**In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.**

**While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.**

A Real Time Memory Manager for the Real Time Specification for Java

By

Rongrong Xu

Supervisor: Professor Trevor W. Pearce

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Science in ISS

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University

May 2003

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitons et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-88789-8

Our file Notre référence

ISBN: 0-612-88789-8

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the Thesis

A Real Time Memory Manager
for the Real Time Specification for Java

Submitted By Rongrong Xu

in partial fulfillment of
the requirements for the degree of
Master of Science in Information Systems Science

Department Chair

Thesis Supervisor

Carleton University

May 2003

Abstract

This thesis addresses the need for providing a real time memory manager, which can be used in a real time Java virtual machine for real time Java applications. The memory manager is designed based on the Real Time Memory Manager Specification proposed here in accordance with the Java Language Specification, the Java Virtual Machine Specification, and the Real Time Specification for Java. The memory manager is realized via interfaces implemented carefully to satisfy the real time requirements. Deterministic evaluation for each interface plus its supporting software components has been done using big-O analysis.

Additional methods are designed and implemented to test the memory manager as it might be used within a real time Java virtual machine. The tests provide black-box results for each interface, which verify that the real time memory manager meets the requirements specified in the Real Time Memory Manager Specification.

All of the software components in the memory manager are implemented in C language, which is expected to fit easily into future development of a real time Java virtual machine. The whole memory manager is constructed in a style that eases understanding, modification, and future integration.

Acknowledgements

Many thanks go to my thesis supervisor, Prof. T.W. Pearce, for his helpful suggestions, endless patience and quiet encouragement.

I would like to thank my friends, Ms. Juemin Yu for her help in C, and Mr. Song Zhang for his help in Java, and some people in the real time group in the Department of System and Computer Engineering.

Finally, I would like to thank my wife Jiamao and my children Leon and Odelia, for their support and valuable suggestions.

Table of Contents

Abstract.....	iii
Acknowledgements.....	iv
Table of Contents.....	v
List of Figures.....	xii
List of Tables.....	xv
List of Acronyms.....	xvi
1 Introduction	1
1.1 Background.....	2
1.1.1 Real Time Systems.....	2
1.1.2 Java.....	3
1.1.3 The Java Virtual Machine	4
1.1.3.1 Class Loader	6
1.1.3.2 Method Area, Heap and Java Stack.....	7
1.1.3.3 Heap Manager	8
1.1.4 The Java Class File	8
1.1.5 Real Time Specification for Java.....	9
1.1.5.1 The History of Real Time Java.....	9

1.1.5.2	Overview of the Seven RTSJ Enhanced Areas.....	11
1.1.5.2.1	Thread Scheduling and Dispatching.....	11
1.1.5.2.2	Memory Management.....	12
1.1.5.2.3	Synchronization and Resource Sharing.....	13
1.1.5.2.4	Asynchronous Events Handling.....	14
1.1.5.2.5	Asynchronous Transfer of Control.....	15
1.1.5.2.6	Asynchronous Thread Termination.....	15
1.1.5.2.7	Physical Memory Access.....	16
1.1.5.3	The RTSJ Memory Classes.....	16
1.1.6	Real Time Memory Management.....	19
1.1.6.1	Manual Memory Management.....	20
1.1.6.2	Automatic Memory Management.....	21
1.2	The Thesis.....	22
1.2.1	The Objective of the Thesis.....	22
1.2.2	The Scope of the Thesis.....	23
1.2.3	The Contributions of the Thesis.....	24
2	Analysis of the RTSJ Memory Classes	26
2.1	The RTSJ Memory Area.....	26
2.2	The Lifetimes of Objects in RTSJ Memory Areas.....	27
2.3	References to Objects in Different Memory Area.....	28
2.4	The RTSJ Memory Classes.....	29
2.4.1	Analysis of <i>MemoryArea</i>	30

2.4.2	Analysis for <i>ScopedMemory</i>	32
2.4.3	Analysis for <i>ImmortalMemory</i>	34
2.4.4	Analysis for <i>HeapMemory</i>	35
3	Real Time Memory Manager Specification	36
3.1	Preconditions for the Memory Manager.....	36
3.1.1	The Thread Manager inside the RTJVM.....	36
3.1.2	Initial Memory Resources.....	37
3.2	Requirements for the Memory Manager.....	38
3.2.1	Requirements on Heap Memory Manipulation.....	38
3.2.2	Requirements on Immortal Memory Manipulation.....	39
3.2.3	Requirements on Scoped Memory Manipulation.....	39
3.2.4	Requirements on Scope Manipulation.....	39
3.3	Functionality in the Memory Manager.....	40
3.4	Interface to the Memory Manager.....	41
4	Design of the Memory Manager	43
4.1	Design of the Memory Manager	43
4.1.1	<i>list</i>	43
4.1.2	<i>memNode</i>	44
4.1.3	<i>scopeNode</i>	45
4.1.4	The nested Scopes in Java, in the Memory Manager and on a Stack.....	46
4.1.5	<i>Mem</i>	49
4.1.6	Memory Manager Parameters.....	49

4.1.7	Initialization of the Memory Manager	50
4.1.8	Invocation of the <i>activateGC()</i> Interface.....	53
4.2	An Example for Memory and Scope Manipulations at Design Level	53
5	Implementation and Evaluation of the Memory Manager	57
5.1	Implementation of Basic Data Structures.....	57
5.2	Implementation and Evaluation of Internal Methods.....	59
5.2.1	Implementation of <i>prepareMemP()</i> , <i>prepareMemNodeP()</i> and <i>prepareScopeNodeP()</i>	60
5.2.1.1	Evaluation of <i>prepareMemP()</i> , <i>prepareMemNodeP()</i> and <i>prepareScopeNodeP()</i>	61
5.2.2	Implementation of <i>initMemNode()</i>	61
5.2.2.1	Evaluation of <i>initMemNode()</i>	61
5.2.3	Implementation of <i>addMemNode()</i> and <i>addScopeNode()</i>	62
5.2.3.1	Evaluation of <i>addMemNode()</i> and <i>addScopeNode()</i>	62
5.2.4	Implementation of <i>initMemList()</i>	63
5.2.4.1	Evaluation of <i>initMemList()</i>	63
5.2.5	Implementation of <i>giveMem()</i>	64
5.2.5.1	Evaluation of <i>giveMem()</i>	64
5.2.6	Implementation of <i>checkMemory()</i> , <i>checkScope()</i> and <i>checkMemScope()</i>	65
5.2.6.1	Evaluation of <i>checkMemory()</i> , <i>checkScope()</i> and <i>checkMemScope()</i>	66

5.2.7	Implementation of <i>checkHeap()</i>	66
5.2.7.1	Evaluation of <i>checkHeap()</i>	66
5.2.8	Implementation of <i>removeMemNode()</i> and <i>removeScopeNode()</i>	67
5.2.8.1	Evaluation of <i>removeMemNode()</i> and <i>removeScopeNode()</i>	67
5.2.9	Implementation of <i>getMem()</i>	67
5.2.9.1	Evaluation of <i>getMem()</i>	68
5.2.10	Implementation of <i>scanOneScope()</i>	69
5.2.10.1	Evaluation of <i>scanOneScope()</i>	69
5.3	Implementation and Evaluation of Interfaces.....	69
5.3.1	The <i>initManager()</i> Interface.....	70
5.3.1.1	Evaluation of the <i>initManager()</i> Interface.....	71
5.3.2	The <i>exceptionCatch()</i> Interface	72
5.3.2.1	Evaluation of the <i>exceptionCatch()</i> Interface.....	72
5.3.3	The <i>getImmortalMem()</i> Interface.....	73
5.3.3.1	Evaluation of the <i>getImmortalMem()</i> Interface.....	74
5.3.4	The <i>getHeapMem()</i> Interface.....	74
5.3.4.1	Evaluation of the <i>getHeapMem()</i> Interface.....	75
5.3.5	The <i>getScope()</i> Interface.....	75
5.3.5.1	Evaluation of the <i>getScope()</i> Interface.....	75
5.3.6	The <i>getScopeMem()</i> Interface.....	76
5.3.6.1	Evaluation of the <i>getScopeMem()</i> Interface.....	77
5.3.7	The <i>scanNestedScope()</i> Interface.....	77
5.3.7.1	Evaluation of the <i>scanNestedScope()</i> Interface.....	78

5.3.8	The <i>endScope()</i> Interface.....	79
5.3.8.1	Evaluation of the <i>endScope()</i> Interface	80
5.3.9	The <i>activateGC()</i> Interface.....	80
5.3.9.1	Evaluation of the <i>activateGC()</i> Interface.....	80
6	Tests for the Memory Manager	81
6.1	The Input Methods.....	82
6.1.1	The <i>simulate_RTJVM_DistributionInfo()</i> Method.....	83
6.1.2	The <i>addInfo()</i> Method	83
6.1.3	The <i>addMemSize()</i> Method	84
6.2	The Output Methods	84
6.2.1	The <i>showManager()</i> Method.....	84
6.2.2	Methods of <i>showMoreCommand()</i> and <i>illInput()</i>	85
6.2.3	Methods of <i>noThisMem()</i> , <i>noMoreNode()</i> and <i>illNestedScope()</i>	85
6.2.4	The <i>showScanInfo()</i> Method.....	85
6.3	The <i>main()</i> Method	86
6.4	Tests for the Memory Manager	89
6.4.1	Tests for the <i>initManager()</i> Interface.....	89
6.4.2	Tests for Interfaces <i>getImmortalMem()</i> , <i>getHeapMem()</i> , <i>getScopedMem()</i> , <i>exceptionCatch()</i> and <i>activateGC()</i>	89
6.4.3	Tests for Interfaces <i>getScope()</i> , <i>scanNestedScope()</i> , and <i>endScope()</i>	90
7	Conclusions and Recommendations	91

References	93
Appendix The Code for the RTSJ Memory Manager	97

List of Figures

Figure 1.1	The JVM Architecture.....	6
Figure 1.2	History of Real Time Java	10
Figure 1.3	The Embedded Memory Manager in a RTJVM.....	23
Figure 2.1	The RTSJ Memory Areas.....	27
Figure 2.2	The Lifetimes of Objects in RTSJ Memory Area	28
Figure 2.3	Relationships among four RTSJ Memory Classes	30
Figure 3.1	Memory Resources in a RTJVM	37
Figure 3.2	The Provided Memory from a RTJVM	38
Figure 3.3	Interface to the Memory Manager	42
Figure 4.1	The <i>memNode</i>	45
Figure 4.2	The <i>scopeNode</i>	45
Figure 4.3	The Compound Scope in Java, in the Memory Manager, and on a Stack.	48
Figure 4.4	The <i>mem</i>	49
Figure 4.5	Distributed Application Memory.....	51
Figure 4.6	The State of Memory Manager after Initialization.....	52
Figure 4.7A	An Example for Scope and Memory Manipulations.....	54
Figure 4.7BC	An Example for Scope and Memory Manipulations	55
Figure 4.7D	An Example for Scope and Memory Manipulations.....	56

Figure 5.1.1	Implementation of <i>memNode</i> , <i>scopeNode</i> and <i>mem</i>	58
Figure 5.2	Interaction among Interfaces and Internal Methods with Big-O Analyses.....	59
Figure 5.2.1	The <i>prepareMemP()</i> Method.....	60
Figure 5.2.2	The <i>initMemNode()</i> Algorithm.....	62
Figure 5.2.3	The <i>addMemNode()</i> Algorithm.....	63
Figure 5.2.4	The <i>initMemList()</i> Algorithm.....	64
Figure 5.2.5	The <i>giveMem()</i> Algorithm.....	65
Figure 5.2.6	The <i>checkHeap()</i> Algorithm.....	66
Figure 5.2.7	The <i>removeMemNode()</i> Algorithm.....	67
Figure 5.2.8	The <i>getMem()</i> Algorithm.....	68
Figure 5.2.9	The <i>scanOneScope()</i> Algorithm.....	69
Figure 5.3.1	The <i>initManager()</i> Algorithm.....	71
Figure 5.3.2	Relationships among Integers, Exceptions and Expressions.....	72
Figure 5.3.3	The <i>exceptionCatch()</i> Algorithm.....	72
Figure 5.3.4	The Calling Structure of the <i>getImmortalMem()</i> Interface.....	73
Figure 5.3.5	The Calling Structure of the <i>getHeapMem()</i> Interface	74
Figure 5.3.6	The <i>getScope()</i> Algorithm.....	76
Figure 5.3.7	The Calling Structure of the <i>getScopeMem()</i> Interface.....	77
Figure 5.3.8	The Calling Structure of the <i>scanNestedScope()</i> Interface.....	78
Figure 5.3.9	The <i>endScope()</i> Interface.....	79
Figure 6.1	An Initial State of the Memory Manager.....	82
Figure 6.2.2	A Nested Scope State	86

Figure 6.3.1	The Calling Structure of the <i>main()</i> Method.....	87
Figure 6.3.2	The States of the Memory Manager after some Tests.....	88

List of Tables

Table 2.1	The Set of Assignment Rules.....	29
Table 2.2	The <i>MemoryArea</i> Class.....	31
Table 2.3	The <i>ScopedMemory</i> Class	34
Table 2.4	The <i>ImmortalMemory</i> Class	34
Table 2.5	The <i>HeapMemory</i> Class.....	35

List of Acronyms

JVM	Java Virtual Machine
RTSJ	Real Time Specification for Java
JLS	Java Language Specification
JVMS	Java Virtual Machine Specification
PC	Program Counter
RTJEG	Real Time Java Experts Group
APIs	Application Programming Interfaces
RTMM	Real Time Memory Manager
RTJVM	Real Time Java Virtual Machine
GC	Garbage Collection
RTMMS	Real Time Memory Manager Specification

Chapter 1

Introduction

Java, originally called Oak, was designed for use in embedded consumer-electronic applications [1]. After several years of experience with the language, it was retargeted to the internet, renamed Java, and substantially revised to become the well-known Java language [2]. So, Java was originally not a real time programming language [3].

In order to make Java available to real time applications as Real Time Java (RTJ) that is created by extending the existing Java classes, Sun and its partners formed the Real Time Java Experts group (RTJEG), which has created the Real Time Specification for Java (RTSJ) [4]. The RTSJ is an open specification, which provides a real time software development platform suitable for a wide range of real time applications [5, 6,7].

To apply the RTSJ for RTJ, a real time Java virtual machine (RTJVM), which is different from traditional Java Virtual Machine (JVM)[8], is required so that all Java programs can be run on the RTJVM. The behavior required in a RTJVM is different from that of a traditional JVM in that a RTJVM must be deterministic, while a traditional JVM does not have such a strict requirement. [9,10]. One of the important issues to ensure deterministic behavior is to have a real time memory manager (RTMM) in RTJVM instead of the heap

manager in a traditional JVM [11]. The RTMM is also called a RTSJ memory manager (RTSJMM), which is designed, implemented, evaluated and tested in this thesis.

This chapter includes two basic parts. The first part reviews background information in real time systems [5], Java [12], the JVM [13], the Java class file [12], and the RTSJ [4]. The second part defines the thesis in term of its objective of the thesis, contribution and the scope.

1.1 Background

This background discusses five important areas, which cover features for real time programming [14], the Java language as specified in the Java Language Specification (JLS)[12] and compared to C and/or C++ languages, the internal architecture of a JVM as specified in the Java Virtual Machine Specification (JVMS)[8], the structure of Java class files [13], and the RTSJ sections related to memory management.

1.1.1 Real Time Systems

A real time system is a system that interacts with surrounding “real world” events, which means that real time systems are all about time [5]. Real-time programming is often called “computing with a deadline”. Real-time systems are [5]:

- *Timely* in that it is possible to predict how long a particular task will take to execute, and it is possible to initiate task execution at a specific time.
- *Concurrent*, with multiple sequences of execution (threads) of varying priority simultaneously active within a task or process.

- *Responsive* to interact with physical interfaces to the surrounding environment in a predictable and rapid manner.
- *Usually embedded* in a system such that the behavior of the real-time software system is indistinguishable from that of the host.
- Highly *predictable* in that they do the right thing all of the time.
- *Robust* in that they will do the right thing under unexpected or even erroneous conditions.

Different real time systems meet the above criteria to different degrees [10]. Some are “soft” in that absolute computing deadline are not always met, while others are “hard” in that all deadlines must be met. The control system in an aircraft is an example of a “hard” real time system [5]. A “soft” real time system can have some missed deadlines some time. Network packet routers are examples of “soft” real-time systems [5].

1.1.2 Java

Java is a simple, object oriented, robust, architecture neutral, interpreted, multithreaded, and dynamic language [1].

- Java is *simple* in that it uses C programming language syntax for the most part, and it can be considered a simplified version of C++. Java source code is compiled into bytecodes, which can be interpreted by a Java virtual machine. Java supports concurrency as one of its language features with multiple threads and thread synchronization features.
- Java is *object-oriented*, even though it uses primitive data types such as “*int*” and “*float*” that are not objects.

- Java is *dynamic* in that loading, linking, and initialization may be delayed until they are needed at runtime. Since Java can load classes as needed at runtime, symbolic references need not be resolved until they are actually used.
- Java is *robust* in that error-prone features of C/C++ (most specifically, pointers) have been reduced or eliminated. Java does not have methods such as *sizeof()*, *malloc()* or *free()* for any memory manipulation. Java has built-in class verification and supports exception handling.
- Java is *architecture neutral* in that once a source program has been compiled into bytecodes, it should run on any JVM without concern for the underlying operating system.
- The standard class libraries offer good classes for programmers to conveniently solve programming problems [15].

However, Java is also [10]:

- *Slow*. Java's compiled bytecodes are interpreted by a JVM, which is slow compared to languages that are compiled into native code. Garbage collection and dynamic class loading and linking occurs at runtime, which make the Java's performance slow.
- *Non-deterministic*. In order to allow a JVM to have the greatest implementation flexibility on heterogeneous platforms, the threading model is built into the language, which resulting in *non-deterministic* behaviors. Garbage collection is a major source of non-determinism as it can occur at any time and suspend all user threads.

1.1.3 The Java Virtual Machine

The term "Java Virtual Machine" means any of three things [8]:

- The JVM Specification proposed by Sun Microsystems,
- A specific implementation of the JVM Specification,
- Or an instance of a specific implementation of the JVM Specification.

A JVM is a byte-code interpreter that implements Sun Microsystems's JVM specification. The JVM is specified to ensure that the semantics defined in the JLS are properly implemented. The Java compiler translates Java source into bytecodes, which are placed into a class file. There are two kinds of classes, one created by a user with the Java compiler, and the other provided by Sun Microsystems as base classes [15]. Base classes contain common or "base" functionality similar to a standard library in other languages. The JVM accepts class files as input.

Figure 1.1 depicts the architecture of a JVM. The method area and heap are shared among all threads, while each thread has its own Java stack, program counter (PC) and native method stack.[13]. As a class file is loaded, the class data is parsed from the binary class file and placed into the method area. All instantiated objects, which are created as the program runs, are allocated from the heap which is garbage collected. As each thread comes into existence, it gets its own program counter and Java stack. The program counter always points at the next bytecode to be executed. The thread's Java stack stores the state of the Java thread, including local variables, actual parameters, return values, and intermediate calculations. The JVM specification refers to the Java stack as the operand stack. The Java stack is composed of stack frames, which contain the state of a particular Java method invocation.

Figure 1.1 The JVM Architecture

1.1.3.1 Class Loader

The class loader is responsible for locating and importing the binary data for classes [13]. It verifies the correctness of the class file, allocates and initializes memory for the class variables, and assists with the resolution of symbolic references. A remarkable amount of

work is needed in the class verification process to ensure that the candidate class file is in the correct format.

The class file verifier works in four steps: checks for proper class file format, checks for proper type data, verifies bytecodes, and verifies symbolic references.

1.1.3.2 Method Area, Heap and Java Stack

The method area contains the following information about a loaded class [13]: class information, constant pool, field information, and method information. To achieve reasonable execution times, the method area information must be readily accessible to the JVM. Typically the JVM generates a method table as an array of direct references to all instance methods of the class, including those inherited from superclasses.

The heap is shared among all threads. Every instance of a class is allocated from the heap, as well as arrays, which in Java are objects [8]. The de-allocation of objects in the heap is done by the garbage collector, which reclaims heap space from unreferenced objects using a daemon thread. The JVM does not specify the structure (binary representation) of an object on the heap [8], however, an object needs to be able to quickly access both its instance data as well as its class data, which is stored in the method area [13].

When a thread comes into existence, it owns a stack, which is composed of frames, each representing the invocation of a method [13]. The JVM performs only two operations on a Java stack, i.e., push and pop. The method currently being executed by a thread is called the current method. The current method's frame is called the current frame. When the thread invokes a method, a new frame is created on that thread's stack, and it

becomes the current frame. The frame holds the method's parameters, local variables, and intermediate computations. When the method completes, either by a normal return or as a result of an exception being thrown, the frame is popped off the stack and discarded. No thread can access another thread's stack [8].

1.1.3.3 Heap Manager

In a JVM, the memory manager is also called the heap manager, which manages heap memory or heap storage [11,25]. The heap manager reclaims heap memory by an automatic storage management system, called the garbage collector. The heap manager handles the heap to ensure that there is enough room for new allocations. Garbage collection can occur at any time, and suspend all user threads. If a computation requires more heap memory than that can be made available by the heap manager, the JVM throws an *OutOfMemoryError*.

1.1.4 The Java Class File

The Java class file produced by a Java compiler is analogous to an unlinked native language object file [13]. A class file contains the complete description of a single Java class or interface. This description includes everything a JVM needs to know about a class or an interface. A class file is in a precisely defined format kept under Sun's version. This ensures compatibility with different JVM's.

The class file is a binary stream of 8-bit bytes [8]. The items in the binary stream are stored sequentially, and multi-byte values are in big-endian ordering. Within the class file, individual items represent the methods, fields, local variables, etc. The items are of

varying length, preceded by the size of the item, allowing the class file to be parsed from front to back.

1.1.5 Real Time Specification for Java

This section introduces the background of RTSJ, covering the history of real time Java, an overview of the RTSJ seven enhanced areas, and the RTSJ memory classes.

1.1.5.1 The History of Real Time Java

Until 1998, many people contributed to the realization of real time Java. Kelvin Nilsen from NewMonics and Lisa Carnahan from National Institute of Standards and Technology (NIST) led one effort, Greg Bollella from IBM led a group of companies related to Java technology and real time, and Sun had an internal real time java project [31]. In the summer of 1998, the three groups merged as a working group. Until early 1999, the group produced a document called *Requirements for Real-Time Extensions for the Java™ Platform: Report from the Requirements Group for Real-Time Extensions for the Java™ Platform* [24].

After 1999, two different groups pursued real time Java applications. The J Consortium led by Hewlett-Packard proposed proprietary extensions for real time Java. This group includes Aonix, Ericsson, Microsoft, Mitre, and NewMonics. The J Consortium released “Real Time Core Extensions” in 2000 [32]. The other group was the Real Time Java Experts Group (RTJEG) including IBM, Lucent, Motorola, and Sun. The RTJEG published “Real Time Specification for Java (RTSJ)” as a "participant review" in September 1999, and released the preliminary edition in June 2000 [4]. In October 2001,

TimeSys delivered a revised specification, a reference implementation, and a test suite to the Java Community Process (JCP) Executive Committee. In March 2003, TimeSys delivered first RTSJ-compliant JVM and embedded Java development kit [33]. Figure 1.2 shows the history of real time java.

Figure 1.2 History of Real Time Java

1.1.5.2 Overview of the seven Enhanced Areas

In RTSJ [4], seven areas are addressed and expressed as fifty-seven classes to provide application-programming interfaces (APIs), and to provide a real time software development platform suitable for real time applications. These seven areas are thread scheduling and dispatching, memory management, synchronization and resource sharing, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination, and physical memory access. A brief discussion of these areas follows.

1.1.5.2.1 Thread Scheduling and Dispatching

Scheduling is the creation of a schedule, i.e., an ordered list specifying how contending access to one or more sequentially reusable resources will be granted. Such resources may be either hardware or software. A schedule is intended to be optimal with respect to criteria.

The specification offers only general guidance for real-time scheduling. For any competition in processing resources, threads with higher priority are generally executed in preference to threads with lower priority. However, such preference does not guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion. Obviously, a stronger semantic statement about the order in which threads should execute is needed whenever you write software with temporal constraints. One way to define these semantics is through algorithms that determine how to choose the next thread for execution.

The specification defines three classes i.e., *Scheduler*, *SchedulingParameters*, and *ReleaseParameters*, and their subclasses that encapsulate temporal requirements. All of these classes are bound to schedulable objects.

The RTSJ introduced two new subclasses for real time threads: *RealtimeThread* (*RT*) and *NoHeapRealtimeThread* (*NHRT*). The *RT* class extends the Java *Thread* class. A *RT* can access objects on the heap and therefore can incur delays because of garbage collection. The *NHRT* class is a subclass of *RealTimeThread*. A *NHRT* cannot access any object on the heap, which means that it can run while the garbage collector is running (and thus avoid delays from garbage collection)[20]. A *NHRT* is suitable for code with a very low tolerance of nonscheduled delays. A *RT*, on the other hand, is more suitable for code with a higher tolerance for longer delays. A regular Java thread will do for code with no temporal constraints.

1.1.5.2.2 Memory Management

The traditional Java memory model is extended to allow the allocation of both long-lived objects and short-lived objects outside of the heap in *memory area*. There are four immediate subclasses of *memory area*. They are *physical memory*, *heap memory*, *immortal memory*, and *scoped memory*. This discussion introduces the concepts behind the different subclasses, and a more detailed discussion of the subclasses is given later.

Physical memory allows objects to be created within memory areas that have particular important characteristics, such as memory attached to a nonvolatile RAM. *Heap memory* in the RTSJ is the same as the heap memory in traditional Java, which is described in section 1.1.3.2. Objects created in *immortal memory* persist until the JVM terminates,

regardless of how many (maybe zero) references to the object may exist at any given time. Immortal memory is never garbage collected, and it is impossible to deallocate objects in immortal memory.

The RTSJ introduces the concept of *scoped memory*. A memory scope or scope is used to give bounds to the lifetime of any objects allocated within it. *Scoped memory areas* associate memory with a scope. When a scope terminates or the system leaves a scope, the system will destroy any objects that were created within the scope by calling their finalizers. A scope is obtained by the invocation and subsequent return from the *enter()* method. When a real-time thread is created, it is associated with a scoped memory area. The association ends when the *run()* method of the real time thread returns.

Scopes may be nested. A nested scope is also called a compound scope. A scope may be a null scope and a compound scope at least has a single scope. A scope is the outermost scope in a compound scope if it is the first scope to nest with other scopes. When a scope is nested into a compound scope and has no further scopes nested inside it, it becomes the current innermost scope in the compound scope.

1.1.5.2.3 Synchronization and Resource Sharing

The specification uses “priority” in synchronization for the real-time application. The system must queue all threads waiting to acquire a resource in priority order. The threads will be queued in first-in-first-out order if the active scheduling policy permits threads with the same priority. The implementation of the synchronized primitive must have a default behavior that ensures there is no unbounded priority inversion. This can be used in real-time threads and in conventional Java code if it runs within the overall RTSJ

implementation. The specification provides a mechanism, by which the programmer can override the default system-wide policy.

Real-time Java programmers always use a combination of regular Java threads, *RTs*, and *NHRTs*. The RTSJ permits locking by different thread types. If an *NHRT* attempts to lock an object that either an *RT* or regular thread has already locked, priority inheritance happens as normal. This approach has a significant problem: The *non-NHRT* that has had its priority boosted cannot execute when the garbage collector is executing. Thus, if a garbage collection is in progress, the boosted thread is suspended until collection completes. To solve this problem, the specification defines queue classes, which allow *NHRTs* to communicate with *RTs* and regular Java threads while avoiding garbage-collector-induced delays in the *NHRTs*.

1.1.5.2.4 Asynchronous Events Handling

There are two classes: *AsyncEvent* and *AsyncEventHandler*, in the asynchronous event facility. Class *AsyncEvent* object represents a hardware interrupt, or represents a computed event. The *AsyncEvent* manages two things: the dispatching of handlers when the event is fired and the set of handlers associated with the event. The application can query this set and add or remove handlers. Class *AsyncEventHandler* is a schedulable object similar to a thread. When the event fires, the system invokes *run()* methods of the associated handlers. Unlike other runnable objects, however, the *AsyncEventHandler* has associated scheduling, release, and memory parameters that control the actual execution of the handler once it is fired. When an event is fired, the system executes the handlers asynchronously, scheduling them according to the parameters. The result is that the

handler appears to have been assigned to its own thread. The implementation of *AsyncEventHandlers* needs fewer system resources than actual threads do. The specification also defines a *Clock* class to represent clocks. Many objects that measure time can be instantiated with any of the *Clock* instances the implementation offers.

1.1.5.2.5 Asynchronous Transfer of Control

The specification defines the asynchronous transfer of control, which allows you identify particular methods by declaring them to throw an *AsynchronouslyInterruptedException* (*AIE*). When such a method is running at the top of a thread's execution stack and the system calls *java.lang.Thread.interrupt()* on the thread, the method will immediately act as if the system had thrown an *AIE*. If the system calls an interrupt on a thread that is not executing such a method, the system will set the *AIE* to a pending state for the thread and will throw it the next time control passes to such a method, either by calling it or returning to it. The system also sets the *AIE*'s state to pending while control is in, returns to, or enters synchronized blocks.

1.1.5.2.6 Asynchronous Thread Termination

The system can use *AIE* directly to implement asynchronous thread termination. It also can combine *AIE* with asynchronous events to implement asynchronous thread termination. Sometimes, by design, all the methods that an instance of *RealtimeThread* uses are declared to throw an *AIE*. In this situation, when the system calls an *interrupt()* on the thread, the effect is similar to the deprecated Java *stop()* method. But, the threads

stop safely because the code was written under the condition that it would not be interrupted.

1.1.5.2.7 Physical Memory Access

The specification defines the *RawMemoryAccess* and *PhysicalMemory* classes for programmers to access physical memory directly from Java code. The *RawMemoryAccess* class defines methods for building an object representing a range of physical addresses and then accessing the physical memory with byte, word, long, and multiple-byte granularity. The *PhysicalMemory* class can be used to build a *PhysicalMemoryArea* object that represents the range of physical memory addresses in which the system can locate Java objects. In order to construct a new Java object in a particular *PhysicalMemory* object, either the *newInstance()* or *newArray()* methods must be used.

The specification implies only *get()* and *set()* methods can be used in physical memory access. The methods allow the physical memory area's contents to be accessed using offsets from the base, and copying to or from byte, short, int, or long arrays. A program can use a physical memory area to contain references to other data values by treating a value as an offset itself. It can also redefine a region of one memory area as a different memory area.

1.1.5.3 The RTSJ Memory Classes

As mentioned before, RTSJ has totally fifty seven classes, in which thirteen classes are related to memory management [4]. A overview of these memory classes follows.

- *MemoryArea* is an abstract base class of all classes dealing with representations of allocatable memory areas, including the physical memory area, immortal memory area, heap and scoped memory areas introduced earlier.
- *ImmortalMemory* is a memory resource that is shared among all threads. Objects allocated in *ImmortalMemory* live until the end of the application and are never subject to garbage collection, although some GC algorithm may require a scan of the *ImmortalMemory*. An immortal object may only contain references to other immortal objects or to heap objects.
- *ImmortalPhysicalMemory* allows objects to be allocated from a range of physical memory with particular attributes, has the same restrictive set of assignment rules as *ImmortalMemory* memory areas, and may be used in any constructor where *ImmortalMemory* is appropriate.
- *HeapMemory* is a singleton object that allows logic within other memory to allocate objects in the Java heap.
- *ScopedMemory* is the abstract base classes dealing with representations of memory spaces with limited lifetime. It has three subclasses: *LTMemory*, *VTMemory* and *ScopedPhysicalMemory*. This memory area is a connection to a particular region of memory, and reflects the region's current status. The scoped memory is valid as long as there are real time threads with access to it. A reference is created for each accessor when either a real time thread is created with a *ScopedMemory* object as its memory area, or a real time thread runs the *enter()* method for the area. When the last reference to the object is removed, finalizers are run for all objects in the memory area, and the area is emptied.

- *VTMemory* is a subclass of *ScopedMemory*. The execution time of an allocation from a *VTMemory* area takes a variable amount of time. However, *VTMemory* areas are not subject to garbage collection and objects within may not be moved.
- *LTMemory* represents a memory area, allocated per *realtimeThread*, or for a group of real time threads, guaranteed by the system to have linear time allocation. The memory area described by a *LTMemory* instance does not exist in the Java heap, and is not subject to garbage collection. Thus it is safe to use a *LTMemory* objects as the memory area associated with a *NoHeapRealTimeThread*, or enter the memory area using the public void *enter()* method within a *NoHeapRealTimeThread*. An *LTMemory* area has an initial size. Enough memory must be committed by the completion of the constructor to satisfy this initial requirement.
- An instance of *ScopedPhysicalMemory* allows objects to be allocated from a range of physical memory with particular attributes, determined by their type. This memory area has the same restrictive set of assignment rules as *ScopedMemory* memory areas.
- *PhysicalMemoryFactory* is available for use by the various physical memory accessor objects to create objects of the correct type that are bound to areas of physical memory with the appropriate characteristics.
- An instance of *RawMemoryAccess* models a range of physical memory as a fixed-size sequence of bytes. This class has one subclass: *RawMemoryFloatAccess*. A full complement of methods allow access to the contents of the physical memory area through offsets from the base, interpreted as byte, short, int, or long data values. A raw memory area cannot contain references to Java objects. *RawMemoryAccess*

allows a real time program to implement device drivers, memory-mapped I/O, flash memory and similar low-level software.

- The *RawMemoryFloatAccess* class holds the methods for accessing a raw memory area float and double types. Implementations of this class are required if and only if the underlying JVM supports floating point data types.
- *Memory parameters* can be given on the constructor of a *RealtimeThread*. These can be used for the purposes of admission control by the scheduler, and for the purposes of pacing the garbage collector to satisfy all of the thread allocation rates. When a reference to a *MemoryParameters* object is given as a parameter to a constructor, the *MemoryParameters* object becomes bound to the created object. Changes to the values in the *MemoryParameters* object affect the constructed object. If given to more than one constructor, then changes to the values in the *MemoryParameters* object affect all of the associated objects.
- The *GarbageCollector* class is an abstract class, which has two subclasses: *IncrementalCollectorExample* and *markAndSweepCollectorExample*. The system provides dynamic and static information characterizing the temporal behavior and imposed overhead of any garbage collection algorithm provided by the system. Implementations are allowed to provide any set of methods in these subclasses as long as the temporal behavior and overhead are sufficiently categorized.

1.1.6 Real Time Memory Management

Memory management is a complex field of computer science. It involves supplying memory as needed, and recycling memory for reuse when it is no longer required [34].

There are two approaches to recycling memory: manual memory management and automatic memory management known as garbage collection [34].

1.1.6.1 Manual Memory Management

In manual memory management, the programmer has direct control over memory allocation and recycling. The key feature of a manual memory manager is that the memory manager does not recycle any memory without manual intervention. Manual memory management was common in early languages such as Algol, C, C++, COBOL and Pascal [34].

In an operating system, the most significant part of memory management is virtual memory. However, it is not possible to build real time applications on a system with virtual memory since the random and long delays introduced when RAM is exhausted and swapping is required is intolerable.

QNX [35] provides dynamic memory allocation using the *malloc()*, *realloc()* or *calloc()* functions, provides the *free()* function to release memory allocation, and provides a microkernel architecture with full memory protection between operating system components such as the filesystem, TCP/IP, and so on.

RTLinux [35] achieves the real time goal of predictability by preallocating the resources the threads will use at run time. There is no memory protection between threads and the kernel, and or between threads themselves, and all the application threads and the RTLinux kernel run in the same address space.

1.1.6.2 Automatic Memory Management

Many modern languages use mainly automatic memory management, for example: BASIC, Java, JavaScript, Modula-3, Perl, Prolog, Scheme and Smalltalk. However, garbage collected memory has always been considered an obstacle to real time programming because the garbage collector (GC) introduces unpredictable latencies. A real time garbage collector is triggered like a task, and when another task with higher priority arrives, the GC is stopped. High priority real time tasks cannot tolerate unbounded garbage collection pauses, and time taken for executing the GC must be bounded. Some solutions that address this real time memory management problem follow [36].

- In the Java Real Time API (v0.2) [38], tasks execute in a pre-allocated space with the GC deactivated. Thus, the interruption of the GC by these tasks is always safe. The *PreallocationContext* class has two methods: *available()* to get the amount of available memory in the area, and *replenish()* to possibly cause a garbage collection to clean up space.
- In RTSJ (V0.8.1) [4], the *memoryArea* abstract class supports scoped, physical and immortal memory. The *GarbageCollector* class provides methods for getting information about the GC behavior.
- In the Real Time Core Extension for the Java Platform [24], this solution has two separate heaps, one for baseline threads, and the other for core tasks. Since tasks and threads may share objects, memory sharing must be supported by the underlying RTOS. The *AllocationContext* class extends the *CoreTask* and supports the *release()*

method, which makes all objects allocated within a context eligible by the GC. Every task has an associated *AllocationContext* object.

- The Portable Executive for Reliable Control (PERC) [10], has a sophisticated real time GC, that is accurate, defragmenting and deterministic. The GC works by dividing its total effect into small bursts. After the GC is preempted by a higher-priority task, it resumes where it left its execution.
- G. Back proposes Java Operating Systems in which each activity has its own heap, distinct from heaps of the other activities [38]. The GC works separately in each heap.
- Another way for real time garbage collection is suggested as stack-based allocation, which is desirable because execution time properties are easier to capture than heap allocation[35].

1.2 The Thesis

This section introduces the objective of the thesis, the contribution of the thesis, and the scope of the thesis.

1.2.1 The Objective of the Thesis

In a traditional Java application, the memory management is not usually an issue due to the automatic reclamation of blocks of memory by the garbage collector in the JVM. The algorithms and implementations for the garbage collection vary in the amount of non-determinacy they add to the execution of a program. So, the memory manager in a traditional JVM cannot satisfy the memory management requirements for real-time Java due to unpredictable latencies.

A RTJVM requires a RTMM that satisfies the requirements of the RTSJ, and ensures that memory manipulations are deterministic for real time Java applications. This memory manager, together with other parts of the real time Java virtual machine, forms a RTJVM, which can be used for real time Java application as specified in the RTSJ. Figure 1.3 shows a real time memory manager embedded in a RTJVM. Since the Java is architecture neutral, therefore, the thesis objective is to design, implement, evaluate and test a general real time memory manager.

Figure 1.3 The Embedded Memory Manager in a RTJVM

1.2.2 The Scope of the Thesis

In order to have a general real time memory manager, it is necessary to identify the related memory classes that contain methods reflecting all necessary functions in the RTMM. From the RTSJ memory classes described in section 1.1.5.3, four classes are found and identified. These classes are *MemoryArea*, *ImmortalMemory*, *ScopedMemory*, and *HeapMemory*, which are satisfied for the RTMM and analyzed in chapter 2. The

other memory classes are not identified due to the following different reasons. The classes *PhysicalMemoryFactory*, *ImmortalPhysicalMemory*, *ScopePhysicalMemory*, *RawMemoryAccess*, and *RawMemoryFloatAccess* focus on physical memory, which is machine-dependent. The classes *VTMemory* and *LTMemory* concerns with different ways of allocation. The *MemoryParameters* class is a class related to parameters. The *garbageCollector* class and two its direct subclasses *IncrementalCollectorExample* and *MarkAndCollectorExample* are all related to garbage collection of heap memory.

Therefore, this thesis focuses on a real time memory manager, its interfaces and its internal software components that support the interfaces. The interfaces are the only components that communicate with threads in a RTJVM. For real time properties, this thesis is also concerned with the deterministic behaviors of the software related to the real time requirements described for the RTSJ memory manager.

The management of threads is outside of the scope of the thesis. However, assumptions will be made for threads in the RTJVM to ensure that the memory manager can properly fit within a RTJVM. The thesis is also not concerned with the heap implementation and some other kinds of the RTSJ memory types such as physical memory.

1.2.3 The Contributions of the Thesis

The following contributions have been made by this thesis:

- The related memory classes described in the RTSJ are identified and analyzed to determine how the functions of these classes might be realized by a RTJVM. The analysis is summarized in a proposed Real Time Memory Manager Specification (RTMMS) for the RTJVM.

- A RTMM is designed based on the proposed RTMM specification. This RTMM can be used in a RTJVM to satisfy real time Java applications as described in the RTSJ.
- The interfaces together with their supporting software components for the memory manager are based on necessary assumptions about the RTJVM, implemented in the C language, and evaluated using complexity analysis to conclude their deterministic behaviors.
- A testing system is designed and implemented to test each internal software component as white-box tests, and to test each interface as black-box tests.

CHAPTER 2

Analysis of the RTSJ Memory Classes

In order to specify a real time memory manager, it is important to identify the related RTSJ memory classes and analyze the methods in these classes. The analyses of the methods are useful in the determination of how these methods might be realized by a RTJVM. The identified memory classes are based on extensions to the traditional Java memory model. These extensions support memory management in a way that the memory model does not interfere with the real time code's ability to provide deterministic behaviors, and allows the allocation of both short-lived and long-lived objects outside of the garbage-collected heap.

2.1 The RTSJ Memory Area

The RTSJ introduces the concept of a memory area to represent an area of memory [4]. The memory area gives the definition of regions of memory for the allocation of objects in, or outside of, the heap. The memory area includes the singleton heap memory area, the singleton immortal memory area, and scoped memory areas currently in use by threads. The heap memory area gives the definition of regions of memory in the traditional Java heap. The immortal memory area and the scoped memory area give

definition of regions of memory outside of the heap. Immortal memory is a memory resource shared among all threads in an application. Figure 2.1 shows the three types of memory areas in the RTSJ memory area.

Figure 2.1 The RTSJ Memory Areas

2.2 The Lifetimes of Objects in RTSJ Memory Areas

The lifetime of an object depends on the memory area in which it is allocated. In heap memory, an object's life depends on garbage collection. The immortal memory area contains objects whose lifetime matches that of the application. Once created, immortal objects continue to exist and are freed only when the Java runtime environment terminates. Immortal objects are never subject to garbage collection (GC) or movement. In a scope, an object lives until the scope terminates. Figure 2.2 shows the lifetimes of objects in different memory areas.

Figure 2.2 The Lifetimes of Objects in RTSJ Memory Area

2.3 References to Objects in Different Memory Areas

Memory areas that exist outside the heap are not garbage-collected. Objects allocated outside of the heap memory area may contain references to the objects in the heap. For example, an immortal object may contain references to the immortal objects or to heap objects. Thus, the garbage collector must be able to scan memory areas outside the heap to find references to objects within the heap, and to preserve the heap integrity. Due to the limited lifetime of scoped objects, the RTSJ limits references to scoped objects by

means of the restricted set of assignment rules summarized in table 2.1. The rules prevent longer-lived objects from referencing objects in scoped memory, which are possibly shorter lived. The heap memory area represents the traditional Java heap, and the RTSJ does not change the determinant of the lifetime of objects on the heap. Heap objects may only contain references to immortal objects or other heap objects. The rules are:

- A reference to an object in *ScopedMemory* can never be stored in an object allocated in the Java heap or in the immortal memory, but can be stored in objects allocated in the same *ScopedMemory* area, or in a nested inner *ScopedMemory* area.
- References to immortal or heap objects may be stored in an object allocated in a *ScopedMemory* area.

Table 2.1 The Set of Assignment Rules

reference to	Heap	Immortal	Scoped
Heap	yes	yes	no
Immortal	yes	yes	no
Scoped	yes	yes	yes, if **

if **: if same, outer or shared scope.

2.4 The RTSJ Memory Classes

In the RTSJ [4], there is one section, called memory management, in which four classes are identified as required in this thesis. They are: *MemoryArea*, *HeapMemory*, *ImmortalMemory*, and *ScopedMemory*. Their relationships are generalization/specialization hierarchy as shown in figure 2.3. They have some attributes in common, and others that are different. The common attributes are abstracted into the generalized class (*MemoryArea*). The different attributes are properties of the specialized

subclasses: *ImmortalMemory*, *HeapMemory* and *ScopedMemory*. The *ScopedMemory* area might have more than one instance. There is only one singleton instance in each of the *ImmortalMemory* area and the *HeapMemory* area.

Figure 2.3 Relationships among four RTSJ Memory Classes

2.4.1 Analysis of *MemoryArea*

The *MemoryArea* class is a *public abstract* class, which means that it may be accessed by any Java codes that access the package in which it is declared (*public*), and that it is incomplete (*abstract*). The *MemoryArea* class is the abstract base class of all classes dealing with representations of allocatable memory areas, including the immortal

memory area, heap memory and scoped memory areas. Table 2.2 shows the *MemoryArea* class in the RTSJ. There are a constructor and seven methods in this class.

Table 2.2 The *MemoryArea* Class

```
public abstract class MemoryArea
{
    protected MemoryArea (long sizeInBytes)
    public void enter (java.lang.Runnable logic)
    public MemoryArea getMemoryArea (java.lang.Object, object )
    public long memoryConsumed ()
    public long memoryRemaining ()
    public synchronized java.lang.Object newArray (java.lang.Class type, int
    number)
    public synchronized java.lang.Object newInstance (java.lang.Class type)
    public long size()
}
```

The constructor and the methods are analyzed as follows:

- In the *MemoryArea()* constructor, the parameter *sizeInBytes* is the size of memory to allocate, in bytes. This constructor is *protected* to restrict the constructor's accessibility from outside the package.
- In the *enter()* method, the parameter *logic* is the *runnable* object (*Java.lang.Runnable*) whose *run()* method should be executed. The method *enter()* associates this memory area to the current real time thread for the duration of the execution of the *run()* method. During this bound period of execution, objects are allocated from the memory area until the *enter()* method is exited.

- The *getMemoryArea()* method returns the *MemoryArea*, in which the given object is located. The parameter *object* is a *Java.lang.Object*. The returned *MemoryArea* is exactly one incarnation, no matter how many instances of the class may eventually be created.
- The *memoryConsumed()* method returns an exact count, in bytes, of all of the memory currently used by the allocated objects.
- The *memoryRemaining()* method returns the total amount of memory currently available for future allocation to objects, measured in bytes.
- The *newArray()* method allocates an array of *type* in this memory area. The parameter *type* is the class of the elements of the new array. The parameter *number* is the number of elements in the array. This method is *synchronized*, which means that mechanisms are provided for synchronizing the concurrent activity of threads such that threads do not interfere with each other.
- The *newInstance()* method allocates an object in this memory area. The parameter *type* is the class of which to create a new instance. This method is also *synchronized*.
- The *size()* method returns the size of the memory area in bytes. The returned value is the current size.

All the methods described above are *public*, which means that they are accessible throughout the package where they are declared.

2.4.2 Analysis of *ScopedMemory*

Table 2.3 shows the RTSJ *ScopedMemory* class [9]. The *ScopedMemory* class has one direct super class, *MemoryArea*, and has one constructor and eleven methods, six of

which are inherited from *MemoryArea*. The *enter()* method is overridden in this class. The inherited methods are analyzed previously in section 2.4.1.

ScopedMemory is a *public abstract* base class.

- The *ScopedMemory()* constructor creates a new *ScopedMemory* area. The parameter *size* is the size of the new *ScopedMemory* area in bytes.
- The *enter()* method overrides its superclass' *enter()* method. Scoped memory allows the allocation of objects using the memory area as a scope, which bounds the lifetime of objects allocated within it. The *enter()* method is a direct way to enter a scope (the other one is by a *RealTimeThread*, to which the scope is attached before the thread executes its *run()* method). The *enter()* method associates this *ScopedMemory* area with the current real-time thread for the duration of the execution of the *run()* method of the given *Java.lang.Runnable*. During this bound period of execution, all objects are allocated from the *ScopedMemory* area until a nested one takes effect, or the *enter()* method is exited. The parameter *logic* is a Java *runnable* object which contains the code to execute and whose *run()* method represents its entry point.
- The *getMaximumSize()* method returns the maximum size the memory area can attain. If this is a fixed size memory area, the returned value will be equal to the initial size.
- The *getOuterScope()* method returns the outer scope of the current scope. If the outer scope is the outermost scoped memory, then the returned memory area is the one associated with the current *RealTimeThread*.
- The methods *getPortal()* and *setPortal()* are related to internet applications and are outside of the scope in the dissertation.

All the methods described above are *public*.

Table 2.3 The *ScopedMemory* Class

```
public abstract class ScopedMemory extends MemoryArea
{
    public ScopedMemory(long size)
    public void enter(java.lang.Runnable logic)
    public int getMaximumSize()
    public memoryArea getOuterScope()
    public java.lang.Object getPortal()
    public void setPortal(java.lang.Object object)
}
```

2.4.3 Analysis of *ImmortalMemory*

Table 2.4 shows the *ImmortalMemory* class in the RTSJ [4]. The *ImmortalMemory* class is *final*, which means that there are not any subclasses. This class has one constructor and eight methods. The constructor and seven methods are inherited from the *MemoryArea* class and are analyzed in the section 2.4.1 The only new method is a *static instance()* method, which returns the singleton *ImmortalMemory* space. The method is *static*, which means that there is only one incarnation.

Table 2.4 The *ImmortalMemory* Class

```
public final class ImmortalMemory extends MemoryArea
{
    public static ImmortalMemory instance()
}
```

2.4.4 Analysis of *HeapMemory*

Table 2.5 shows the *HeapMemory* class in RTSJ [4]. The *HeapMemory* class is *final* and has one constructor and eight methods. The constructor and seven methods are inherited from *MemoryArea*, and are analyzed before in section 2.4.1. The *instance()* method is *static* and returns the singleton *HeapMemory* object.

Table 2.5 The *HeapMemory* Class

```
public final class HeapMemory extends MemoryArea  
{  
    public static HeapMemory instance()  
}
```

Chapter 3

Real Time Memory Manager Specification

Based on the JLS, JVMs, and RTSJ, a real time memory manager specification (RTMMS) is outlined in this chapter. The RTMMS covers the memory manager's preconditions, requirements, functionality and interfaces. The specification provides guidelines for the design of the memory manager in the next chapter.

3.1 Preconditions for the Memory Manager

Since the memory manager is a key part of the Java virtual machine [22], it requires two preconditions from the RTJVM to satisfy the requirements from the RTSJ, the JVMs, and the JLS. The preconditions allow the memory manager to be designed and implemented as a black box.

3.1.1 The Thread Manager inside the RTJVM

The memory manager requires the thread manager in the RTJVM control the synchronization of threads while acquiring their memory from the memory manager, i.e., the internal access to the memory manager is synchronized outside the memory manager by the thread manager. This means that the thread manager is similar to the thread

manager in JRE 2.0 to support concurrent threads [3]. This precondition is important in satisfying concurrency requirements in the RTSJ.

3.1.2 Initial Memory Resource

The memory manager [22] requires initialization information from the RTJVM, which includes a block of initial memory resource. This precondition must be satisfied during the initialization of the RTJVM, and the memory manager must be triggered to initialize as soon as this precondition is satisfied [23]. Figure 3.1 shows the memory resources in a RTJVM. The black arrow is a pointer to the shaded memory block passed to the memory manager.

Figure 3.1 Memory Resources in a RTJVM

The provided memory is not acquired by the memory manager, but is given to the memory manager during RTJVM initialization. The provided memory, as shown in figure 3.2, is considered as two parts: the first part, called the manager memory resource, is used to support the memory manager and the second part, called the RTSJ memory resource, is used to provide memory as heap memory, immortal memory or scoped memory.

Figure 3.2 The Provided Memory from a RTJVM

3.2 Requirements for the Memory Manager

Based on the preconditions and the analyses of the identified RTSJ memory classes, the following requirements need to be satisfied for the memory manager for both memory manipulation and scope manipulation.

3.2.1 Requirements on Heap Memory Manipulation

The memory manager is required to provide heap memory as it is originally used in traditional Java. The memory manager is not required to be deterministic during the heap memory manipulation. The garbage collector is not part of this research, however, a method to invoke garbage collection is included. To allocate heap memory, the memory manager first checks if GC is required. If GC is required to start, the memory manager invokes the *activateGC()* interface to start the garbage collector in the RTJVM. Otherwise, the memory manager offers the required heap memory if there is memory available, or throws an exception if there is insufficient memory available.

3.2.2 Requirements on Immortal Memory Manipulation

The memory manager is required to offer immortal memory. Since objects in immortal memory live until the end of the application, there is no requirement to handle the return of immortal memory to the memory manager. The memory manager must have absolutely deterministic behaviors during immortal memory manipulation. The memory manager can allocate immortal memory until there is no more memory available. If there is insufficient memory available, the memory manager throws an exception.

3.2.3 Requirements on Scoped Memory Manipulation

The memory manager is required to deal with scoped memory manipulation, which includes allocating scoped memory, locating a scope, associating scoped memory with a scope, and recovering scoped memory. Scoped memory manipulation is required to have deterministic behaviors. Scoped memory is used for short-lived objects, which are different from those in immortal memory.

Locating a scope requires identifying the required scope. Recovering scoped memory requires removing the scoped memory from a scope before the scope exits. The recovered scoped memory is recycled for future use by the memory manager.

3.2.4 Requirements on Scope Manipulation

The memory manager is required to manipulate scopes for scoped memory manipulation, and must have absolutely deterministic behaviors during scope manipulation. Scope

manipulation includes scanning a scope or compound scope, nesting a scope into a compound scope, and exiting a scope from a compound scope.

Scanning a scope requires returning memory information (address, size and contents) associated with the scope. Scanning a compound scope requires returning memory information associated with every scope in the compound scope by scanning each scope. Entering a scope requires nesting a new scope as the innermost scope in a compound scope, and exiting a scope requires recovering associated scoped memory and removing the scope from a compound scope.

3.3 Functionality in the Memory Manager

With requirements for the memory manager described in the last section, the functionality of the memory manager is specified in this section. The functionality allows the memory manager to support a real time Java virtual machine.

The memory manager can:

1. Initialize itself within a RTJVM,
2. Provide heap memory,
3. Activate garbage collection,
4. Provide immortal memory,
5. Enter a scope,
6. Nest an entered scope into its compound scope,
7. Associate an entered scope with scoped memory,
8. Provide scoped memory associated with its scope in the compound scope,
9. Recover scoped memory in a scope before the scope exits,

10. Remove a scope from a compound scope when the scope exits,
11. Scan a scope to obtain scoped memory information in the scope,
12. Scan a compound scope.
13. Throw an exception if a requirement cannot be met.

3.4 Interfaces to the Memory Manager

The functionality of the memory manager, described in the last section, is realized via its nine interfaces, and supported by internal software components. Figure 3.3 shows the interfaces, i.e., *initManager()*, *getHeapMem()*, *activateGC()*, *getImmortalMem()*, *getScope()*, *getScopedMem()*, *scanNestedScope()*, *endScope()*, and *exceptionCatch()*. Each interface responds to some partial functionality for the memory manager. The responsibilities for each interface are shown as follows:

- The *initManager()* interface is for part 1, i.e., to initialize the memory manager, so the memory manager is ready for uses within the RTJVM.
- The *getHeapMem()* interface is for part 2 to provide heap memory.
- The *activateGC()* interface is for part 3 to invoke the garbage collector in RTJVM,
- The *getImmortalMem()* interface is for part 4 to provide immortal memory.
- The *getScope()* interface is for parts 5 and 6 to allow a thread to enter a scope, and to nest this scope (or to create the scope as a new compound scope).
- The *getScopedMem()* interface is for parts 7 and 8 to provide a thread scoped memory associated with a scope in a compound scope.
- The *endScope()* interface is for parts 9 and 10 to remove a scope.

- The *scanNestedScope()* interface is for parts 11 and 12 to provide scoped memory information in a scope or in a compound scope.
- The *exceptionCatch()* interface is for part 13. This function throws different exceptions for different failures.

The interfaces, except the initialization interface, are used to communicate with threads. The memory manager is not required to have deterministic behaviors for both *initManager()* interface and *getHeapMem()* interface plus their supporting software components, but is required to be deterministic for all other interfaces plus their supporting software components.

Figure 3.3 Interfaces to the Memory Manager

Chapter 4

Design of the Memory Manager

The RTMMS described in the last chapter provides guidelines for the design of the memory manager. This chapter presents the design of the memory manager from low level to high level, followed by an example with both memory and scope manipulations at the design level.

4.1 Design of the Memory Manager

The design of the memory manager contains: the basic data structures, the public parameters, the principles of scope manipulation, and memory manipulation. The basic internal data structures include the memory node structure (*memNode*), the scope node structure (*scopeNode*), memory manipulation structure (*list*) and the memory structure (*mem*). The *memNode* is designed to directly relate to application memory resources as described in section 3.1.2, the *scopeNode* is for scope manipulation, the *list* is for both *memNode* manipulation and *scopeNode* manipulation, and the *mem* is for organization of *memNode* lists. The details on these topics are discussed in the following sections.

4.1.1 *List*

The manipulation of scopes, scope memory and immortal memory are required to be deterministic. To satisfy this requirement, a linked list data structure is used in a stack style [26]. The manipulation for a list is designed to always occur at the first node in the list. This allows nodes in the list to work as “last in, first out” elements on a stack. When a new node is required to be added in a list, it is inserted as the first node in the list as the stack “pushes” it on the top of the stack. When a node is required to be removed from a list, the first node in the list is removed as the node is “popped off” the stack.

All lists in the memory manager belong to this kind of list. They are used to organize both memory nodes and scope nodes as introduced in section 4.1.2 and section 4.1.3, respectively.

4.1.2 *memNode*

A *memNode* refers to a memory node that directly relates to a memory resource. The *memNode* data structure contains three elements: one pointer (*link*) pointing to the next node in the list, one pointer to a memory unit, and an integer (*thisNodeSize*) indicating the size of the memory referenced by the *memoryResourcePointer* pointer. Figure 4.1 shows the *memNode* structure. The memory referenced by a memory node may be heap memory, immortal memory, or scoped memory.

The type for a pointer to a *memNode* is *memoryNodePointer*.

Figure 4.1 The *memNode*

4.1.3 *scopeNode*

A scope is expressed as a data structure called a *scopeNode*, shown in figure 4.2. There are four elements in the scope structure: one pointer (*next*) to its outer scope, one pointer (*scopeMemHeadP*) to a list of memory nodes (*memNodes*) containing objects allocated in the scope, one integer *sNum* indicating the number of all its outer scopes, and one integer *nNum* indicating the number of scoped memory nodes in this scope. The default *sNum* is one and the default *nNum* is zero. So, in a scope, there is only one list for scoped memory nodes, and the memory units referenced by memory nodes in the list may be of different *thisNodeSizes*.

The type for a pointer to a *scopeNode* is *scopePointer*.

Figure 4.2 The *scopeNode*

A compound scope is expressed as a scope list as described in section 4.1.1. The innermost scope in a compound scope is the first scope in a scope list. When a new *scopeNode* is nested into a scope, it is added as the first node and its *sNum* is set to one more than the *sNum* in its outer scope node. The new scope is now the current scope. The memory manipulation in a compound scope only occurs in its current scope.

4.1.4 Comparison of Scopes in Java, in the Memory Manager and on a Stack

In Java, a scope means a pair of {}, and a nested scope means that a scope contains other scope(s), such as {..{..}..}, or {..{..}..{..}..}. Figure 4.3 demonstrates a group of nested scopes in Java, in the memory manager and on a stack. In Java, as shown in figure 4.3A, *scope y* contains *scope x* ... *scope b* and *scope a*, *scope x* contains *scope b* and *scope a*, similarly, *scope b* contains *scope a*, and *scope a* is the innermost scope. The *scope y* is entered first, the memory manipulation in *scope y* stops when *scope x* is entered and continues after exiting *scope x*. When *scope x* is entered, scoped memory manipulation only occurs in *scope x* until its inner scope is entered and manipulation in *scope x* does not continue until its inner scope is exited. For the innermost scope, when *scope a* is nested, memory manipulation occurs in *scope a* until it is exited.

Figure 4.3B shows the corresponding scope manipulation in the memory manager. Scope *y* is first added as the first node, with *sNum* equal to 1 and *nNum* equal to 0. The memory manipulation in *scope y* occurs with updating the *nNum* until *scope x* is added, and continues after *scope x* is removed. When *scope x* is added as the node, the *sNum* value is set to 2 and the *nNum* value is set to 0. Scoped memory manipulation then occurs in *scope x* with updating the *nNum* until its inner scope is added as the first node, and

continues after its inner scope node is removed. The most inner scope, *scope a*, is the last one added to the front of the scope list, with *sNum* set to one more than that in *scope b* (25 in this case) and *nNum* equal to 0. Memory manipulation occurs with updating the *nNum* until it is removed from the linked list. Before the current scope is removed, all its memory nodes must be recovered. The recovery of memory nodes involves recycling the memory nodes back into the memory manager.

Figure 4.3C demonstrates that the above scope manipulations are similar to the manipulations on a stack. The *scopes y, x, ... b, and a* are pushed on the stack respectively, and are popped off the stack in the reverse order. Memory manipulation only occurs in the current scope, which is the top scope in the stack.

Figure 4.3 The Nested Scope in Java, in the Memory Manager, and as a Stack

4.1.5 *mem*

Free memory resources available for allocation are organized into lists of memory nodes, in which each memory node in a list is of the same size. There is a *mem* data structure designed to organize the free memory node lists in the memory manager via an array of memory node pointers expressed as *memHeadP[]*. These memory nodes are available for the memory manager to offer the memory as heap memory, immortal memory and scoped memory. A used scoped memory node or a used heap memory node may be recovered back to the respective memory list in *mem* based on its *thisNodeSize* value.

Figure 4.4 shows the *mem* data structure.

The type for pointer of *mem* is *memPointer*.

Figure 4.4 The *mem* Structure

4.1.6 Memory Manager Parameters

In the memory manger, there are the following public parameters: integers *initialNum*, *nodeNum*, *scopeNum* and *countNum*; an array of integers expressed as *Di_Info[][]* for distribution information and four arrays of integers expressed as *nodeCount[thisNodeSize]*, *heapCount[thisNodeSize]*, *immortalCount[thisNodeSize]* and

scopeCount[thisNodeSize] for recording the respective memory nodes, *memNode* pointers expressed as *tempMemNode[initialNum]* for memory node manipulation, *scopeNode* pointers expressed as *tempScope[initialNum]* for scope manipulation and as *tempNestScope[initialNum]* for the outermost scope. These parameters need $4*4 + 3*4 * initialNum + 4*6*countNum$ bytes memory arranged from the manager memory resource, as described in section 3.1.2.

4.1.7 Initialization of Memory Manager

The initialization information supplied to the memory manager consists of a block of memory (as described in section 3.1.2) and two arrays of integer information. One array is *Di_Info[][]*, introduced in last section, containing distribution information related to memory nodes as $[(n1, m1), (n2, m2)... (nk, mk)]$, where n_i is the number of memory units with a size of m_i bytes (i from 1 to k). The other array contains information related to memory for the memory manager parameters described before. The initialization includes following four steps:

1. Based on the distribution information, the memory manager first initializes a *mem* pointer, *memNode* pointers, and *scopeNode* pointers, and then divides the provided memory resource into the memory manager resource and the application memory resource, and further divides the application memory resource into memory units of the required size. Each unit is managed by a *memNode* structure via a pointer (*memoryResourcePointer*) and *thisNodeSize* is set equal to the size of the unit. Figure 4.5 shows the distributed application memory based on the provided distribution information.

Figure 4.5 Distributed Application Memories

2. The memory manager organizes all memory nodes referencing to memory units into k linked lists. The memory nodes in each list refer to memory with the same size, i.e., the parameter *thisNodeSize* in each memory node has the same value as $m1$, $m2$... or mk .
3. The memory manager uses *mem* to organize these k linked lists via a array of k memory node pointers (*memHeadP[k]*) described in section 4.1.5. After this step of

initialization all memory node pointers are organized in their linked lists. The nodes are of size equal to $m1$ (bytes) in the first list, equal to $m2$ in the second list, similarly equal to mk in the last list. Figure 4.6 shows the state of the memory manager after initialization. The memory resource is distributed via pointers from memory nodes organized in lists.

Figure 4.6 The State of Memory Manager after Initialization

4.1.8 Invocation of the *activateGC()* Interface

The *activateGC()* interface is invoked if the number of available memory nodes in any free memory linked list is less than twice the number of used heap memory nodes with the same *thisNodeSize*, i.e. $heapCount[i] > nodeCount[i]/2$, where i is from 1 to k .

The implementation of this interface is merely for arbitrary tests, since this interface depends on the specification of a RTJVM and also depends on the implementation of the garbage collector inside the RTJVM.

4.2 An Example for Memory and Scope Manipulations at Design Level

Figure 4.7 shows an example with scope manipulation and memory manipulation in both *mem* and in the current scope. Assume that there are only three lists of memory nodes in *mem* with memory sizes 8-byte, 16-byte and 32-byte respectively. This example is also used in the testing system described in chapter 6.

Figure 4.7A shows a memory manager state. *Scope1* has n allocated scoped memory blocks and a link to its $s-1$ outer scopes. There are no nodes allocated to immortal memory and heap memory. In the three lists of memory nodes, the nodes have *thisNodeSize* equal to 8, 16 or 32 respectively.

In figure 4.7B, a new scope has been created within *scope1*. The new scope is the current scope expressed as *C-scope*, with *nNum* equal to zero and *sNum* equal to $s + 1$. A 32-byte heap node, a 16-byte heap node, an 8-byte immortal node, a 32-byte immortal node, and a 16-byte immortal node have also been allocated. Note that the *mem* lists have been modified: node 1 in the 8-byte list, nodes 1 and 2 in both the 32-byte list and the 16-byte

list have been removed from their lists and allocated as immortal memory and heap memory.

In figure 4.7C, a 32-byte scoped node, a 16-byte scoped node, an 8-byte scoped node, an 8-byte immortal node, an 8-byte scoped node and a 32-byte scoped node have been allocated. So, node 3 in the three lists, nodes 2 and 4 in the 8-byte list and node 4 in the 32 byte list have been removed from their lists and allocated as scoped memory and immortal memory. The parameter $nNum$ is updated to 5.

In figure 4.7D shows the memory manager state after *C-scope* exits. First, scoped memory in *C-scope* has been recovered. The order for recovering the scoped memory is in reverse with the order in which the scoped memory is acquired. The recovered scoped memory nodes are returned back in their respective memory lists. Secondly, *C-scope* exits from the nested scope. The *scope1* is the current scope again.

Figure 4.7A An Example for Scope and Memory Manipulations

Figure 4.7B An Example for Scope and Memory Manipulations

Figure 4.7C An Example for Scope and Memory Manipulation

Figure 4.7D An Example for Scope and Memory Manipulation

Chapter 5

Implementation and Evaluation of the Memory Manager

The implementation of the memory manager using the C language [28] is based on the design of the memory manager in chapter 4 and satisfies the RTMMS specified in chapter 3. The implementation of the memory manager begins from three basic low-level data structures to nine high-level interfaces via sixteen supporting methods. The supporting methods are internal in the memory manager. Each interface may invoke one or more supporting methods. Each supporting method may be invoked by one or more interface. Some supporting methods are only invoked by other supporting methods.

Following the implementation of internal methods and interfaces, their deterministic behaviors are evaluated using big-O analysis [29,30]. The deterministic behaviors in each interface are deduced from that of its invoked internal methods. The deterministic evaluation for the memory manager is made based on the evaluation of each interface. The deterministic evaluation provides a guideline for the implementation of the software components in the memory manager.

5.1 Implementation of Basic Data Structures

Figure 5.1.1 shows the implementation of three basic data structures from section 4.1 and shows their pointers types.

Figure 5.1.1 Implementation of *memNode*, *scopeNode* and *mem*

A	<pre>struct memNode { struct memNode * link; int thisNodeSize; void * memoryResourcePointer; }; typedef struct memNode * memoryNodePointer;</pre>
B	<pre>struct scopeNode { memoryNodePointer scopeMemHeadP; struct scopeNode * next ; int sNum; int nNum; }; typedef struct scopeNode * scopePointer;</pre>
C	<pre>struct mem { memoryNodePointer memHeadP[mk]; }; typedef struct mem * memPointer;</pre>

5.2 Implementation and Evaluation of Internal Methods

There are sixteen internal methods in the memory manager. All of them are implemented with deterministic behaviors to ensure that the supported interfaces have deterministic behavior, and are evaluated using big-O analysis. Figure 5.2 provides a calling graph showing the interaction among interfaces and internal methods, and showing a summary of the big-O analysis.

Figure 5.2 Interactions among Interfaces and Internal Methods with Big-O Analyses

5.2.1 Implementation of *prepareMemP()*, *prepareMemNodeP()* and *prepareScopeNodeP()*

These methods are implemented in a similar way. The *prepareMemP()* method initializes a pointer to point to a *mem* structure. Figure 5.2.1 shows the implementation of the *prepareMemP()* method.

The *prepareMemNodeP()* method initializes pointers to memory nodes, expressed as *tempMemNode[]* as described in section 4.1.6. These *memNode* pointers are used for memory manipulation in the memory manager.

The *prepareScopeNodeP()* method initializes *scopeNode* pointers to scopes, expressed as *tempScope[]* for scope manipulation and *tempNestScope[]* for creation of new compound scope as described in section 4.1.6.

Figure 5.2.1 The *prepareMemP()* Method

```
void prepareMem()

//pre:  memPointer m;
        struct mem n;
//post: a mem pointer is created;

{      m=&n;          }
```

5.2.1.1 Evaluation of *prepareMemP()*, *prepareMemNodeP()* and *prepareScopeNodeP()*

The *prepareMemP()* algorithm is deterministic in $O(1)$. Both *prepareMemNodeP()* and *prepareScopeNodeP()* have algorithms in $O(n)$, where n is the *initialNum* as introduced in section 4.1.6.

5.2.2 Implementation of *initMemNode()*

The *initMemNode()* method uses a for-loop to initialize $n_1 + n_2 + \dots + n_j + n_k$ nodes according to the given distribution information discussed in section 4.1.7. The j th node in n_i nodes is expressed as:

$$tempMemNode[\sum_{p=1}^{i-1} n_p + j]$$

The parameter *thisNodeSize* is m_i in the n_i nodes, where i is from 1 to k . The distributed application memory is pointed to by the *memoryResourcePointer* in *tempMemNode[i]* (i is from 1 to $n_1 + n_2 + \dots + n_k$, as introduced in section 4.1.7). The public parameters *nodeCount[i]* is initialized to n_i , all *heapCount[i]*, *immortalCount[i]* and *scopeCount[i]* is initialized to 0 (i is from 1 to k). The initialized nodes can be used as heap memory, immortal memory or scoped memory.

5.2.2.1 Evaluation of *initMemNode()*

The *initMemNode()* method uses a for-loop to give the parameter *thisNodeSize* a value (as $m_1, m_2, \dots$, or m_k) and to assign the pointer *memoryResourcePointer* a block of

respective memory in each node. So, the *initMemNode()* algorithm, as shown in figure 5.2.2, is in $O(n)$, where n is the sum of $n1, n2, .. nk$.

Figure 5.2.2 The *initMemNode()* Algorithm

```
void initMemNode()

//pre:  There are enough memory nodes prepared for initialization;
//post: The  $n1 + n2 + ... + nk$  nodes are initialized with thisNodeSize as  $m1, m2, ..$ , or  $mk$ , and with memoryResourcePointer to respective memory;
{
    use a for-loop to initialize  $n1 + n2 + ... + nk$  nodes such that thisNodeSize has
    a value as  $m1, m2, ..$ , or  $mk$ , and its memoryResourcePointer to respective
    memory;
}
```

5.2.3 Implementation of *addMemNode()* and *addScopeNode()*

The *addMemNode()* method is used to add a memory node in a memory list, either in a scope or in the *mem* structure. This method takes two pointers, for example, l and p , as its input. l is a pointer to the first node in a list, and p points to a node to be inserted. The insertion of a node is realized via linking the node pointed to by p as the first node in the list.

The *addScopeNode()* method is used to add a scope in a scope list and works in a similar way as the *addMemNode()* method.

5.2.3.1 Evaluation of *addMemNode()* and *addScopeNode()*

Figure 5.2.3 shows the *addMemNode()* algorithm. Both method have the same algorithm and are deterministic $O(1)$.

Figure 5.2.3 The *addMemNode()* Algorithm

```
memoryNodePointer addMemNode ( memoryNodePointer l, memoryNodePointer p)

//pre: l points to the head of a list and p points to the node to be inserted in the list;
//post: return p, whose pointed node links to the first node pointed by l;

{
    link node referenced by p to the first node pointed to by l;
    return p;
}
```

5.2.4 Implementation of *initMemList()*

The *initMemList()* method is used to form k memory lists in *mem* structure, as described in section 4.1.5. The *initMemList()* method invokes the *addMemNode()* method, as described in section 5.2.3, to link memory nodes with the same size together. This method has two integers, *first* and *last*, as its inputs to indicate the linked nodes *tempMemNode[n]* from $n = \textit{first}$ to *last*.

5.2.4.1 Evaluation of *initMemList()*

The *initMemList()* method invokes the *addMemNode()* method, which is $O(1)$ as described in last section, to link nodes together. So, the *initList()* algorithm, shown in figure 5.2.4, is implemented in $O(n)$, where n is the number of nodes $(i2-i1+1)$ to be linked in the list.

Figure 5.2.4 The *initMemList()* Algorithm

```
memoryNodePointer initMemList(int first, int last)

//pre: The nodes created by the initMemNode() method is enough for uses;
//post: return the head pointer pointing to a memory list with (last-first+1) nodes;
{
    use a for-loop to link (last-first+1) nodes together using addMemNode();
    return the head pointer of the list;
}
```

5.2.5 Implementation of *giveMem()*

The *giveMem()* method takes an integer s , representing a required memory size, as its input, and returns a pointer to a node referenced by $memHeadP[m(i+1)]$ such that $mi < s \leq m(i+1)$ where i is from 1 to $k-1$. Based on the discussion in section 4.1.7, the required memory size s should be between 1 and mk bytes. The method causes invocation of the *exceptionCatch()* interface if s is not from 1 to mk , as introduced in section 3.3. The *exceptionCatch()* interface is discussed later in section 5.3.2

5.2.5.1 Evaluation of *giveMem()*

The *giveMem()* algorithm is shown as in figure 5.2.5. It is deterministic in $O(1)$ as the best case if s is not from 1 to mk and in $O(n)$ as the worse case if the *giveMem()* method needs to find the memory node at most mk for n .

Figure 5.2.5 The *giveMem()* Algorithm

```
memoryNodePointer giveMemSize( int s)

// Pre:
//Post: returns a pointer to the node referenced by memHeadP[m(i+1)] such that
       mi < s < m(i+1).
{
    if s is not between 1 and mk,
        invoke exceptionCatch( noThisMem)
    else
        return a pointer such that it satisfies the postcondition;
}
```

5.2.6 Implementation of *checkMemory()*, *checkScope()* and *checkMemScope()*

The *checkMemory()* method is used to check if the required memory is available via the public parameters *nodeCount()* as introduced in section 4.1.6. The *checkMemory()* method returns 1 if the respective *nodeCount(i)* (*i* is 1, 2, ..., or *k*) is more than 0 and returns 0 if the required memory is not available.

The *checkScope()* method takes an integer as its input to indicate the targeted compound scope and is used to check the current scope in the compound scope. The *checkScope()* method returns 0 if the current scope exists, otherwise 1 is returned.

The *checkMemScope()* method is used to check if there are memory nodes in a current scope. There are two ways to implement the *checkMemScope()* method. One is to check the parameter *nNum* in the scope. The other is to check the memory head pointer *scopeMemHeadP* in a scope. If *nNum* is 0 or the *scopeMemHeadP* is null, no memory nodes exists in the scope. The first way is adapted for implementation of the *checkMemScope()* method.

5.2.6.1 Evaluation of *checkMemory()*, *checkScope()* and *checkMemScope()*

The *checkMemory()* method is used to check if the parameter *nodeCount[i]* (*i* is from 1 to *k*) is larger than 0, the *checkScope()* method is used to check if the targeted compound scope exists, and the *checkMemScope()* method is used to check if there is any scoped memory in a scope, i.e., *nNum* is larger than 0, so all the three methods are $O(1)$.

5.2.7 Implementation of *checkHeap()*

The *checkHeap()* method is used to check if the *activateGC()* interface needs to be invoked after heap memory is required, as described in section 4.1.8.

5.2.7.1 Evaluation of *checkHeap()*

The *checkHeap()* algorithm is shown in figure 5.2.6. The best case for this *checkHeap()* method is $O(1)$ without invocation of *activateGC()* interface, and the worst case depends wholly on the *activateGC()* interface.

Figure 5.2.6 The *checkHeap()* Algorithm

```
void checkHeap()

// pre: The heapCount[i] and nodeCount[i] are known, i is from 1 to k;
//       The precondition for the activateGC() interface should be satisfied;
// post: The postcondition for the activateGC() interface should be satisfied;
//       do nothing or invoke activateGC();
{
    if ( heapCount[i] > nodeCount[i]/2 )    activateGC(); // i is from 1 to k
}
```

5.2.8 Implementation of *removeMemNode()* and *removeScopeNode()*

The *removeMemNode()* method is used to remove a memory node from a memory list. The method takes a memory node pointer, as its input, pointing to a node to be removed, sets the pointer to its next node and returns the pointer.

The *removeScopeNode()* method is implemented in a way similar to the *removeMemNode()* method. It is used to remove a scope from a compound scope.

5.2.8.1 Evaluation of *removeMemNode()* and *removeScopeNode()*

These two methods are simply in $O(1)$. Figure 5.2.7 shows the *removeMemNode()* method.

Figure 5.2.7 The *removeMemNode()* Method

```
memoryNodePointer removeMemNode(memoryNodePointer p)

//pre:  none;
//post: return a pointer pointing to the next node;

{
    p=p->link;
    return p;
}
```

5.2.9 Implementation of *getMem()*

The *getMem()* method returns a pointer to a acquired memory node using the *removeMemNode()* method introduced in last section. It first sets a pointer *p* to the first memory node in the list, sets the list head pointer to the second node in the list, unlinks the first node from the second node in the list, and return the pointer *p*.

5.2.9.1 Evaluation of *getMem()*

The *getMem()* method invokes *removeNode()* method, which is $O(1)$ as described in last section, and is invoked by the interfaces *getHeapMem()*, *getImmortalMem()* and *getScopedMem()*. Figure 5.2.8 shows the *getMem()* algorithm, which is deterministic with $O(1)$.

Figure 5.2.8 The *getMem()* Algorithm

```
memoryNodePointer getMem( int size)

// pre: The size must be m1, m2, ... , or m1;
       The respective memory node in its linked list is available;
       The precondition for the removeNode() method should be satisfied;
//post: The postcondition for removeNode() should be satisfied;
       The respective memory list is updated in the memory manager;
       return a node pointer pointing to the required memory;
{
    a memory node pointer p points to first node in respective memory linked list;
    call removeNode() to remove and update the memory linked list;
    return p pointing to the node, whose link is set to null;
}
```

5.2.10 Implementation of *scanOneScope()*

Scanning a scope requires scanning each memory node from the first node to the last node in the scope.

5.2.10.1 Evaluation of *scanOneScope()*

The *scanOneScope()* algorithm is shown in figure 5.2.9. Since this method uses a for-loop to scan memory nodes from the first node to the last node, it is an $O(n)$ algorithm where n is equal to parameter *nNum*.

Figure 5.2.9 The *scanOneScope()* Algorithm

```
void scanOneScope()
//pre: The memory node number nNum is known in scanned scope;
//post: All memory nodes in scanned scope are scanned;
{
    use a for-loop to scan every memory node:
        if the node is not the last node, continue for-loop;
}
```

5.3 Implementation and Evaluation of Interfaces

The implementation of the nine interfaces is based on the three data structures and their supporting methods described in the previous sections.

The *initManager()* interface is for initialization of the memory manager. The memory manager is available for use after its initialization. The *exceptionCatch()* interface is

implemented to notify a respective thread that an exception is thrown during the communication between the thread and the memory manager. The *activateGC()* interface is implemented to notify the RTJVM outside of the memory manager to start garbage collection.

The interfaces, *getImmortalMem()*, *getHeapMem()* and *getScopedMem()*, provide functions for memory manipulation for a RTJVM thread to get the required memory. The interfaces *getScope()*, *scanNestedScope()* and *endScope()* provide scope manipulation. The interfaces *getScope()*, *getScopedMem()*, and *endScope()* allow the scoped memory to be recycled for reuse. So, scoped memory is different from immortal memory and heap memory. The *scanNestedScope()* interface can scan memory information in any nested scope in deterministic behaviors, which makes the memory manager useful for supporting debug systems in RTJVM.

5.3.1 The *initManager()* Interface

The *initManager()* interface is used to initialize the memory manager. Based on the manager memory resource, as assumed in section 3.1.2, it initializes the memory manager to contain k lists of memory nodes as described in section 4.1.7.

5.3.1.1 Evaluation of the *initManager()* Interface

The *initManager()* interface is not required to have deterministic behaviors, since both scope and memory manipulations occur after the initialization of the memory manager. However, a deterministic *initManager()* interface allows the memory manager to be initialized in a predictive period.

The *initManager()* interface invokes methods *prepareMemP()*, *prepareMemNodeP()*, *prepareScopeP()*, *initMemNode()* and *initMemList()* to initialize the memory manager. These invoked methods are $O(1)$, $O(n)$, $O(n)$, $O(n)$ and $O(n)$ respectively as discussed in previous sections. Therefore, the *initManager()* algorithm is deterministic in $O(n)$. Figure 5.3.1 shows the *initManager()* algorithm.

Figure 5.3.1 The *initManager()* Algorithm

```
void initManager()

//pre: The preconditions for the invoked methods must be satisfied;
       The nodeCount[i] are known for i from 1 to k;
//post: The postconditions for the invoked methods must be satisfied;
        2* initialNum scope pointers available for uses;
        initialNum - (n1+n2 + ... + nk) memory node pointers available for uses;
        k memory linked lists are available for uses;
{
    invoke prepareMemP(),
           prepareMemNodeP()
           prepareScopeP();
           initMemNode();

    m-> memHeadP[1] = initMemList( 1, m1);
    m-> memHeadP[2] = initMemList(m1+1, m1+m2 );
    ...
    m-> memHeadP[l] = initMemList(m1+..mj+1, m1+m2+..mk);
}
```

5.3.2 Implementation of the *exceptionCatch()* Interface

This interface is implemented to catch exceptions during communication between a RTJVM thread and the memory manager. When an exception is caught, it is invoked with two input integers and returns an integer. The different input integers correspond to different exceptions. The relationships between the integers and exceptions are shown in Figure 5.3.2. This interface is merely for arbitrary tests, since this interface depends on the specification of a RTJVM. The return value is 1, 2, or 3+j, where 1 means that the required memory size is illegal, 2 means that memory nodes with the required size is not available, and 3+j means that the jth compound scope does not exist.

Figure 5.3.2 Relationships among Integers, Exceptions and Expressions

int i	int j	exception	expression	return
1	0	The memory size is not from <i>l</i> and <i>mk</i>	<i>noThisMem</i>	1
2	<i>thisNodeSize</i>	No memory with required memory size	<i>noMoreNode</i>	2
3	nested scope number	The indicated nested scope does not exist	<i>illNestScope</i>	3+j

5.3.2.1 Evaluation of the *exceptionCatch()* Interface

This interface is simply in $O(1)$ since it returns the integer based on its inputs. Figure 5.3.3 shows the *exceptionCatch()* algorithm.

Figure 5.3.3 The *exceptionCatch()* Algorithm

```
int exceptionCatch (int i int j)

// Pre:  different input integers stand for different exceptions;
//Post:  return i

{
    if i=1 or 2 , return i;
    else return 3+j;
}
```

5.3.3 Implementation of the *getImmortalMem()* Interface

The *getImmortalMem()* interface is for allocating immortal memory. This interface first invokes the *giveMem()* method to get a pointer to the required memory node as described in section 5.2.5. If the required memory size is not from 1 to *mk*, the *exceptionCatch(1, 0)* interface is invoked to throw an exception. Otherwise, the *getImmortalMem()* interface invokes the *checkMemory()* method to determine whether the required memory is available.

If the required memory is available, the interface invokes the *getMem()* method to acquire the memory. If the required memory is not available, the *exceptionCatch(2, thisNodeSize)* interface (discussed in section 5.3.2) is invoked. Figure 5.3.4 shows the calling structure of the *getImmortalMem()* interface.

Figure 5.3.4 The Calling Structure of the *getImmortalMem()* Interface

5.3.3.1 Evaluation of the *getImmortalMem()* Interface

The *getImmortalMem()* interface directly invokes three methods, which are: the *giveMem()* method, the *checkMem()* and *getMem()* method. These invoked methods are either $O(n)$ or $O(1)$ as described in previous sections. The factor of n is mk . So, the *getImmortalMem()* interface is also deterministic in $O(n)$.

5.3.4 Implementation of the *getHeapMem()* Interface

The *getHeapMem()* interface is used to allocate heap memory. Figure 5.3.5 shows the calling structure of the *getHeapMem()* interface, which first invokes the *checkHeap()* method to check whether garbage collection is needed. If necessary, the *activateGC()* interface is invoked to process garbage collection as described in the previous section. Otherwise, the *getHeapMem()* interface works in a way similar to the *getImmortalMem()* interface.

Figure 5.3.5 The Calling Structure of the *getHeapMem()* Interface

5.3.4.1 Evaluation of the *getHeapMem()* Interface

Heap memory manipulation is not deterministic in traditional Java. Even so, the evaluation is still done for this interface to conclude its best case and its worse case. The *getHeapMem()* interface first calls the *checkHeap()* method, and then works in the same way as the *getImmortalMem()* method. So, for the *getHeapMem()* interface the best case is $O(n)$ and the worse case depends on the *activateGC()* interface.

5.3.5 Implementation of the *getScope()* Interface

The *getScope()* interface is implemented for adding a new scope as the current scope in a compound scope, or creating a new compound scope. It takes an integer to indicate the targeted compound scope, as its input.

The interface first invokes the *checkScope()* method to check the compound scope. If the compound scope does not exist, it creates a new compound scope. Otherwise, it adds a new scope as the current scope into the compound scope. The new current scope is always empty when it is created.

5.3.5.1 Evaluation of the *getScope()* Interface

This interface must be deterministic to make sure that the scope manipulation and scoped memory manipulation have deterministic behaviors.

Figure 5.3.6 shows the *getScope()* algorithm. Since the *checkScope()* method and the *addScopeNode()* method are all in $O(1)$ as described in sections 5.2.6 and 5.2.3, the *getScope()* interface is also deterministic in $O(1)$.

Figure 5.3.6 The *getScope()* Algorithm

```
void getScope()

// pre: The initialized scopes are available for uses;
//       The preconditions for invoked methods must be satisfied;
// post: The postconditions for invoked methods must be satisfied;
//       A new scope is created as a new compound scope,
//       or a new scope is added as the current scope into the compound scope;
{
    if ( checkScope() )
        set a new scope as a new compound scope;
    else
        call addScopeNode() to add a new scope in the compound scope;
}
```

5.3.6 Implementation of the *getScopeMem()* Interface

The *getScopeMem()* interface allocates scoped memory to the current scope. It takes an integer as its input (*s*) to indicate the targeted compound scope, and returns a memory node pointer related to the acquired memory. Figure 5.3.7 shows the calling structure of the *getScopeMem()* interface.

The *getScopeMem()* interface first invokes the *checkScope()* method to check for a compound scope. If a compound scope does not exist, then *getScopeMem()* interface invokes the *exceptionCatch(3, s)* interface, as discussed in section 5.3.2. Otherwise, the *getScopeMem()* interface works in a way similar to the *getImmortalMem()* interface, discussed in section 5.2.3, except that the *getScopeMem()* interface adds the acquired memory into the current scope in the compound scope.

Figure 5.3.7 The Calling Structure of the *getScopeMem()* Interface

5.3.6.1 Evaluation of the *getScopeMem()* Interface

The *getScopeMem()* interface first invokes the $O(1)$ *checkScope()* method to check the compound scope. If the compound scope does not exist, then the interface is $O(1)$. Otherwise, the interface works in a way similar to the $O(n)$ *getImmortalMem()* interface, followed by invoking the $O(1)$ *addMemNode()* method. Therefore, the *getScopeMem()* is deterministic in $O(n)$.

5.3.7 Implementation of the *scanNestedScope()* Interface

Scanning a compound scope includes scanning each of its scopes, and each memory node in each scope. Scanning proceeds from the innermost scope to the outermost scope. The *ScanNestedScope()* interface is implemented for scanning a compound scope. Figure 5.3.8 shows the calling structure of the interface. The interface takes an integer (s) as its input to indicate the targeted compound scope.

The *scanNestedScope()* interface first invokes *checkScope()* to check for a compound scope. If the compound scope does not exist, the interface invokes the *exceptionCatch(3,s)*. Otherwise, the interface begins to scan each scope in the compound scope. While scanning a scope, the *scanNestedScope()* interface scans each memory node in the scope using the *scanOneScope()* method described in section 5.2.10.

Figure 5.3.8 The Calling Structure of the *scanNestedScope()* Interface

5.3.7.1 Evaluation of the *scanNestedScope()* Interface

This interface first invokes the $O(1)$ *checkScope()* method. If the compound scope exists, the $O(n)$ *scanOneScope()* method is invoked. Since scanning all memory nodes in each scope is linear, the *scanNestedScope()* interface is $O(n)$, where n is the sum of the number of memory nodes in each scope of the whole compound scope.

5.3.8 Implementation of the *endScope()* Interface

The *endScope()* interface takes an integer (*s*) as its input to indicate the targeted compound scope. Figure 5.3.9 shows the calling structure of the *endScope()* interface.

The *endScope()* interface first invokes the *checkScope()* method to check the compound scope. If it does not exist, the interface invokes *exceptionCatch(3, s)*. Otherwise, the *endScope()* interface invokes the *checkMemScope()* method to check memory in the current scope. If the current scope does not have any memory nodes, the interface invokes the *removeScopeNode()* method to remove the current scope from the compound scope.

If the current scope has memory nodes, the interface invokes the *removeMemNode()* method, followed by the *addMemNode()* method, to recover and recycle the memory nodes. After recycling the memory nodes, the *removeScopeNode()* method is invoked to remove the current scope from the compound scope.

Figure 5.3.9 The *endScope()* Interface

5.3.8.1 Evaluation of the *endScope()* Interface

The *endScope()* interface is $O(1)$ if the *exceptionCatch(illNestedScope)* interface is invoked in the *checkScope()* method or if there is no memory in the current scope. Otherwise the *endScope()* interface is $O(n)$ since a for-loop is used to recover each memory node in the current scope prior to removing the current scope. The factor n is the number of memory nodes in the scope. Therefore, The *getScopeMem()* interface is deterministic in $O(n)$.

5.3.9 Implementation of the *activateGC()* Interface

The *activateGC()* interface is implemented to start the garbage collector, which is not in the memory manager. This interface is invoked when the *checkHeap()* method detects that a free memory linked list is less than two times of the number of used heap memory nodes with the same *thisNodeSize*. The interface returns integer *size*.

5.3.9.1 Evaluation of the *activateGC()* Interface

Since the interface just returns an integer when it is invoked, the interface is deterministic in $O(1)$.

Chapter 6

Tests for the Memory Manager

This chapter introduces a testing system used to test the real time memory manager. The purpose of this testing system is to test if the memory manager satisfies the RTMMs described in chapter 3. In order to achieve this purpose, the testing system simulates the communication between Java threads and the memory manager via interactions between users and the memory manager.

Testing the memory manager requires testing all interfaces, which are the *initManager()* interface, the *getHeapMem()* interface, the *activateGC()* interface, the *getImmortalMem()* interface, the *getScope()* interface, the *getScopeMem()* interface, the *scanNestedScope()* interface, the *endScope()* interface, and the *exceptionCatch()* interface.

The testing system includes a *main()* method, three input methods and seven output methods. These methods are invoked from *main()*, an interface or internal methods. The testing system provides a status screen to show the interaction between a user and the memory manager, the state of the memory manager and the commands that can be entered by the user. The way that the information is entered to the memory manager by a user is similar to the way that the information is passed from the Java threads to the memory manager, and the way that the information is returned from the memory manager

to the status screen is similar to the way that the information is passed from the memory manager to the Java threads.

Figure 6.1 shows the status screen for the memory manager after entering 183, 8, 180, 16, 177, 32, which stand for a block of memory distributed in 183 8-byte nodes, 180 16-byte nodes and 177 32-byte nodes. These memory nodes can be used as immortal memory, scoped memory or heap memory. The distribution information are entered as:

183' 8' 180' 16' 177' 32'

where ' stands for clicking "enter" in the keyboard.

Figure 6.1 An Initial State of the Memory Manager

```

*****
Distributed memory information: 183, 8, 180, 16, 177, 32

Memory:           8_byte  16_byte  32_byte
Initial Node Num   183      180      177
Available Node Num 183      180      177
Used Heap Node Num  0        0        0

Enter commands: h, i, gn, mn, sn, en, d (n=1,2,3,4,5), or c in details!

*****

```

6.1 The Input Methods

In the testing system, there are two methods implemented to enter the necessary data to simulate the uses of the memory manager in the RTJVM, and there is one method implemented to automatically add information into the application memory. The added information is characters in this testing system.

6.1.1 The *simulate_RTJVM_DistributionInfo()* Method

Before the initialization of the memory manager, the user enters the memory distribution information, invoking the testing program to call a standard C method *malloc()* to obtain memory space. This simulates the situation that memory distributed information and memory space are given to the memory manager during the initialization of the RTJVM and before the initialization of the memory manager.

The *simulate_RTJVM_DistributionInfo()* method is designed for this purpose. It first invokes a *printf()* method to prompt the user to enter the memory size and then invokes a standard C *scanf()* method to read $n1, m1, n2, m2, \dots, nk, mk$ for the memory manager as described in section 4.1.7.

To illustrate the testing system, three sizes of memory nodes are used, i.e., the parameter *thisNodeSize* is 8, 16 or 32 bytes, as described in section 4.3. So the user enters $n1, 8, n2, 16, n3$ and 32. The testing program then gets a block of memory with $8*n1 + 16*n2 + 32*n3 + 4*4 + 4*6*countNum + 3*4*initialNum$ bytes before its initialization, where $8*n1 + 16*n2 + 32*n3$ bytes of memory are for the application memory resource, and $4*4 + 4*6*countNum + 3*4*initialNum$ is discussed in section 4.1.6.

6.1.2 The *addInfo()* Method

An *addInfo()* method is implemented to add characters into application memory. When the *initNode()* method is invoked to initialize the memory nodes, it stores characters into distributed memory. The contents are used to test the *scanNestedScope()* interface, which scans each memory node for memory information related to the address, size and content.

The contents could be stored to memory at any time, but it is reasonable to add the contents during the initialization of the each node.

6.1.3 The *addMemSize()* Method

The *addMemSize()* method allows a user to enter a required memory size to the memory manager. Due to the sizes of the memory blocks used in testing (8, 16 and 32 bytes), the entered memory size must be between 1 and 32. This is to simulate a situation where a thread requests a certain size of memory from the memory manager, which corresponds to the requirement of offering the required memory.

6.2 The Output Methods

There are seven methods used to output information for the tests of the memory manager. All of them are discussed in the following sections.

6.2.1 The *showManager()* Method

This method provides a small window showing the state of the memory manager. This method is invoked after each interface invocation. The displayed state includes: initial number of distributed memory nodes, the number of distributed memory nodes available, the number of used heap memory nodes, and commands for the user to enter. The *showManager()* method is implemented based on the public parameters of various nodes. An initial state of the memory manager is shown in figure 6.1. The commands are introduced in section 6.3.

6.2.2 Methods of *showMoreCommand()* and *illInput()*

As introduced in the status screen, entering *c* causes the invocation of the *showMoreCommand()* method to show the commands in more detail.

The *illInput()* method responds to an illegal command by showing a message stating that an illegal command has been entered.

6.2.3 Methods of *noThisMem()*, *noMoreNode()*, and *illNestedScope()*

Three methods are invoked by the *exceptionCatch(i)* interface to represent the cases when *i* is 1, 2 and 3 respectively.

If the entered memory size is not from 1 to 32, the *exceptionCatch(1,0)* method is invoked, resulting in calling the *noThisMem()* method to show a message stating that a memory size outside of the permitted range has been entered.

When memory is required and there are no memory nodes available, the *exceptionCatch(2,thisNodeSize)* method is invoked, resulting in a call to the *noMoreNode()* method. The method displays a message indicating that there are no more nodes of the required size available. For example, if an 8-byte immortal memory is required, but no 8-byte memory node is available, then the *noMoreNode()* method is invoked.

When an attempt is made to access a nested scope that does not exist, the *exceptionCatch(3, nested scope number)* interface is invoked, resulting in a call to the *illNestedScope()* method to show that the nested scope does not exist.

6.2.4 The *showScanInfo()* Method

This method is implemented in the *scanOneScope()* method and is used to show the state of a compound scope. The state of a compound scope includes the scopes in the compound scope, all memory nodes in a scope, and the address, size and content of each memory node.

Figure 6.2.2 shows a state of a compound scope. The S1 stands for S1 compound scope, a $\rightarrow$ for a scope, and the tuple (integer1, char, interger2) for a memory node with memory address integer1, memory content char, and memory size integer2. In figure 6.2.2, there are three scopes in the compound scope S1. The current scope has one memory node, the outermost scope has three memory nodes, and the other scope has none. In the outermost scope, the node (8325548 b 8) is added first, followed by node (8325972 j 32) and then node (8325556 c 8).

Figure 6.2.2 A Nested Scope State

S1 $\rightarrow$ (8326004 k 32) $\rightarrow \rightarrow$ (8325556 c 8) (8325972 j 32) (8325548 b 8)

6.3 The *main()* Method

The calling structure of the *main()* method, shown in figure 6.3.1, is used to start tests for the memory manager. The *main()* works in three steps. The first step is to provide the memory manager a block of memory based on the distributed information input by the user via the *simulate_RTJVM_DistributionInfo()* method. The second step is to initialize the memory manager via the *initManager()* interface and the *showManager()* method. The third step is to loop testing the seven interfaces based on commands entered by the

user, until the done command *d* is entered. The *showManager()* method is always invoked after each interface is tested.

Figure 6.3.1 The Calling Structure of the *main()* Method

When *h* is entered, the *getHeapMem()* interface is invoked and tested. Command *i* is for testing *getImmortalMem()* interface, *gn* for the *getScope()* interface, *mn* for *getScopeMem()* interface, *sn* for the *scanNestedScope()* interface, *en* for the *endScope()* interface, and *d* is for exiting the *main()* via invoking the *done()* method. Compared to figure 6.1 showing the initial state, figure 6.3.2 shows states after the following information and commands are entered:

183' 8' 180' 16' 177' 32' h' 12' i' 1' i' 13' h' 9' g1' g1' m1'
 1' m1' 31' m1' 8' g4' g2' m4' 12' m4' 19' g1' g4' m4'
 21' i' 2' m1' 7' g1' m1' 6' m1' 14' e1' s1' s2' s3' s4' s5'

where 183' 8' 180' 16' 177' 32' are discussed before, *h' 12'* stands for getting a heap memory block with a size of 12 bytes (the manager returns a 16-byte memory node), *i' 1'* stands for getting a immortal memory block with a size of 1 byte (the manager returns a 8-byte memory node), *g1'* stands for creating a compound scope with id 1 if it does not exist or allocating a new current scope in the compound scope with id 1 if it exists. *m1' 31'* stands for associating a memory node (a 32-byte memory node for the 31-byte block required) with the current scope in the compound scope with id 1. *e1'* stands for ending the current scope in the compound scope with id 1. *s1'* stands for scanning the compound scope with id 1.

In figure 6.3.2, two 16-byte nodes is used as heap memory, a 16-byte node and two 8-byte nodes are used as immortal memory, the other seven nodes are used as scoped memory.

Figure 6.3.2 The States of the Memory Manager after some Tests

```
*****
Distributed memory information: 183, 8, 180, 16, 177, 32

Memory:          8_byte  16_byte  32_byte
Initial Node Num    183      180      177
Available Node Num  178      176      174
Used Heap Node Num   0        2        0

Enter commands: h, i, gn, mn, sn, en, d (n=1,2,3,4,5), or c in details!
*****
S1--> (7295668 e 8) --> (7295652 c 8) (7299980 m 32) (7295644 b 8) -->
*****
S2-->
*****
compound scope 3 is NULL!
*****
S4--> (7300044 o 32) --> (7300012 n 32) (7297148 y 16)
*****
compound scopes 5 is NULL!
*****
```

6.4 Tests for the Memory Manager

As described before, testing the memory manager requires tests on the nine interfaces. These tests are outlined in the following sections. The memory manager passed all tests with the testing system.

6.4.1 Tests for the *initManager()* Interface

The tests for the *initManager()* interface are done with different distributed information. As mentioned before, the memory size in the testing system is fixed at 8, 16, or 32-byte. So, tests for the *initManager()* interface actually focus on with the different number of memory nodes with different memory size. The number such as -9, 0, 1, 10, 100, or 200 is selected for tests.

6.4.2 Tests for Interfaces *getImmortalMem()*, *getHeapMem()*, *getScopedMem()*, *exceptionCatch()* and *activateGC()*

General tests for these interfaces focus on two aspects: the first one is to test the memory size boundary and the second one is to test the response from the memory manager if the required memory is not available. Memory sizes such as -9, 0, 1, 7, 8, 9, 15, 16, 17, 31, 32, 33, 100 bytes are used for the boundary test. The *exceptionCatch()* interface is called for these tests.

For the *getHeapMem()* interface, additional tests are required for testing the *activateGC()* interface when the free memory space is less than twice the used heap memory of the same memory size. The *getScopedMem()* interface also requires additional tests for testing if the required compound scope exists, for testing adding memory nodes into

different scopes in a compound scope, and for testing adding nodes into different compound scopes. The *exceptionCatch()* interface is also tested if the required compound scope does not exist.

6.4.3 Tests for Interfaces *getScope()*, *scanNestedScope()*, and *endScope()*

The *getScope()* interface is first tested by adding a new scope into a compound scope. If the compound scope does not exist, the compound scope is created with this new scope. Otherwise, the new scope is added into the nested scope. The *getScope()* interface is also tested for adding many scopes into a compound scope or different compound scopes.

The *scanNestedScope()* interface is tested for scanning a compound scope each time after the command *sn* is entered. If the compound scope does not exist, the *exceptionCatch()* interface is tested and the status screen shows a message stating that the scanned compound scope does not exist. Otherwise, the test continues for scanning a compound scope with more than one scope containing different memory nodes. Tests also focus on the memory information in each memory node, the order of the memory nodes added into a scope, and the order of scopes added into a compound scope.

The *endScope()* interface is tested for ending a scope in a compound scope. If the compound scope does not exist, the *exceptionCatch()* interface is tested and the message stating that the compound scope does not exist is shown. Otherwise, the interface first recovers any memory nodes allocated to the scope, and then removes the scope. The status screen reflects the number of memory nodes recovered. The *scanNestedScope()* interface can be used to show that the scope is removed.

Chapter 7

Conclusions and Recommendations

This thesis identified three important differences, which are between traditional Java and the RTJ, between the JVM and the RTJVM, and between the traditional JVM memory manager and the RTJVM memory manager. It also identified and analyzed the related memory classes described in the RTSJ to determine how the functions of these classes might be realized by a RTJVM. The thesis proposed a RTMMS, and designed and implemented a RTMM based on the proposed RTMMS.

The provided RTMM is implemented in C, is easy to understand, is ready to be embedded into a RTJVM for RTJ applications, and satisfies requirements specified in the JLS, the LVM and the RTSJ.

The memory manager contains nine interfaces, one is to initialize the memory manager, one is to notify the thread for garbage collection, one is to notify the calling thread about the thrown exception, the other six interfaces are to provide different memory areas. All of the interfaces are implemented in deterministic behaviours with worst case as $O(n)$ performance.

The nine interfaces are supported via sixteen internal methods. The implementation of these internal methods is based on three basic data structures. All internal methods are

evaluated during their implementation using big-O analysis and are found to be deterministic in $O(n)$ as a worst case.

The testing system is designed and implemented, and found to be successful to test the heap memory manipulation, immortal memory manipulation, scoped memory manipulation and scope manipulation. The testing system provides input methods to simulate requirements from a thread to the memory manager and output methods to simulate a reply from the memory manager to a thread. Using the testing system, the memory manager was fully tested to satisfy the RTMMS proposed in this thesis.

The following future work is recommended:

- Extend some scope manipulation functionality such that it can satisfy the *getPortal()* method and *setPortal()* method specified in the RTSJ *ScopeMemory* class.
- Extend some functionality of the memory manager such that it can satisfy the other RTSJ memory classes, which include *VTMemory*, *LTMemory*, *PhysicalMemoryFactory*, *ImmortalPhysicalMemory*, *ScopedPhysicalMemory*, *RawMemoryAccess*, *RawMemoryFloatAccess*, and *MemoryParameters*.
- Design, implement and test software components for garbage collection in the RTJVM, so the memory manager can start a garbage collector via its *activateGC()* interface.
- A Real Time Java Virtual Machine Specification (RTJVMS) needs to be proposed based on the JLS, the JVMMS and the RTSJ. With the RTJVMS, software components can be developed to fulfill the whole RTJVM, and the RTSJ's APIs can be implemented for real time Java applications.

References

1. C. S. Horstmann and G. Cornell, *Core Java 1.2*, volume 1-Fundamentals, 1999, The Sun Microsystems Press.
2. K. Arnold and J. Gosling, *The Java Programming Language*, The Java Series, Addison-Wesley, 1996.
3. Sun Microsystems, *Java 2 Platform API Speciation*, Version 1.3, 2001, <http://Java.Sun.com/j2se/1.3/docs/>.
4. G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin and M. Turnbull, *The Real-Time Speciation for Java*, 2000, Addison-Wesley.
5. C. M. Krishna and K. G. Shin, *Real-Time Systems*, WCB/McGraw-Hill, 1997,
6. R. G. Hayes, *Real-Time Java*, <http://home.earthlink.net/~rghayes/gradpaper.doc>.
7. M. Stolz, *Implementing the Real-Time Specification for Java*, Embedded Systems Conference Europe Stuttgart, 10 October 2001.
8. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd Edition, 1999, <http://Java.sun.com>.
9. D. Hardin, *Bring Java's Benefits to Real-Time Developers*, JavaOne, Sun's 2002 Worldwide Java Developer Conference.

10. K. Nilsen, *Issues in the Design and Implementation of Real-Time Java*,
<http://www.sys-con.com/Java/iss1/real/htm>.
11. *The Heap Manager*, Transvirtual Technologies, Inc., <http://www.kaffe.org>.
12. J. Gosling, B. Joy and G. Steele, *The Java Language Specification*,
<http://Java.sun.com/docs/books/jls/html/index.html>.
13. Bill Venners, *Inside the Java Virtual Machine*, McGraw-Hill, 1998,
14. R. Johnsonbaugh and M. Kalin, *Applications Programming in C++*, Prentice Hall,
1999.
15. D. Flanagan, *Java in a Nutshell*, 1999, O'Reilly.
16. E. D. Jensen, *A Proposed Initial Approach to Distributed Real-Time Java*, The
MITRE Corporation, <http://www.real-time.org>.
17. K. Nilsen, *Adding Real-Time Capabilities to Java*, Communications of the ACM,
June 1998, Vol. 41, No. 6.
18. S. Ortiz Jr., *The Battle Over Real-time Java*, Industry Trends, June 1999.
19. M. Kolehmainen, *Synchronization in the Real-Time Specification for Java*, 2001,
[http://www.google.ca/search?hl=en&ie=ISO-8859-1&q=Synchronization+in+The
+Real-time+specification+for+Java%2C+&meta=](http://www.google.ca/search?hl=en&ie=ISO-8859-1&q=Synchronization+in+The+Real-time+specification+for+Java%2C+&meta=).
20. R. Yerraballi, *Real-Time Java: A Specification*, 2000, CSE 5326: Real-
Time Systems Design, <http://data.uta.edu/~ramesh/cse5326/rtJava.html>.
21. *Memory Manager*, <http://www.optimalcode.com/memmgr.htm>.
22. *The Memory Manager*,
<http://www.q-software-solutions.com/lccwin32/manual/MemoryManager.html>.
23. N. C. Myers, *Memory Management in C++*, <http://www.cantrip.org/wave12.html>.

24. *Requirements for the Real-time Extensions for the Java Platform*, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, MD 20899-0001, 1999.
25. *KaffeVM*, Transvirtual Technologies, Inc., <http://www.kaffe.org>.
26. D. A. Bailey, *Data Structures in Java for the Principled Programmer*, WCB/McGraw-Hill, 1998.
27. *Introductory C Programming Class Notes*,
<http://www.eskimo.com/~scs/cclass/class.html>.
28. S. G. Kochan, *Programming in C*, Hayden Books, 1990.
29. W. Li, *Introduction to Information and System Science*, School of Computer Science, Carleton University, 2001.
30. M. A. Weiss, *Data Structures And Algorithm Analysis In C*, The Benjamin/Cummings Publishing Company, Inc., 1993.
31. Peter Dibble, *Introduction to the Real Time Java Platform: Brief History of Real Time Java*, InformIT, 2002.
32. The Real time Java Working Group, *Real Time Core Extensions*,
<http://www.J-consortium.org/rtjwg/index.shtml>
33. TimeSys Delivers First RTSJ-Compliant Java (TM) Virtual Machine and Embedded Java Development Kit, <http://www.timesys.com>
34. *The Memory Management Reference*, <http://www.memorymanagement.org>
35. RTOS State of the Art Analysis: Deliverable D1.1 - RTOS Analysis
<http://www.mnis.fr/opensource/ocera/rtos/book1.html>

36. M. Banatre, G. Cabillic and Frederic Parain, *Java Embedded Real Time Systems: an Overview of Existing Solutions*, IEEE, 2000
37. Proposed Java Real Time API, v0.2, *Java Software Division of Sun Microsystems*, 1998, <http://www.sdct.itl.nist.gov/carnahan/real-time/sun/api>
38. G. Back, P. Tullmann, L. Stoller, W. Hsieh, and J. Lepreau, *Java Operating Systems*, 1998, <http://www.cs.utah.edu/projects/flux>.

Appendix B

The Code for the RTSJ Memory Manager

```

//*****//
//|
//
//
//
//*****//

////////////////////////////////// preprocessor ///////////////////////////////////
#include <iostream.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

////////////////////////////////// memory manager variables ///////////////////////////////////
// pointer to memory
void * p;
// parameters for different kinds of memory nodes
long nodeCount[100], scopeCount[100], heapCount[100], immortalCount[100];
// distribution info
long DI_at_Size[100], DI_at_Num[100];
// numbers of used member nodes & used scope nodes
int nodeNum=0, scopeNum=0;
// parameter for methods prepareMemNodeP() and prepareScopeNodeP()
int initialNum=10000, countNum=100;

////////////////////////////////// methods and interfaces ///////////////////////////////////
void printUU()
{
    printf("*****");
}

// print the some space
void printUD(int i)
{
    if(i==1) { printf("\n"); printUU(); printf("\n"); printUU(); printf("\n"); }
    else { printUU(); printf("\n"); printUU(); printf("\n\n"); }
}

// a message shows how the memory manager works in a RTJVM
void preface_for_RTSJ_MemoryManager()
{
    printUD(1);
    printf(" This is to simulate a RTSJ memory manager in a RTJVM! \n\n");
    printf(" There three steps for this RTSJ memory manager to work: \n\n");
    printf(" Step 1: a RTJVM offers distribution info and required memory; \n");
    printf(" Step 2: the RTSJ memory manager begins to initialize; \n");
    printf(" Step 3: the RTSJ memory manager are ready for tests.\n\n");
    printUD(0);
}

```

```

}

// a message shows the commands to be entered as h, i, gn, sn, en, d, or c.
void showMoreCommand()
{
    printUD(1);
    printf(" h  for geting a heap memory; \n");
    printf(" i  for geting an immortal memory; \n");
    printf("gn forcreating compound scope n or adding a scope to compound scope n;\n");
    printf("mn for associating scope memory with current scope at compound scope n; n;\n");
    printf(" sn for scanning showing the details in the nested scope n;  \n");
    printf(" en for ending the current scope at compound scope n.\n");
    printf(" d  for exit!  \n");
    printf(" c  in details  \n");
    printUD(0);
}

// showing the state of the memory manager
// the initial numbers of memory nodes with 8, 16, or 32-byte
// the available numbers of memory nodes with 8, 16, or 32-byte
// the numbers of heap memory nodes with 8, 16, or 32-byte
void showManager()
{
    printUD(1);
    printf("  Distributed Memory Information: %d %d %d %d %d %d\n\n",
        DI_at_Num[8],  DI_at_Size[8],
        DI_at_Num[16], DI_at_Size[16],
        DI_at_Num[32], DI_at_Size[32] );
    printf(" memory:           8_byte  16_byte  32_byte.\n");
    printf(" Initial Node Num      %d      %d      %d \n",
        DI_at_Num[8], DI_at_Num[16], DI_at_Num[32]);
    printf(" Available Node Num     %d      %d      %d \n",
        nodeCount[8],  nodeCount[16],  nodeCount[32]);
    printf(" Used Heap Node Num     %d      %d      %d \n",
        heapCount[8],  heapCount[16],  heapCount[32]);
    printf(" enter h, i, gn, mn, sn, en, or c for details!\n");
    printf("\n");  printUD(0);
}

// a message shows illegal memory size if 0>thisNodeSize or thisNodeSize >32
void noThisMem()
{
    printUD(1);  printf("illegal memory size,try again (1...32)!\n");  printUD(0); }

// a message shows that the memory node is not available
void noMoreNode(int s)

```

```

{
    if(s == 8)
    { printUD(1);   printf("no 8-byte memory node.\n");   printUD(0);   }
    else if (s==16)
    { printUD(1);   printf("no 16-byte memory node.\n");   printUD(0);   }
    else
    { printUD(1);   printf("no 32-byte memory node.\n");   printUD(0);   }
}

// a message shows the illegal command entered
void illInput()
{   printUD(1);   printf(" illegal commands, try again! \n");   printUD(0);   }

// a message shows the illegal compound scope i
void illNestedScope(int i)
{   printUD(1);   printf(" illegal nested scope %d, try again!\n", i); printUD(0);   }

//// a message shows the exit of testing the memory manager
void done()
{   printUD(1);   printf("Done! \n");   printUD(0);   exit(0);   }

// the exceptionCatch (int i, int j) interface with three exceptions
integer exceptionCatch (int i, int j) // an interface
{
    if (i==1)           {   noThisMem();           }
    else if (i==2)       {   noMoreNode(j);         }
    else if (i==3)       {   illNestedScope(j);     }
}

// enter the distributed information parameters: the number of memory node with
// memory size 8, 16, and 32-byte
// check if the parameters are legal
void simulate_RTJVM_DistributionInfo()
{
    long s,t=1;
    DI_at_Size[8]=8,   DI_at_Size[16]=16,   DI_at_Size[32]=32;
    printUD(1);   printf("Step1:\n");   printf("Please provide distribution info.\n");
    while(t)
    {
        printf("m1=8, enter n1="); scanf("%d", &s);   DI_at_Num[8]=s;
        printf("m2=16, enter n2="); scanf("%d", &s);   DI_at_Num[16]=s;
        printf("m3=32, enter n3="); scanf("%d", &s);   DI_at_Num[32]=s;
        if (((DI_at_Num[8]<0)|| (DI_at_Num[16]<0) ||(DI_at_Num[32]<0))<0)
        {

```

```

        printUD(1);   printf("n1, n2 or n3 is less then 0, try again!\n");
        printUD(0);   t=1;
    }
    //for long 2,147,483,647
    else if (((DI_at_Size[8])*(DI_at_Num[8]) +
            (DI_at_Size[16])*(DI_at_Num[16]) +
            (DI_at_Size[32])*(DI_at_Num[32]))>2147483647 )
    {
        printUD(1); printf(" total memory > 319848 bytes, try again!\n");
        printUD(0);   t=1;
    }
    else    t=0;
}
// 4 is for p, 3*4 is for initialNum, scopeNum, nodeNum,
// 6*4 100 is for 6 arrays of integers, 3*4*initialNum is for 3 arrays of pointers
p =malloc((DI_at_Size[8])*(DI_at_Num[8]) + DI_at_Size[16])*(DI_at_Num[16])
+
(DI_at_Size[32])*(DI_at_Num[32])+4*4+6*4*countNum+3*4*initialNum );
}

```

// the chars from a to z are produced for the distributed memory

char addInfo(char c)

```

{
    if      (c==' ') return 'a';  else if (c=='a') return 'b';  else if (c=='b') return 'c';
    else if (c=='c') return 'd';  else if (c=='d') return 'e';  else if (c=='e') return 'f';
    else if (c=='f') return 'g';  else if (c=='g') return 'h';  else if (c=='h') return 'i';
    else if (c=='i') return 'j';  else if (c=='j') return 'k';  else if (c=='k') return 'l';
    else if (c=='l') return 'm';  else if (c=='m') return 'n';  else if (c=='n') return 'o';
    else if (c=='o') return 'p';  else if (c=='p') return 'q';  else if (c=='q') return 'r';
    else if (c=='r') return 's';  else if (c=='s') return 't';  else if (c=='t') return 'u';
    else if (c=='u') return 'v';  else if (c=='v') return 'w';  else if (c=='w') return 'x';
    else if (c=='x') return 'y';  else if (c=='y') return 'z';  else      return ' ';
}

```

//memory node structure

struct memNode

```

{
    struct memNode * link;           //pointer link to the next node
    char * memoryResourcePointer;    //pointer pointing to the memory needed
    int thisNodeSize;               // thisNodeSize with 8, 16 or 32-byte
};

```

// memonode node pointer type

typedef struct memNode * memoryNodePointer;

memoryNodePointer tempMemNode[initialNum];

struct memNode temp[initialNum];


```

// prepare initialNum memNode pointer
void prepareMemNodeP()
{
    int t;    for (t=0; t<initialNum; t++)    tempMemNode[t]= &temp [t];    }

//scope node structure
struct scopeNode
{
    memoryNodePointer scopeMemHeadP;    //link to the next node
    int sNum;                        // number of scope node nested inside this scope
    int nNum;                        // number of memory node within this scope
    struct scopeNode * next ;        // point to next scope
};

// scope node pointer type
typedef struct scopeNode * scopePointer;
scopePointer tempScope[initialNum ],
tempNestScope[initialNum ];        struct scopeNode q[2* initialNum];

// prepare initialNum scope node pointers
void prepareScopeNodeP()
{
    int temp;
    for (temp=0; temp<initialNum; temp++)
    {
        tempScope[temp]=&q[temp];        tempScope[temp]->sNum=1;
        tempScope[temp]->nNum = 0;        tempNestScope[temp]=NULL;
    }
}

// memory structure
struct mem
{
    memoryNodePointer memHeadP[initialNum];    };

// mem type pointer
typedef struct mem * memPointer;
memPointer m;        struct mem n;

// prepare a mem pointer
void prepareMemP()
{
    m=&n;    }

// initialize the memory nodes
// provide contents and thisNodeSize to memory nodes
void initMemNode()
{

```

```

long t, t1, t2, t3;    char *temp,temp1=' ';
printUD(1);  printf("  Step 2: initialization of the memory manager\n");
temp=p;
t1=DI_at_Num[8];    t2=t1+DI_at_Num[16];    t3=t2+DI_at_Num[32];
for ( t=1; t<t3+1; t++)
{
    *temp = addInfo(temp1);    temp1 = *temp;
    tempMemNode[t]->memoryResourcePointer = temp;
    if (t<t1+1)
    {    tempMemNode[t]->thisNodeSize=DI_at_Size[8];    temp +=8;    }
    else if (t<t2+1)
    {    tempMemNode[t]->thisNodeSize=DI_at_Size[16];    temp +=16;    }
    else
    {    tempMemNode[t]->thisNodeSize=DI_at_Size[32];    temp +=32;    }
}
nodeNum=t3+1;
nodeCount[8]=(DI_at_Num[8]);
nodeCount[16]=(DI_at_Num[16]);
nodeCount[32]=(DI_at_Num[32]);

}

// add memNode p into list a as its first node
memoryNodePointer addMemNode ( memoryNodePointer a, memoryNodePointer p)
{    p->link = a;    a= p;    return a;    }

// produce a list of memory nodes from node Begin to node End
memoryNodePointer initMemList(int nodeBegin, int nodeEnd)
{
    int t=0;
    for (t=nodeEnd; t>nodeBegin-1;t--)
    {
        tempMemNode[nodeNum]=
            addMemNode(tempMemNode[nodeNum], tempMemNode[t]);
    }
    nodeNum++;
    return tempMemNode[nodeNum-1];
}

// initialize memory manager
// produce three memory node lists with memory sizes of 8, 16, and 32-byte
void initManager()
{
    int t1, t2, t3;  t1=DI_at_Num[8]; t2=t1+DI_at_Num[16]; t3=t2+DI_at_Num[32];

    prepareMemNodeP();

```

```

        prepareScopeNodeP();
        prepareMemP();
        initMemNode();
        m->memHeadP[8] = initMemList( 1, t1);
        m->memHeadP[16]= initMemList( t1+1,t2);
        m->memHeadP[32]= initMemList( t2+1, t3);
    }

// this method add the required memory size form a user
int addMemSize()
{
    int s;  printf("Enter the memory size. The memory size is: ");
    scanf("%d", &s); return s;
}

// find the right memory node for the required memory
memoryNodePointer giveMem()
{
    int s, t=1;
    while (t)
    {
        s = addMemSize();
        if ( ( s>0) &&( s <9))      {      s=8;   t=0;      }
        else if (( s>8) &&(s <17))  {      s=16;  t=0;      }
        else if ((s>16)&&( s < 33)) {      s=32;  t=0;      }
        else                        {      exceptionCatch (1, 0);t=1;  }
    }
    return m->memHeadP[s];
}

//remove the first node in the memNode, return p
memoryNodePointer removeMemNode(memoryNodePointer p)
{      p= p->link;      return p;      }

// get a memory node
memoryNodePointer getMem(memoryNodePointer p)
{
    tempMemNode[nodeNum]= p;
    m->memHeadP[p->thisNodeSize]=removeMemNode(p);
    nodeNum++;
    return tempMemNode[nodeNum-1];
}

// an interface for garbage collection
void activateGC()
{      printUD(1);  printf(" GC needed to work! \n");  printUD(0);  exit(0);  }

```

```

//check if the heap memory used is more then 2/3 of original heap memory
void checkHeap()
{
    if ((nodeCount[8]<2*heapCount[8])||(nodeCount[16]<2*heapCount[16])
        ||(nodeCount[32]<2*heapCount[32]))
        activateGC();
}

// check if the memory required is available
int checkMemory( int thisNodeSize)
{
    if (nodeCount[thisNodeSize] >0)    return 1;
    else    {    exceptionCatch (2, thisNodeSize);    return 0;    }
}

// The interface first check if the acquired memory available
// if no, it call exceptionCatch(),
// if yes, it get acquired heap memory
memoryNodePointer getHeapMem()
{
    int thisNodeSize;    tempMemNode[nodeNum]=giveMem();
    thisNodeSize =tempMemNode[nodeNum]->thisNodeSize;
    checkHeap();
    if (checkMemory(thisNodeSize))
    {
        heapCount[thisNodeSize] ++;    nodeCount[thisNodeSize] --;
        return getMem(tempMemNode[nodeNum]);
    }
    else    {    exceptionCatch (2, thisNodeSize);    return NULL;    }
}

// The interface first check if the acquired memory available
// if no, it call exceptionCatch(),
// if yes, it get acquired immortal memory
memoryNodePointer getImmortalMem()
{
    int thisNodeSize;    tempMemNode[nodeNum]=giveMem();
    thisNodeSize =tempMemNode[nodeNum]->thisNodeSize;
    if (checkMemory(thisNodeSize))
    {
        immortalCount[thisNodeSize] ++;    nodeCount[thisNodeSize] --;
        return getMem(tempMemNode[nodeNum]);
    }
    else    {    exceptionCatch (2, thisNodeSize);    return NULL;    }
}

```

```

// The interface called for adding a scope node inside scope s as its current scope
scopePointer addScopeNode(scopePointer s)
{
    tempScope[scopeNum]->next =s;
    tempScope[scopeNum]->scopeMemHeadP=NULL;
    tempScope[scopeNum]->nNum=0;
    tempScope[scopeNum]->sNum=s->sNum+1;
    s=tempScope[scopeNum];    scopeNum++;
    return s;
}

// remove the current scope
scopePointer removeScopeNode(scopePointer s)
{    return s->next;    }

// check if the scope exists
int checkScope( int i)
{    if (!tempNestScope[i]) return 1;    else return 0; }

// The interface called for either creating a new compound scope i or adding a new scope
// into the compound scope as its current scope
void getScope(int i)
{
    if (checkScope(i)) {tempNestScope[i]=tempScope[nodeNum]; nodeNum++; }
    else    tempNestScope[i]=addScopeNode(tempNestScope[i]);
}

// The interface first check if the compound scope exists
// if yes, it try to get the required memory; otherwise, it calls exceptionCatch()
// if it fails to get acquired memory it call exceptionCatch(),
// if it get acquired memory, it add the memory into the current scope as first node.
memoryNodePointer getScopedMem(int s)
{
    int thisNodeSize;
    if (checkScope(s))    {    exceptionCatch (3, s); return NULL;    }
    else
    {
        tempMemNode[nodeNum]=giveMem();
        thisNodeSize =tempMemNode[nodeNum]->thisNodeSize;
        if (checkMemory(thisNodeSize))
        {
            scopeCount[thisNodeSize] ++;    nodeCount[thisNodeSize] --;
            tempMemNode[nodeNum]=getMem(tempMemNode[nodeNum]);
            tempNestScope[s]->scopeMemHeadP =
                addMemNode(tempNestScope[s]-

```

```

        >scopeMemHeadP, tempMemNode[nodeNum]);
    tempNestScope[s]->nNum++; nodeNum++;
    return tempNestScope[s]->scopeMemHeadP;
}
else { exceptionCatch (2, thisNodeSize); return NULL; }
}

// this method show the information in one memory node
void showScanInfo(char*p, int i)
{ printf("%d %c %d), p, *p, i);}

// scan all memory nodes form p in a scope.
void scanOneScope(memoryNodePointer p, int j)
{
    int i;
    for (i=1; i<j+1;i++)
    {
        showScanInfo ( (p->memoryResourcePointer), (p->thisNodeSize));
        p=p->link;
    }
}

// scan all the memory nodes in each scope of the compound scope
// use scanOneScope to scan the memory nodes in scope
// produce Sn as compound scope, -> as a scope
// and (address, content, memory size) as a memory node
void scanNestedScope( int i)
{
    int j, k;
    tempScope[scopeNum] = tempNestScope[i];
    if (checkScope(i)) exceptionCatch (3,i);
    else
    {
        k=tempNestScope[i]->sNum; printf("S%d ", i);
        for ( j=1; j<k+1; j++)
        {
            printf("--> ");
            scanOneScope(tempScope[scopeNum]->scopeMemHeadP,
                        tempScope[scopeNum]->nNum);
            tempScope[scopeNum+1] = tempScope[scopeNum]->next;
            scopeNum++;
        }
        scopeNum++;
    }
}
}

```

```

// this method check if a memoryNodePointer has memory nodes
// if yes, it return 1; if no it return 0.
int checkMemScope ( memoryNodePointer p)
{ if (p) return 1; else return 0; }

// The interface called to end a compound scope i
// step 1: It check if the compound scope i exists
// step 2: if i exists, goto step 3; otherwise it call the exceptionCatch()
// step 3: it check if there is memory nodes in the current scope other
// step 4: if yes, recover the memory and remove the current scope
// step 5: if no, remove the current scope
void endScope(int i)
{
    int j;
    tempScope[scopeNum]=tempNestScope[i];
    if (checkScope(i)) exceptionCatch (3,i);
    else if (!checkMemScope (tempScope[scopeNum])->scopeMemHeadP)
        tempNestScope[i]=removeScopeNode(tempNestScope[i]);
    else there are nodes in the first scope
    {
        while (checkMemScope ((tempScope[scopeNum])->scopeMemHeadP))
        {
            tempMemNode[nodeNum]=
                (tempScope[scopeNum])->scopeMemHeadP;
            (tempScope[scopeNum])->scopeMemHeadP=
                removeMemNode((tempScope[scopeNum])->scopeMemHeadP);
            j=tempMemNode[nodeNum]->thisNodeSize ;
            scopeCount[j]++;
            m->memHeadP[j] = addMemNode(m->memHeadP[j],
                tempMemNode[nodeNum]);
            nodeNum++;
        }
        tempNestScope[i]=removeScopeNode(tempNestScope[i]);
        scopeNum++;
    }
}

// the test system implemented to test the memory manager
// enter the command h for heap memory manipulation
// enter the command i for immortal memory manipulation
// enter commands gn, mn, en, or sn for scope memory manipulation (n: 1,2,3,4 or 5)
// enter the command c for the details of each commands.
// enter themmend d if tests are done
void testManager()
{

```

```

char c0, c1 ;
while (1)
{
    showManager();      c0=getchar();      c0=getchar();
    if (c0 == 'h')      getHeapMem();
    else if (c0 == 'i')  getImmortalMem();
    else if (c0 == 'g')
    {
        c1=getchar();
        if (c1=='1')    getScope(1);  else if (c1=='2')  getScope(2);
        else if (c1=='3') getScope(3);  else if (c1=='4')  getScope(4);
        else if (c1=='5') getScope(5);  else      exceptionCatch(3,0);
    }
    else if (c0 == 'm')
    {
        c1=getchar();
        if (c1=='1')    getScopedMem(1);
        else if (c1=='2') getScopedMem(2);
        else if (c1=='3') getScopedMem(3);
        else if (c1=='4') getScopedMem(4);
        else if (c1=='5') getScopedMem(5);
        else      exceptionCatch(3,0);
    }
    else if (c0 == 's')
    {
        c1=getchar();
        if (c1=='1')    scanNestedScope(1);
        else if (c1=='2') scanNestedScope(2);
        else if (c1=='3') scanNestedScope(3);
        else if (c1=='4') scanNestedScope(4);
        else if (c1=='5') scanNestedScope(5);
        else      exceptionCatch(3,0);
    }
    else if (c0 == 'e')
    {
        c1=getchar();
        if (c1=='1')    endScope(1);  else if (c1=='2')  endScope(2);
        else if (c1=='3') endScope(3);  else if (c1=='4')  endScope(4);
        else if (c1=='5') endScope(5);  else  exceptionCatch(3,0);
    }
    else if (c0 == 'c')  showMoreCommand();
    else if (c0 == 'd')  done();
    else    illInput();
}
}
//////////////////////////////////////////////////////////////// main ///////////////////////////////////////////////////////////////////

```



```
void main()
{
    preface_for_RTSJ_MemoryManager();
    simulate_RTJVM_DistributionInfo();
    initManager();
    testManager();
}
//////////////////////////////// End of Memory Manager //////////////////////////////////
```