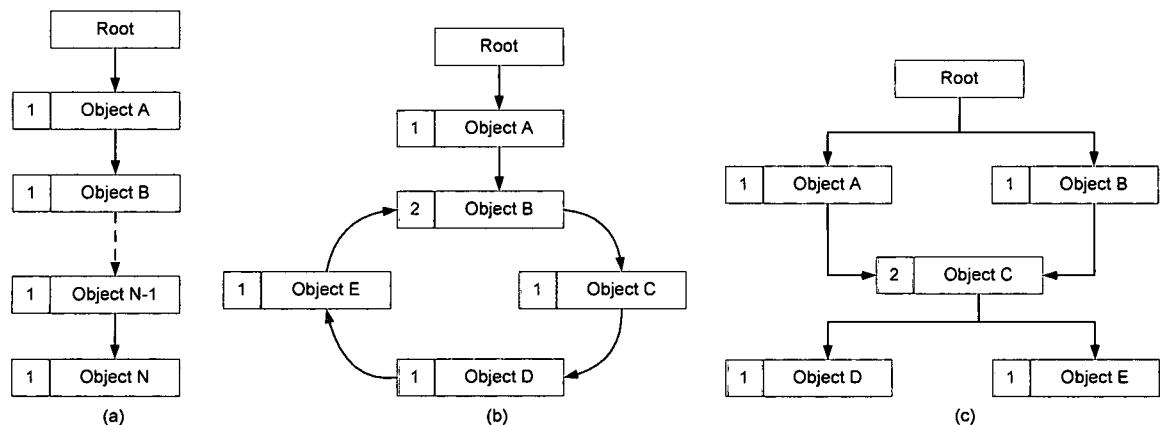


Figure 1.2 Object Pointer Structures

Figure 1.2 shows the main ways in which the objects can point to one another and the reference count associated with each object. Figure 1.2(a) shows a sequential structure in which each object points to another object in the chain in a sequential order. The root points to Object A and Object A references Object B and Object B

would reference the next object in the chain and so on down to the end of the change. In this type of structure the reference count for all the objects is one. Figure 1.2(b) shows a cyclic structure in which the root point to object A and object A reference Object B. This then continues until the last Object Points back to the Second Object, in this case Object B. Object B has a reference count of two since it has two objects referencing it, but all objects have a reference count of one. This is not the only cyclic structure that can be created but this one is shown since it can create a problem which will be shown later in this section. Lastly, Figure 1.2(c) shows a cascading structure where the root points to both Object A and Object B. These two objects then reference Object C and Object C then reference both Objects D and Object E. Since Object B has two objects pointing to it, it has a reference count of two and all the other objects have a reference count of one. Based on these structures it can be seen how the reference collector would work in an incremental fashion. When the collector is activated it will scan a complete structure till the end and will then release control to the program if needed and then continue with the next structure or object. The collector identifies objects as dead or alive based on the reference count. If the count is zero the object is considered dead and if it has a value it is considered to be alive. But there are some issues that can arise from this.

Based on the structures shown in Figure 1.2 certain issues can arise if certain objects have their reference count decremented to zero. The two structures that can cause the main issues are the sequential structure and the cyclic structure. The cascading structure does not really cause many issues unless it is combined with one of the other two structures. Figure 1.3 shows what can happen if a certain reference is broken.

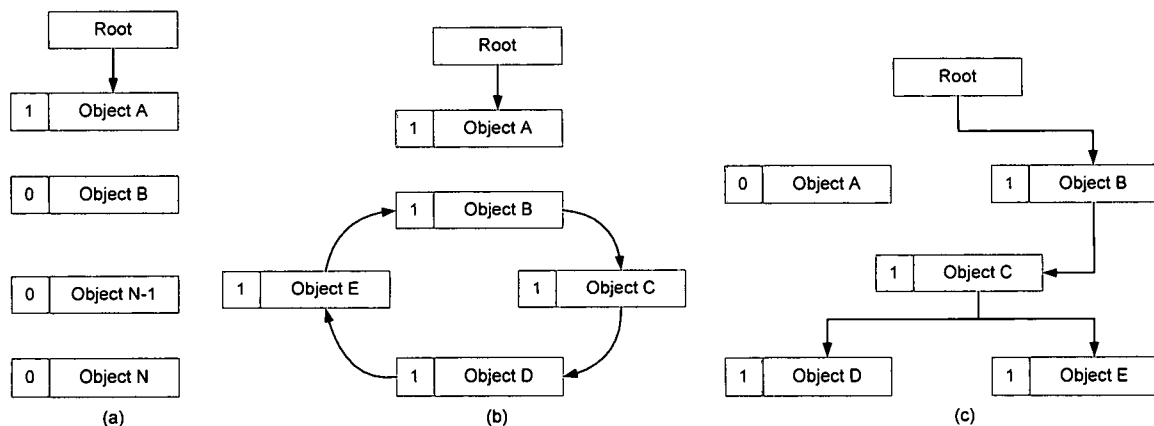


Figure 1.3 Potential Hazards with Reference Counting

Figure 1.3 shows the hazards that can arise if a certain reference to another object is removed. In Figure 1.3(a) it can be seen that if the reference from Object A to Object B is removed the reference count for Object B becomes zero and the collector will treat it as a dead object and remove it. Once this happens the object it reference will get a reference count of zero and treated as dead and also removed. This process will repeat all the way to the end of the sequential structure. The issue arises due to the objects referencing one another in a sequential order [5]. This is a potential hazard since if the chain is sequential and can take very long for the collector to execute since the collector will not release control to the program till it completely goes through the whole structure deallocating all the objects. Figure 1.3(b) shows a different type of issue that can arise in cyclic data. This issue is the most serious problem in which cyclic data is not collected [5]. When object A has the reference to Object B removed Object B still has a reference count of one since the last object (Object E) has a reference back to it. The collector will see all the objects as being alive even though Object B is really dead which would cause all the other objects other then Object A to be treated as dead also. But, since all the reference counts remain at one, the system treats the objects as alive and creates a memory leak. This memory cannot be reclaimed and reused due to this, thus reducing the actual size of heap memory accessible. If too many of these cyclic structures where created the heap could be eventually be filled to the point where no new objects could ever be

created, due to the large number of memory leaks. Figure 1.3(c) shows the last structure the cascading structure. If one of the reference is removed there is not as big of an issue since the remaining objects might have other objects referencing it. But if the tree structure was created using a sequential structure inside of it the same issue that arose from the cascading structure could arise.

In order to get around some of these issues there has been various designs of the reference counting collector created to deal with them. One method is to create a hybrid system combining the mark-sweep collector with the reference counting collector. Another reference count collector design by Weizenbaum [8] implements a method to reduce the issue of the sequential/Cascading structure and the amount of time it could take to deallocate all the objects in the chain. The Weizenbaum method uses a stack memory to place the objects in the cascading structure onto the stack but does not deallocate them at this time. It does this by deallocating the first object with a reference count of zero and then places the object's references on the stack. When the object on the stack is deallocated the object it references is then placed on the stack and so on down the line. By doing this, the collector can execute faster releasing control to the program to continue creating less delay. The collector goes back and deallocates the objects on the stack as time permits. So by doing this it does not remove the cascading deallocation but merely allows the time to do so to be more spaced out evenly across the system [4]. But, there can be an issue that can arise here too. If there is a large amount of data on the stack to be deallocated in memory, if the system does not have enough time to deallocate objects on the stack the heap memory can become full very quickly.

1.1.3 Copying Collector

The last collector that will be explained is the copy collector. The copy collector uses the method of copying live objects from one partition in heap memory to another or from one physical heap memory to another physical heap memory. New objects are allocated in the heap labeled as ToSpace and the other heap is labeled the FromSpace heap. The copy collector is activated in the same way most other collectors are activated and this is when the heap memory becomes too full to allocate more objects. When the copy collector is activated it flips the names attached to the

heaps and the ToSpace becomes FromSpace and vice versa. New objects are then allocated in the new ToSpace and the collector scans the new FromSpace for live objects and copies it to ToSpace. Any object that is not copied is considered to be dead and remains in FromSpace to be overwritten with new objects when the collector activates again and flips the heap memories again.

There are several types of Copy collector algorithms that could be used. One of the more famous copy collectors is the Cheney Copy Collector [3]. The Cheney Copy Collector uses the ToSpace and FromSpace method in which the ToSpace is where new object allocation happens and the FromSpace is where live objects are copied from by the collector. Figure 1.3 shows the copy collector and how it behaves when activated.

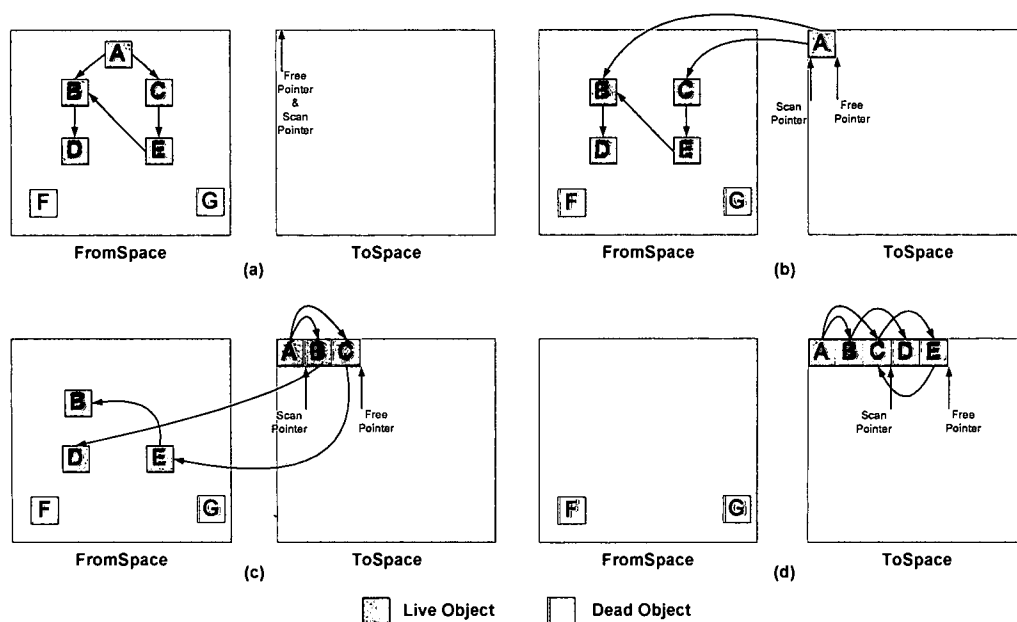


Figure 1.4 Cheney Copy Collector Example

Once ToSpace and FromSpace have been swapped the collector begins moving live object from FromSpace to ToSpace. Figure 1.4(a) shows the heap memory right after the swap. The FromSpace contains both live and dead objects and ToSpace memory is empty with the scanning pointer and free pointer pointing to the start of the ToSpace heap. The collector then copies the first live object, in this case Object A, to ToSpace. This is shown in Figure 1.4(b), where Object A is now in ToSpace, but still retains references to Objects B and C in FromSpace. The free

pointer is now updated in ToSpace to point to the next free location in the ToSpace heap. Figure 1.4(c) shows that next both object B and C are copied to ToSpace since they were both reference by Object A. The pointer to Object B and C are update in Object A to point to their new locations in ToSpace. At the same time the scan pointer has scanned Object A and marked it as alive and the free object pointer is updated to the next free space in the ToSpace. Objects B and C retain their pointers to Objects D and E in FromSpace. It should also be noted that the pointer in Object E that pointed to Object B still points to the location of Object B in FromSpace and not its new location in ToSpace. Figure 1.4(d) shows the last step where Objects D and E are copied to ToSpace and the references to them from Objects B and C are updated to point to their new locations in ToSpace. Also, the pointer in Object E is updated to point to the new location of Object B. Pointer Values inside an object are not update till the object is copied as in the case of Object E. At this same time this is going on Objects B and C are scanned and marked as alive and the pointer is updated. The free object pointer is also updated to point to the next free location in ToSpace. Also, since Objects F and E are considered to be dead they are not copied and remain in FromSpace to be later overwritten by new objects when the heaps swap during the next collection cycle. Know that all the live objects have been copied the last step would be for Objects D and E to be scanned and marked as alive and at this time the scan pointer and the free pointer would be at the same location. The only time these two pointers are at the same location is at the start of collection and at the end of collection. After collection any new objects that are allocated would be placed at the current location of the free pointer. This Cheney collector is not incremental, which means it will execute till it completely copies the live objects in FromSpace to ToSpace before releasing control to the program.

The reason the Cheney Copy Collector does not work incrementally is that since it allocates new objects at the free pointer when scanning happens will mark objects as alive or mark them as having descendents and must be scanned again later to make sure they exist. This is not very efficient and causes many problems if attempted to be implemented incrementally. This can also be an issue if attempting to use the Cheney Copy Collector in a real-time environment since if a sequential

structure we found that is very large in size it can take a large amount of time for it to be copied. This creates a non-deterministic behavior which is detrimental in a real-time environment. So to counter this, a modified version of the copy collector was created called the Baker Copy Collector.

The Baker Copy Collector is one of the most popular real-time garbage collecting algorithms [1]. The Baker Copy Collector algorithm is based on the Cheney Copy Collector algorithm, but with modifications to make it more useful and efficient in a real-time environment. The main idea of a ToSpace and FromSpace is still utilized as well as the idea of a scan and free pointer, but this is where the similarities end. Baker's collector implements an additional pointer called the new pointer. The new pointer is located at the end of the heap and points to the next free location in ToSpace for new objects to be created. This is different from the Cheney's design in that Cheney allocated objects in line with the objects being copied. Figure 1.5 shows the Baker Copy Collector in Action.

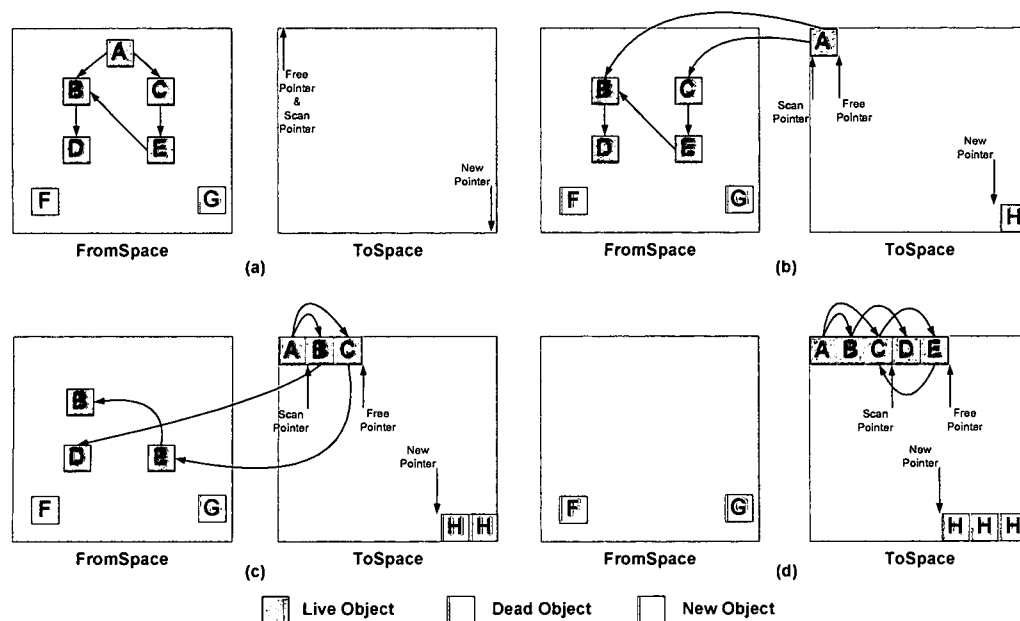


Figure 1.5 Baker Copy Collector Example

The collection scheme shown in Figure 1.5 is identical to the Cheney Collector previously described and shown in Figure 1.4. But, the difference is that the new pointer is introduced and points to the end of the heap as shown in Figure

1.5(a). After Object A is copied a new object can be allocated and is placed at the end of the heap as shown in Figure 1.5(b). Figure 1.5(c) and 1.5(d) show that new objects can be created after each of the remaining copy steps. This allows a deterministic behavior to be obtained by the collector. The longest delay is based on the size of the largest object and not on the size of heap as in the Cheney Copy Collector. Due to this, the Baker Copy Collector is best suited to small objects to keep copy time to a minimum. By knowing what the largest object that will be created the programmer can in theory time how long it takes to copy and can modify his program to allow this worst case scenario so that critical data is not missed.

1.2 Summary

Three different garbage collections schemes that have been described all have their advantages and disadvantages. The choice on which collector to use is left to the programmer to decide; however this is often the most difficult thing to do. This is since a certain collector may be appealing for one of its strengths and a different one for a different strength. Often through these tough decisions hybrid collectors are created to best suit the program being written. However, still too often the collectors create too large an overhead and can cause more problems than needed. So, based on this problem of slow execution time of the collector, the use of dedicated hardware based on a hybrid design seems appealing. This can be a difficult task since hardware is both time and cost consuming to develop when compared to software development, so the hardware collector must lend itself to use in a variety of environments. The remaining chapters will explain a hybrid design in which the Baker Copy Collector and the Reference Count Collector are utilized to create the final design and its implementation into a JAVA environment.

Chapter 2

Hardware, Design, and Testing Approach Overview

Chapter 1 explained three different classical garbage collectors. By combining aspects of each one, one can increase the performance of the collector. This design is based on Baker's Copy Collector. Due to limitations in the hardware and the copy collector design, a hybrid approach is utilized. This Chapter gives a generalized overview of the Xilinx Virtex 4 Field Programmable Field Array (FPGA) device and the hybrid design with more detailed descriptions of the SMM hardware and software interface designs given in later chapters.

The purpose of a garbage collector is to keep memory from fragmenting and making sure that any unused data is removed, thereby keeping as much memory available for the system. This creates a system that is more stable and runs more reliably. The main issue with garbage collection arises when one wants to run at real-time speeds, which is the time it takes to execute. Depending on the amount of memory that must be managed, the collector can cause slow program execution. Current software designed garbage collectors use ten to twenty percent of a programs total execution time [5]. In a real-time system, if a garbage collector was to use this amount of time it would eventually lead to system failure or the missing of critical data. More importantly, typical collectors run when memory allocation fails. Hence the system must wait until the collector to fully finish before continuing with the real-time tasks. This inserts a random pause in the real-time execution causing potential critical tasks to be missed. By creating a hybrid hardware garbage collector system to manage the memory one can benefit greatly from the increase in speed custom hardware allows. By allowing direct access to the memory and removing slow software function calls, it allows a more predictable and deterministic system. This paper describes this hybrid system and its implementation into a software environment. This chapter gives a simple hardware overview of the FPGA used, the hybrid system, and the general interfaces between the

hardware and software. The testing environment for the SMM is all described with results shown in chapter 6.

2.1 *Virtex 4 FPGA and Xilinx ML403 Development Board*

The device that was used to implement the SMM design is the Virtex-4 FX FPGA manufactured by Xilinx. At the time of the SMM design implementation, the only available FX series chip was the FX12 (V4FX12) and was the newest FPGA chip developed by Xilinx. The Virtex-4 FX was chosen for the inclusion of a hard silicon PowerPC 405 RISC processor core (PPC405). This processor is a 32 bit processor with a 5-stage execution pipeline, 16KB data level 1 cache, and 16KB instruction level 1 cache [10]. The PowerPC processor lies within the FPGA fabric which allows the PowerPC to be attached to custom created hardware or external devices.

As well as containing the PPC405 core, the virtex-4 FPGA also contains other useful features. It contains internal memory, called Block RAM (BRAM), which allow the creation of dedicated memory in flexible data widths and memory block sizes. BRAM runs at speeds up to 500 MHz and is located within the FPGA. This allows easy access to BRAM without the overhead associated with accessing external memory. The V4FX12 FPGA contains 36 18-Kbit-blocks of BRAM memory.

The V4FX12 contains 12,312 logic units and is the smallest of the FX series chips. Xilinx tools are used to compile a design, the use the logic units to create various components within the design.

The FPGA was mounted in a development board containing components useful during the design and testing process. The model of the board is the ML403. Figure 2.1 shows the Xilinx ML403 Development Board.

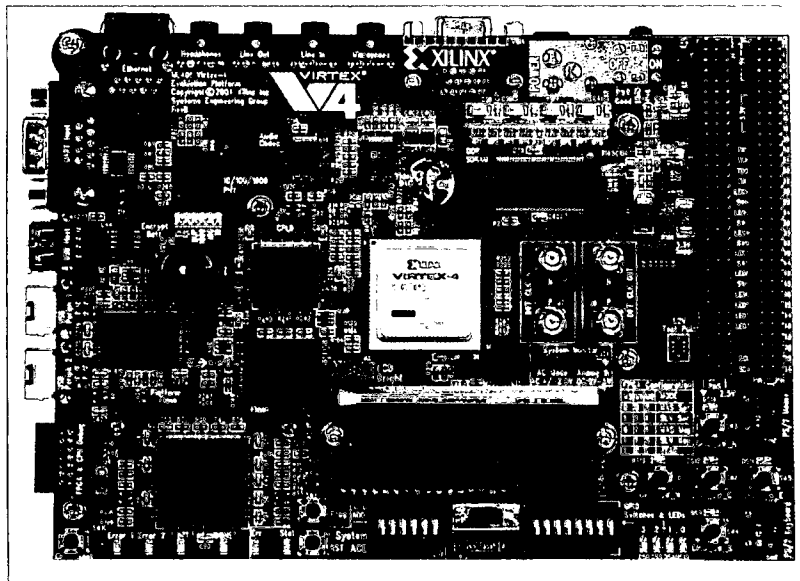


Figure 2.1 Xilinx ML403 Development Board

The ML403 contains Ethernet, RS232, SRAM, SDRAM, and External I/O pins, as well as various other devices. The devices allow a wide variety of designs to be implemented, as well as a range of operating systems such as Linux, OS/2, and Wind River to be run in the ML403 development board. The operating system chosen to implement the SMM design was Open Source Linux based on the 2.4.26 kernel using the PowerPC development kernel. Once running, Linux supports the use of readily available software compilers and created a system in which custom hardware could be easily constructed in the FPGA logic. This also allowed the SMM design to be used within currently available java compilers, such as the open source GNU java compiler GCJ and University of Washington's JRate Real-time JAVA compiler.

2.2 Hybrid Approach

The hybrid design is based on Baker's Copy Collector. Objects are copied from one memory space to another during collection. The initial intent was to try and mimic this design, but in hardware instead of software. It quickly became apparent that this would not be possible due to limitations in the hardware, as well the complexity required in moving the design into hardware. The differences between the Baker Copy Collector and the Hybrid Approach used will be shown in the rest of this section. The similarities

between the two designs will be shown and then the hybrid modifications made to make the hardware implementation possible.

The Backer Copy Collector uses either multiply memories or a single memory with two distinct partitions to store objects in. One memory is labeled the ToSpace memory in which new objects are allocated and the FromSpace in which live objects are copied from to the ToSpace during collection. The only other similarity is in that new objects are allocated at the end of the memory space and copied objects are placed at the beginning of the memory space. But, this is where the similarities end, since the method to scan the memory could not be implemented in identical fashion and the marking of live or dead objects was done differently.

The Backer Copy collector uses three pointers to track where the data is copied to, where new objects are allocated, and a scan pointer marking data as alive or dead. The hybrid design uses two additional memories called the to-copy memory and is-copied memory that are equal to the maximum number of objects that can be allocated. These two memories are used to mark which object need to be copied and which objects have been copied. In Chapter 4 this will be explained in greater detail and why it was implemented in this fashion.

The hybrid approach used not only used the Baker Copy Collector Scheme to Copy data from ToSpace and FromSpace it also implemented the idea of reference counting. In classical reference counting designs the reference count stored in an object is the number of other objects that references it. In the SMM design, this is reversed and the root object contains the number of objects it reference to plus one. If the reference count is zero then the object is considered dead unless it is referenced by another object. This will be explained in greater detail in chapter 4. Figure 2.2 shows the overall design of the SMM system.

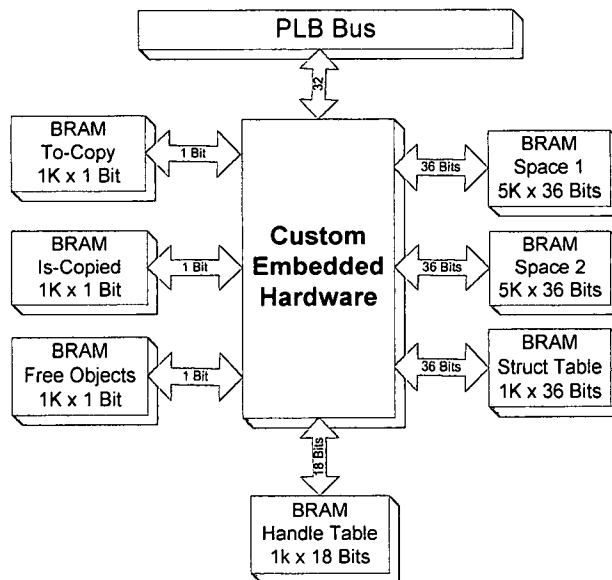


Figure 2.2 SMM System Block Diagram

In figure 2.2, it can be seen that the system uses the internal FPGA Bram for the required memories and tables. This was done due to lack of logic available resources on the ML403 development board. The system has two BRAM Memories sized at 5k x 36 bits for use as the ToSpace and FromSpace object storage memory. It contains a 1k x 36 bit BRAM memory used to store the structure definitions as well as a 1k x 18 bit memory for storing the handle Table which contains pointers to the objects in ToSpace for each object. The object ID returned is the address location in the Handle Table. There are then three BRAM memories of size 1k x 1 bit. One of these memories is used to mark which object ID's are used and which ones are free. When an object ID is allocated a value of 1 is written into the Free Objects Memory at the address that equals the address in the Handle Table. The other two memories of this size are used to mark which objects have been copied and which objects need to be copied during garbage collection.

Based on the SMM design shown in figure 2.2 external devices located on the ML403 development board where chosen. The devices chosen where based on the minimum requirements needed to execute in the final test environment. This was also done to allow majority of the FPGA logic to be used for the SMM implementation. Figure 2.3 shows the complete system bock diagram with all bus connections for each device.

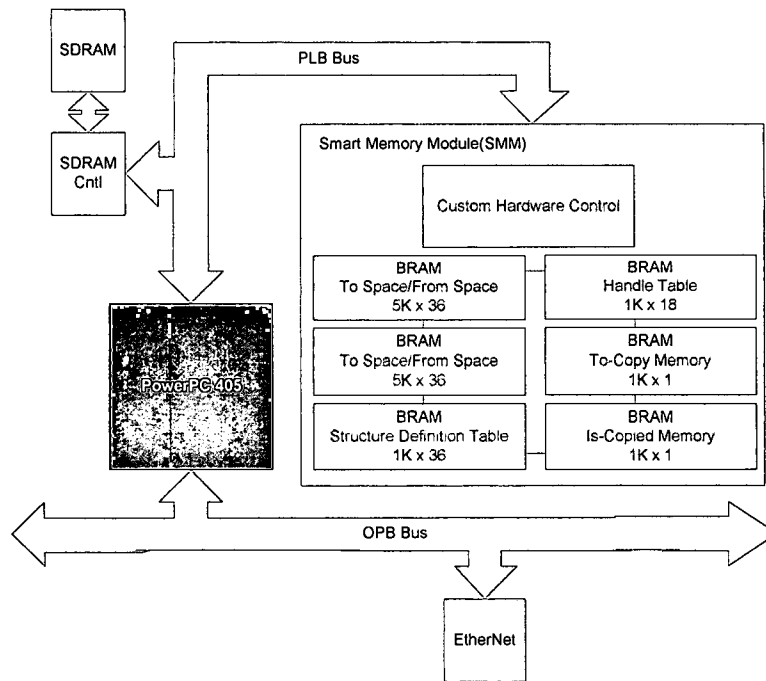


Figure 2.3 Complete System Bus Connections

The available devices shown in Figure 2.3 are the Ethernet, SDRAM, PowerPC, and the SMM Hardware. This utilized 99% of the available logic on the FPGA and did not allow any additional device support to be added.

In order to interact with the SMM, an interface had to be generated between the Linux Operating System executing on the PowerPC processor and the SMM hardware. This was accomplished by creating custom Linux Driver using already existing Linux functions and creating hardware on the SMM side to accept the function calls.

Lastly, in order to handle interactions between the interface design and the garbage collector, a handshake mechanism had to be developed. This handshake handled access to the various system memories. If the interface was enabled, it would grant access to the interface allowing it to read and write to the different memories and when the collector was activated it grant it access to the memories. This issue here is who has a higher priority over the other if the interface and the garbage collector both ask for access to the memories the same time? In this case, the interface was given priority to complete in order to create less delay in the software program before the collector grabs control and executes a complete collection cycle and releases control of the memory. Figure 2.4 shows the internals modules of the SMM Hardware and their interconnections

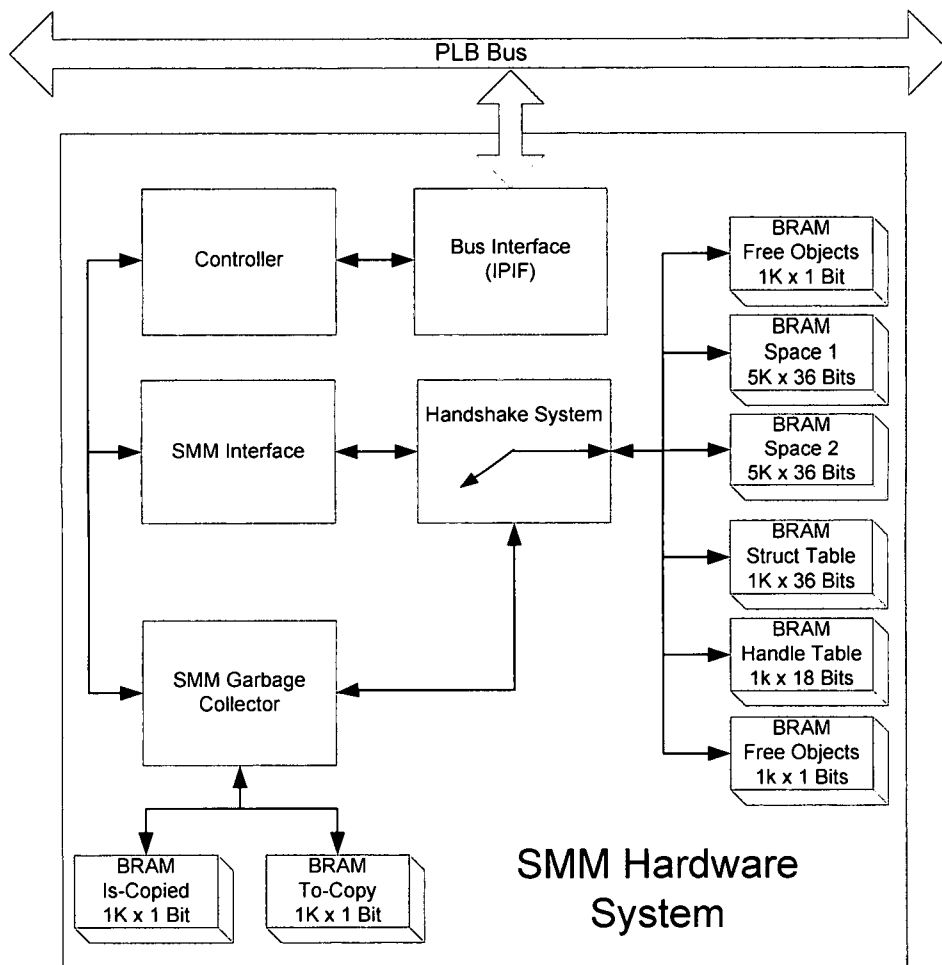


Figure 2.4 SMM Internal Module Connections

Figure 2.4 shows that all bus data comes in through the Bus interface (IPIF) and is routed to the SMM controller. The controller then routes the incoming data to either the SMM Interface or the Garbage Collector and also tells the handshake system which process should have control of the memory. The handshake system then routes the memory access lines to the appropriate module, the SMM interface or garbage collector. In Chapter 3 the hardware interface will be discussed in greater detail and the software driver in chapter 5.

2.3 Testing Environment

The Boeing navigational system for the Scan Eagle UAV was used to show the full capabilities of the hardware SMM system. The application executed various level of

processing goals. The higher the processing rate of a specific task, the more data it created and the more crucial it was that the data was not missed due to garbage collection. The test application was based on a frame system in which each task has a specific amount of time to be executed and if a frame is missed the system can malfunction. The high rate processing consisted of tactical steering of the UAV and the low rate processing consisted of navigational steering. The system was toggled between the high rate and low rate processes once per second with each one being placed within three different threads executing at rates of 20 Hz, 5 Hz, and 1Hz.

Chapter 3

Interface Design

Chapter 2 provided an overview of the smart memory module (SMM) hybrid garbage collection system. Chapter 3 will describe the hardware interfaces provided by the SMM to the software environment. The SMM supports ten software function calls to support memory allocation. The garbage collector function is transparent to the programmer. It allows the programmer to not agonize about the memory management aspect of the system.

3.1 *Implemented Functions*

The functions supplied range from data handling, system management, and system verification. The functions included are, *New Structure Definition*, *New Object*, *New Array*, *Clone*, *Put Object Data*, *Get Object Data*, *Bump Reference Count*, *Decrement Reference Count*, *Get Reference Count*, and *Reset*. The following sections will explain each of these functions in greater detail.

3.1.1 New Structure Definition

The *New Structure Definition* function allows the creation of a new data object definition. The size of the object and a pointer mask is passed to the SMM. The SMM then returns a structure ID, which is used during creation of a new object. Figure 3.1 shows the simple block diagram of the *New Structure Definition* function.

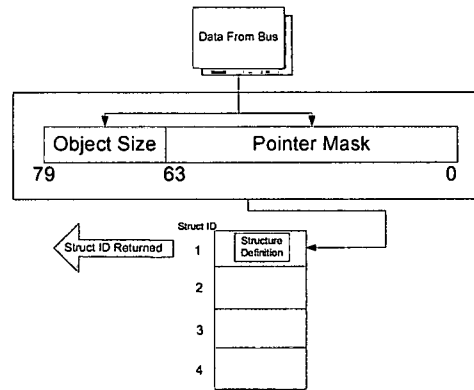


Figure 3.1: Simple Block Diagram of New Structure Definition Function

Data comes in from the PLB bus and is then stored in the structure table memory. The structure ID is then returned from the SMM. Figure 3.2 shows the state machine for the *New Structure Definition* function showing a more detailed view of the flow of data shown in figure 3.1.

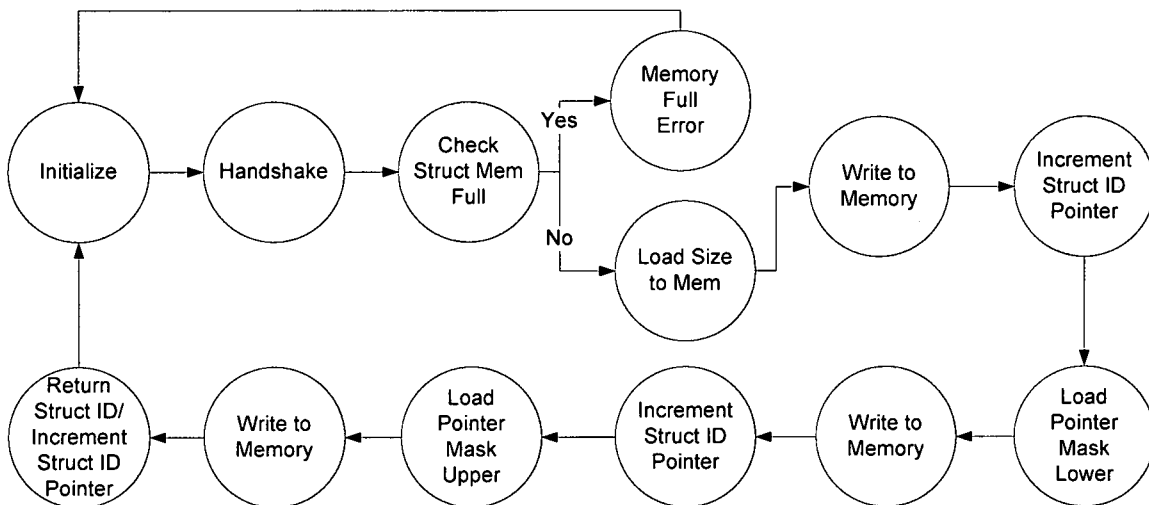


Figure 3.2: New Structure Definition State Diagram

The first state is the initialize state and the state machine remains in this state until the function is activated by the software program. The function then moves to the handshake state, where it verifies that another function is not being executed and that it has control of memory. It then verifies that there is space available in the structure memory. If there is no memory available, an error code is returned. Otherwise, the size

of the object is loaded and written into memory. This is followed by the writing of the lower 32-bits and upper 32-bits of the pointer mask into the structure table memory. Throughout this process, the structure table memory pointer is updated to point to the next free space. Once completed, a new structure definition takes three 32-bit memory locations in the structure table memory. The first address in memory of the structure definition is returned as the structure ID. The state machine then goes back to initialize and waits until it is called again.

3.1.2 New Object

The *New Object* function allows the creation of new data objects in current ToSpace Memory. A structure ID is passed to the SMM interface and the SMM locates the structure definition in the structure table memory. Figure 3.3 shows the simple block diagram of the *New Object* function.

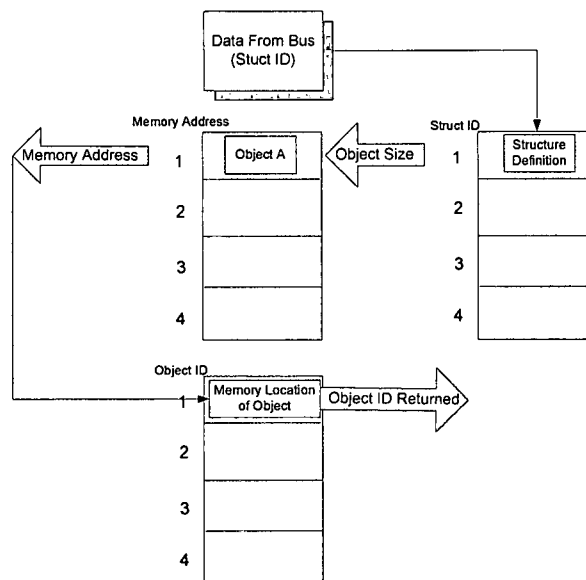


Figure 3.3: Simple Block Diagram of New Object Function

Data comes in from the PLB bus accessing the structure definition from the structure table memory. The data stored in the structure definition is then used to create the new object in ToSpace Memory. The root address of the object is stored in the handle table memory. The address in the handle table is returned as the object ID. Figure 3.4

shows the state machine for the *New Object* function. It shows a more detailed view of the flow of data, which was shown in figure 3.3.

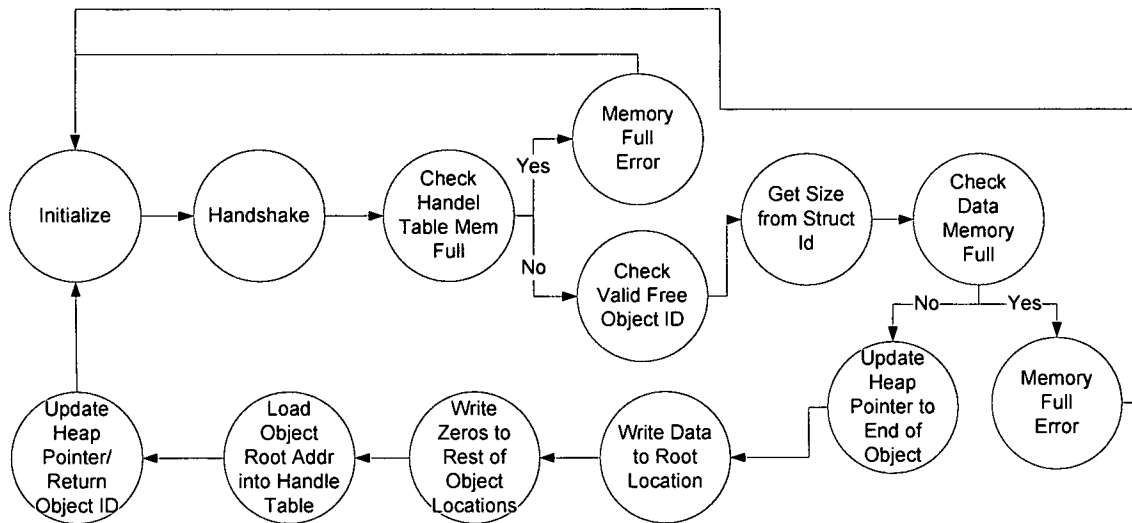


Figure 3.4: New Object State Diagram

The first state is the initialize state and the state machine remains here until the function is activated. The function then moves to the handshake where it verifies that another function is not being executed and that it has control of memory. It then verifies that there is space available in the handle table memory. If there is no memory available, an error code is returned. If there is memory available in the handle table memory, the system then obtains the next free object ID. The system obtains the size of the object from the structure definition and uses this to check if there is enough memory available in ToSpace. If there is not enough memory, an error code is returned. Otherwise, the structure ID and reference count is written into the root location of the object. All other memory locations in the object have zeros written into them. The address of the root location is then written into the handle table for later use in accessing the object. The address in the handle table is returned as the object ID and then returns to the initialize state. The memory allocation for an object is the size stored in the structure definition plus one. This allows the size stored in the structure definition to be the size of accessible data storage and the first memory address in an object to be used to store the reference

count and structure ID. Once completed, a one is placed into the object ID look up table signifying that this specific object ID is in use.

3.1.3 New Array

The *New Array* function allows the creation of new array in ToSpace memory. When a new array is created the SMM Interface is sent a size and if the array is a pointer type. All data stored in an array must be all raw data or pointers. Figure 3.5 shows a simple block diagram of the *New Array* function.

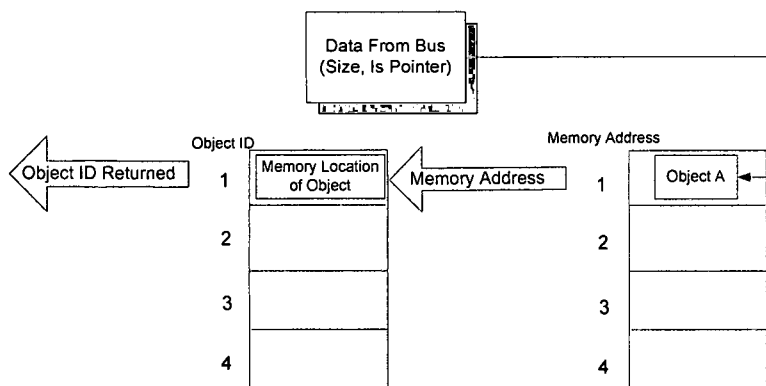


Figure 3.5: Simple Block Diagram of New Array Function

Data comes in from the PLB bus and is used to create a new array in the current ToSpace memory. The address of root location is stored in the handle table memory. The address in the handle table is returned as the object ID. Figure 3.6 shows the state machine for the *New Array* function showing a more detailed view of the flow of data shown in figure 3.5.

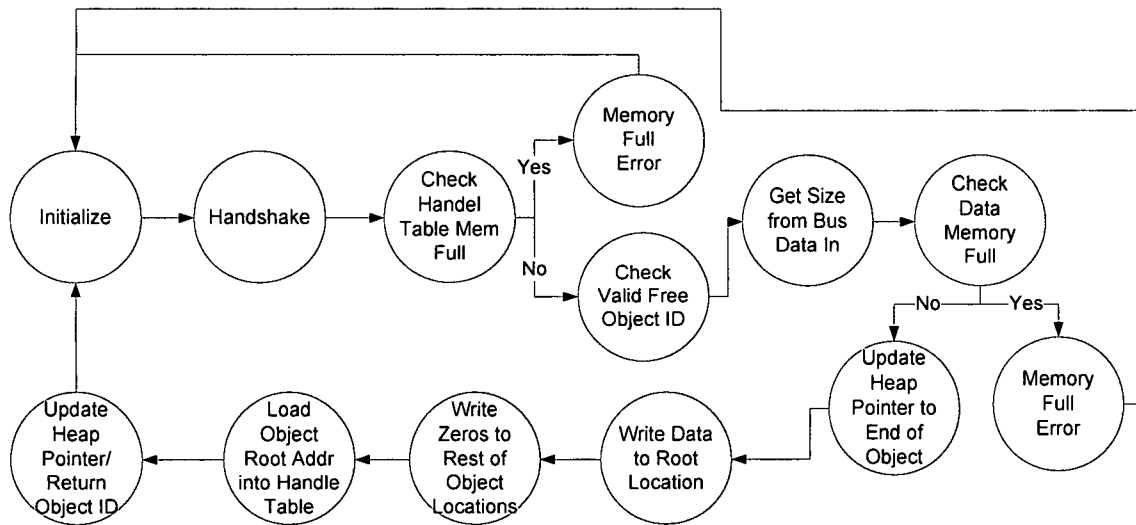


Figure 3.6: New Array State Diagram

The first state is the initialize state and the state machine remains here until the function is activated by the software program. The function then moves to the handshake where it verifies that another function is not being executed and that it has control of memory. It then verifies that there is space available in the handle table memory. If there is no memory available, an error code is returned. Otherwise, the system obtains the next free object ID. The system then obtains the size of the array from the data passed in from the PLB bus and uses this to check if there is enough memory available in ToSpace memory. If there is not enough memory, an error code is returned. Otherwise, the array size, data or pointer type, and reference count is written into the root location. All other memory locations in the array get zeros written into them. The address of the root location is written into the handle table for later use in accessing the object. The address in the handle table is then returned as the object ID and then goes back to the initialize state. The memory allocation for an array is the size passed in plus one. This allows the size stored in the structure definition to be the size of accessible data storage and the first memory address in an array to be used to store data used during garbage collection. Once the function has completed, a one is placed into the object ID look up table signifying that this specific object ID is in use.

3.1.4 Clone

The purpose of the *Clone* function is to make an exact copy of a data object or array currently stored in memory. The SMM interface is passed an object ID and the SMM uses this to determine if a data object or array is being cloned. Figure 3.7 shows the block diagram of the *Clone* function.

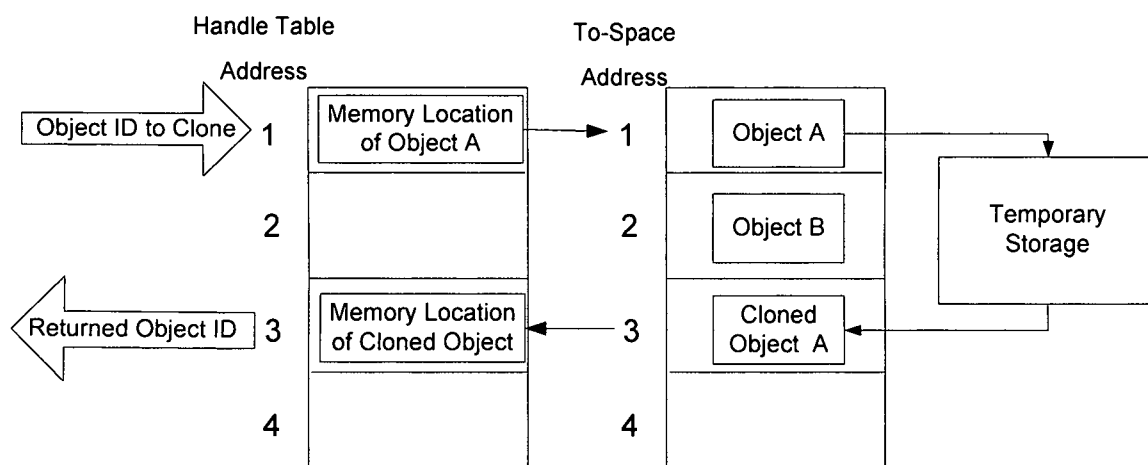


Figure 3.7: Simple Block Diagram of Clone Function

Data comes in from the PLB bus and is used to locate the address to the object or array to be cloned. The object or array is cloned and the address of the root location is saved to the handle table memory and returned as the object ID. Figure 3.8 shows the state machine for the *Clone* function showing a more detailed view of the flow of data shown in figure 3.7.

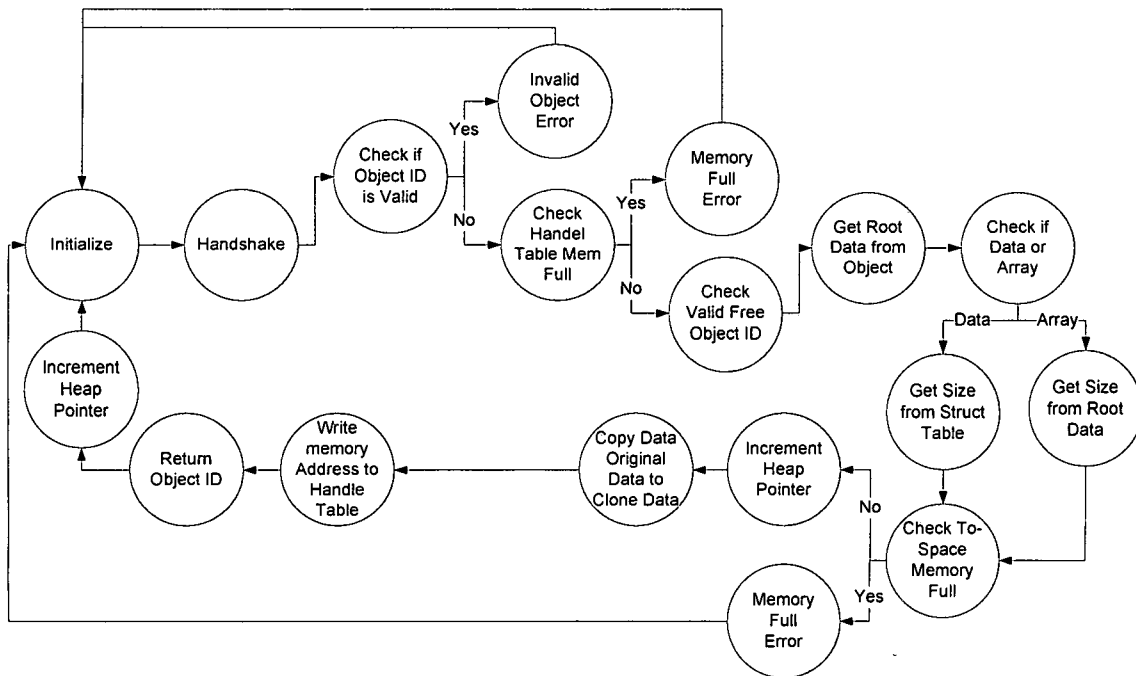


Figure 3.8: Clone State Diagram

The first state is the initialize state and the state machine remains here until the function is activated by the software program. The function then moves to the handshake where it verifies that another function is not being executed and that it has control of memory. Next it verifies that the object ID passed in is valid. If it is not valid, it returns an error code. Otherwise, it verifies that there is space available in the handle table memory. If there is no memory available, an error code is returned. If there is memory available in the handle table memory, the system then obtains the next free object ID. The SMM then obtains the data from the root location of the object or array to be cloned. If an object is being cloned, the structure ID stored in the root data is used to obtain the size of the object to clone. If an array is being cloned, the size of the array is obtained directly from the root data. Based on the size obtained, the SMM checks if there is enough memory space in ToSpace memory. If there is not enough memory available, an error code is returned. Otherwise, the heap pointer in ToSpace Memory is updated based on the size obtained. Then the data is copied from the object or array being cloned to the memory location of the new object or array. The root location of the new object or array is then written into the handle table and the address returned as the object ID. It then returns to the initialize state and waits till it is accessed again. Once the function has

completed, a one is placed into the object ID look up table signifying that this specific object ID is in use.

3.1.5 Put Object Data

The *Put Object Data* function is used to place data into an object or array. The SMM interface receives the object ID and offset from the software and uses it to determine where to place it. Data being placed in an object can be either a pointer or data. All data placed in an array must be either all data or pointers. Getting data from the systems SDRAM to the SMM is a two step process. Figure 3.9 shows step 1, the loading of the data from the SDRAM.

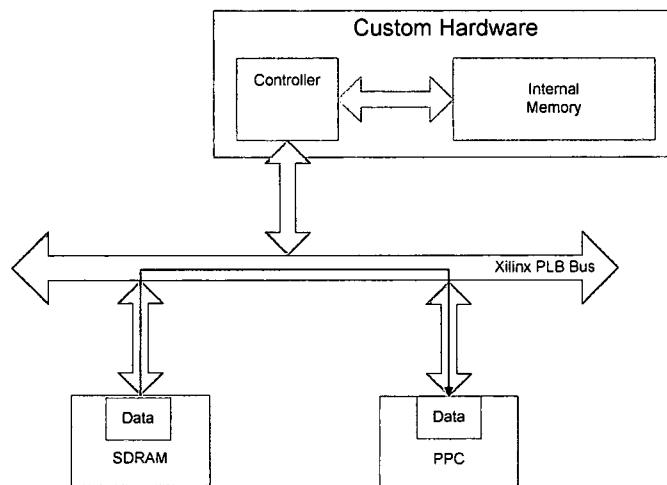


Figure 3.9: Step 1: Loading Data to Internal PowerPC Registers

Data is obtained from the SDRAM and stored in internal PowerPC registers. Figure 3.10 shows step 2, the sending for the data to the SMM.

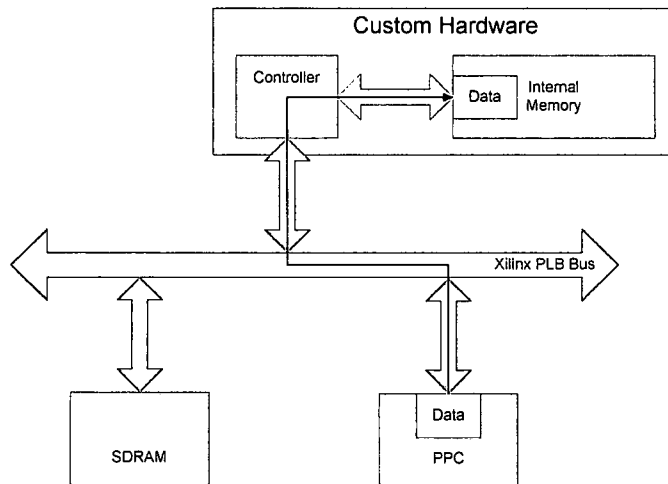


Figure 3.10: Step 2: Sending Data to the SMM from Internal PowerPC Registers

Data is then sent from the internal PowerPC registers to the SMM. Figure 3.11 shows the state machine of how data is handled by the *Put Object Data* function once the SMM receives it.

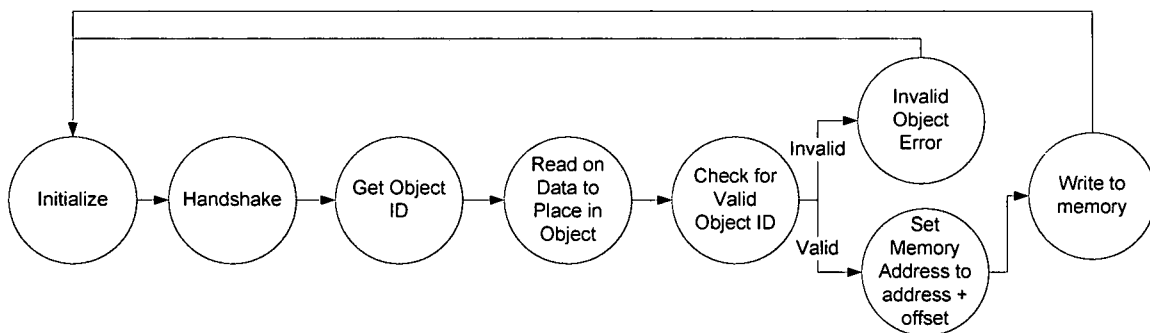


Figure 3.11: Put Object Data State Diagram

The first state is the initialize state and the state machine remains in this state until the function is activated by the software program. The function then moves to the handshake where it verifies that another function is not being executed and that it has control of memory. Next, the object ID is obtained and is then verified that it is valid. If the object ID is not valid, an error code is returned. Otherwise, the address location of the object is obtained from the handle table. From here, the offset sent to the SMM is added to the address to obtain to location in the object or array that the data is to be placed. The data is written into this desired location and returns to the initialize state.

3.1.6 Get Object Data

The *Get Object Data* function retrieves data in an object or array. The SMM interface receives the object ID and offset from the software and uses to determine what data to return. Data is then sent back to the software program regardless if it is raw data or a pointer. In the case where a pointer type is being returned, the SMM does not take charge of dereferencing the pointer, this is left to the programmer to handle. When data is returned from the SMM, it is a two step process to get the data from the SMM to SDRAM. Figure 3.12 shows the first step of getting data from the SMM and storing it into an internal PowerPC register.

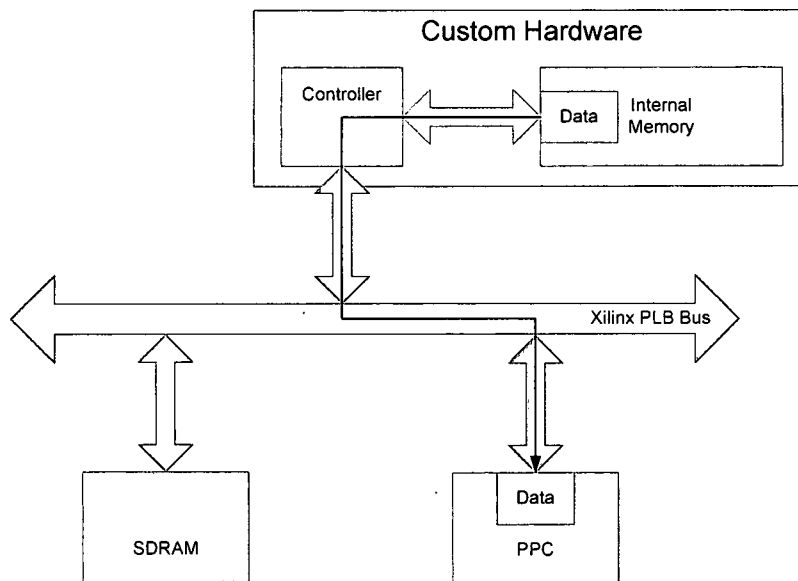


Figure 3.12: Step 1: Returned Data from the SMM to Internal PowerPC Registers

Data is obtained from the SMM and stored in internal PowerPC registers. Figure 3.13 shows the second step of sending the returned data to the SDRAM.

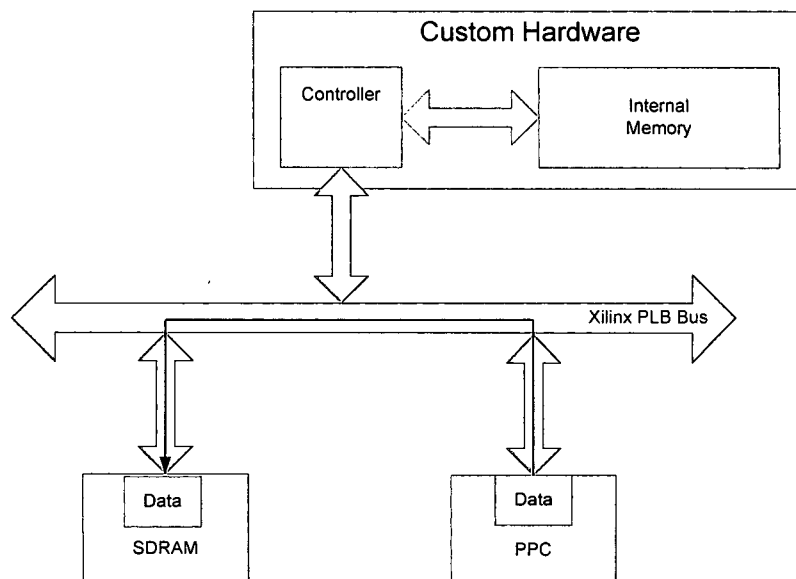


Figure 3.13: Step 2: Sending Data from Internal PowerPC Registers to SDRAM

Data is then sent from the internal PowerPC registers to the SDRAM. Figure 3.14 shows the state machine of how the data is handled by the *Get Object Data* function.

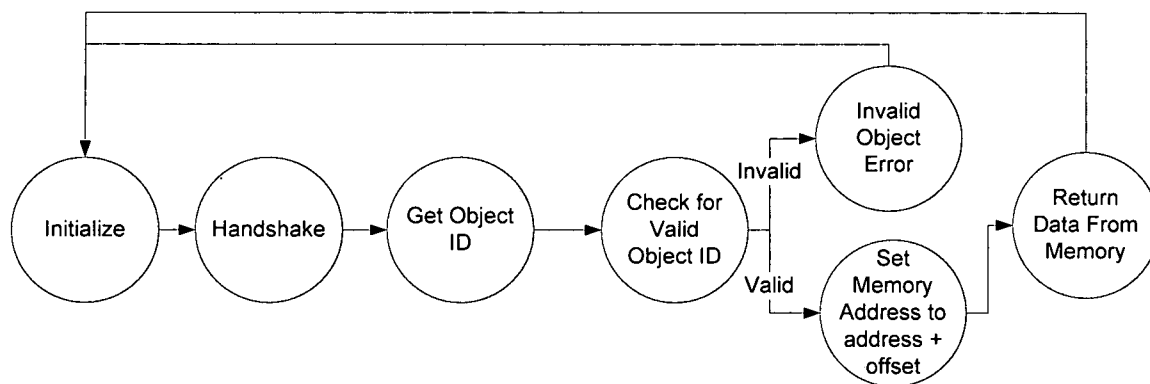


Figure 3.14: Get Object Data State Diagram

The first state is the initialize state and the state machine remains in this state until the function is activated by the software program. The function then moves to the handshake where it verifies that another function is not being executed and that it has control of memory. Next, the object ID of the object or array is obtained and is then verified that it is valid. If the object ID is not valid, an error code is returned. Otherwise, the address location of the object is obtained from the handle table. From here the offset

is added to the address to obtain to location in the object or array of the data to be returned. The data is then returned from the SMM and then goes back to the initialize state.

3.1.7 Bump Reference Count

The *Bump Reference Count* function is used to increase the reference count by one. The reference count is stored in the root location of an object or array. The SMM interface receives an object ID and uses this to locate the root location in ToSpace memory from the handle table. Figure 3.15 shows the process of the *Bump Reference Count* function

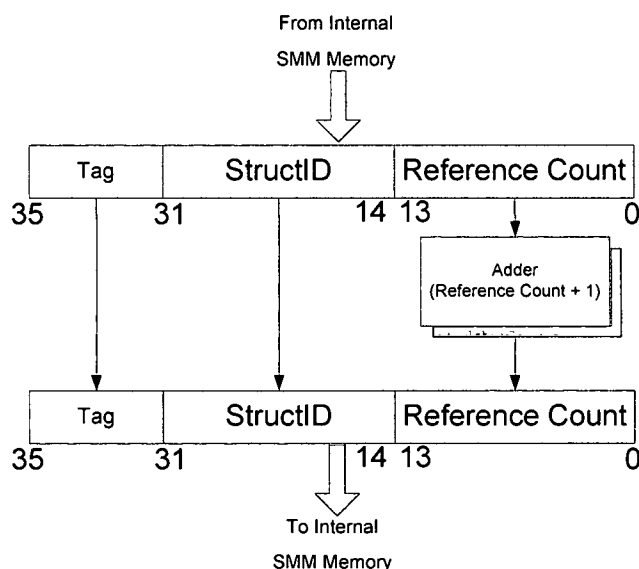


Figure 3.15: Incrementing the Reference Count Stored in Internal SMM Memory

Data comes in from ToSpace memory and the reference count is increased by one. The Tag and Struct ID are left untouched. Figure 3.16 shows the state machine of how the *Bump Reference Count* function is handled in the SMM.

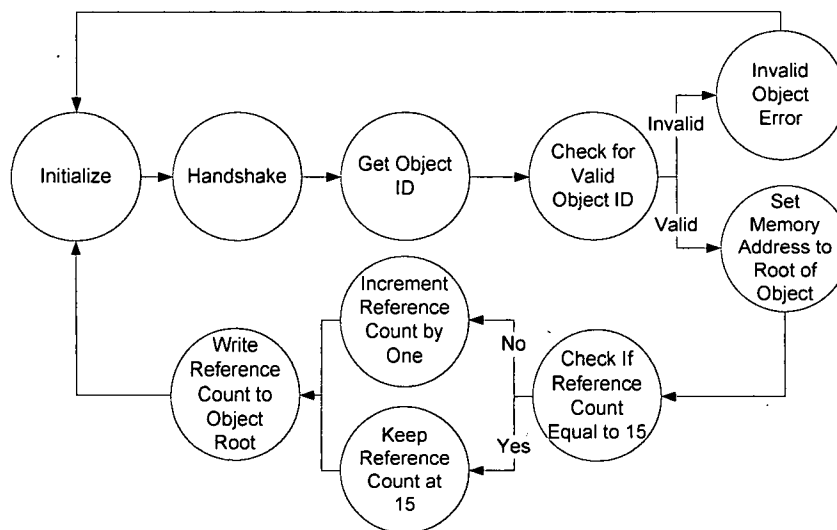


Figure 3.16: Bump Reference Count State Diagram

The first state is the initialize state and the state machine remains here until the function is activated by the software program. The function then moves to the handshake state where it verifies that another function is not being executed and that it has control of memory. Next, the object ID of the object or array is obtained and is verified that it is a valid object ID. If the object ID is not valid an error code is returned. Otherwise, the address location of the object is obtained from the handle table. The data is then obtained from the root location of the object and the reference count is checked. If it is already at the maximum reference count allowed of 15, it remains at 15. Otherwise, it is increased by one. It is then written back into the root location of the object and returns back to the initialize state.

3.1.8 Decrement Reference Count

The *Decrement Reference Count* function is used to decrease the reference count by one. The reference count is stored in the root location of an object or array. The SMM interface receives an object ID and uses this to locate the root location address in ToSpace memory of the object from the handle table. Figure 3.17 shows the process decreasing the reference count.

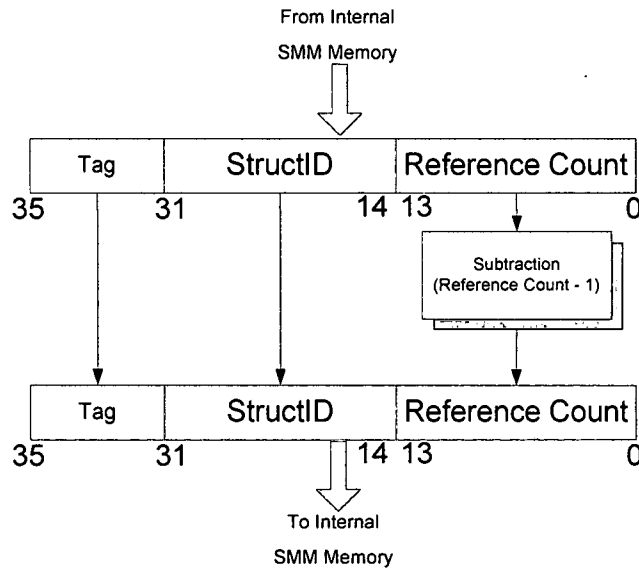


Figure 3.17: Decrementing the Reference Count Stored in Internal SMM Memory

Data comes in from ToSpace memory and the reference count is decreased by one. The Tag and Struct ID are left untouched. Figure 3.18 shows the state machine of how the *Decrement Reference Count* function is handled in the SMM.

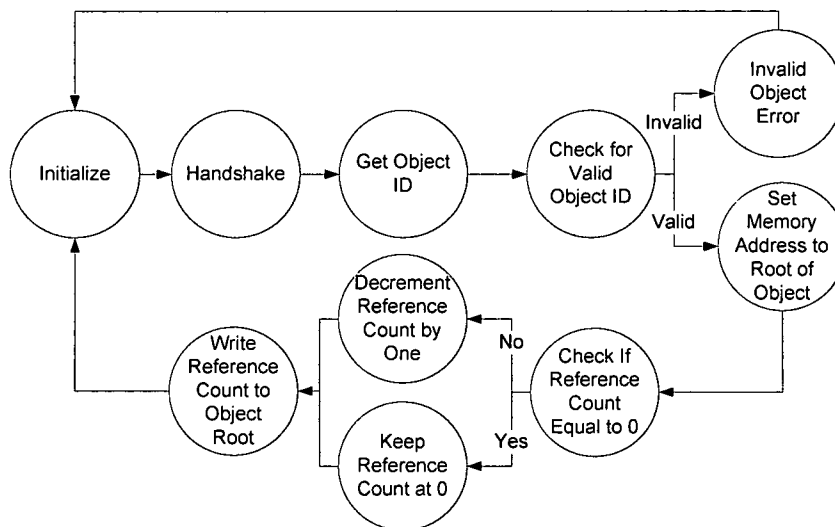


Figure 3.18: Decrement Reference Count State Diagram

The first state is the initialize state and the state machine remains here until the function is activated by the software program. The function then moves to the handshake state where it verifies that another function is not being executed and that it has control of

memory. Next, the object ID of the object or array is obtained and is verified that it is a valid object ID. If the object ID is not valid an error code is returned. Otherwise, the address location of the object is obtained from the handle table. The data is then obtained from the root location of the object and the reference count is checked. If it is already at the minimum reference count allowed of zero, it remains at zero. Otherwise, it is decreased by one. It is then written back into the root location of the object and returns back to the initialize state.

3.1.9 Get Reference Count

The *Get Reference Count* function is used to return the current reference count of an object. This function was implemented for testing purposes since data contained within the SMM is transparent to the software. To obtain the reference count the SMM is passed an object ID and the SMM uses this to locate the root location of the object. Figure 3.19 shows a simple overview of where the reference count is stored in the root of an object.

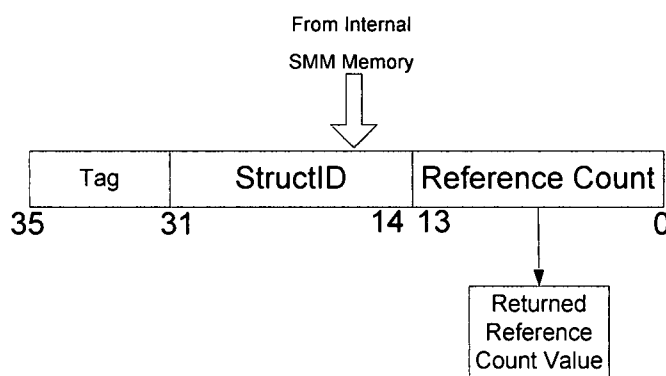


Figure 3.19: Returning of Reference Count

Data comes in from ToSpace memory and the reference count is returned from the SMM. Figure 3.18 shows the state machine of how the *Get Reference Count* function is handled in the SMM.

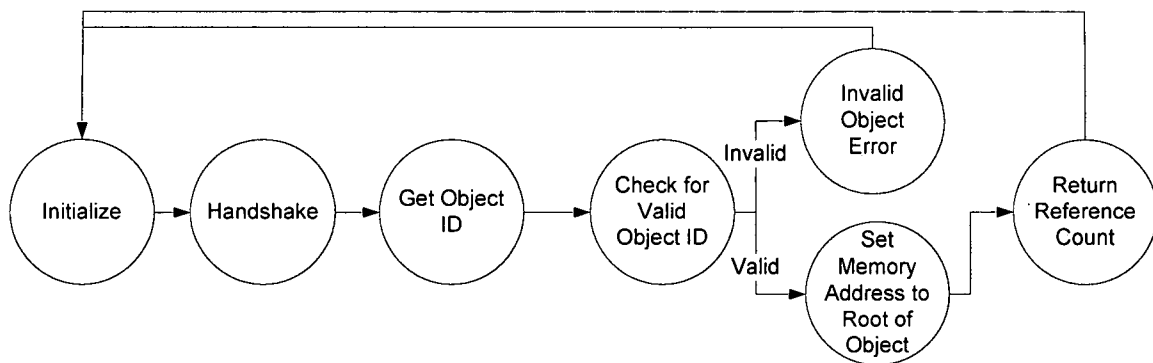


Figure 3.20: Get Reference Count State Diagram

The first state is the initialize state and the state machine remains in this state until the function is activated by the software program. The function then moves to the handshake where it verifies that another function is not being executed and that it has control of memory. Next the object ID of the object or array is obtained and is verified that it is a valid object ID. If the object ID is not valid, an error code is returned. Otherwise, the address location of the object is obtained from the handle table. The data is then obtained from the root location of the object and the reference count is returned and then returns back to the initialize state.

3.1.10 Reset

The purpose of the *Reset* function is to reset the system to its initial boot up state. This function was implemented for use when running multiple test programs. Figure 3.21 shows the state machine of the *Reset* function.

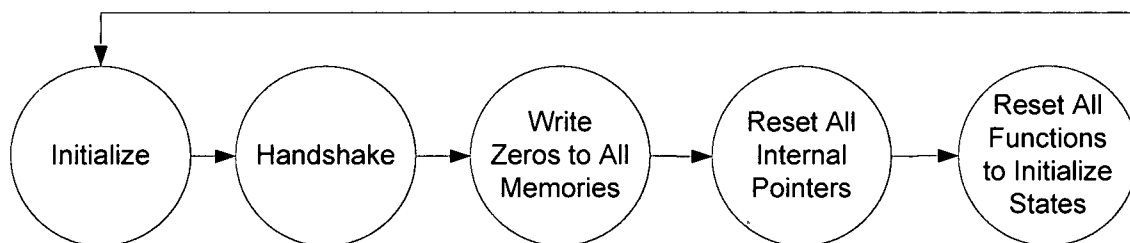


Figure 3.21: Reset State Diagram

The first state is the initialize state and the state machine remains in this state till the function is activated by the software program. The function then moves to the handshake state where it verifies that another function is not being executed and that it has control memory. It then writes zeros to all locations in each internal SMM memory. Next, it resets the pointers to their initial positions. It then resets all the functions to their initialize state and then returns to its own initialize state.

3.2 Implementation

In order to implement the smart memory module, the Xilinx PLB IPIF bus interface was used. The Xilinx IPIF is a bi-directional interface between a user's custom hardware and the PLB 64-bit bus standard. Figure 3.22 shows the block diagram of the IPIF structure provided by Xilinx.

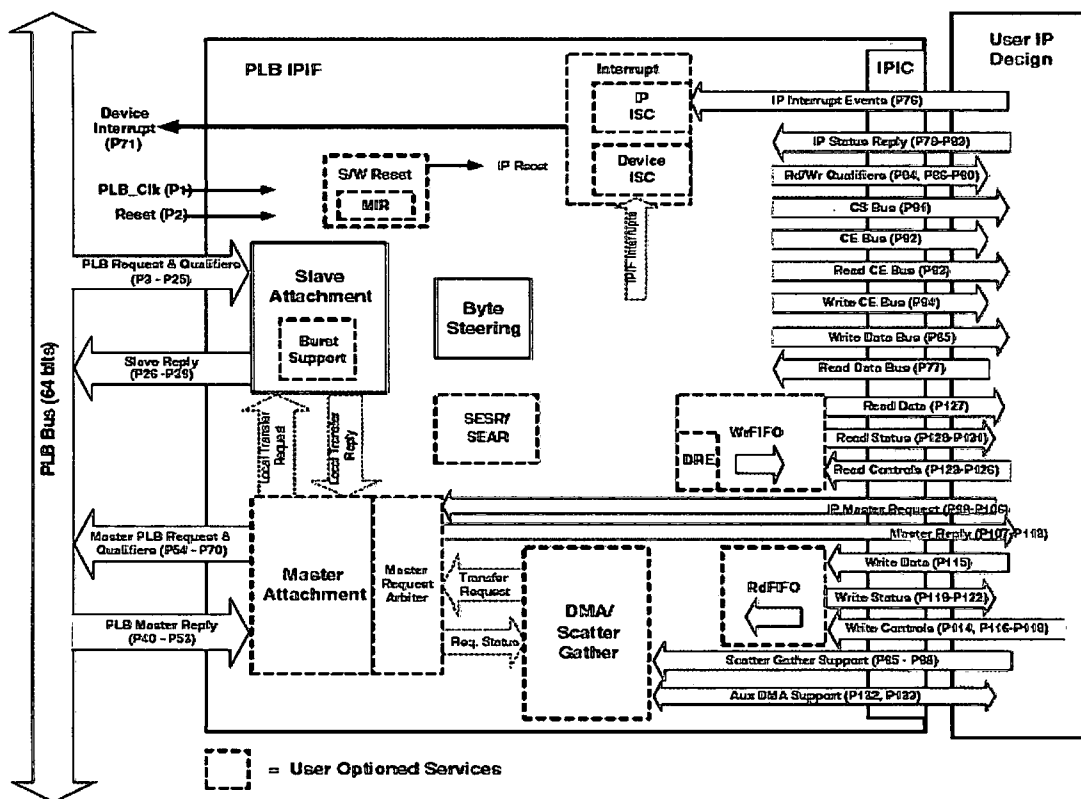


Figure 3.22: PLB IPIF Block Diagram [9]

The IPIF is attached to the custom SMM hardware and a software driver developed to allow data to be sent to the SMM. The IPIF handles the addressing and

interconnections to the bus allowing for focus on developing the hardware and not the bus interface. Figure 3.23 shows the block diagram of the bus, IPIF, and SMM connections.

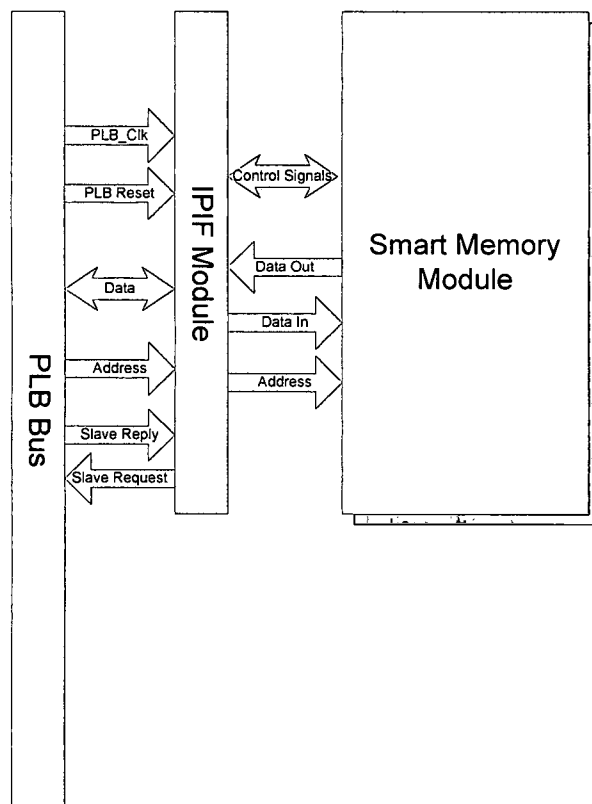


Figure 3.23: Hardware System Block Diagram

The PLB bus attaches to the IPIF module and the SMM then attaches to the IPIF and obtains data that the IPIF receives from the bus.

3.3 Summary

To the PowerPC, the SMM looks like any other hardware device on the development board and the functions were developed to use this. The functions described allow full functionality to be obtained and is based on memory mapping to the IPIF. The hardware and functions allow the SMM to be adapted and included into various software and hardware platforms. By using already available hardware core's,

such as the IPIF, the SMM can easily be adapted to other Xilinx development boards which contain PowerPC based FPGA's. In chapter 4 the internal hardware configuration of the SMM will be described and the software driver will be described in chapter 5.

Chapter 4

Garbage Collector Hardware Design

The main function of the garbage collector is to free space in memory by removing objects that are considered to be dead. The design of the hardware garbage collector chosen is based on the Baker Copy Collector. It includes ideas taken from the reference counting garbage collector, creating a hybrid system. This merging of multiple designs was done due to limitations in the ML403 development board and Viretex-4 FPGA used, as well as complexity issues that were more easily implemented and more efficient using different methods. This chapter is dedicated to discussing the design of the collector, how it updates the number of free objects, and the handshake between the interface and the collector.

4.1 Garbage Collector

The garbage collector has the most important task in the SMM design. It manages the amount of free memory space in the system. This is done by scanning memory for objects which are considered to be dead and removing them. The idea of using two memory spaces is taken from the Baker Copy Collector. The ToSpace memory is used for new object allocation and to copy live objects from FromSpace to the ToSpace memory. Figure 4.1 shows the coping scheme of the copy collector.

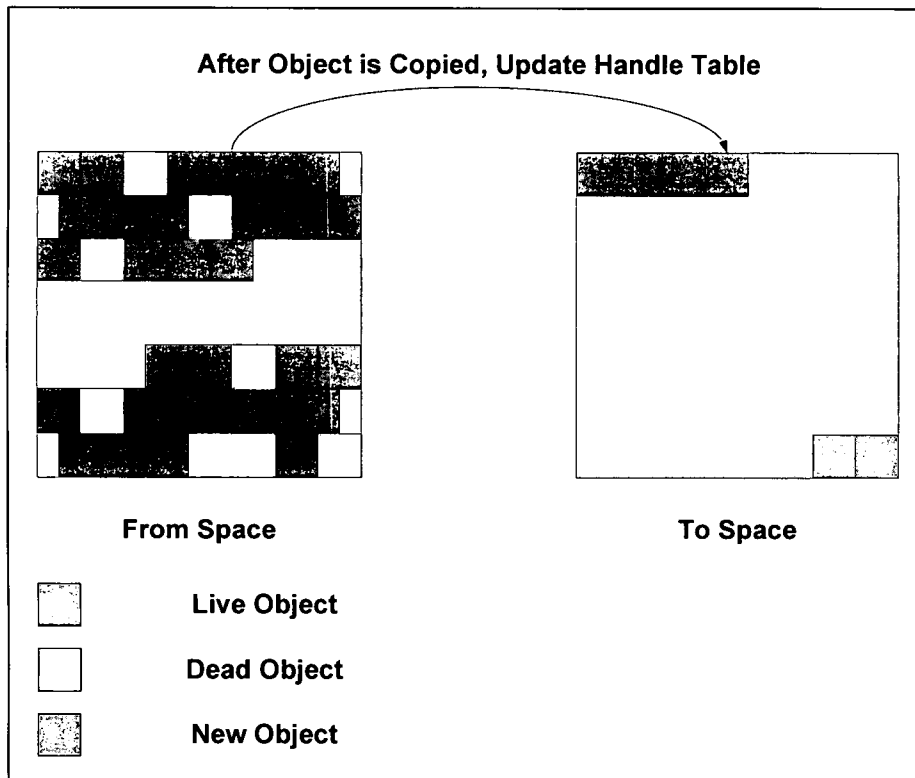


Figure 4.1 Basic Copy Collection Scheme

Figure 4.1 shows that data is copied from FromSpace to ToSpace and is placed at the top of the memory. As objects are copied, the memory is defragmented removing any holes created by dead objects. The design adds new objects at the end of the memory space as not interference with the garbage collector. This is identical to the method applied by the Baker Copy Collector. The Baker Copy Collector and the SMM are also activated in similar fashion. They use pointers to calculate the amount of free space available in memory and once this value passes a predefined threshold, the garbage collector is activated. The threshold value is determined based on how fast memory is being allocated and must be set at the time the SMM hardware is compiled. Figure 4.2 shows how the amount of free memory is calculated using two pointers in ToSpace Memory.

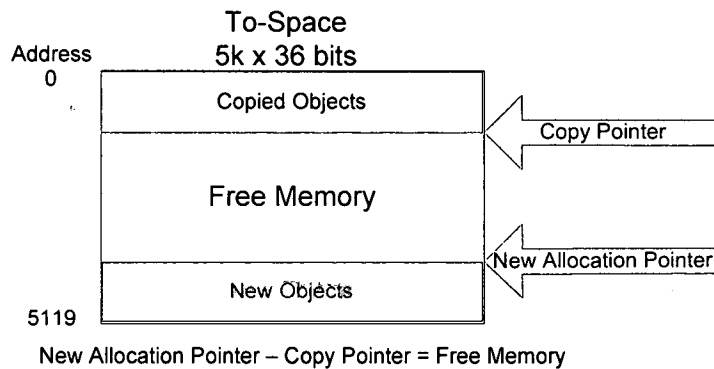


Figure 4.2 ToSpace Free Memory Calculation

Figure 4.2 shows that the amount of free memory is calculated based on the difference between the New Allocation Pointer and the Copy Pointer. This value is then compared against the predefined threshold value. If the threshold value is greater than the amount of free memory then the garbage collector is activated. However, there is an issue that could arise. If all the objects are alive, the collector could just copy to the new ToSpace and see that it is over threshold and activate again. To counter this, the collector only compares the amount of free memory against the threshold when memory is allocated. If there is enough free memory, the object is allocated and the collector is activated. If there is not enough memory to allocate the object, an error code is returned and the collector activates. Once the collector is activated, it does a number of steps to prepare the system to copy live objects and prepare for new object allocations. The first step is to make sure the collector has control of all memory. Once the handshake verifies the collector has control, the collector flips the object memory spaces so that the old ToSpace becomes the new FromSpace and the old FromSpace becomes the new ToSpace. At the same time it sets up the Copy Pointer and the New Allocation Pointer to the correct addresses in the new ToSpace. It then zeros out the is-copied and to-copied memories to prepare it so that objects to copy and that have been copied can be marked. This is a critical step, since if the memories were not cleared data that should not be copied might be copied and data that should be copied could be skipped. Next, the system starts at address zero in the new FromSpace and locates the root of the first object ID and checks if the reference count is not zero. If the reference count is not zero, it places a one in the to-copy memory at the address equivalent to the object ID of the current object. It

then checks if the object is an array or data object. If it is an array, it copies the data in copied one way and if it is an object, it is copied another way. Figure 4.3 shows the different between an array and an object.

<u>Data Object</u>		<u>Array</u>	
Address		Address	
0	Root	0	Root
1	Data/Pointer	1	Misc Data
2	Data/Pointer	2	Misc Data
3	Data/Pointer	3	Data/Pointer
4	Data/Pointer	4	Data/Pointer
5	Data/Pointer	5	Data/Pointer
6	Data/Pointer	6	Data/Pointer
7	Data/Pointer	7	Data/Pointer

Figure 4.3 Data Object and Array Storage Structure

Figure 4.3 shows the difference in of how objects and arrays are allocated in memory. If the object to be copied is determined to be an object, the root location is copied to the new ToSpace and the handle table is updated to point to this new memory location. At the same time, it also uses the structure ID stored in the root location to obtain the pointer mask and size associated with the object. Then, the collector begins copying the rest of the object based on the size obtained. As it copies each memory location, it checks the pointer mask to determine if the data is a pointer to another object or is just data. If it is just data, it copies it directly to the new ToSpace. If the data is a pointer the data will be an object ID to another object in memory. The collector will write a one into the to-copy memory at the address equivalent to the object ID contained in the pointer. This process then continues till the whole object is copied. It then writes a one into the is-copied memory marking the object as being completely copied. If it is determined that an array is to be copied the root location is copied and the address stored in the handle table is updated. The size of the array is obtained from the root of the array, as well as if the data in the array is pointers or data. It then copies the first two locations of the array as is and then begins copying the rest of the array. If the array contains all pointers the collector updates the to-copy memory as each location in the array is copied. Otherwise, the data it is copied directly to the new ToSpace. It then writes a one into the

is-copied memory to mark the array as copied. This cycle repeats till all allocated objects are checked and all objects with reference counts other than zero are copied. The collector then scans the to-copy and is-copied memory verifying that the values in both are identical. Figure 4.4 shows the process of matching the to-copy to the is-copied memory.

	<u>To-Copy Memory</u>			<u>Is-Copied Memory</u>	
Address	0	1	Copied	1	1
	1	1	Copied	1	1
	2	1	Needs Copying	0	0
	3	0	Dead Object	0	0
	4	1	Copied	1	1
	5	0	Dead Object	0	0
	6	0	Dead Object	0	0
	7	1	Needs Copying	0	0

Figure 4.4 Comparing the to-copy Memory to the is-copied Memory

Figure 4.4 shows that the collector scans comparing the to-copy memory to the is-copied memory and what it determines based on the values stored in them. If the values in both memories are one the objects have been copied and if the values are both zero the object is considered to be dead. Lastly, if the value in to-copy is one and the value is zero in is-copied, the object still remains to be copied. When this happens, the collector then copies the object or array in the same fashion as previously described. This case arises when an object is considered to be alive due to another live object referencing it. This process is repeated until the to-copy and is-copied memory has identical values for every address.

4.2 Handle Table Updating

Once all the objects have been copied to the new ToSpace, the collector then activates a process to update the handle table to free any objects that are considered to be dead. To complete this task, an internal handshake inside the collector gives control of the to-copy and is-copied memories to the handle table update process. Figure 4.5 shows

the updating of the free objects and handle table memories based on the values in the to-copy and is-copied memories.

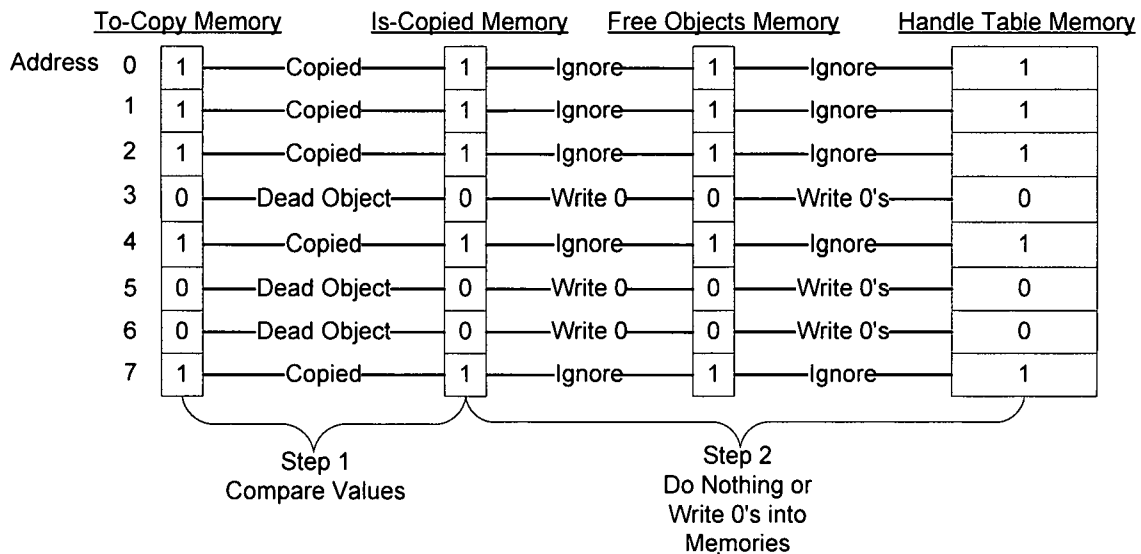


Figure 4.5 Updating Free Objects and Handle Table Memory

First, the to-copy and is-copied memories are scanned for any locations that contain zero for both values. Once found, the handle table memory gets zeros written into it at the location equivalent to the addresses location where the zero values were found. At the same time, the free objects memory gets a zero written into it allowing the system to know that this specific object ID is free and can be used for allocation of a new object or array. This is done for all 1024 locations in the to-copy and is-copied memories. Once complete, the collector has completed and releases control of all the memories required by the interface.

4.3 Interface/Collector Handshake

In order to successfully have the interface and collector access the same memory a handshaking method is used. The handshake works by handling the routing of data in and out of the both the interface and the garbage collector. Figure 4.6 shows the basic design behind the handshake system.

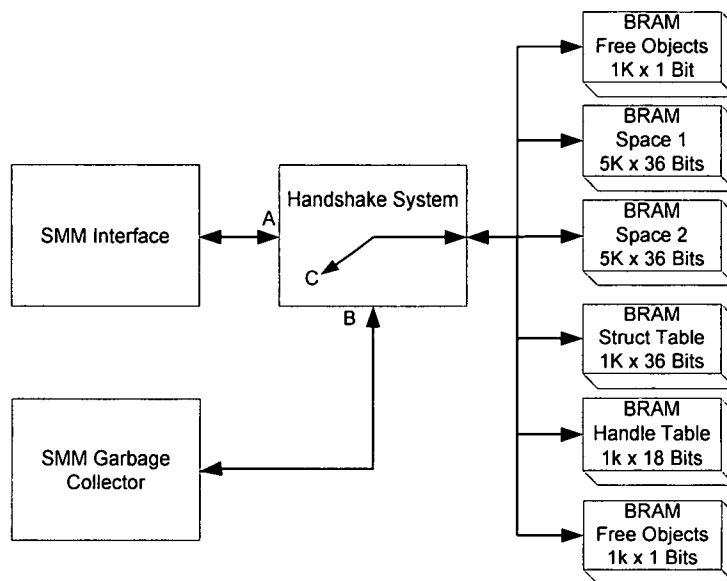


Figure 4.6 Handshake Basic System Design

Figure 4.6 shows that if points A and C are attached control is given to the interface and if points B and C are attached control is given to the garbage collector. This design is simple in nature, but does introduce some complexity and questions. If the interface or the garbage collector requests control of memory, no issues arose. However, if both request access to memory at the same time one must be given priority over the other. This is in order to not create data collisions on the memory inputs and outputs. In the SMM, the interface was given priority over the garbage collector in order to reduce delays to the executing program. Also, once the interface or the collector does grab control of the memory it does not release control until it completes its current task.

4.4 Summary

The design of the garbage collector was created by using ideas from both the Baker Copy Collector and the Reference Counting Collector, with modifications to suit hardware implementation. Modifications were also done to unforeseen hardware limitations. The idea of allowing the collector to fully execute and copy all live objects was initially done to reduce complexity. In an ideal situation, more handshaking would be done between the interface and collector to allow the collector to pause and allow the interface to take control as needed causing the collector to execute between interface calls. Overall, the garbage collector worked as desired and with the use of the basic

handshaking done the memories where shared without corrupting data or causing data collisions on the memory inputs and outputs.

Chapter 5

Linux to Hardware Java Interface

The previous two chapters discussed the hardware side of the SMM system. This chapter will discuss the software interface design. The software interface is used to connect a software program to the SMM hardware. This allows support of all ten hardware interface functions described in chapter 3 as well as one additional function. The design uses already existing Linux commands to develop the driver using the C++ programming language. It then uses the Cygnus Native Interface (CNI) to create a java wrapper for the C++ interface code. Both the C++ and the CNI wrapper will both be described in greater detail in the rest of this chapter.

5.1 *Linux Custom Driver*

The purpose of the Linux Custom Driver is to allow a software program to connect and utilize the SMM hardware. It is written in C++ allowing for utilization of already existent Linux functions to be used to read and write data to the processor local bus (PLB). This allows the creation of the driver less complex, due to not having to recreate functions that are already tightly coupled to the Linux system.

The Linux Custom Driver also allows the programmer to call these functions. The functions handle the formatting of data to allow proper sending and receiving to the SMM hardware. Figure 5.1 shows the Linux custom driver interface with the whole system.

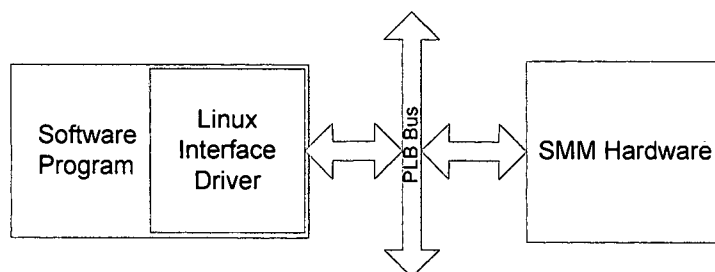


Figure 5.1 Linux Custom Driver Interface to SMM System

There are four Linux functions that are utilized by the custom driver. Those functions are *ioremap*, *ioremap*, *in_le32*, and *out_le32*. Functions, *ioremap* and *ioremap*, setup and remove memory maps that allow access to the SMM hardware. The functions *in_le32* and *out_le32* allow the sending and receiving of data from the SMM Hardware. These four functions as well as each of the eleven interface functions will be described in the following sections.

5.1.1 ioremap Linux Function

The *ioremap* function is a Linux supplied function to map bus memory into CPU space, making devices on the bus accessible to the user. The bus address assigned to the device and the address size is passed the *ioremap* function. *Ioremap* then performs a platform specific sequence of operations to make the bus memory CPU accessible. It then returns a virtual address used to access the device. The address as returned is not fully used, but is the base address of the device and the user must apply offsets to read and write to different inputs and outputs attached to the bus. Figure 5.2 shows the C code that implements the *ioremap* function.

```

volatile unsigned long * ioremap(unsigned long physaddr, unsigned size)
{
    static int axs_mem_fd = -1;
    unsigned long page_addr, ofs_addr, reg, pgmask;
    void* reg_mem = NULL;
    // looks like mmap wants aligned addresses?
    pgmask = getpagesize()-1;
    page_addr = physaddr & ~pgmask;
    ofs_addr = physaddr & pgmask;
    // Don't forget O_SYNC, esp. if address is in RAM region.
    // Note: if you do know you'll access in Read Only mode,
    // pass O_RDONLY to open, and PROT_READ only to mmap
    if (axs_mem_fd == -1) {
        axs_mem_fd = open("/dev/mem", O_RDWR|O_SYNC);
        if (axs_mem_fd < 0) {
            perror("AXS: can't open /dev/mem");
            return NULL;
        }
    }
    // memory map
    reg_mem = mmap(
        (caddr_t)reg_mem,
        size+ofs_addr,
        PROT_READ|PROT_WRITE,
        MAP_SHARED,
        axs_mem_fd,
        page_addr
    );
    if (reg_mem == MAP_FAILED) {
        perror("AXS: mmap error");
        close(axs_mem_fd);
        return NULL;
    }

    reg = (unsigned long )reg_mem + ofs_addr;
    return (volatile unsigned long *)reg;
}

```

Figure 5.2 ioremap Function Code

5.1.2 iounmap Linux Function

The *iounmap* function is a Linux supplied function to unmap bus memory that has been placed CPU space. The virtual address assigned by *ioremap* and the address size is passed the *iounmap* function. *Iounmap* then performs a platform specific sequence of operations to remove the device from CPU Space. Once *iounmap* is called the device is no longer accessible by the CPU. Figure 5.3 shows the C code that implements the *iounmap* function.


```

int iounmap(unsigned long start, unsigned long length)
{
    unsigned long ofs_addr;
    ofs_addr = start & (getpagesize()-1);

    // do some cleanup when you're done with it
    return munmap((long*)start-ofs_addr, length+ofs_addr);
}

```

Figure 5.3 iounmap Function Code

5.1.3 in_le32 Linux Function

The *in_le32* function is a Linux supplied function to read data from a device attached to the bus. The virtual address plus the offset associated with the port to read from is passed to the *in_le32* function. The function then handles the bus interfacing and the reading of the data from the desired device. The function is implemented using inline assembly to make it more efficient. Figure 5.4 shows the C code that implements the *in_le32* function.

```

extern inline unsigned in_le32(volatile unsigned long *addr)
{
    unsigned ret;
    __asm__ __volatile__ ("lwbrx %0,0,%1;\n"
                          "twi 0,%0,0;\n"
                          "isync" : "=r" (ret) :
                          "r" (addr), "m" (*addr));
    return ret;
}

```

Figure 5.4 in_le32 Function Code

5.1.4 out_le32 Linux Function

The *out_le32* function is a Linux supplied function to send data to a device attached to the bus. The virtual address of the device plus an offset and data is passed to the *out_le32* function. The function then handles the bus interfacing and the sending of the data to the desired device. The function is implemented using inline assembly to make it more efficient. Figure 5.5 shows the C code that implements the *out_le32* function.

```
extern inline void out_le32(volatile unsigned long *addr, long val)
{
    __asm__ __volatile__ ("stwbrx %1,0,%2; eieio" : "=m" (*addr) :
                          "r" (val), "r" (addr));
}
```

Figure 5.5 out_le32 Function Code

5.1.5 New Structure Definition

The *New Structure Definition* function allows a programmer to send appropriate data to the SMM Hardware to store the definition in the structure table. Figure 5.6 shows the *New Structure Definition* C++ software code.

```
unsigned long sendStructDef(unsigned long size, unsigned long ptr_mask)
{
    out_le32(r1_upper, size);
    out_le32(r0_lower, ptr_mask);
    out_le32(r0_upper, ptr_mask+1);
    out_le32(r3_upper, SENDSTRUCTDEF_CONTROL);
    Status = in_le32(r6_upper);

    while(Status != SENDSTRUCTDEF_STATUS_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    unsigned long return_val = in_le32(r5_upper);
    out_le32(r3_upper, INITIALIZE_STATES);
    while(Status != INITIALIZE_STATES_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    return return_val;
}
```

Figure 5.6 New Structure Definition Function Code

The *New Structure Definition* function is passed a size and pointer mask. The data is then sent to the SMM Hardware and then the required control value to activate the function in hardware is sent. The software interface then waits for a status value to be returned letting the software function know the hardware has completed. The function then reads in the assigned structure ID and then sends the initialize states control value and waits for it to complete. It then returns the obtained structure ID to the program.

5.1.7 New Object

The *New Object* function allows a programmer to send appropriate data to the SMM Hardware to create a new object to hold data or pointers. Figure 5.7 shows the *New Object* C++ software code.

```
unsigned long New(unsigned long struct_ID)
{
    out_le32(r2_upper, struct_ID);
    out_le32(r3_upper, NEW_CONTROL);
    Status = in_le32(r6_upper);
    while(Status != NEW_STATUS_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    unsigned long return_val = in_le32(r5_lower);
    out_le32(r3_upper, INITIALIZE_STATES);
    while(Status != INITIALIZE_STATES_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    return return_val;
}
```

Figure 5.7 New Object Function Code

The *New Object* function is passed a structure ID previously obtained from the sending of the *New Structure Definition* function. The data is then sent to the SMM Hardware and then the required control value to activate the function in hardware is sent. The software interface then waits for a status value to be returned letting the software function know the hardware has completed. The function then reads the assigned object ID. It sends the initialize states control value and waits for it to complete. It then returns the obtained object ID to the program.

5.1.8 New Array

The *New Array* function allows a programmer to send appropriate data to the SMM Hardware to create a new array to hold either data or pointers. Figure 5.8 shows the *New Array* C++ software code.

```

unsigned long newArray(unsigned long size, unsigned long arePointers)
{
    out_le32(r4_lower, arePointers);
    out_le32(r1_upper, size);
    out_le32(r3_upper, NEWARRAY_CONTROL);
    Status = in_le32(r6_upper);
    while(Status != NEWARRAY_STATUS_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    unsigned long return_val = in_le32(r7_lower);
    out_le32(r3_upper, INITIALIZE_STATES);
    while(Status != INITIALIZE_STATES_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    return return_val;
}

```

Figure 5.8 New Array Function Code

The *New Array* function is passed a size for the array and if all locations in the array are pointers or data. This data is then sent to the SMM Hardware and then the required control value to activate the function in hardware is sent. The software interface then waits for a status value to be returned letting the software function know the hardware has completed. The function then reads in the assigned object ID and then sends the initialize states control value and waits for it to complete. It then returns the obtained object ID to the program.

5.1.9 Clone

The *Clone* function allows a programmer to send appropriate data to the SMM Hardware to create an exact copy of an array or object already allocated in memory. Figure 5.8 shows the *Clone* C++ software code.

```

unsigned long cloneObject(unsigned long obj_ID, unsigned long struct_ID)
{
    out_le32(r2_lower, obj_ID);
    out_le32(r3_upper, CLONEOBJ_CONTROL);
    Status = in_le32(r6_upper);
    while(Status != CLONEOBJ_STATUS_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    unsigned long return_val = in_le32(r7_upper);
    out_le32(r3_upper, INITIALIZE_STATES);
    while(Status != INITIALIZE_STATES_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    return return_val;
}

```

Figure 5.9 Clone Function Code

The *Clone* function is passed an object ID which is a pointer to the object or array to be cloned. This data is then sent to the SMM Hardware and then the required control value to activate the function in hardware is sent. The software interface then waits for a status value to be returned letting the software function know the hardware has completed. The function then reads in the assigned object ID and then sends the initialize states control value and waits for it to complete. It then returns the obtained object ID to the program.

5.1.10 Put Object Data

The *Put Object Data* function allows a programmer to send appropriate data to the SMM Hardware to place data into an object or array. Figure 5.10 shows the *Put Object Data* C++ software code.

```

void putObjectData(unsigned long obj_ID,unsigned long offset,unsigned long
data)
{
    out_le32(r2_lower, obj_ID);
    out_le32(r3_lower, offset);
    out_le32(r1_lower, data);
    out_le32(r3_upper, PUTOBJECTDATA_CONTROL);
    Status = in_le32(r6_upper);
    while(Status != PUTOBJECTDATA_STATUS_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    unsigned long test_val = in_le32(r6_lower);
    out_le32(r3_upper, INITIALIZE_STATES);
    while(Status != INITIALIZE_STATES_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    return;
}

```

Figure 5.10 Put Object Data Function Code

The *Put Object Data* function is passed an object ID, an offset, and the data to be placed in the object or array. The object ID is used to determine which object or array to place the data and the offset is used to determine in what location within in the object or array the data should be placed. This data is then sent to the SMM Hardware and then the required control value to activate the function in hardware is sent. The software interface then waits for a status value to be returned letting the software function know the hardware has completed. It then sends the initialize states control value and waits for it to complete. It then returns control to the rest of the program.

5.1.11 Get Object Data

The *Get Object Data* function allows a programmer to get appropriate data to the SMM Hardware to get data from an object or array. Figure 5.11 shows the *Get Object Data C++* software code.

```

unsigned long getObjectData(unsigned long obj_ID, unsigned long offset)
{
    out_le32(r2_lower, obj_ID);
    out_le32(r3_lower, offset);
    out_le32(r3_upper, GETOBJECTDATA_CONTROL);
    Status = in_le32(r6_upper);
    while(Status != GETOBJECTDATA_STATUS_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    unsigned long return_val = in_le32(r6_lower);
    out_le32(r3_upper, INITIALIZE_STATES);
    while(Status != INITIALIZE_STATES_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }

    return return_val;
}

```

Figure 5.11 Get Object Data Function Code

The *Get Object Data* function is passed an object ID and an offset. The object ID is used to determine which object or array to get the data from and the offset is used to determine in what location within the object or array the data should be obtained from. This data is then sent to the SMM Hardware and then the required control value to activate the function in hardware is sent. The software interface then waits for a status value to be returned letting the software function know the hardware has completed. The function then reads in the returned data and sends the initialize states control value and waits for it to complete. It then returns the obtained data to the program.

5.1.12 Bump and Decrement Reference Count

The *Bump Reference Count* and *Decrement Reference Count* functions allow a programmer to get appropriate data to the SMM Hardware to increment or decrement the reference count of an object or array. Figure 5.12 shows the bump reference count C++ software code and figure 5.13 shows the C++ software code for decrement reference count.

```

void bumpRefCount(unsigned long obj_ID)
{
    out_le32(r2_lower, obj_ID);
    out_le32(r3_upper, BUMPREFCOUNT_CONTROL);

    Status = in_le32(r6_upper);
    while (Status != BUMPREFCOUNT_STATUS_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    out_le32(r3_upper, INITIALIZE_STATES);
    while(Status != INITIALIZE_STATES_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }

    return;
}

```

Figure 5.12 Bump Reference Count Function Code

```

void decRefCount(unsigned long obj_ID)
{
    out_le32(r2_lower, obj_ID);
    out_le32(r3_upper, DECREFCOUNT_CONTROL);

    Status = in_le32(r6_upper);
    while (Status != DECREFCOUNT_STATUS_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    out_le32(r3_upper, INITIALIZE_STATES);
    while(Status != INITIALIZE_STATES_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }

    return;
}

```

Figure 5.13 Decrement Reference Count Function Code

Both the *Bump Reference Count* and *Decrement Reference Count* functions are passed an object ID. The object ID is used to determine which object or array to increment or decrement the reference count in. This data is then sent to the SMM hardware and then the required control value to activate the function in hardware is sent. The software interface then waits for a status value to be returned letting the software function know the hardware has completed. The function then sends the initialize states control value and waits for it to complete.

5.1.13 Get Reference Count

The *Get Reference Count* function allows a programmer to get appropriate data to the SMM Hardware to obtain the current reference count for an object or array. Figure 5.14 shows the *Get Reference Count* C++ software.

```
unsigned long getrefcount(unsigned long obj_ID)
{
    out_le32(r2_lower, obj_ID);
    out_le32(r3_upper, GETREFCOUNT_CONTROL);
    Status = in_le32(r6_upper);
    while (Status != GETREFCOUNT_STATUS_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    unsigned long return_val = in_le32(r8_upper);
    out_le32(r3_upper, INITIALIZE_STATES);
    while (Status != INITIALIZE_STATES_COMPLETE)
    {
        Status = in_le32(r6_upper);
    }
    return return_val;
}
```

Figure 5.14 Get Reference Count Function Code

The *Get Reference Count* function is passed an object ID. The object ID is used to determine which object or array to obtain the reference count from. This data is then sent to the SMM Hardware and then the required control value to activate the function in hardware is sent. The software interface then waits for a status value to be returned letting the software function know the hardware has completed. The function then reads the returned reference count. The function then sends the initialize states control value and waits for it to complete. It then returns the obtained reference count to the executing program.

5.1.14 Get Garbage Collection Count

The *Get Garbage Count* function allows a programmer to get appropriate data to the SMM Hardware to obtain the current number of times the garbage collector has been activated. This is done by reading a single register from the hardware SMM. The register is always available and gets updated every time the garbage collector runs. The function call to obtain the count is one step using the `in_le32` function and the correct address to the register storing the count. This function was implemented for testing

purposes since the garbage collector is transparent to the software. This function is not required for actual implementation of the SMM.

5.1.15 Reset

The *Reset* function allows a programmer to get appropriate data to the SMM Hardware to initiate a hard reset inside the SMM hardware. This is done by writing a specific control value to the SMM hardware. It then waits for a status value to be returned stating that the reset is complete. During actual execution of a program, the reset function would not be used since one would not want to delete all the stored data. This function was implemented for testing purposes to allow the test suite to execute different tests one right after another without rebooting the whole system.

5.2 CNI Interface Code

Since the main goal is to interact with the SMM hardware using JAVA code, some of the C++ interface code had to have a CNI wrapper placed around it. CNI allows C++/Java integration. Since all the SMM defined interface functions utilize the four main Linux supplied functions written in C, only these needed to have a CNI wrapper placed around them. The functions that used the CNI wrapper are *ioremap*, *iounmap*, *in_le32*, and *out_le32*.

Using CNI to implement the C Linux functions, the functions are turned into JAVA objects that can be used within a main JAVA program. Since the other software interface functions shown in section 5.1 above were created using these four functions, the rest of the interface functions were rewritten in JAVA and used the CNI created objects to do the actual bus interfacing to the SMM hardware. Figure 5.15 shows the interconnection of the Software program and interface to the CNI interface.

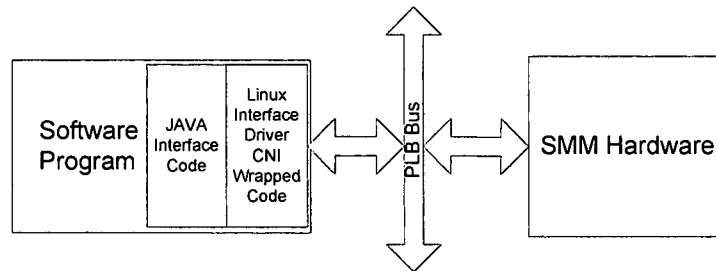


Figure 5.15 Linux Custom Driver with CNI Interface to SMM System

5.3 Summary

By creating C language version for each interface function, it allowed early testing to be done on the SMM hardware. It also allowed additional functions to be added as needed for system debugging. This is because C++ is natively supported by the Xilinx toolset and Linux. Once the system had been tested and verified to be working, it was then ported to JAVA and integrated into the full environment. This made debugging easier for the JAVA code, since the hardware was known to work and if any issues were found, they were contained in the software. Also, by using the CNI it allowed an easy method of talking to the SMM hardware from the JAVA code using the already existent Linux functions, since they have already been proven to work. Lastly, the use and design of the interface functions allow easy integration of the SMM into different programs with little work done by a programmer.

Chapter 6

Implementation Results

To test the SMM, a test suite was created to exercise various features. Each case was written to run on either the SMM hardware or the software equivalent system. This allowed a greater comparison of performance increase to be shown as well as deterministic behavior. The test suite, as well as the results obtained will be discussed in the following sections of chapter 6.

6.1 *Test Suite Description and Results*

The test suite was designed to test full functionality of the SMM system. The tests are designed to try and break the SMM system and cause failure in such a way that the reason for the failure could be identified. The test cases test the system functionality, replicate real situations that could be encountered, and time various parts of the system. Each test case is a set of sub tests. Each test prints out either a pass or failure for each test sub test inside in each test case. This was done to limit the amount of data printed out and making verification easier. The test code handles the verification of each test and if the test fails the reason for failure is shown.

The test suite was written in JAVA and is compiled using the open source GCJ Java Compiler. Each test case was run on the hardware version of the SMM and the software equivalent version of the SMM. Each test case will be explained and the results show in sections 6.1.1 through 6.1.12. The JAVA code for the test suite can be seen in Appendix B.

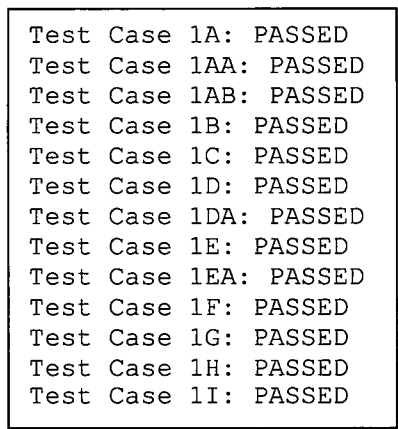
6.1.1 Test Case 1 – Interface and Initial Garbage Collector Testing

Test case 1 is designed to test the interface functions of the SMM. It is comprised of thirteen different sub tests that exercise the interface functions and test them for proper functionality. Each sub test builds upon the previous sub tests executed before it.

In Test Case 1A a *New Structure Definition* is created and then based on this structure an object is created. It then does a *Get Reference Count* on this object to verify that during creation of the object the object has a reference count of one. If reference count returned from the SMM is one the test passed. Next, in Test Case 1AA, data is placed in the first location in the object created during test 1A. It then obtains the data back from the SMM verifying that the data placed in the object equals the data returned from the object. If the data is equal then the test has passed. The next test case, Test Case AB, two large arrays are created in order to pass the threshold level set to activate the garbage collector. The count for the number of garbage collection cycles is obtained and verified to be one. If the count is one, the test passes. Next, in Test Case 1B, the Object ID of the first object created during Test Case 1A is compared to the object ID of the first array created during Test Case 1AB. If the Object ID's are not the same, the test passes. Test Case 1C and 1D place data into the two arrays created in Test Case 1AB. The data is obtained from the two arrays and compared to the data sent to them. Test Case 1C is considered to pass if the data placed in the first array is the same as the data returned from it. This is also the same for Test Case 1D, but for the second array. The next interface functions to be tested are *Bump Reference Count* and *Decrement Reference Count* within an array. In the previous test cases, two arrays have been created and data placed in them one of these arrays is used for Test Case 1DA. The *Bump Reference Count* command is sent, which increases the reference count by one in the array. The reference count and the data that was previously stored in the array are checked. If the reference count returned is now two, Test Case 1DA passes. If the data that was previously stored in the array is also still correct and has not been corrupted then Test Case 1E passes. Next, we test *Decrement Reference Count* in the same manner as in Test Case 1EA. The *Decrement Reference Count* is sent to the array in which the reference count was bumped previously to two. The reference count is checked and should be one again, if it is Test Case 1EA passes. If the stored data is still not corrupted then Test Case 1F passes also. Test 1G tests the activation of the garbage collector when a reference count is decremented to zero. The *Decrement Reference Count* is sent to the first array previously created making the reference count equal to zero. This will force the collector to activate, since memory threshold has still been exceeded, treating the array as garbage.

If the garbage collector count is now at two Test Case 1G passes. If the data that was in the array is attempted to be accessed and the value that was stored is not returned then Test Case 1H passes. This helps to verify that dead objects are being collected. The last test, Test Case 1I, tests the *Clone* function. In Test Case 1A an object was created and data placed in it. Here we clone that object and verify that the data that was stored in the original object is equal to the data in the newly created cloned object. If the data is equal then Test Case 1I passes.

Each of the sub tests cases in Test Case 1 are executed sequentially in the order they were explained. Based on the criteria for each test case to pass or fail, Figure 6.1 shows the screen shot of the output produced by test case 1.



```
Test Case 1A: PASSED
Test Case 1AA: PASSED
Test Case 1AB: PASSED
Test Case 1B: PASSED
Test Case 1C: PASSED
Test Case 1D: PASSED
Test Case 1DA: PASSED
Test Case 1E: PASSED
Test Case 1EA: PASSED
Test Case 1F: PASSED
Test Case 1G: PASSED
Test Case 1H: PASSED
Test Case 1I: PASSED
```

Figure 6.1: Screen Capture for Test Case 1 Outputs

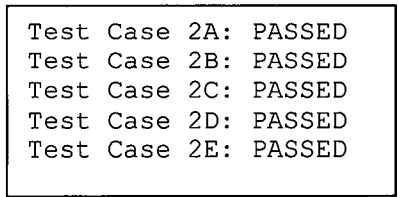
As each test case executes, it prints out if it passed or failed the test. If a test case was to fail, the reason for failing would also be shown. Since all the test cases in Test Case 1 display passed next to them, it verifies that the interface functions behave in the desired fashion.

6.1.2 Test Case 2 – Pointer Mask and Data Offset Testing

In Test Case 2 the pointer mask in structures, pointer types in arrays, garbage collection with pointers, and the use of the offset to place and obtain data is tested. There are 5 sub tests in Test Case 2 to test each of these situations. Again, as in Test Case 1 each of the sub tests build upon the previous one to complete all the sub tests.

In Test Case 2A a structure with a complex pointer mask is created and two objects based on this structure are created. It then places data in all the non-pointer locations in the first object. It places a pointer to the first object in pointer locations in object two. It then checks that all the data in the first object is equal to what was sent to and if it is then Test Case 2A passes. Next, the reference count on the first object is decremented to zero in Test Case 2B. Once the reference count has been decremented a large array is allocated to force garbage collection. Once garbage collection has completed, it is verified that the first object with a reference count of zero is kept alive and copied from FromSpace to ToSpace due to the reference located in object two. If all the data in object one is still correct then the pointers in object two are removed and Test Case 2B passes. With the pointer now removed, in Test Case 2C, the reference count of the large array previously created is decremented to zero and a small array is created to force garbage collection. With no pointers to object one, object one should be treated as being dead and should be collected. This is verified by checking if object one still exists. If the object no longer exists and then Test Case 2C passes. In Test Case 2D it is verified that an object that is double referenced is copied from FromSpace to ToSpace and not collected by the garbage collector. This is done by creating three different objects of the same size and same pointer masks. In object one data is placed in a non-pointer location. In object two a pointer to object one is placed and in object 3 a pointer to object 2 is placed. Then the reference counts of object one and two are decremented to zero to label them as dead. A large array is then created to force garbage collection. Once garbage collection is complete due to object three pointing to object two, object two should be copied, and likewise object one should be copied now since object two has been copied. This is verified by checking that the data in object one is still correct using the *Get Object Data* function. If the data is correct then Test Case 2D passes. The last test case, Test Case 2E, tests the creation of an object larger than 64 words longs. This test is done since the pointer mask is a maximum of 64 bits long and all locations outside the first 64 words in an object can only be data. The test creates an object of size 100 and places data in location 99 and then verifies that the data is placed and returned correctly. If the data returned equals the data placed than Test Case 2E passes.

As each of the tests in Test Case 2 is executed they return a pass or fail based on the criteria previously explained for each of the sub tests. Figure 6.2 shows the screen shot of the output produced by test case 2.



```
Test Case 2A: PASSED
Test Case 2B: PASSED
Test Case 2C: PASSED
Test Case 2D: PASSED
Test Case 2E: PASSED
```

Figure 6.2: Screen Capture for Test Case 2 Outputs

If all the tests cases display passes as they do in Figure 6.2 it verifies that the Pointer Mask and the placing and obtaining of data based on offset behaves as expected. If a test case was to fail the reason for failing would also be shown.

6.1.3 Test Case 3 – Multi-Thread Testing

For Test case 3 the SMM is tested in a multi threaded environment. Unlike Test Case 1 and 2 there are no sub tests in Test Case 3. The test code creates multiple threads and attempts to complete each one in the order they are created. Once a thread is executed it locks the SMM till completion and the next thread cannot start till the previous one completes. Test Case 3 uses test cases 1 and 2 creating threads that execute each one of them. Since each thread is just the execution of a previous test cases, Test Case 1 and 2, and they previously passed if anything fails it is due to the multi threaded setup.

When Test Case 3 is executed, each thread print outs the results for Test Case 1 and 2. Figure 6.3 shows the screen shot of the output produced by test case 3.


```
Test Case 3A: PASSED
Test Case 3B: PASSED
Test Case 3C: PASSED
Test Case 3D: PASSED
Test Case 2A: PASSED
Test Case 3A: PASSED
Test Case 3B: PASSED
Test Case 3C: PASSED
Test Case 3D: PASSED
Test Case 2A: PASSED
Test Case 3A: PASSED
Test Case 3B: PASSED
Test Case 3C: PASSED
Test Case 3D: PASSED
Test Case 2A: PASSED
Test Case 3: PASSED
```

Figure 6.3: Screen Capture for Test Case 3 Outputs

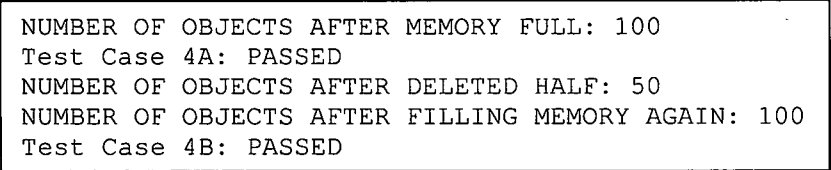
If all the threads execute successfully, the final print out shows passed for each thread as show in Figure 6.3. If a test case was to fail, the reason for failing would also be shown. This test is critical for proper functionality since the final demo environment used a highly multi threaded environment. Since all sub-tests display passed next to them, it verifies that the SMM hardware and Software Driver behave as expected in the multi threaded environment.

6.1.4 Test Case 4 – Basic Garbage Collector and Memory Bounds Testing

Test Case 4 tests the memory bounds of the SMM, as well as to test the basic functionality of the garbage collector. Each sub test exercises if the error conditions are returned when a particular situation arises and that during garbage collection that dead objects are collected. The first test, Test Case 4A, alternates between creating new objects and new arrays all of the same size. As each object or array is created it is also filled with data. This happens until the memory full error code is returned. At the same time once the threshold is met the garbage collector is activated in between each memory allocation till the memory is full. It then verifies the data in all of the objects and arrays created making sure there is no data corruption during allocation and the coping from FromSpace to ToSpace. If all the data is still verified to be correct, Test Case 4A passes. Test Case 4B uses the maximum number of objects and arrays obtained in Test Case 4A and decrements the reference count to zero on half of objects and arrays so that the

collector will treat them as dead and collect them. It then allocated objects of the same size objects and arrays used in Test Case 4A till memory is full again. This number should be the same as the maximum number of objects and arrays allocated in 4A. If the numbers are the same then Test Case 4B passes.

Since Test Case 4B builds on results from Test Case 4A they must be run sequentially and once complete it is verified if it passes or fails. Figure 6.4 shows the screen shot of the output produced by test case 4.



```
NUMBER OF OBJECTS AFTER MEMORY FULL: 100
Test Case 4A: PASSED
NUMBER OF OBJECTS AFTER DELETED HALF: 50
NUMBER OF OBJECTS AFTER FILLING MEMORY AGAIN: 100
Test Case 4B: PASSED
```

Figure 6.4: Screen Capture for Test Case 4 Outputs

As each sub-test executes it prints out if it passed or failed the test, as well as some additional data to verify functionality as can be seen in Figure 6.4. If a test case was to fail the reason for failing would also be shown. The first line shows the total number of object allocated to fill memory. It then prints out what half of the object would be based on the maximum number of objects allocated would be. It then prints out the maximum number of object allocated again to fill memory. Since all sub-tests display passed next to them and the number shown are correct, it verifies the memory allocation is working correctly as well as the garbage collector for simple objects and arrays with no pointers.

6.1.5 Test Case 5 – Null Pointer and Out of Range Pointer Testing

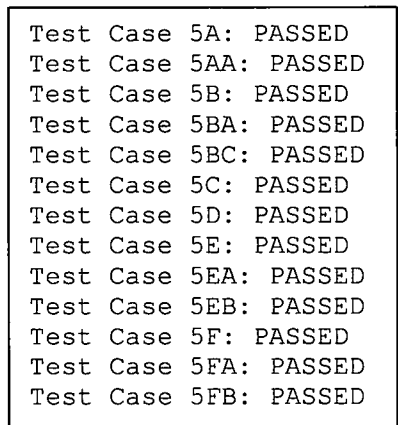
Test Case 5 is designed to test the null and out of range pointer situations. Null pointers and out of range pointers are ignored by the collector and are treated as data. This is because null pointers point to nothing and the out of range pointers can either be an error on the programmer's part or point to objects in another heap outside the SMM. There are thirteen sub test cases in Test Case 5 which exercise these two situations. The first test, Test Case 5A, just creates an object and places data in it and verifies that the data stored in the object is stored and returned correctly. If the data is stored correctly and

returned correctly then it passes successfully. Next, in test case 5AA another object is created with a pointer mask of one. The object ID for object one is stored in the pointer location in this newly created object. The reference count of object one is then decremented to zero. A large array is then created to force garbage collection. The garbage collection count is then verified to be one and if it is then Test Case 5AA passed. Next, because of the reference in object 2 to object 1, both object 1 and 2 are copied from FromSpace to ToSpace even though object 1 is technically considered dead due to a reference count of 1. The data and pointer value in objects 1 and 2 are then verified and if the data is correct then Test Case 5B passes. Test Case 5BA decrements the reference count of the large array created to force garbage collection and then places a null pointer in object two overwriting the pointer to object one previously contained in it. It then verifies the collector has been activated and has a count of two, if it does then Test Case 5BA passes. A large array is then allocated to force garbage collection in Test Case 5BC. If the collector count is now three then Test Case 5BC passes. Since there is no longer a reference to object 1 from object 2, object one should not be collected. Test Case 5C checks to make sure that the data that was in object 1 no longer exists and if this is the case then Test Case 5C passes. It then gets the data from the pointer location in object two verifying it is still zero and was not changed during collection. If the value is still zero Test Case 5D has passed which concludes the full testing of the null pointer situation.

Test Case 5E test starts the out of range pointer testing. It first creates an object, object 1, placing and paces data in it. It then retrieves the data and if the data returned equals the data sent to the object, Test Case 5E passes. Another object is then created, object 2, and it has a pointer mask of 1 associated to it. It then places a pointer to object 1 in the pointer location and decrements the reference count of object 1 to zero. A large array is then allocated to force the garbage collector to be activated. This activates the garbage collector and the value of the collector count at this time should be four. If the count is four then Test Case 5EA passes. Next, a large array is allocated to again force garbage collection in Test Case 5EB. The collector count is again verified to now be 5, if this is the case then Test Case 5EB passes. Next the data in Object 1 is verified to still exist and be correct due to the reference to object 1 from object 2. If the data still exists

and is correct in object one then Test Case 5F passes. The reference count of the large array used to previously force garbage collection is decremented to zero forcing garbage collection again in Test Case 5FA. The garbage collection count should now be 6 and if it is the then Test Case 5FA passes. The next test case, Test Case 5FB, replaces the reference in object 2 to object 1 with a value that is outside the allowed pointer value range. It then forces garbage collection by allocating a large array to go over the memory threshold value. If the garbage collector count is now at 7, then Test Case 5FB passes. It then checks to verify that the object 1 has been collected by the garbage collector and if it has been then Test Case 5G passes. The last test case, Test Case 5H, verifies that during the execution of the garbage collector that the out of range pointer was not changed. If the value remains unchanged then Test Case 5G passes.

As each of the sub tests completes the results of either passing or failing are returned. Figure 6.5 shows the screen shot of the output produced by test case 5.



```
Test Case 5A: PASSED
Test Case 5AA: PASSED
Test Case 5B: PASSED
Test Case 5BA: PASSED
Test Case 5BC: PASSED
Test Case 5C: PASSED
Test Case 5D: PASSED
Test Case 5E: PASSED
Test Case 5EA: PASSED
Test Case 5EB: PASSED
Test Case 5F: PASSED
Test Case 5FA: PASSED
Test Case 5FB: PASSED
```

Figure 6.5: Screen Capture for Test Case 5 Outputs

Since all sub-tests display passed next to them, it verifies the null pointer and out of range pointer situations are working correctly and the garbage collector is treating them appropriately. If a test case was to fail, the reason for failing would also be shown.

6.1.6 Test Case 6 – Timing Statistics

Test Case 6 was used to time key interface functions and garbage collection. This supplied a comparison between the hardware and software and allowed the calculation of the actual speed up of the hardware implementation. The first interface function that was

tested was *Put Object Data*. An object is created and data is placed into the object at the same offset and location a million times. The average was taken to obtain the time to execute a single *Put Object Data* call. *Get Object Data* was tested in the same manor and for the same number of calls. The average again was calculated for *Get Object Data* to get the results for a single function call. Next, the time to fill heap memory by allocating arrays of a specific size and the time to full the structure table was found. The last timing result is used to determine the time it takes for the collector to run 2000 times when memory is full. It does this by filling memory and then decrementing the object ID of the last object to zero and then adding an object back after collection and repeating this cycle. The average is then calculated to obtain the time it takes to run one collection cycle in the garbage collector. Figure 6.5 shows the timing results obtained from Test Case 6.

```
Prototype Software Collector Timing Results
=====
Put Object Data - Average Time 1M iterations = 4.9005462300000035E-6
Get Object Data - Average Time 1M iterations = 4.506009119999987E-6
Fill memory with arrays time = 0.005077326666651061
Fill memory with structs time = 0.00689618666663705
Test Case 6A: PASSED
Collector - Average Time 2000 iterations = 0.0025144330866669405

FPGA Hardware Collector
=====
Put Object Data - Average Time 1M iterations = 4.699240500000002E-6
Get Object Data - Average Time 1M iterations = 4.776275710000007E-6
Fill memory with arrays time = 0.004900273333333871
Fill memory with structs time = 0.004500229999990779
Test Case 6A: PASSED
Collector - Average Time 2000 iterations = 4.0983135166652575E-4
```

Figure 6.6: Screen Capture for Test Case 6 Outputs (Time Shown in Seconds)

The values returned, shown in Figure 6.6, show the time taken to put data and get data from an object, time to fill memory with arrays, time to fill the structure memory, and the average time it take for the garbage collector to execute one time. Since this is running identically every time and is in hardware, it should be deterministic. This deterministic behavior will be shown in section 6.3. The timing results will also be

compared in greater detail in section 6.2. This test could also be extended to test and compare the rest of the functions if it felt it is needed.

6.1.7 Test Case 7 – Error Condition Testing

Test Case 7 tests the error codes included in the SMM system. The error codes that can be returned in case of an issue is detected by the SMM. Table 6.7 shows a list of error codes supported by the SMM hardware and software equivalent.

Table 6.7: List of Supported Error Codes

Error Detected	Error Code Returned	Hardware Supported	Software Supported
Memory Full	255	Yes	Yes
No More Object ID's	254	Yes	Yes
No More Struct ID's	253	Yes	Yes
Invalid Object ID	252	Yes	Yes
Invalid Struct Size	251	No	Yes
Invalid Struct Type	250	No	Yes
Object ID out of Range	249	No	Yes
Offset out of Range	248	No	Yes
PTR Data out of Range	247	No	Yes

Based on the error codes supported by the SMM hardware a test to activate each one was created. There are six sub tests that execute with in Test Case 7. Test Case 7A first places 50 arrays in the ToSpace memory, completely filling the heap memory. If all 50 arrays are created successfully with no error code being returned, Test Case 7A passes. Next another array is attempted to be allocated, but with no space available a memory full error should be returned. If the error code is returned, Test Case 7B passes. The next test, Test Case 7C, fills the structure table with the maximum number of structures it can handle of 1021. If this is successful with no error code returned, Test Case 7C passes. It then attempts to create one more structure which should cause the no more struct ID's error to be returned. If the error code is returned then Test Case 7D passes. Test Case 7E tests the creation of the maximum number of objects. Since the

memory was previously filled with arrays the system is first reset and then starts by creating 1024 arrays of size one. If all 1024 arrays are created then Test Case 7E passes. Lastly, another array is attempted to be created and if the no more object ID's error code is returned, Test Case 7F passes. Figure 6.7 shows the screen shot of the output produced by test case 7.

Test Case 7A: PASSED
Test Case 7B: PASSED
Test Case 7C: PASSED
Test Case 7D: PASSED
Test Case 7E: PASSED
Test Case 7F: PASSED

Figure 6.7: Screen Capture for Test Case 7 Outputs

As each sub-test executes, it prints out if it passed or failed the test. If a test case was to fail the reason for failing would also be shown. Since all sub-tests display passed next to them, it verifies the error codes are being returned when they should be and the test suite is treating them appropriately.

6.1.8 Test Case 8 – More Garbage Collector Testing

In test Case 8, the conditions under which the garbage collector is activated is tested. The original design was implemented to activate the collector when the free memory threshold was exceeded. But, it was later decided that activating when memory was completely full and when there were no freer object ID's would also be useful. The main situations which are tested in Test Case 8 are the memory full and no more free object ID's case for garbage collection activation. Test Case 8A creates three arrays, where the first two fill memory and the third should cause a memory full error. If this all happens and the first two arrays are created successfully and the third causes the memory error code to be returned then Test Case 8A passes. Next, the reference count of the first array is decremented to zero. Then an array is attempted to be created causing the memory full error to be returned. Then array one is attempted to be accessed. If the array is accessed successfully then Test Case 8B fails otherwise it passes. The system is then reset and 1024 arrays of size one is created to fill the object handle table. If all 1024 arrays are created successfully then Test Case 8C passes. Next another array is attempted

to be created and if the no more object ID's error is returned then Test Case 8D passes. Lastly, the reference count of the first array created is decremented to zero. Then another array is attempted to be created causing the no more object ID's error to be returned again. Then array 1 is attempted to be accessed and if successful in accessing it Test Case 8E fails otherwise it passes. Figure 6.8 shows the screen shot of the output produced by test case 8 from the hardware SMM.

```
Test Case 8A: PASSED
Test Case 8B: Failed
  Expected ID 4 Valid:
  Actual Values:
    Object Id 4 = -1
Test Case 8C: PASSED
Test Case 8D: PASSED
Test Case 8E: Failed
  Expected Valid
  Object Id:
  Actual Value: -1
```

Figure 6.8: Screen Capture for Test Case 8 Outputs for Hardware

Failures appear when running this test case in hardware due to certain features not being implemented. For Test Case 8B the activation of the collector when the memory full error code was returned was not implemented. Also, for Test Case 8E activating the collector when the no more object ID's error code was returned was not implemented. The plan was to implement these features if time allowed. These two situations are not considered critical for the system to work correctly. Figure 6.9 shows the screen shot of the output produced by test case 8 from the software SMM.

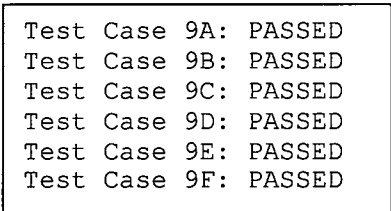
```
Test Case 8A: PASSED
Test Case 8B: PASSED
Test Case 8C: PASSED
Test Case 8D: PASSED
Test Case 8E: PASSED
```

Figure 6.9: Screen Capture for Test Case 8 Outputs for Software

For the Software SMM all tests pass for test case 8, since the software was used to model the ideal SMM system and used to test functionality.

6.1.9 Test Case 9 – FACET Example

Test Case 9 is created as a prototype of the FACET event channel created by Washington University. This was important since this would be similar to how the SMM would be used for a final demo for the Air Force Research Labs. This test uses pointers inside object and arrays to verify the garbage collector is treating them correctly. First, an array is created with pointers and it then creates the number of objects equal to the size of the array initially created. After it creates each object it places the object ID into the array and places a data value into the object. It then decrements the reference count on the objects to zero. At this time, the memory is filled to right before the threshold value. An array is then created to force the garbage collector to activate. If the collector has activated, Test Case 9A passes. Test Case 9B then verifies that due to the references in the array to the object that contain zero reference counts that all the objects still exist and have not been collected. If all the objects still exist, Test Case 9B passes. The reference count in the array is then decremented to zero causing the collector to activate again. The collector count is then verified again and if correct, Test Case 9C passes. The array and all objects created are checked to see if they still exist. If the array has been collected then Test Case 9D passes and if all the objects have been collected then Test Case 9E passes. The last test then decrements the reference count to zero in the large array used to force garbage collection and then is verified that it has been collected. If it has been collected, Test Case 9F passes. This whole cycle of sub tests then repeats in a loop for 10,000 times. Figure 6.10 shows the screen shot of the output produced by test case 9.



```
Test Case 9A: PASSED
Test Case 9B: PASSED
Test Case 9C: PASSED
Test Case 9D: PASSED
Test Case 9E: PASSED
Test Case 9F: PASSED
```

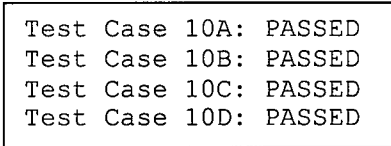
Figure 6.10: Screen Capture for Test Case 9 Outputs

During each pass of the test case the only time a print out happens is if a fail is observed or at the completion of the test. If the test executed 10,000 times successfully then the PASSED result is returned for each test case. Since all sub-tests display passed,

next to them it verifies the FACET prototype test case is working correctly. If a test case was to fail, the reason for failing would also be shown. This test is critical since it prototypes the main system environment the SMM was tested in. By passing this test, it allows a better chance of the system working in the final testing environment.

6.1.10 Test Case 10 – Extended Interface Testing

Test Case 10 was created to extend the testing done in test case 1. This was required to extend testing to additional interface tests not done in Test Case 1. Test Case 10A just creates an object and places data in it and then gets the data from the object. If the data sent to the object equals the data from the object, Test Case 10A passes. The object is then cloned creating a new object which is an exact copy of the original object. The data is retrieved from the cloned object and if it matches the data sent to the original object, Test Case 10B passes. An array is then created and data placed in it and then retrieved from it. If the data obtained from the array matches the data sent to the array, Test Case 10C passes. Lastly, the array is cloned and the data in the cloned array is obtained and matched to the data sent to the original array. If the data matched, Test Case 10D passes. Figure 6.11 shows the screen shot of the output produced by test case 10.



```
Test Case 10A: PASSED
Test Case 10B: PASSED
Test Case 10C: PASSED
Test Case 10D: PASSED
```

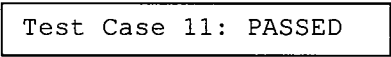
Figure 6.11: Screen Capture for Test Case 10 Outputs

As each sub-test executes it prints out if it passed or failed the test. If a test case was to fail, the reason for failing would also be shown. Since all sub-tests display passed, next to them it verifies that test case 10 is working correctly.

6.1.11 Test Case 11 – Extended System Testing

Test Case 11 was a test case created for debugging purposes only. It was created to debug an issue that arose in the FACET even channel test case (Test Case 9). By creating a test case that narrowed down where the issue was being generated the SMM

could be fixed. First, a group of arrays are created to fill memory just before the threshold level. Then the first array created has the reference count decremented to zero. The test then enters a loop that executes for a set number of times. Inside the loop a new object and new array are created. The reference counts for both are decremented to zero. Each time the object and array are created the garbage collector is activated and then again when the reference counts are decremented to zero. Figure 6.12 shows the screen shot of the output produced by test case 11.



Test Case 11: PASSED

Figure 6.12: Screen Capture for Test Case 11 Outputs

This helped to resolve an issue that initially arose where memory was filling up due to the collector not coping correctly and over allocating memory during the copy process. Once this test case was working correctly, it caused Test Case 9 to work correctly. This test could have been removed in the final test suite, but was left to allow additional system verification.

6.1.12 Test Case 12 – Extended Pointer Reference Testing

Test Case 13 extended the pointer testing to include activating the garbage collector to verify that when garbage collection was activated it handled pointers correctly and copied objects that had pointers to them even if they had a zero reference count. This happens in the case of a sequential structure where the first object is alive with a positive reference count and all the objects it references have a reference count of zero. The first thing done is a new object is created of size 64 based on a structure that labels every forth word in the object as a pointer starting from location zero. It then creates 64 objects and places data in them. These object Id's are then placed in each location in the original object created. At the same time it also decrements the reference count of the 64 new objects to zero causing them to be thought of as dead objects. An array is then created to activate garbage collection and only the objects that are referenced based on the pointer mask should be copied and all other objects collected. If the objects that should be collected are collected and the objects that are referenced by

the original object are copied, Test Case 12A passes. Next, the reference count of the original object is decremented to zero causing it to be collected as well as all the objects it referenced. This frees up all the ToSpace heap memory for the next test.

An array is created with a size of three with all words in the array being pointers. A loop is executed for 100 iterations in which new arrays are created and the object ID for that array is stored in the array created in the previous execution of the loop. This creates a sequential chain of a 101 arrays that all reference one another. While this is happening, all the reference counts of the arrays are decremented to zero other than the original array created. The next array created after the loop executes creates an array with an out of range pointer. Lastly an array is created to force garbage collection to activate. Once collection has completed, it is verified that every array in the sequential chain has been copied successfully and that the out of range pointer valid is still the same. If both these criteria are met then Test Case 12B passes. Next, the original array which contains a reference count of 1 is decremented to zero activating garbage collection. All the arrays should then be collected leaving only the array with the out of range pointer left. It is then verified that all the arrays that should have been collected where and if this is the case, Test Case 12C passes.

Next, the test creates a chain where some objects created later in the loop reference objects previously created. It does this by creating an array of pointers which references the first array in the chain. As each additional array is created they reference the array created right before it. Then an array is created with an out of range pointer and an array is also created to force garbage collection. Once collection has completed, all arrays created should be copied and the out of range pointer should be unchanged. If both of these criteria are met then Test Case 12D passes. The original array in the chain has the reference count decremented to zero causing garbage collection to be activated again. At this time, all the objects should be collected leaving only the array with the out of range pointer. It is then verified that the arrays that should have been collected are and if this did happen, Test Case 12E passes.

The last structure to be created and tested is the cyclic structure. This type of structure can cause the most problems in a copy collector if not handled correctly, so as not to get stuck in an infinite loop. An initial array is created with a reference count of

one. Then 100 more arrays are created with reference to each one being placed in the previous array. Once all 100 arrays are created the last array has a reference to the first array placed in it creating the cyclic structure. Next, an array is created to force the collector to activate and once the collector completes then each array is verified to have been copied as well as verifying that the last object still references the first object. If all this holds true, Test Case 12F passes. The last test is to decrement the reference count of the original array to zero starting the collection process and all the arrays should now be collected. If all the arrays are verified to be collected, Test Case 12F passes

Figure 6.13 shows the screen shot of the output produced by test case 12.

Test Case 12A: PASSED
Test Case 12B: PASSED
Test Case 12C: PASSED
Test Case 12D: PASSED
Test Case 12E: PASSED
Test Case 12F: PASSED
Test Case 12G: PASSED

Figure 6.13: Screen Capture for Test Case 12 Outputs

As each sub-test executes, it prints out if it passed or failed the test. Since all sub-tests display passed next to them, it verifies that test case 12 is working correctly. If a test case was to fail, the reason for failing would also be shown. By passing these test cases, the garbage collector was verified to handle pointers contained in data objects and arrays correctly for various structures. This was the final test since it was considered to be the most difficult since there can be cases where objects have pointers to them, but have a zero reference count. This would normally be treated as garbage, but due to a pointer pointing to it; it must be copied and preserved for later use.

6.2 Hardware vs. Software Comparison

The main goal in creating the hardware SMM was to increase the speed of execution of the garbage collector and make it deterministic. The speed in which the interface executed was also crucial, since if it took longer for the hardware interface to execute when compared to the software equivalent, the speed up shown by the garbage collector could be nullified and the systems would execute at equal speeds or the

hardware potentially being slower. Table 6.14 shows the timing results and the net gain or loss in time the hardware SMM provides over the software version.

Table 6.14: Timing Comparison Between Hardware and Software

Function Timed	Hardware Time (Milliseconds)	Software Time (Milliseconds)	Net Gain or Loss in Time (Milliseconds)
Put Object Data	0.004699240	0.004900546	0.000201306
Get Object Data	0.004776276	0.004506009	-0.000270267
Fill Memory with Arrays	4.900273333	5.077326667	0.000177053
Fill Memory with Structures	4.500229999	6.896186667	0.002395957
Garbage Collection	0.471040795	2.544219235	2.07317844

Execution Time in Milliseconds Between Hardware and Software

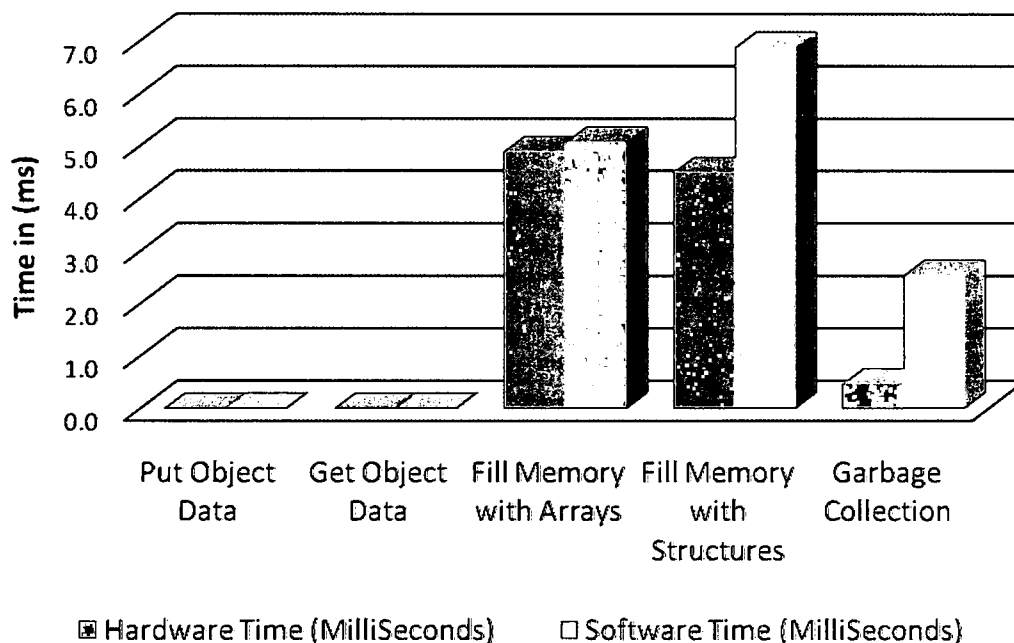


Figure 6.14: Comparing Execution Times Between Hardware and Software

The hardware SMM showed execution speed increases in all functions other than *Get Object Data*. This is due to the memory setup on the BRAM was chosen to take two clock cycles to read data out of memory and one to write data into memory. This easily could have been reversed and the *Put Object Data* would have been slightly slower in the hardware than software. With the execution time difference in *Put Object Data* positive and the *Get Object Data* negative, there is no advantage in the hardware SMM over the software SMM.

For filling memory with arrays, the hardware SMM is only slightly faster than the software SMM since there are additional checks the hardware has to do and the hardware has to write to more than one memory to complete the task. For filling of the structure table memory, the SMM hardware shows a significant speed increase since there is only one memory write it must complete. Once the data is sent to the SMM, it does not have to compete for the bus with all other operations going on like the software SMM has to.

The most significant speed increase comes from garbage collection. The hardware collector is on the order of 5 times faster than the software collector and there are a few different reasons for this. One reason is that the hardware collector can run from start to stop without any interference from other devices. The software collector has the Linux operating system to compete with as well as bus contention with other devices. Based on the worst case scenario, execution time for the software can vary from run to run. In hardware the worst case scenario can be determined and will not change each time it is executed. This deterministic behavior will be shown in more detail in section 6.3.

Showing the increase in speed between the hardware and software collector, it can be seen how this would affect a real-time system. By reducing the amount of time it takes for the collector to execute the real-time system is less likely to miss mission critical data. This is major concern in real-time systems since there is a very limited amount of time in each frame to execute a number of tasks. By making a garbage collector that can execute very fast and within the allowed amount of time, this issue can be resolved. However, due to the issue that garbage collection can execute at various times, the SMM system would benefit even more by allowing it to interleave with the real-time system by executing only during times when the program is not accessing it.

This can be done by allowing for garbage collection to be paused while it is activated, allowing calls from the real-time system to be handled. The only issue that needs to be addressed here would be how fast objects are being allocated so as not to fill memory faster than the collector can free up memory. By creating a hardware based system, this can be done.

6.3 Full Demo Environment Results

The full demo environment was explained in section 2.3 and it was used to test the full functionality of the SMM hardware. By running the Scan Eagle Flight Scenario without garbage collection and then again with the software and hardware collector, the need for garbage collection can be seen [7]. Also the benefits of the increase execution speed of the hardware SMM over the software SMM can be seen. Without garbage collection, the system was highly unstable and ran out memory before any valid timing or data could be collected due to many of frames being missed. Figures 6.15 and 6.16 show what happens when the software and the hardware collector are added to the system

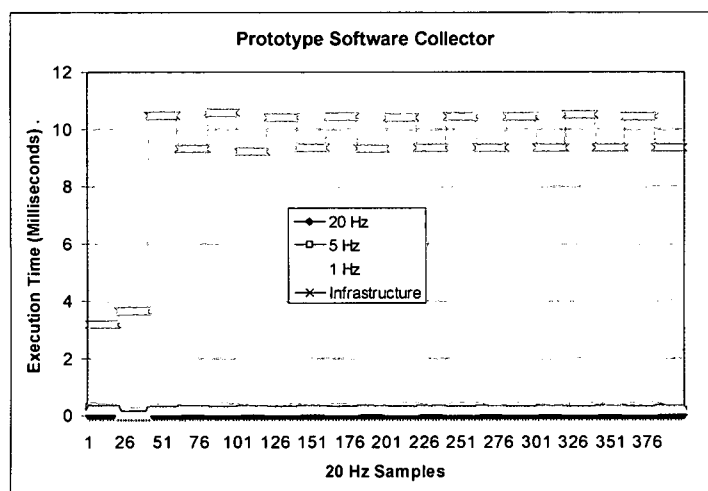


Figure 6.15 Prototype Software Collector Timing Results for Each Rate [7]

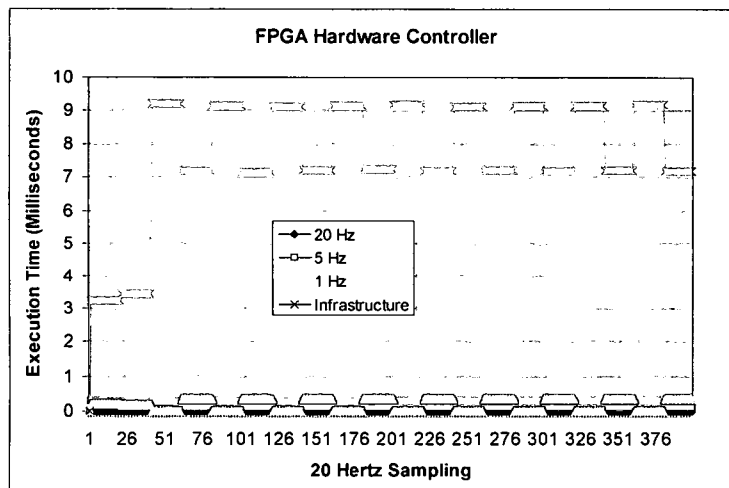


Figure 6.16 FPGA Hardware Collector Timing Results for Each Rate [7]

By adding garbage collection, the system was able to run stable without missing frames at the various rates being executed. While both the hardware and software SMM's meet the needs, the software SMM was still not within the desired time desired. The goal was to keep the infrastructure execution time under 10 milliseconds under high processing load and low proceeding load. Figure 6.15 shows that the software SMM was capable of this for nominal load conditions, but not capable of doing this under peak load conditions. This is due to the amount of time the software SMM takes to execute tying up the rest of the system. Figure 6.16 shows that the hardware SMM was capable of meeting the goal of under the 10 millisecond for execution time for both the nominal and peak load conditions. This is due to the hardware garbage collector being on the order of 5 times faster than the software collector. With the hardware collector, the system performed approximately 2.1 milliseconds faster under nominal load conditions and 1.2 milliseconds faster under peak load conditions when compared to the software collector.

One major reason for this is that the hardware collector is deterministic and allows one to know the worst case scenario for execution time for garbage collection to complete. In software this is harder to determine since the Linux OS can and will often have higher priorities then the software collector, causing execution time to vary from run to run. This can be seen in Figures 6.15 for the software and 6.16 for the hardware. In figure 6.15 for the software the infrastructure time difference from high to low various

from transition to transition. Figure 6.16 the hardware shows the difference between the high and low values is consistently the same.

Chapter 7

Future Improvements and Conclusions

As with most designs, there is always something that can be done to further the performance. For the SMM hardware, we have already shown that it executed faster than software equivalent and is also deterministic in nature. However, there are some enhancements that can only improve upon this design to make it even more efficient. The rest of this chapter will describe some of these changes that could be implemented as well as some final conclusions.

7.1 *Future Improvements*

There are three main things that could be done to increase the overall performance of the SMM system. These are moving to a new board to allow hardware modifications to be done, embedding the interface driver into the kernel, and creating a plug-in SMM module that could be utilized on various boards. The rest of this section will describe each of these changes in greater detail.

7.1.1 Hardware Modifications

Due to limitations in the FPGA, some modifications had to be made to make it fit in the available logic, as well as talk to available memory. The current design utilized internal FPGA Bram Memory which reduced the number of objects that could be created. By using a board that contained multiple SRAM's or SDRAM's, the number of objects that could be created would be increased, allowing for a greater range of programs to be used with the SMM hardware. The use of a larger FPGA would allow for modifications to the SMM hardware, allowing use of more efficient methods.

7.1.2 Embedded Kernel Module Driver

The current software interface design must be attached to every software program and the bus address must be mapped to CPU space every time a program executes. This could be greatly enhanced by the creation of an embedded kernel module that handles the

creation of the SMM as a Linux device. Allowing the interface functions, that talk to the SMM hardware, to always be available. This would only require the bus address to be mapped into CPU space once at start up and the virtual address obtained by the executing program. By creating a more tightly coupled interface with the Linux kernel, the system has less overhead by not having to map the bus address and then unmap it every time the SMM must be accessed by different programs. This would also allow multiple programs to access the SMM hardware at the same time and create a true multi-threaded device.

7.1.3 Garbage Collector Hardware Plug-in Module

The creation of a plug-in module that could be used by various development and production boards would allow greater functionality and use of the SMM. The design for the SMM comes from the garbage-collected memory module (GCMM) created by Kevin Nielson [6]. The only real difference is in the use of dedicated hardware in place of a RISC processor executing software to handle the interfacing and control of garbage collection and memory allocation. By using FPGA hardware, execution time can be greatly reduced beyond the GCMM. This is done by completely removing the software aspect the GCMM still uses as the control unit. The FPGA based plug-in module would allow the SMM to be moved from board to board without having to restructure the SMM to work on various platforms. The only modifications that would have to be done when moving from one board to another would be the interface code which would be based on platform and operating system. In this case, the moving of the SMM module from one Xilinx board to another would require no change other than attaching the device to the Xilinx project, but all interface code could be used by only updating the assigned bus address for that board. Figure 7.1 shows the SMM plug-in module proposed.

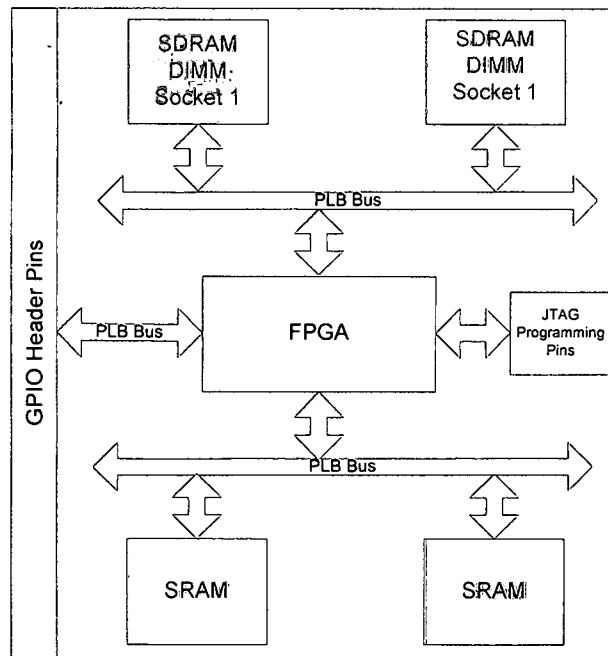


Figure 7.1 Proposed SMM Plug-In Module

The proposed SMM plug-in module contains an FPGA attached two SDRAM DIMM sockets, two SRAM memories, and GPIO header pins. The FPGA would contain the SMM controller and handle all incoming calls and the sending of data over the GPIO pins attached the main system board. The SRAM memories would be used to store the Structure Table and the Handle Table. And the SDRAM DIMM sockets would allow various memories sizes to be used. Ideally, these would be the same size and would be used as the ToSpace and FromSpace memories for the SMM. Having two individual memories for ToSpace and FromSpace, the coping of data from one to the other during collection would be more efficient since it could be reading from one and writing to the other at the same time.

The only drawback to this design is that the GPIO pins run slower than 100 MHz, which is the speed of the PLB bus. The maximum speed at which memory could be allocated would have to be determined. This could be overcome by using alternative interfaces then the GPIO pins such as rocket-io. However, this would reduce the number of boards that the plug-in module could be attached to.

7.2 Conclusions

This thesis has described the design and implementation of a hardware garbage collection system name the smart memory module (SMM) for use in real-time applications. It has been shown that by embedding garbage collection into custom hardware that greater speed and deterministic behavior can be achieved. This was done using a software equivalent design of the SMM allowing for an accurate comparison to be done. The software model also allowed testing of the design as well as early integration in the final testing environment. The software model allowed potential problems to be identified early and corrected before being coded into VHDL hardware code. This was more efficient, since the modification of the software code was faster than modifying the VHDL code. Problems could be debugged in software faster than hardware, since the software is fully transparent to the programmer.

It was also shown that with greater resources the system could be enhanced to further increase execution time significantly. By moving to a board with more logic, the system could utilize faster methods to achieve the intended goal. By embedding the interface functions into the kernel as a kernel module, one would decrease system overhead, as well as allow multiple programs to share the SMM. Lastly, by creating a dedicated SMM plug-in module that the SMM could more widely use across multiple platforms, it would allow greater flexibility to the SMM and lend itself to a wide range of uses.

Appendices

Appendix A: SMM Hardware VHDL Code

See attached cd under folder “Hardware VHDL” to view the SMM VHDL system files.

Appendix B: JAVA Test Suite Code

See attached CD under folder “Java Test Suite” to view the test suite code and build files.

References

1. Baker, Henry G. "List Processing in Real-Time on a Serial Computer." Communications of the ACM. 21(4):280-94. 1978.
2. Boehm, Hans-Juergen. Weiser, Mark. "Garbage Collection in an Uncooperative Environment." Software Practice and Experience, 18(9):807-820, 1988.
3. Cheney, C. J. "A Non-Recursive List Compacting Algorithm." Communications of the ACM, 13(11):677-678, November 1970.
4. Fieler, Joseph Thomas. "A Real-Time Garbage Collection Design for Embedded Systems." Master's Thesis. University of Dayton. 2004.
5. Jones, Richard. Lins, Rafael. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons. Chichester. 1996.
6. Nilsen, Kevin. "A High-Performance Architecture for Real-Time Garbage Collection." In Workshop: Garbage Collection in Object-Oriented Systems, 1991.
7. Weber, John, Cytron, Dr. Ron k, Pl, Edward. "Final Report for the Adaptive Real-Time Computing for Information Exchange Program (ARTEC)." Unpublished Final Report. 2006.
8. Weizenbaum, J. "Symmetric List Processor." Communications of the ACM, 6(9):524-544, September 1963.
9. Xilinx. "PLB IPIF (v2.02a)." Technical Data Sheet. 2005
10. Xilinx. "Virtex-4 Family Overview (v2.0)" Technical Data Sheet 2007

R00 2593184