JAVA VIRTUAL MACHINE IMPLEMENTATION ON MICRO-C/OS-II REAL-TIME OPERATING SYSTEM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF
NATURAL AND APPLIED SCIENCES OF
ÇANKAYA UNIVERSITY

BY
ALP BÜLENT BURÇ SÜRMELİ

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR
THE DEGREE OF MASTER OF SCIENCE
IN THE DEPARTMENT OF
COMPUTER ENGINEERING

SEPTEMBER 2005

Approval of the Graduate School of Natural and Applied Sciences.

[illegible]

Prof. Dr. Yurdahan GÜLER

Director

I certify that this thesis satisfies all requirements as a thesis for the degree of Master of Science.

[illegible]

Prof. Dr. Mehmet Reşit TOLUN

Head of the Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

[illegible]

Prof. Dr. Turhan ALPER

Supervisor

Examining Committee Members

Prof. Dr. Turhan ALPER	[illegible]
Asst. Prof. Dr. Reza HASSANPOUR	[illegible]
Dr. Erhan GÖKÇAY	[illegible]

ABSTRACT

JAVA VIRTUAL MACHINE IMPLEMENTATION ON MICRO-C/OS-II REAL-TIME OPEATING SYSTEM

SÜRMELİ, Alp Bülent Burç
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Turhan ALPER
September 2005, 86 pages

Virtual Machine (VM) implies mostly the transperancy of executable code from the underlying computer hardware . So, a virtual machine is expected to have an independent instruction set, program description syntax and mostly a different program execution architecture independent from the underlying computer hardware.

Java Virtual Machine (JVM), has the capability of executing the platform independent object file called *class* file which is an output of the compilation process which takes the source files written by using the java syntax and semantic rules as an input.

Micro-C/OS-II is a real-time operating system which is certifed to be real-time operating system supporting safety-critical software development with FAA (Federal Aviation Agency) certification. MicroC/OS-II is already ported to more than 20 different hardware manufacturers computer architectures.

The aim of this thesis work is realizing a JVM core by using the pure MicroC/OS-II system calls. So, the resulting JVM core will have the capability to be ported easly to any computer architecture which is ported by MicroC/OS-II. Also this JVM core will be, at least, a soft real-time execution environment where applications developed in Java can be deployed and executed as they are deployed and executed in other JVMs. At the end, this thesis work is also expected to be a baseline and a guide for the future developers and designer who will be improving this core to support the safety-critical real-time software development.

Keywords : Java Virtual Machine, K Virtual Machine, Java, Virtual Machine, MicroC/OS-II, uCOS , real-time

ÖZ

MICRO-C/OS-II GERÇEK ZAMANLI İŞLETİM DİZGESİ ÜZERİNDE JAVA SANAL MAKİNESİ GERÇEKLEŞTİRİMİ

SÜRMELİ , Alp Bülent Burç
Yüksek Lisans , Bilgisayar Mühendisliği Bölüm
Tez Yöneticisi : Prof . Dr . Turhan ALPER
Eylül 2005, 86 sayfa

Sanal Makine deyimi çoğunlukla işletilebilir kodun alttaki bilgisayar donanımına transparan olması kavramına karşılık gelir. Dolayısıyla, bir sanal makinenin bağımsız bir komut kümesi, program tanımlama söz dizimi ve çoğunlukla da üzerinde çalıştığı bilgisayar donanımından bağımsız bir program koşma mimarisi olur.

Java Sanal Makinesi, Java programlama dili söz dizim ve anlambilim kuralları kullanılarak geliştirilen kaynak kodun derlenmesi ile elde edilen ve sınıf adıyla anılan platformdan bağımsız amaç kütükleri koşabilme kabiliyetine sahiptir.

Micro-C/OS-II güven-kritik gerçek-zamanlı yazılım geliştirimini destekleyen Amerikan Federal Havacılık Ajansı (FAA) sertifikalı bir gerçek-zamanlı işletim dizgesidir. Micro-C/OS-II işletim dizgesi hali hazırda yirmiden (20) fazla donanım üreticisinin değişik bilgisayar mimarileri üzerine taşınmış durumdadır.

Bu tez çalışmasının amacı Micro-C/OS-II işletim dizgesi çağrılarını kullanarak bir Java Sanal Makinesi çekirdeğini çalışır hale getirmektir. Bu sayede, elde edilecek Java Sanal Makinesi çekirdeği Micro-C/OS-II işletim dizgesinin üzerine taşındığı bütün bilgisayar mimarilerine kolayca taşınabilecektir. Bununla beraber, bu Java Sanal Makinesi, Java programla dili kullanılarak geliştirilmiş uygulamaların diğer Java Sanal Makinelerine yüklenip koşuldukları gibi yüklenip koşulabileceği hafif gerçek-zamanlı bir işletim ortamı sağlayacaktır. Yapılan çalışma sonucunda ortaya konulan tez, gelecekte bu Java Sanal Makinesi çekirdeğinin kabiliyetlerini güven-kritik gerçek-zamanlı yazılım geliştirilebilmesine imkan verecek şekilde geliştirecek yazılım mühendisleri için yetkin bir referans ve rehber olmaya aday bir çalışmadır.

Anahtar Kelimeler : Java Sanal Makinesi , Java , Sanal Makine , MicroC/OS-II , uCOS , gerçek-zamanlı

To My Family

ACKNOWLEDGEMENTS

First and foremost, I express sincere appreciation to Prof . Dr . Turhan ALPER for his patience and guidance through out the development of this thesis. And, I would like express my sincere appreciation to my mother and sister for their patience, encourgements and endurance. I would like to express my gratitude to my colleagues Ahmet B. ÇAVDAR, Burak CODUR, Serden ÇİVRİL and Emre H. KÖK for their encourgements, guidance and help to accomplish my thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Background of the Development Process

There are a great number of operating systems ported on top of the different computer architectures. All of them tend to be similar in the category of services they provide as Tasking Services, Mutex Services, Semaphore Services, Message Box Services, Memory Management Services, Event Handling Services, Timing Services and I/O Services. Even if they have the given categories of services individually, the system call signatures and the quality of these services varies significantly. So, some of them are certified to be real-time or not, or cluster aware or not and also some further evaluation takes place for the discrimination of each for addressing specific needs of special domains.

This variance of the knowledge required about application development not only for different platforms but to solve similar problems is the driving force to have a abstraction layer like an application programming interface (API) like Java Virtual Machine has. So, using one API for all is a factor of improving the software development productivity which is also recognized by software developing society. And there is serious amount of work carried out and in progress to port the JVM to different computer architectures. Also, there are a significant number of applications developed by Java programming language to run on these JVMs ported.

There is a vital amount of work to make the JVM portable to variety of platforms because of the reasons outlined. This thesis work is proposing a way of porting a specialized JVM core to different platfroms easly by using an abstraction provided through the system calls of MicroC/OS-II real-time operating system.

There are also some other efforts to improve Java API to address the real-time requirements of sophisticated application domains requiring mission-critical or safety-critical execution. As a result of these efforts Java community published a specification declaring the services and their quality which is called Real-Time Java Specification (RTJS). By implementing a JVM core on top of an real-time operating system like MicroC/OS-II is also a choice made on purpose to evaluate the possibility of meeting some of RTJS requirements.

This thesis is pretend to be a guide and baseline for the JVM designers and implementors . Even though resolving the optimization problems of JVM domain is far beyond of this thesis work , already it may give an idea unintentionally.

1.2 Utilization of Proposed Java Application Deployment Capability

Our contribution is providing a single way of building the JVM dedicated for the java application compiled into a class file that may be deployable to any platform on which the MicroC/OS-II is already deployable. Having such a deployment capability, a developer can develop and produce the java application on any host. And then by going through the steps explained in thesis, it is possible to build the executable java application by using a C cross compiler producing object code that is deployable to the host platform. The application ready for execution can be installed onto the target computer environment through a media which is capable to boot up the system. The media can be either a booting network resource (an serial cable or ethernet e.g.) or a booting storage device (like diskettte or flash memory e.g.). The Figure 1.1 below illustrates a sample utilization scenario for the proposed java application deployment capability.

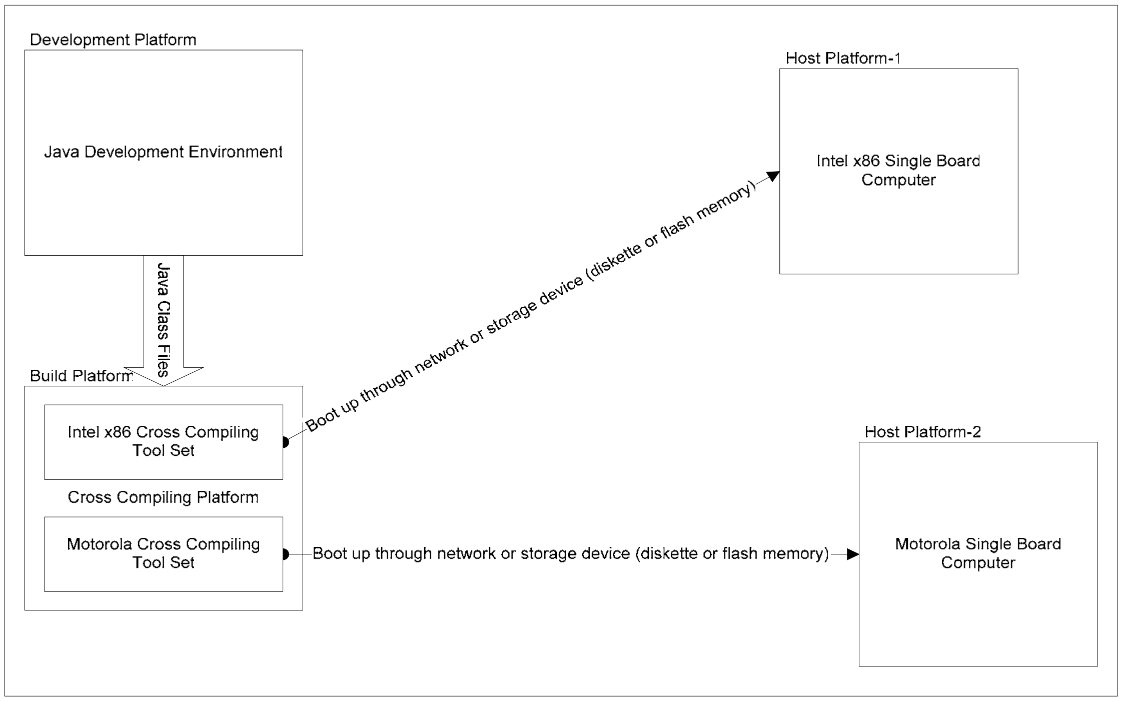

Development Platform
Java Development Environment
Java Class Files
Build Platform
Intel x86 Cross Compiling Tool Set
Cross Compiling Platform
Motorola Cross Compiling Tool Set
Boot up through network or storage device (diskette or flash memory)
Host Platform-1
Intel x86 Single Board Computer
Boot up through network or storage device (diskette or flash memory)
Host Platform-2
Motorola Single Board Computer

Figure 1.1: Utilization of the proposed java application deployment.

CHAPTER 2

AN OVERVIEW OF JVM ARCHITECTURE

2.1 Developer Point Of View

The use of the JVM from the developers point of view consists of two major steps to go as seen in Figure 2.1 . One of them is the compile-time environment in which the Java source files compilied into class files and the other is the run-time environment in which the executable (interpretable) class files deployed to JVM.

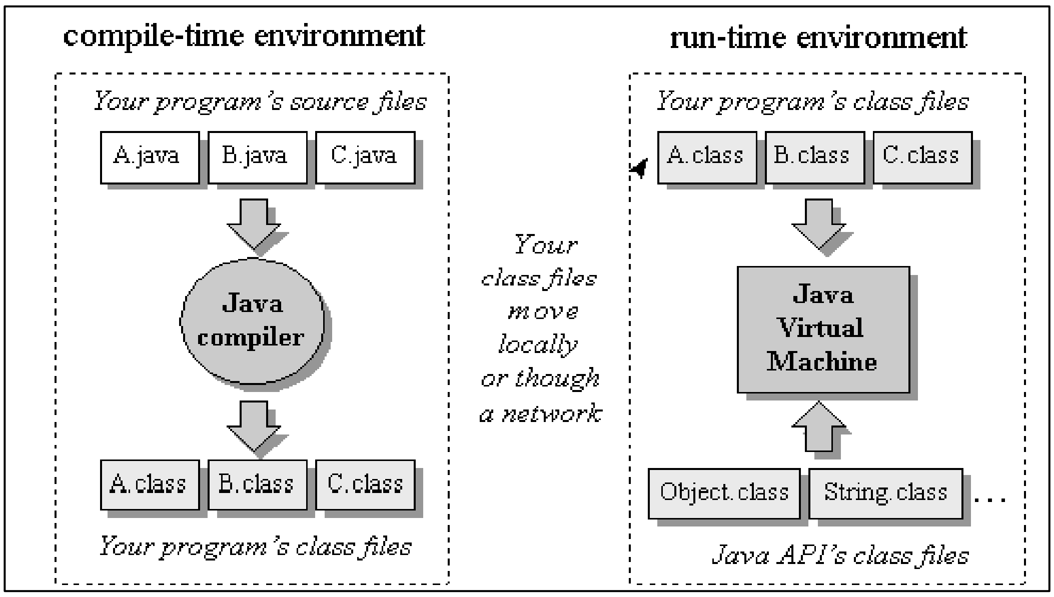


Figure 2.1: The phases of running an application on the JVM.

2.2 Architectural Point of View

The JVM word implies the instance of a set of components which is created for running only an application, which means the life cycle of the JVM is has an one to one dependency with application deployed.

A JVM architecture, which is specialized for running an instance of an application (class) at a time, has some specialized components developed for the purpose of accomplishing its function. The Figure 2.2 shows the architectural view of the JVM .

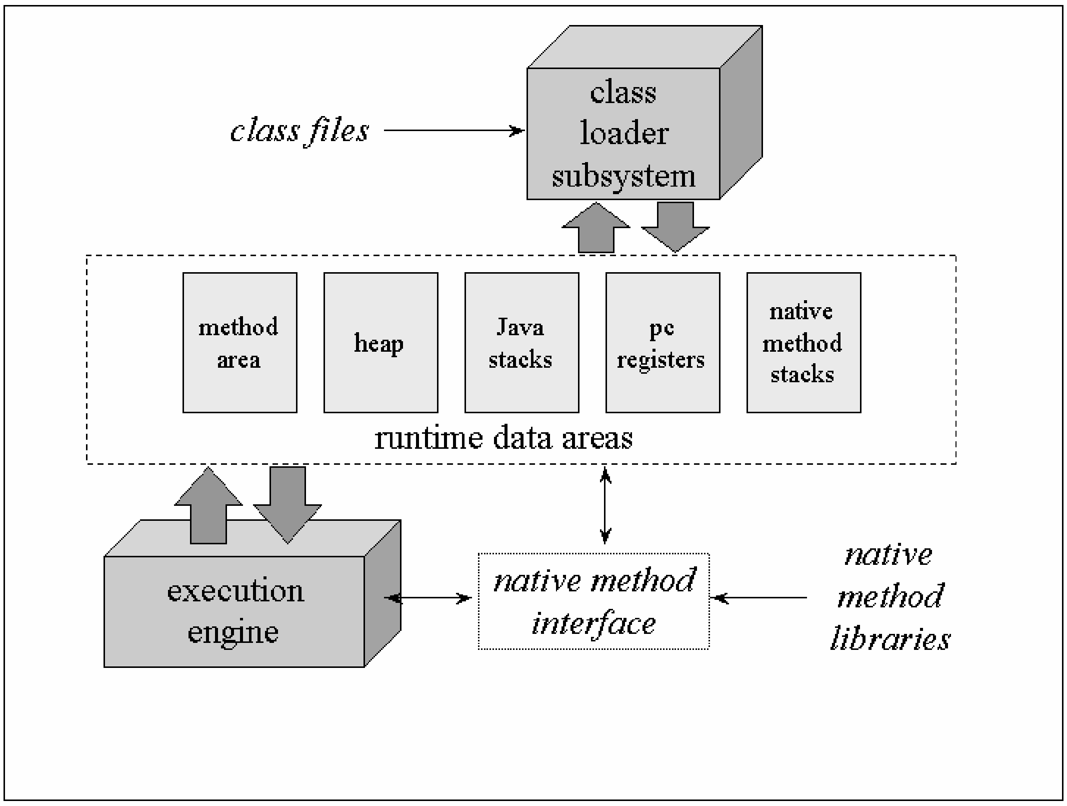


Figure 2.2: JVM functional block diagram.

As seen in Figure 2.2, an application, which is a collection of class files, is required to be loaded prior to its execution. The load operation performs a specialized parsing operation which prepares the some of the static data required to execute the application in the runtime data area. Upon loading the application the entry method of the entry class which is the implicitly defined "init()" function is invoked and then the rest of the application logic execution sequence takes place.[1]

The execution engine as a JVM function is responsible for executing the JVM instructions extracted from the application source code during the compilation process also responsible for managing and handling some of the volatile data areas which resides in runtime data areas upon a method invocation, a class creation or a thread execution.

Native methods are also required by JVMs , naturally , to map the JVM functions to the appropriate systems calls of the target platform and also extending the JVM API to support a wider range of input-output operations.

The specialized portions of the functional block diagram given in Figure 2.2 is detailed in the following sections to clarify their responsibility and capability.

2.2.1 Class Loader

The class loader can be thought as the application entry point to the JVM core. Class Loader takes the specialized binary class file, verifies and builds the data structures required during the life-time of the application execution .

Comparing with the conventional loader of the operating system shells like in DOS or UNIX, class loader has similar behavior except for some more detailed process about the class execution. This capability is also a result of the data provided in the class file format.[1]

A class file is a binary file consisting of the description for a class or an interface regarding the Java nomenclature. A class is an encapsulation of attributes and methods together which is thought be related conceptually. And an interface is a specialized class to describe method signatures for a group of logically related objects. And an interface does not have any concrete object instance at any time.

Each `class` file contains the definition of a single class or an interface . A `class` file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, and eight consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high bytes come first.

2.2.2 Runtime Data Areas

The Java virtual machine defines various runtime data areas that are used during execution of a program. Some of these data areas are created on Java virtual machine start-up and are destroyed only when the Java virtual machine exits. Other data areas are per thread. Per-thread data areas are created when a thread is created and destroyed when the thread exits.

2.2.3 The pc Register

The Java virtual machine can support many threads of execution at once. Each Java virtual machine thread has its own `pc` (program counter) register. At any point, each Java virtual machine thread is executing the code of a single method, the current method for that thread. If that method is not `native`, the `pc` register contains the address of the Java virtual machine instruction currently being executed. If the method currently being executed by the thread is `native`, the value of the Java virtual machine's `pc` register is undefined. The Java virtual machine's `pc` register is wide enough to hold a `returnAddress` or a native pointer on the specific platform.[2]

2.2.3.1 Java Virtual Machine Stacks

Each Java virtual machine thread has a private *Java virtual machine stack*, created at the same time as the thread. A Java virtual machine stack stores frames.[2] A Java virtual machine stack is analogous to the stack of a conventional language such as C: it holds local variables and partial results, and plays a part in method invocation and return. Because the Java virtual machine stack is never manipulated directly except to push and pop frames, frames may be heap allocated. The memory for a Java virtual machine stack does not need to be contiguous.[1]

The Java virtual machine specification permits Java virtual machine stacks either to be of a fixed size or to dynamically expand and contract as required by the computation.[2] If the Java virtual machine stacks are of a fixed size, the size of each Java virtual machine stack may be chosen independently when that stack is created. A Java virtual machine implementation may provide the programmer or the user control over the initial size of Java virtual machine stacks, as well as, in the case of dynamically expanding or contracting Java virtual machine stacks, control over the maximum and minimum sizes.[3]

The following exceptional conditions are associated with Java virtual machine stacks:

- If the computation in a thread requires a larger Java virtual machine stack than is permitted, the Java virtual machine throws a `StackOverflowError`.

- If Java virtual machine stacks can be dynamically expanded, and expansion is attempted but insufficient memory can be made available to effect the expansion, or if insufficient memory can be made available to create the initial Java virtual machine stack for a new thread, the Java virtual machine throws an `OutOfMemoryError`.[1]

2.2.3.2 Heap

The Java virtual machine has a *heap* that is shared among all Java virtual machine threads.[2] The heap is the runtime data area from which memory for all class instances and arrays is allocated.

The heap is created on virtual machine start-up. Heap storage for objects is reclaimed by an automatic storage management system (known as a *garbage collector*); objects are never explicitly deallocated.[1] The Java virtual machine assumes no particular type of automatic storage management system, and the storage management technique may be chosen according to the implementor's system requirements. The heap may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger heap becomes unnecessary. The memory for the heap does not need to be contiguous.[3]

A Java virtual machine implementation may provide the programmer or the user control over the initial size of the heap, as well as, if the heap can be dynamically expanded or contracted, control over the maximum and minimum heap size.[2]

The following exceptional condition is associated with the heap:

- If a computation requires more heap than can be made available by the automatic storage management system, the Java virtual machine throws an `OutOfMemoryError`.[1]

2.2.3.3 Method Area

The Java virtual machine has a *method area* that is shared among all Java virtual machine threads. The method area is analogous to the storage area for compiled code of a conventional language or analogous to the "text" segment in a UNIX process. It stores per-class structures such as the runtime constant pool, field and method data, and the code for methods and constructors, including the special methods used in class and instance initialization and interface type initialization.[4]

The method area is created on virtual machine start-up. Although the method area is logically part of the heap, simple implementations may choose not to either garbage collect or compact it.[3] This version of the Java virtual machine specification does not mandate the location of the method area or the policies used to manage compiled code. The method area may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger method area becomes unnecessary. The memory for the method area does not need to be contiguous.[1]

A Java virtual machine implementation may provide the programmer or the user control over the initial size of the method area, as well as, in the case of a varying-size method area, control over the maximum and minimum method area size.

The following exceptional condition is associated with the method area:

- If memory in the method area cannot be made available to satisfy an allocation request, the Java virtual machine throws an `OutOfMemoryError`.[1]

2.2.3.4 Runtime Constant Pool

A *runtime constant pool* is a per-class or per-interface runtime representation of the `constant_pool` table in a `class` file.[1] It contains several kinds of constants, ranging from numeric literals known at compile time to method and field references that must be resolved at run time. The runtime constant pool serves a function similar to that of a symbol table for a conventional programming language, although it contains a wider range of data than a typical symbol table.

Each runtime constant pool is allocated from the Java virtual machine's method area. The runtime constant pool for a class or interface is constructed when the class or interface is created by the Java virtual machine.[1]

The following exceptional condition is associated with the construction of the runtime constant pool for a class or interface:

- When creating a class or interface, if the construction of the runtime constant pool requires more memory than can be made available in the method area of the Java virtual machine, the Java virtual machine throws an `OutOfMemoryError`.[1]

2.2.3.5 Native Method Stacks

An implementation of the Java virtual machine may use conventional stacks, colloquially called "C stacks," to support `native` methods, methods written in a language other than the Java programming language.[1] Native method stacks may also be used by the implementation of an interpreter for the Java virtual machine's instruction set in a language such as C. Java virtual machine implementations that cannot load `native` methods and that do not themselves rely on conventional stacks need not supply native method stacks.[1] If supplied, native method stacks are typically allocated per thread when each thread is created.

The Java virtual machine specification permits native method stacks either to be of a fixed size or to dynamically expand and contract as required by the computation. If the native method stacks are of a fixed size, the size of each native method stack may be chosen independently when that stack is created.[1] In any case, a Java virtual machine implementation may provide the programmer or the user control over the initial size of the native method stacks. In the case of varying-size native method stacks, it may also make available control over the maximum and minimum method stack sizes.

The following exceptional conditions are associated with native method stacks:

- If the computation in a thread requires a larger native method stack than is permitted, the Java virtual machine throws a `StackOverflowError`.
- If native method stacks can be dynamically expanded and native method stack expansion is attempted but insufficient memory can be made available, or if insufficient memory can be made available to create the initial native method stack for a new thread, the Java virtual machine throws an `OutOfMemoryError`.

2.2.4 Execution Engine

Execution Engine component of the JVM is expected to execute the JVM instructions in the given order. This responsibility of the execution engine includes the maintenance of the data structures required during the instruction execution.[4] Frame is a data structure which encapsulates the fields required during the method execution in a JVM.

2.2.4.1 Frames

A *frame* is used to store data and partial results, as well as to perform dynamic linking , return values for methods, and dispatch exceptions.[2]

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes, whether that completion is normal or abrupt (it throws an uncaught exception).[2] Frames are allocated from the Java virtual machine stack of the thread creating the frame. Each frame has its own array of local variables, its own operand stack, and a reference to the runtime constant pool of the class of the current method.

The sizes of the local variable array and the operand stack are determined at compile time and are supplied along with the code for the method associated with the frame. Thus the size of the frame data structure depends only on the implementation of the Java virtual machine, and the memory for these structures can be allocated simultaneously on method invocation.

Only one frame, the frame for the executing method, is active at any point in a given thread of control. This frame is referred to as the *current frame*, and its method is known as the *current method*. The class in which the current method is defined is the *current class*. Operations on local variables and the operand stack are typically with reference to the current frame.[1]

A frame ceases to be current if its method invokes another method or if its method completes. When a method is invoked, a new frame is created and becomes current when control transfers to the new method. On method return, the current frame passes back the result of its method invocation, if any, to the previous frame. The current frame is then discarded as the previous frame becomes the current one.

Note that a frame created by a thread is local to that thread and cannot be referenced by any other thread.[2]

2.2.4.2 Local Variables

Each frame contains an array of variables known as its *local variables*. The length of the local variable array of a frame is determined at compile time and supplied in the binary representation of a class or interface along with the code for the method associated with the frame.

A single local variable can hold a value of type `boolean`, `byte`, `char`, `short`, `int`, `float`, `reference`, or `returnAddress`. A pair of local variables can hold a value of type `long` or `double`.[1]

Local variables are addressed by indexing. The index of the first local variable is zero. An integer is be considered to be an index into the local variable array if and only if that integer is between zero and one less than the size of the local variable array.

A value of type `long` or type `double` occupies two consecutive local variables. Such a value may only be addressed using the lesser index. For example, a value of type `double` stored in the local variable array at index *n* actually occupies the local variables with indices *n* and *n* +1; however, the local variable at index *n* +1 cannot be loaded from. It can be stored into. However, doing so invalidates the contents of local variable *n*.[2]

The Java virtual machine does not require *n* to be even. In intuitive terms, values of types `double` and `long` need not be 64-bit aligned in the local variables array. Implementors are free to decide the appropriate way to represent such values using the two local variables reserved for the value.

The Java virtual machine uses local variables to pass parameters on method invocation. On class method invocation any parameters are passed in consecutive local variables starting from local variable *0*. On instance method invocation, local variable *0* is always used to pass a reference to the object on which the instance method is being invoked (`this` in the Java programming language). Any parameters are subsequently passed in consecutive local variables starting from local variable *1*.

2.2.4.3 Operand Stacks

Each frame contains a last-in-first-out (LIFO) stack known as its *operand stack*.[2] The maximum depth of the operand stack of a frame is determined at compile time and is supplied along with the code for the method associated with the frame.

Where it is clear by context, the operand stack of the current frame will sometimes be refered to as simply the operand stack.

The operand stack is empty when the frame that contains it is created. The Java virtual machine supplies instructions to load constants or values from local variables or fields onto the operand stack. Other Java virtual machine instructions take operands from the operand stack, operate on them, and push the result back onto the operand stack. The operand stack is also used to prepare parameters to be passed to methods and to receive method results.

For example, the *iadd* instruction adds two `int` values together. It requires that the `int` values to be added be the top two values of the operand stack, pushed there by previous instructions. Both of the `int` values are popped from the operand stack. They are added, and their sum is pushed back onto the operand stack. Subcomputations may be nested on the operand stack, resulting in values that can be used by the encompassing computation.

Each entry on the operand stack can hold a value of any Java virtual machine type, including a value of type `long` or type `double`.[1]

Values from the operand stack must be operated upon in ways appropriate to their types. It is not possible, for example, to push two `int` values and subsequently treat them as a `long` or to push two `float` values and subsequently add them with an *iadd* instruction. A small number of Java virtual machine instructions (the *dup* instructions and *swap*) operate on runtime data areas as raw values without regard to their specific types; these instructions are defined in such a way that they cannot be used to modify or break up individual values. These restrictions on operand stack manipulation are enforced through `class` file verification.

At any point in time an operand stack has an associated depth, where a value of type `long` or `double` contributes two units to the depth and a value of any other type contributes one unit.

2.2.4.4 Dynamic Linking

Each frame contains a reference to the runtime constant pool for the type of the current method to support *dynamic linking* of the method code.[3] The `class` file code for a method refers to methods to be invoked and variables to be accessed via symbolic references. Dynamic linking translates these symbolic method references into concrete method references, loading classes as necessary to resolve as-yet-undefined symbols, and translates variable accesses into appropriate offsets in storage structures associated with the runtime location of these variables.[3]

This late binding of the methods and variables makes changes in other classes that a method uses less likely to break this code.

2.2.4.5 Normal Method Invocation Completion

A method invocation *completes normally* if that invocation does not cause an exception to be thrown, either directly from the Java virtual machine or as a result of executing an explicit `throw` statement. If the invocation of the current method completes normally, then a value may be returned to the invoking method. This occurs when the invoked method executes one of the return instructions, the choice of which must be appropriate for the type of the value being returned (if any).[1]

The current frame is used in this case to restore the state of the invoker, including its local variables and operand stack, with the program counter of the invoker appropriately incremented to skip past the method invocation instruction. Execution then continues normally in the invoking method's frame with the returned value (if any) pushed onto the operand stack of that frame.

2.2.4.6 Abrupt Method Invocation Completion

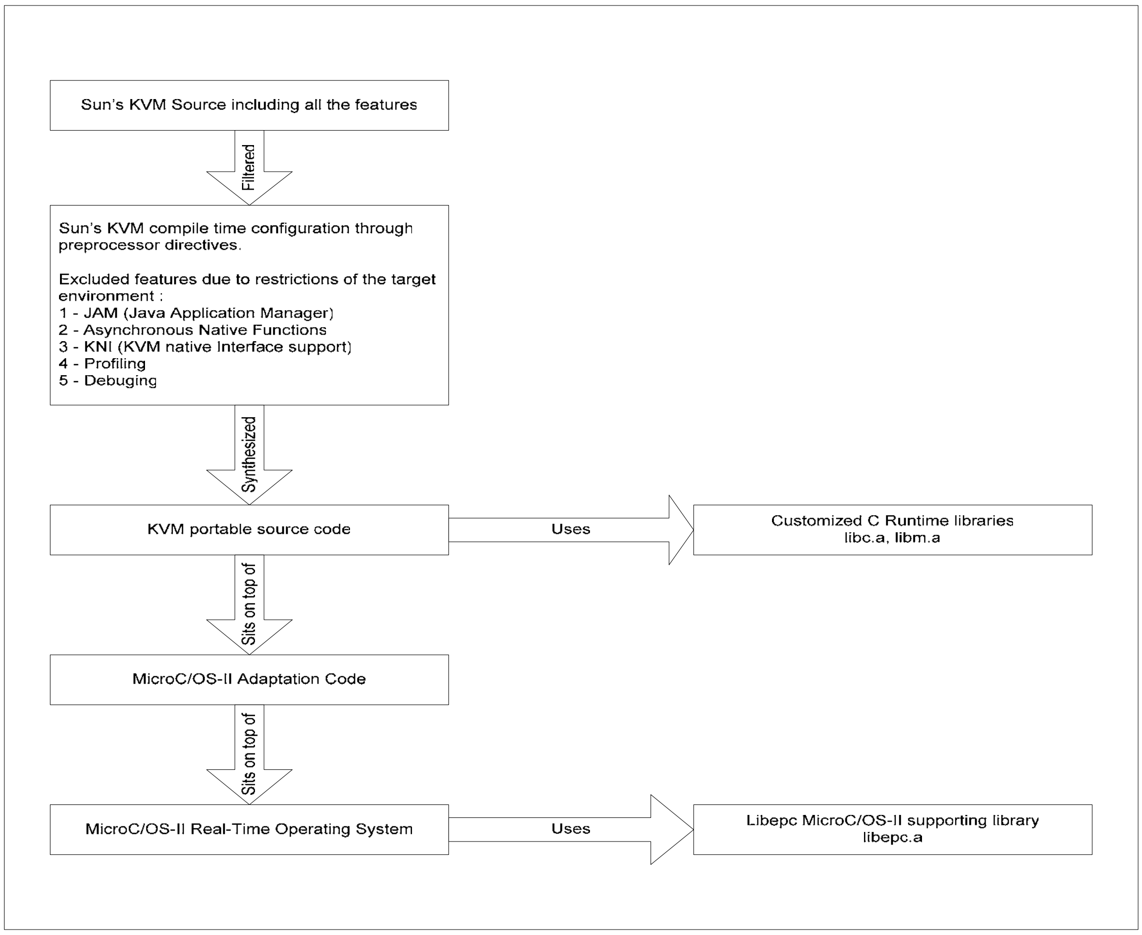
A method invocation *completes abruptly* if execution of a Java virtual machine instruction within the method causes the Java virtual machine to throw an exception, and that exception is not handled within the method. Execution of an *athrow* instruction also causes an exception to be explicitly thrown and, if the exception is not caught by the current method, results in abrupt method invocation completion. A method invocation that completes abruptly never returns a value to its invoker.[1]

CHAPTER 3

THE PROPOSED JVM CORE DEPLOYMENT ARCHITECTURE

3.1 Development-Time Environment

The development-time environment is choosen to be emphasized distinctly for its suppport for different Operating Systems with the use of the abstraction. The Figure 3.1 summarizes the Development-Time Environment.



Sun's KVM Source including all the features
Filtered
Sun's KVM compile time configuration through preprocessor directives.
Excluded features due to restrictions of the target environment :
1 - JAM (Java Application Manager)
2 - Asynchronous Native Functions
3 - KNI (KVM native Interface support)
4 - Profiling
5 - Debuging
Synthesized
KVM portable source code
Uses
Customized C Runtime libraries libc.a, libm.a
Sits on top of
MicroC/OS-II Adaptation Code
Sits on top of
MicroC/OS-II Real-Time Operating System
Uses
Libepc MicroC/OS-II supporting library libepc.a

Figure 3.1: Development-Time environment.

3.2 Java Application Development-Time Environment

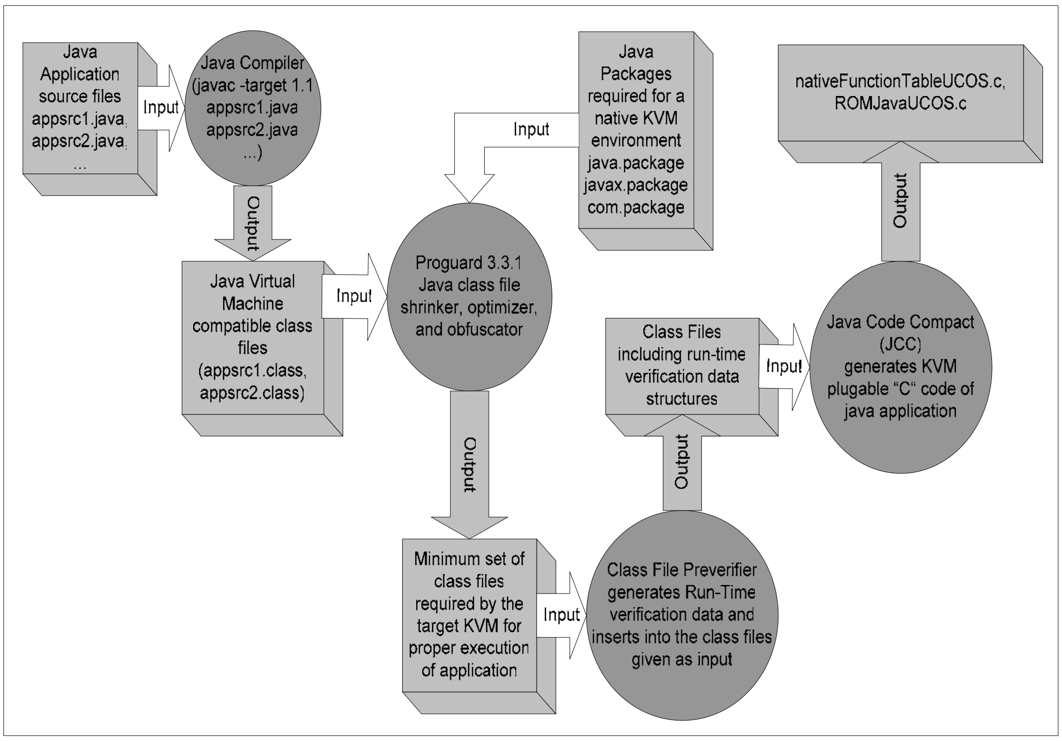
Java Application Development-Time environment defines the steps to be taken to make java application source ready for deployment onto the proposed virtual machine.

The process starts when application source files (*.java) are ready for compilation into class files with the use of Sun Java compiler *javac* having the version 1.4.2_01. The *javac* tool is required to be called with the *target* option to specify the complying release of the virtual machine core on which the class file is planned to be executed.

The output of the *Javac* compiler is taken as an input for the Proguard tool(version 3.3.1). Proguard tool is a shrinker, optimizer and obsfucator tool for java class files.[5] Proguard takes the application classes and the other classes, required by the native virtual machine environment, as input and populates the minimum set of class files needed to execute the java application. Proguard tool is required mostly for its optimization capability to reduce the size of target dedicated virtual machine. These class files are also packed into a compressed file for the use of tools in the chain.

Generated compressed file is provided to the Java Code Compact (JCC) tool. Java Code Compact tool is a virtual machine stub generator. The stub is a set of C files generated by Java Code Compact which are used while building the dedicated virtual machine for java application developed.

Java Application Development-Time sequence of activities are completed with the generation of stub which is an output of the JCC tool. The Figure 3.2 summarizes the Java Application Development-Time Environment.



Java Application source files appsrc1.java, appsrc2.java, ...
Input
Java Compiler (javac -target 1.1 appsrc1.java appsrc2.java ...)
Output
Java Virtual Machine compatible class files (appsrc1.class, appsrc2.class)
Input
Java Packages required for a native KVM environment java.package javax.package com.package
Input
Proguard 3.3.1 Java class file shrinker, optimizer, and obfuscator
Output
Minimum set of class files required by the target KVM for proper execution of application
Input
Class File Preverifier generates Run-Time verification data and inserts into the class files given as input
Output
Class Files including run-time verification data structures
Input
Java Code Compact (JCC) generates KVM plugable "C" code of java application
Output
nativeFunctionTableUCOS.c, ROMJavaUCOS.c

Figure 3.2: Java Application Development-Time environment.

3.3 Build-Time Environment

Build-Time Environment represents the build process of virtual machine with the use of compiler and linker.

Being the DOS port of the GNU, DJGPP tools are used as the build tool set. The compiler is choosen to be “gcc” (version 4.0.1) and the linker is choosen to be “ld” (version 2.16.1).

Target virtual machine is proposed to be running with the capability of booting the system. For this implementation, target host is selected to be an intel 386 or higher CPU based platform. So, the executed booting procedure supports the machines complying with the platform specifically mentioned. Booting capability is the function of the entry point defined in the libepc.a library.

The functions required by the MicroC/OS-II Real-Time operating system which is platform dependent (like basic I/O handling and Interrupt Handling) are also provided by methods resides in the libepc.a library.

The portable KVM code which make use of the string(string), standard input/output(stdio), time(time), standard library(stdlib), system call(syscalls), error handling(errno), re-enterency(reent), locale(locale), floating point computation (mathfp, math and common), 64 bit arthmetic operations(division and mod) functions of a compilation environement is also required to be implemented. Such a need for a customized run-time C library is supplied by customizing the multi platform portable *newlib* library. Newlib library is a source code distribution of a Redhat community. The distribution comes with the build scripts which is configurable for a supported platform.

The similar adaptation acrivities is needed to be carried out to port the implementation to the platforms other than intel 386 architecture. The Figure 3.3 summarizes the Build-Time Environment.

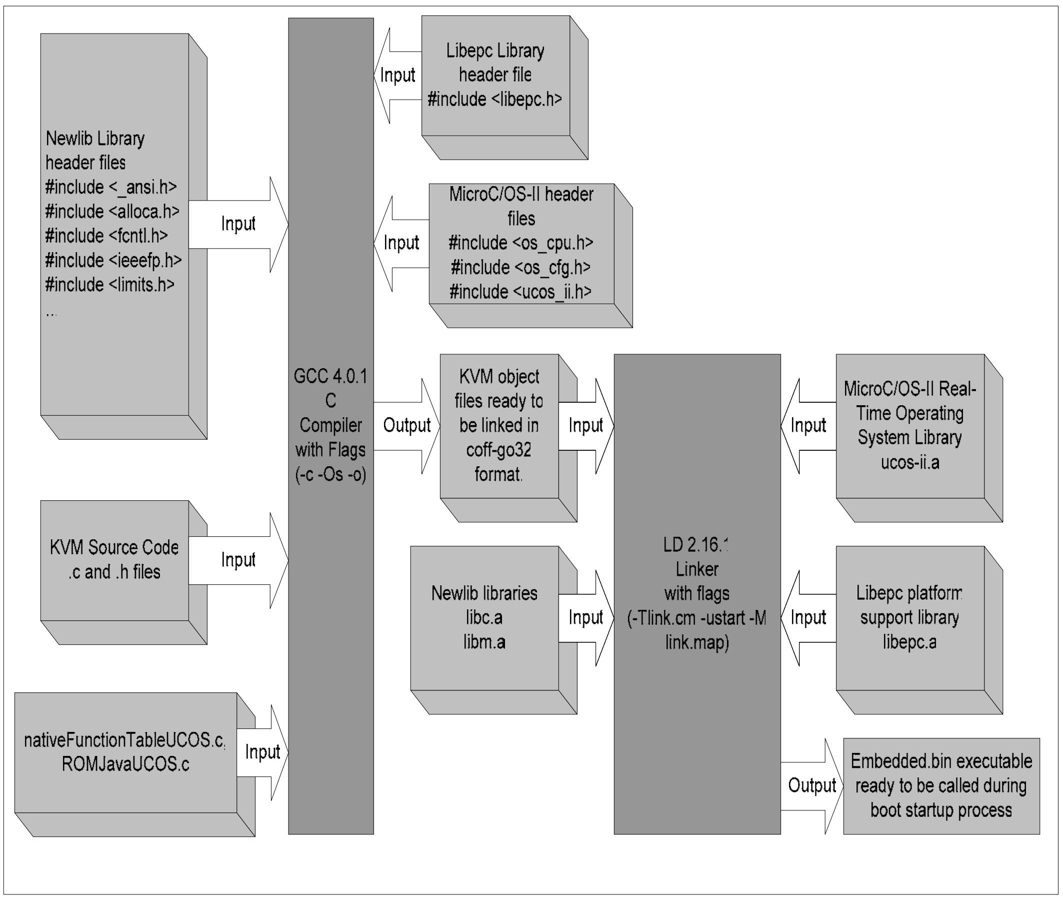

Newlib Library header files #include <_ansi.h> #include <alloca.h> #include <fcntl.h> #include <ieeefp.h> #include <limits.h> ..
Input
GCC 4.0.1 C Compiler with Flags (-c -Os -o)
Libepc Library header file #include <libepc.h>
Input
MicroC/OS-II header files #include <os_cpu.h> #include <os_cfg.h> #include <ucos_ii.h>
Input
Output
KVM object files ready to be linked in coff-go32 format.
Input
LD 2.16.1 Linker with flags (-Tlink.cm -ustart -M link.map)
Input
MicroC/OS-II Real-Time Operating System Library ucos-ii.a
KVM Source Code .c and .h files
Input
Newlib libraries libc.a libm.a
Input
Input
Libepc platform support library libepc.a
nativeFunctionTableUCOS.c, ROMJavaUCOS.c
Input
Output
Embedded.bin executable ready to be called during boot startup process

Figure 3.3: Build-Time environment.

3.4 Run-Time Environment

Run-Time environment describes the execution partitioning of the JVM core among all target environment. As mentioned previously, the JVM core makes use of the MicroC/OS-II system calls as much as it can to keep its platform independency property due its strict dependency to the MicroC/OS-II system calls. The Figure 3.4 summarizes the Run-Time Environment.

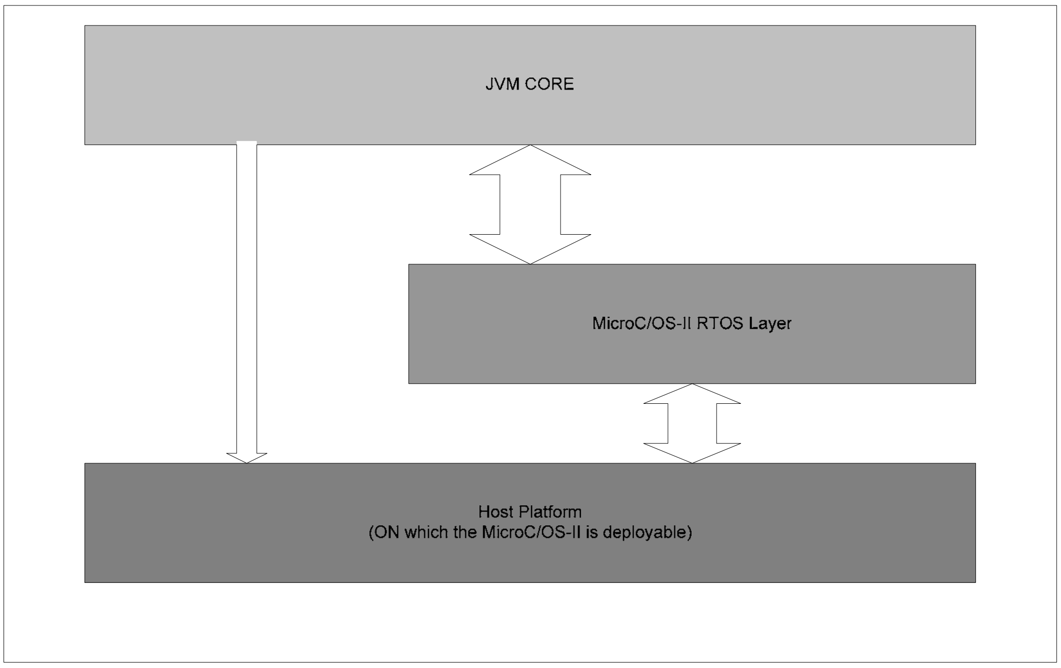


Figure 3.4: Run-Time environment.

CHAPTER 4

K VIRTUAL MACHINE DEPLOYMENT DETAILS

4.1 K Virtual Machine File Structure and Functional Responsibilities

The portable K Virtual Machine consists of a set file which are common for all platforms. And these files includes the data structures and functions which are required by the K virtual machine.

The files constituting the KVM C source code repository are listed in Table 4.1.

Table 4.1: K Virtual Machine Files and their Responsibilities.

File	Description
`StartJVM.c`	Virtual machine startup and command line argument reading.
`bytecodes.c`	The definition of Java bytecodes for the redesigned bytecode interpreter (since KVM 1.0.2).
`cache.h`, `cache.c`	Inline caching operations for speeding up method lookup and for supporting "fast" bytecodes.
`class.h`, `class.c`	Internal runtime data structures and operations for representing Java classes.
`events.h`, `events.c`	Event system implementation.
`execute.h`, `execute.c`	Interpreter execution macros and operations needed by the redesigned bytecode interpreter (since KVM 1.0.2).
`fields.h`, `fields.c`	Internal runtime data structures and operations for representing fields and methods.
`fp_math.h`, `fp_math.c`	High-level floating point function interface.
`frame.h`, `frame.c`	Stack frame and exception handling operations.
`garbage.h`, `garbage.c`, `collector.c`	Garbage collector and memory management.
`global.h`, `global.c`	Miscellaneous global variables and definitions.
`hashtable.h`, `hashtable.c`	Hash table implementation that is used internally by the virtual machine.

Table 4.1: Continues.

`interpret.h`, `interpret.c`	Bytecode interpreter. Note that starting from KVM 1.0.2 the actual interpreter code and bytecode definitions are located in other files (`bytecodes.c`, `execute.h`, `execute.c`).
`kni.h`, `kni.c`	K Native Interface (KNI) support.
`Loader.h`, `loader.c`	Class loader and class format checks required by the class file verifier.
`log.h`, `log.c`	Logging/diagnostic operations for debugging and profiling.
`long.h`	Special macros to handle 64-bit operations in a portable fashion.
`main.h`	Compilation options and system-wide default settings.
`messages.h`	Error and warning messages.
`Native.h`, `native.c`, `nativeCore.c`	Native function table operations and core native library functions.
`pool.h`, `pool.c`	Runtime data structures and operations for representing constant pools.
`profiling.h`, `profiling.c`	Data declarations and operations for profiling virtual machine execution.
`property.h`, `property.c`	Operations for accessing Java system properties.
`rom.h`	Macros needed by the ROMizer (JavaCodeCompact tool).
`runtime.h`	Function templates for certain machine-specific operations that need to defined for each KVM port.
`stackmap.c`	Stackmap operations that are used for supporting exact garbage collection.
`Thread.h`, `thread.c`	Internal runtime data structures and operations for multithreading and Java thread management.
`verifier.h`, `verifier.c`, `verifierUtil.h`, `verifierUtil.c`	Classfile verifier (see Chapter 13for details).

4.2 K Virtual Machine System Configuration Options

The following definitions allow controlling which components and features to include in target port.

```
#define IMPLEMENTS_FLOAT 1
```

Turns floating point support in KVM on or off. Its value should be ‘1’ in those implementations that are compliant with *CLDC Specification* version 1.1, and ‘0’ in those implementations that are compliant with *CLDC Specification* version 1.0. [6]

`#define PATH_SEPARATOR ':'`

Its value should be path separator character used in CLASSPATH. This definition is meaningful only when utilizing the default class loader for command line based systems.[7]
(Defined in `VmCommon/h/loader.h`.)

`#define ROMIZING 1`

Turns class prelinking/preloading (JavaCodeCompact) support on or off. If this option is turned on, KVM prelinks all the system classes directly in the virtual machine, speeding up application startup considerably.

`#define USE_JAM 0`

Includes or excludes the optional Java Application Manager (JAM) component in the virtual machine.

`#define ASYNCHRONOUS_NATIVE_FUNCTIONS 0`

This option instructs the KVM to use optional asynchronous native functions.

`#define USE_KNI 1`

This option was introduced in KVM 1.0.4. When enabled, the system will include some code that is needed by the K Native Interface (KNI).[8]

4.3 K Virtual Machine Memory Allocation Settings

The following definitions affect the amount of memory KVM allocates.

#define DEFAULTHEAPSIZE 256*1024

Its value shows the Java heap size that KVM allocates upon virtual machine startup. This value is commonly overridden from makefiles. Note that, starting from KVM 1.0.3, it is possible to override the heap size value from the command line (in those ports that support command line operation.) The heap size value must be a number that is divisible by four. The number must be in the range of 16k to 64 M.[7]

#define INLINECACHESIZE 128

Its value shows the the size of a special inline cache area that KVM reserves upon virtual machine startup if the ENABLEFASTBYTECODES option is turned on. The inline caching mechanism speeds up method lookups in the KVM by utilizing a technique popularized by Deutsch & Schiffman in the early 1980s. The size here is expressed as a number of inline cache entries (each entry requires 12-16 bytes depending on target platform.) [8]

#define STACKCHUNKSIZE 128

The execution stacks of Java threads inside the KVM grow and shrink automatically as necessary. This value defines the default size of a new stack frame chunk when a new stack chunk needs to be allocated. Reducing the default stack chunk size will make the creation of new Java threads less expensive, but will slow down the execution of the VM when running programs that require a lot of stack space (that is, programs that have a lot of nested method calls.) [7]

#define STRINGBUFFERSIZE 512

This option defines the size (in bytes) of a statically allocated area that the virtual machine uses internally in various string operations.

4.4 K Virtual Machine Garbage Collection Options

The following option turns on compacting garbage collection. Note that currently compaction cannot be used on those platforms that have segmented (noncontiguous) memory architecture.

#define ENABLE_HEAP_COMPACTION 1

The following option, if set to a non-zero value, causes a garbage collection to occur on every allocation. This makes it easier to find garbage collection problems.

Since this option makes the virtual machine run extremely slowly, the option should be turned off in production builds.

#define EXCESSIVE_GARBAGE_COLLECTION 0

4.5 K Virtual Machine Interpreter Execution Options

The following macros allow turning on and off certain features controlling interpreter execution. The default values for a production release are shown below.

#define ENABLEFASTBYTECODES 1

With this option runtime bytecode replacement and method inline caching is turned on or off. This option improves the performance of the virtual machine by about 10-20%, but increases the size of the virtual machine by a few kilobytes. Note that bytecode replacement cannot be performed on those target platforms in which bytecodes are stored in non-volatile memory such as ROM. [7]

#define VERIFYCONSTANTPOOLINTEGRITY 1

This option instructs the virtual machine to verify the types of constant pool entries at runtime when performing constant pool lookups. Reduces runtime performance slightly, but is generally recommended to be kept on for safety and security reasons. Additional definitions and interpreter macros:

#define BASETIMESLICE

The value of this variable determines the basic frequency (as a number of bytecodes executed) in which the virtual machine performs thread switching, event notification and other periodically needed operations. A smaller number reduces event handling and thread switching latency, but causes the interpreter to run more slowly.

#define DOUBLE_REMAINDER(x, y) fmod(x, y)

A compiler macro, defined in interpret.h, which is used to find the modulus of two floating point numbers.

#define SLEEP_UNTIL (wakeupTime)

This macro makes the virtual machine sleep until the current time (as indicated by the return value of the function CurrentTime_md ()) is greater than or equal to the wakeup time. The default implementation of SLEEP_UNTIL is a busy loop. Most ports should usually provide a more efficient implementation for battery conservation reasons. Refer to Section 12.4 “Battery power conservation” for further details.

4.6 K Virtual Machine Interpreter Execution Techniques

Since the release 1.0.2, KVM has an interpreter design that gives up to 15-30% better performance than KVM 1.0 without any loss of ANSI C portability.

The actual performance improvement percentage depends on the target platform and the capabilities of the C compiler that is used for compiling the KVM. The performance improvement is the result of the following four techniques that can be used independently of each other:

■ restructuring the interpreter code so that virtual machine registers will be placed into local C variables when the interpreter is running.

■ splitting uncommonly used Java bytecodes into a separate interpreter loop subroutine. This allows the C compiler to do a better job in optimizing the code for more frequently used bytecodes.

■ moving the test for Java thread rescheduling from the top of the interpreter loop to branch bytecodes. This reduces the overhead of the time slice counter that is used for controlling thread switching.

Padding out the bytecode space is also an alternative in order to allow the C compiler to produce better code for the main switch statement of the interpreter. These techniques do not depend on any compiler-specific features, and are therefore portable across a wide variety of C compilers. Each of the techniques and the corresponding macros are discussed in more detail below.

4.7 K Virtual Machine Java Level Debuging Options

The KVM 1.0.2 release introduced a new Java-level debugger interface that allows the KVM to be plugged into third party Java debugger environments and integrated development environments (IDEs) that supports the JDWP (Java Debug Wire Protocol) protocol. The macros in this subsection are related to the Java-level debugger options.

Note – It is important to notice that there is a fundamental difference between the debugging facilities intended for Java-level debugging and VM-level debugging. Java-level debugging facilities are related to the debugging of the Java programs that the KVM executes. VM-level debugging facilities are used for debugging the KVM itself at the native (C) code level.

#define ENABLE_JAVA_DEBUGGER 0

This option inserts a large amount of debugger support code that is needed for plugging KVM into a third-party Java debugger or integrated development environment such as Forte or Borland JBuilder.

4.8 K Virtual Machine Level Debuging and Tracing Options

KVM provides a large number of debugging and tracing facilities that can be used for inspecting the behavior of the KVM itself at the native (C) code level. These facilities can be extremely helpful during porting efforts.

All the VM-level debugging and tracing options should be turned off in a production release.

4.8.1 Including and Excluding Debuging Code

#define INCLUDEDEBUGCODE 0

This option makes possible to include a large amount of debugging and logging code that is useful when porting the virtual machine onto a new platform. This option should be turned off in production builds.

#define ENABLEPROFILING 0

This option is used to turn on or off certain profiling features that allow monitoring virtual machine execution and get execution statistics. Turning this option on slows down the virtual machine execution speed considerably. This option should be turned off in production builds.

4.8.2 Tracing Options

In KVM 1.0, all the tracing options were compilation flags that could be changed only by recompiling the virtual machine. In KVM 1.0.2, all these tracing options were changed into global variables that can be controlled from the command line.

This makes it much easier to turn individual tracing options on and off. These global variables (and command line switches) are available only if the virtual machine has been compiled with the INCLUDEDEBUGCODE mode turned on.

These options can be controlled directly by changing their default values in file VmCommon/src/global.c, or by defining a graphical user interface that sets and resets these options.

Additionally, whether the tracing messages printed out terse or more verbose can be controlled by modifying the following option:

```
#define TERSE_MESSAGES 0
```

KVM also contains a stack trace printing facility that can be turned on to help debugging of exceptions and errors in more detail (at the cost of some additional memory footprint). By default, this mode is turned on automatically when the INCLUDEDEBUGCODE flag is turned on.

```
#define PRINT_BACKTRACE 0
```

4.9 K Virtual Machine Error Handling Macros

The interpreter uses the internal error handling macros shown in Figure 4.1. If there is a call to the macro THROW (error), anywhere inside the “normal code,” the VM jumps immediately to error handling code. Uses of this macro can be nested, either lexically or dynamically. The THROW jumps to the innermost CATCH error handling code. (The various TRY, THROW, and CATCH macros are defined in VmCommon/h/global.h.)

```
TRY {
normal code
} CATCH (error) {
error handling code
} END_CATCH
always continue here
```

Figure 4.1: K Virtual Machine error handling macros

By default, this behavior is emulated using setjmp and longjmp. These macros have been illustrated in Figure 4.2.

```
VM_START {
normal VM code
} VM_FINISH (value) {
code to execute before VM shuts down
} VM_END_FINISH
```

Figure 4.2: K Virtual Machine Shutdown Macro

Rather than calling the normal C exit function, the proper way to exit from the VM is to call macro VM_EXIT (value). Calling this macro will cause the control of the VM to be immediately transferred to the code that follows the VM_FINISH (value) macro. The value to be passed to this code typically represents the exit code that the VM will return when it shuts down.

4.10 K Virtual Machine Startup Conventions

Virtual machine startup practices can vary significantly in different KVM ports. By default, KVM supports regular command line based Java virtual machine startup, but the virtual machine can easily be modified for those environments in which command line based startup is not desired.

4.10.1 Command Line Startup

This subsection describes the virtual machine startup conventions when launching KVM from the command line. The file VmExtra/src/main.c provides a default implementation of main (). The virtual machine is called from the command line as follows:

```
kvm [option]* className [arg]*
where each option is one of
-version
-classpath <list of directories>
-heapsize <heap size parameter>
```

The required className argument specifies the class whose method static main (String argv []) is to be called. All arguments beyond the class name are uninterpreted strings that are made into a single String [] object and passed as the single argument to the main method.

The -classpath option allows the user to define the directories from which the KVM reads the class files. The parameter <list of directories> is a single string in which the directories are separated by the PATH_SEPARATOR character.

The value of the PATH_SEPARATOR character is typically ';' on Windows platforms, and ':' on UNIX platforms.

The -heapsize option (introduced in KVM 1.0.3) allows the user to manually set the Java heap size that KVM allocates upon virtual machine startup. The heap size can range from 16 kilobytes to 64 megabytes. The heap size can be specified either in bytes (such as 32768), kilobytes (such as 32k or 32K), or megabytes (such as 1m or 1M). Note that when the heap size is defined in bytes, the KVM automatically rounds up the heap size number to the next number that is divisible by four.

Additionally, if the virtual machine has been compiled with the INCLUDEDEBUGCODE mode turned on, the tracing options are also available.

When the Java-level debugging interface is in use, additional command line options are available to control the debugger. The default implementation of main (int argc, char **argv) calls the function StartJVM () with an argv in which all of the options have been removed and an argc that has been decremented appropriately.

4.10.2 Romizied Virtual Machine Startup

In this port of the KVM for MicroC/OS-II since there is not any command line capability. The class files embedded in to the virtual machine, including the entry point of the java application, is invoked after the initialization process is completed internally in the KVM_Start function of the StartJVM.c file. This configuration of the KVM also requires specifying the command line options from the source code by modifying the preprocessor directives in the global.h or main.h files.

4.11 K Virtual Machine 64 Bit Support

It is not required the compiler to support 64-bit arithmetic. However, having a 64-bit capable compiler makes porting much easier.

4.11.1 Setup

If the compiler supports 64-bit integers, the types long64 and ulong64 should be defined in one of the platform-dependent include files. The meaning of these two types is shown below in Table 4.2.

Table 4.2: K Virtual Machine 64 Bit Definition Description.

Type	**Description**
long64	A signed 64-bit integer.
ulong64	An unsigned 64-bit integer.

It should be considered setting one of the two compiler constants BIG_ENDIAN or LITTLE_ENDIAN to a non-zero value. This is only required while using the Java Code Compactor, but KVM can produce better code if it knows the endianness of target machine.

If the compiler does not support 64-bit integers, 1 preprocessor constant COMPILER_SUPPORTS_LONG must be set to zero. It must be defined exactly as one of BIG_ENDIAN or LITTLE_ENDIAN2 to have a non-zero value.

The types long64 and ulong64 are defined to be a structure consisting of two fields, each an unsigned long word, named high and low. The high field is first if target machine is big endian; the low field is first if target machine is little endian.

The functions shown in Table 4.2 must be defined. If target platform supports floating point, the functions shown in Table 4.3 must also be defined. Any of these functions can be implemented as a macro instead.

Table 4.3: K Virtual Machine 64 Function Correspondance.

Function or Constant		**Java equivalent**
long64	Ll_mul(long64 a, long64 b);	a * b
long64	Ll_div(long64 a, long64 b);	a / b
long64	Ll_rem(long64 a, long64 b);	a % b
long64	Ll_shl(long64 a, int b);	a << b
long64	Ll_shr(long64 a, int b);	a >> b
long64	Ll_ushr(long64 a, int b);	a >>> b

Table 4.4: K Virtual Machine 64 bit type correspondance.

Function or Constant		**Java equivalent**
long64	Float2ll(float f);	(long)f
long64	double2ll(double d);	(long)d
float	ll2float(long64 a);	(float)a
double	ll2double(long64 a);	(double)a

4.12 K Virtual Machine Floating-Point Support

For CLDC 1.1 compliant implementations, the floating-point functionality is always enabled by default.[8] It can be disabled by changing the IMPLEMENTS_FLOAT flag in main.h. The majority of the support needed in the virtual machine for implementing floating-point is done to the Java bytecodes defined in bytecodes.c. The specific modifications needed are described in the sections below.

4.12.1 Floating-Point Bytecodes Implementation

The file bytecodes.c represents one of the major components that must be changed to support floating-point. This file contains Java bytecodes executed by the KVM interpreter. Many of the modifications involve checking for NaNs.

Among the bytecodes that require modifications are D2I, D2L, F2I, and F2L. The modifications and checks for NaNs are described in Section 10.4

"Porting." The x86 specific changes are implemented in fp_bytecodes.c (located in directory kvm/VmExtra/src/fp). Specific details of the changes are also documented with comments in that file.[8]

Table 4.5 lists the trigonometric functions that are now implemented in the KVM for floating-point support. Listed with each function are the corresponding file(s) in which the function is implemented.

Table 4.5: K Virtual Machine Supported TrigonometricFunctions.

Function	**File(s)**
sin	k_sin.c s_sin.c
cos	k_cos.c s_cos.c
tan	k_tan.c s_tan.c
sqrt	e_sqrt.c w_sqrt.c
ceil	s_ceil.c
floor	s_floor.c
abs	s_fabs.c

The implementation of the trigonometric functions is taken directly from the JDK1.3.1 sources with no changes except to the function names. The trigonometric files are specified in directory kvm/VmExtra/src/fp.

4.12.2 Floating-Point Support Porting to x86 Architecture

The traditional x87 FPU is fully IEEE 754 compliant. However, the IEEE 754 standard explicitly allows rounding to reduced precision, but greater exponent range, which does not always match the floating-point model used in the Java language and the JVM.[8] Therefore, additional work is needed to implement floating point.

Additionally, the P4 processor contains the SSE2 instruction set extension, which is another IEEE 754 compliant implementation. However, SSE2 is more amenable to Java's semantics.[8]

To implement floating-point for the x86 platforms, checks involving NaNs are needed for the following Java bytecodes: FCMPL, FCMPG, DCMPL, DCMPG, FREM, and DREM. These bytecodes needed additional checks to behave as mentioned in the Java™ Virtual Machine Specification. The Java™

Virtual Machine Specification describes what each of these bytecodes should do or return when a NaN value is encountered.[8]

The file fp_bytecodes.c under kvm/VmExtra/src/fp contains the x86-specific implementation for the floating-point bytecodes. Each function in this file implements an algorithm for a specific floating-point bytecode that needs modification. Each of these bytecodes checks the value that is on the stack to see if it is a NaN. If a NaN value is encountered, it is handled as a special case according to the Java™ Virtual Machine Specification. These functions are executed only if the variable PROCESSOR_ARCHITECTURE_X86 is set in the platform-specific header file machine_md.h.[8]

4.12.3 Floating-Point strict-fp implementation on x86 Architecture

The x86 is designed to operate on 80-bit double extended floating-point values rather than the 64-bit and 32-bit double and float values used in the Java programming language.[8] The x86 can be made to round to float or double precision. Unfortunately, this rounding does not exactly emulate the pure float and double called for by Java, since an extended exponent range is available. The extended exponent range means the overflow and underflow thresholds are different than for pure float and double.

To implement strictfp, the bytecodes DMUL and DDIV must be changed. The problem is, while doing these operations on subnormal numbers (very small IEEE 754 values with less precision than normal numbers) rounding occurs, producing an incorrect result. In addition, double-rounding can occur if the obvious code generation algorithm is used. The solution is to implement the following algorithms for DMUL and DDIV.

Multiply (DMUL)

■ load multiplier

■ scale multiplier by multiplying multiplier by 2-15360

■ load multiplicand

■ multiply scaled multiplier by multiplicand

■ rescale product by 215360

■ store rescaled product

■ reload stored rescaled product

Divide (DDIV)

For strictfp floating-point on x86, the initial scaled quotient must be smaller than the actual quotient for the rounding to work properly. Thus, the algorithm is:

■ load dividend

■ load divisor

■ compute initial_quotient by either:

■ initial_quotient = (2-15360 × dividend)/divisor

■ initial_quotient = dividend/(divisor × 215360)

■ rescale initial_quotient to get the right significant bits. Compute:

quotient = initial_quotient × 215360

■ store rescaled quotient

■ reload stored rescaled quotient

The bytecodes for FADD and FSUB did not need to be changed since if those operations have subnormal results, the results are exact (that is, no rounding occurs). [8]

CHAPTER 5

JAVA APPLICATION DEPLOYMENT SUPPORT TOOLS

5.1 Class File Preverifier

The class file verifier supported by Java 2 Standard Edition (J2SE) is not suitable for small, resource-constrained devices. The J2SE verifier requires a minimum of 50 kB binary code space and at least 30-100 kB of dynamic RAM at run time. In addition, the CPU power needed to perform the iterative dataflow algorithm in the standard JDK verifier can be substantial.

Preverification is designed and implemented as new, two-phase classes file verifier that is significantly smaller than the existing J2SE verifier. The runtime part of the new verifier requires about 15 kB of Intel x86 binary code and only a few hundred bytes of dynamic RAM at run time for typical class files. The runtime verifier performs a linear scan of the byte code, without the need of a costly iterative dataflow algorithm. The new verifier is especially suitable for KVM, a small-footprint Java virtual machine for resource-constrained devices. [7]

The new class file verifier operates in two phases, as illustrated in Figure 5.1:

■ First, Java class files have to be run through a special preverifier tool in order to augment the class files with additional attributes to speed up runtime verification. The preverification phase is typically performed on a development workstation, where the application developer writes and compiles the applications.

■ At runtime, the runtime verifier component in the KVM utilizes the additional attributes generated by the preverifier to perform the actual class file verification efficiently.

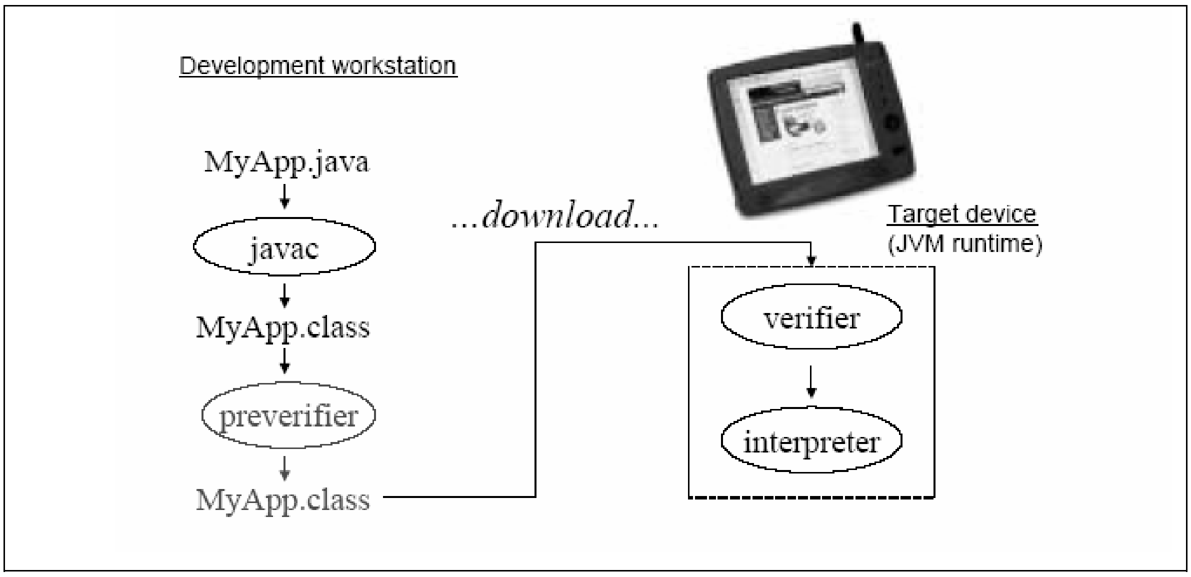


Figure 5.1: The phases of running an application on the JVM.

5.1.1 Using Class File Preverifier

The preverification phase is usually performed at application development time. For example, if a preverifier weren't used, Foo.java would be typically compiled with javac like this:

```
javac -classpath kvm/classes Foo.java
```

However, when using the preverifier, output of javac would be placed in a separate directory and then the resulting class files would be transformed using the preverifier. For example:

```
javac -classpath kvm/classes -d mydir Foo.java
preverify -classpath kvm/classes -d . mydir
```

The above preverifier command transforms all class files under mydir/ and places the transformed class files in the current directory (as specified by the –d option).

5.1.2 Class File Preverifier Options

More generally, the preverifier is invoked as follows:

```
preverify <options> <input files>
```

Preverifier options and accepted input file formats are explained in more detail below. The preverifier accepts a number of arguments and options.

-classpath <directories> | <JAR files>

■ This option indicates the directories or JAR file(s) in which the KVM/CLDC Java library classes are located. The directory separator is platform-specific. On Solaris a colon is used. On Win32 a semicolon is used. The JAR file specified must be in a valid Java Archive format and must end with.jar, .JAR, .zip or .ZIP suffix.

-d <directory>

■ The directory in which output classes will be written. The default output directory is ./output.

-cldc1.0

■ This option checks for the existence of language features prohibited by CLDC 1.0 (native methods, floating point, and finalizers).

-nofinalize

■ This option checks for the use of finalizers in application classes. When this option is specified, an error is reported if finalizers are detected in any of the input files.

-nonative

■ This option checks for the use of native methods in application classes. When this option is specified, an error is reported if native methods are detected in any of the input files.

-nofp

■ This option checks for the use of floating point operations in application classes. When this option is specified, an error is reported if floating point operations are detected in any of the input files.

@<filename>

■ The name of a text file from which command line arguments will be read.

Note – When the command line arguments are read from a file, parameters must all be specified on a single line and the parameters to the -classpath and –d options must be enclosed within double quotes. When the corresponding options are used from the command line, quotes are not required (unless the directory/file name parameter contains spaces.)

5.2 Java Code Compact (JCC)

This utility allows Java classes to be linked directly in the virtual machine, reducing VM startup time considerably.

At the implementation level, the JavaCodeCompact utility combines Java class files and produces a C file that can be compiled and linked with the Java virtual machine.

In conventional class loading, javac is used to compile Java source files into Java class files. These class files are loaded into a Java system, either individually, or as part of a jar archive file. Upon demand, the class loading mechanism resolves references to other class definitions.

JavaCodeCompact provides an alternative means of program linking and symbol resolution, one that provides a less-flexible model of program building, but which helps reduce the VM's bandwidth and memory requirements.

JavaCodeCompact can:

■ combine multiple input files

■ determine an object instance's layout and size

■ load only designated class members, discarding others.

5.2.1 Java Code Compact (JCC) Options

JavaCodeCompact accepts a large number of arguments and options. Only the options currently supported by KVM are given below.

■ filename

This option designates the name of a file to be used as input, the contents of which should be included in the output. File names with a .class suffix are read as single class files. File names with .jar or .zip suffixes are read as Zip files. Class files contained as elements of these files are read. Other elements are silently ignored.

■ -o output filename

This option designates the name of the output file to be produced. In the absence of this option, a file is produced with the name ROMjava.c.

■ -nq

This option prevents JavaCodeCompact from converting the byte codes into their "quickened" form. This option is currently required by KVM.

■ -classpath path

Specifies the path JavaCodeCompact uses to look up classes. Directories and zip files are separated by the delimiting character defined by the Java constant java.io.File.pathSeparatorChar. This character is usually a colon on the Unix platform, and a semicolon on the Windows platform.

Multiple classpath options are cumulative, and are searched left-to-right. This option is used in conjunction with the -c cumulative-linking option, and with the -memberlist selective-linking option.

■ -memberlist filename

This option provides selective loading as directed by the indicated file. This file is an ASCII file, as produced by JavaFilter, containing the names of classes and class members.

■ -v

This option turns on the verbosity of the linking process. It is cumulative. Currently up to three levels of verbosity are understood. It is only of interest as a debugging aid.

■ -arch Architecture

Specify the architecture for which romized image is generated. At this time," KVM" must be specified as the architecture.

5.2.2 Executing Java Code Compact (JCC)

The JavaCodeCompact utility is used to build the platform-specific file nativeFunctionTablePlatform.c, which contains tables necessary for calling native methods.

This file must be built even if the ability of JavaCodeCompact to pre-load classes is not used. If system classes are not ROMized (in other words, all system classes are loaded dynamically), Step 4 may be skiped.

The simplest method for using the JavaCodeCompact utility is to either use the Makefile provided or to modify it for the target platform. The following lists the steps that the Makefile performs:

1. Compile all the .java files in the api/src directory. The resulting class files are verified and merged into a single zip file classes.zip. This zip file is copied to the tools/jcc directory.
2. Compile the sources for JCC.
3. Copy classes.zip to classesPlatform.zip. Remove from this platform dependent zip file any classes or packages that should not be used on target platform.

4. Execute host system's equivalent of the following command in the jcc directory:

```
env CLASSPATH=classes \
JavaCodeCompact -nq -arch KVM \ -o ROMjavaPlatform.c classesPlatform.zip
```

The "env CLASSPATH-classes" sets an environment variable indicating that the code for executing JavaCodeCompact can be found in the subdirectory called classes. Next on the command line is the name of the class whose main method is to be executed (JavaCodeCompact), and the arguments to that method.

5. Execute host system's equivalent of the following command in the jcc directory:

```
env CLASSPATH=classes \
JavaCodeCompact -nq -arch KVM_Native -o nativeFunctionTablePlatform.c
classesPlatform.zip
```

This command creates the file containing the native function tables necessary to link native methods to the corresponding C code.

5.2.3 Java Code Compact (JCC) Limitations

The current implementation of JavaCodeCompact requires that the class files that compacted constitute a "transitive closure." If class A is compacted, and class A's constant pool references class B, then class B must also be included as part of the compaction.

Class A includes Class B in its constant pool if any of the following conditions are true:

■ Class A is a direct subclass of class B, or class A directly implements class B.

■ Class A creates an instance of class B, or an array of class B.

■ Class A calls a method that is defined in class B.

■ Class A checks to see if an object is an instance of type B, or casts an object to type B.

Note that the following do not cause class B to be included in class A's constant pool. Under certain circumstances, it may be possible to compact A without also compacting B.

■ Class A has an instance variable of type B

■ Class A has a method whose argument or return type includes type B in its signature.

■ Class A creates an instance of class B using the Class.forName() method.

JavaCodeCompact will fail and give an error message if a class file that it requires is not included.

CHAPTER 6

VIRTUAL MACHINE DEPLOYMENT SUPPORT LIBRARIES

6.1 Overview

Even if the MicroC/OS-II is a platform-independent real-time operating system, it requires some minor platform dependent functions for its proper execution in the target environment.[9]

It is also a need of the KVM code to be supported by some of the library functions including string operations (like strcpy, strcmp, strlen) memory operations (like memmove, memcpy, malloc, free) and also some standard input output functions(like fprintf and sprintf).

The functions required by MicroC/OS-II and KVM code either are unfortunately unlikely to be supported by the development environments (DJGPP) standard C Runtime library "libc.a". The appropriate library for such an environment should satisfy the following requirements:

1. The library should include a crt0.c (C Run-Time Environment Initializer) file which implements handling some booting up procedures.
2. The library should not include any function depending on operating system or ROM-BIOS.
3. The library should not include any object reference, which requires for its execution, like some definitions in the memory region of embedded application, in order to decrease applications complexity and memory footprint.

Since these library requirements are unlikely to be supported by any development environment, it is choosen to make use of two different libraries for the satisfaction of these requirements.

One of these libraries is the Libepc library which includes specialized functions for the required booting up procedure as well as the primary input/output functions for supporting the rest.[10]

The other library is a kind of customizable C run-time library which includes all the headers, types, macros and functions of the standard C run-time library complying with the ANSI and POSIX specifications. This library is called the Newlib (developed and distributed by a commuity of RedHat known as Cygnus) and designed to be used for the different target the platforms after completing the configuration customization steps declared in its documentations.[11]

In order to improve re-usability and not to modify the KVM source the hiearchy of library dependencies is planned to be as in the Figure 6.1.1.

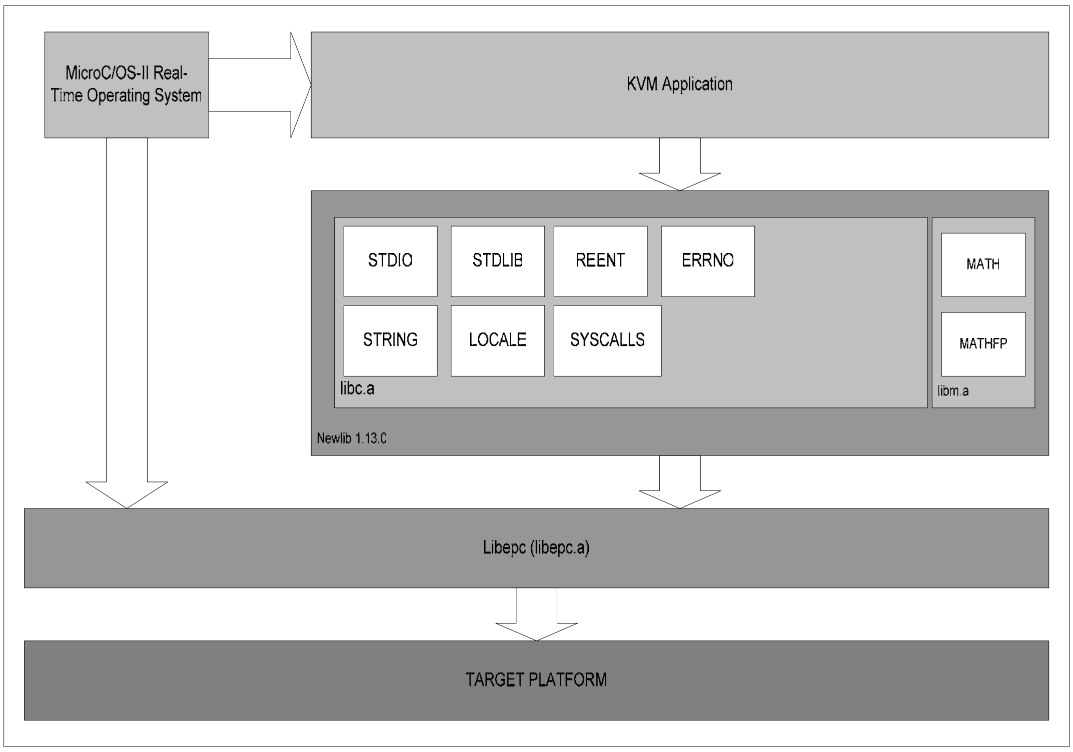


Figure 6.1: The Library use and dependency

6.2 Use of Newlib Library

Newlib is a freely-available C runtime library with a portable and flexible architecture that makes it suitable for use in resource-constrained embedded systems. Newlib is an actively supported and mature product, and is the preferred choice for GNU based embedded environments.

This section discuss some of newlib's functionality and portabilty features, and provides a strategy on how to integrate newlib into an embedded environment based on MicroC/OS-II Real Time Operating System (RTOS).

6.2.1 Newlib Standard Input/Output Features

Some of newlib's functionality provides useful enhancements to a "typical" embedded setting (whatever that is), while others allow newlib to be about as POSIX-like as a compact C runtime setting can be. These capabilities can be a big help when porting desktop-tested applications to an embedded environment.[11]

Newlib contains a complete implementation of the C standard printf() and family. By the implementation's own admission, "this code is large and complicated" 3, but essential for systems that need full ANSI C input and output support, including capabilities for representing and parsing floating point numbers.

Many embedded systems do not use floating point math, however, and great pains are taken in most embedded runtime libraries to cull this code-bloating functionality whenever possible. Newlib approaches this problem in two ways: a FLOATING_POINT macro that allows selective disabling of floating point support in each of the library functions that can offer it, and an iprintf() function that only knows how to display integer objects.

If an embedded system needs floating point support in only a few of the standard input and output functions, then newlib can be rebuilt to exclude floating point from places where it isn't needed. Floating point can be omited for everything except scanf(), for example, by either undefining the FLOATING_POINT macro everywhere except in the scanf.c source file, or by modifying newlib's Makefile to do the same thing.

For situations where only integer output is required, newlib provides the iprintf() function: a version of the printf() function built with the FLOATING_POINT macro undefined. It behaves exactly like printf(), except that it does not understand the %f, %F, %g, and %G type specifiers, and therefore has a much smaller code footprint.

Newlib's standard input and output facilities are surprisingly complete, even beyond the printf() et al implementations. The complete C file API is also provided, complete with read and write buffering, seeking, and stream flushing capabilities. Variations like sprintf(), fprintf() and vfprintf() (takes va_list arguments) are also included, which makes a newlib environment look strikingly similar to one expected to be seen in a more C language programming environment. An unfortunate limitation of newlib's stdio library is that it requires at least a minimal malloc() for complete and proper operation. Fortunately, newlib includes a pretty good dynamic memory allocator that is straightforward to set up and use. One can also build a malloc() based on a fixed size memory block allocator, to eliminate fragmentation worries in systems where this is a concern. If use of stdio is constrainted to just iprintf(), the malloc() is not needed.[11]

6.2.2 Newlib Libm Support

Newlib includes a complete IEEE math library called libm. In addition to offering the standard math functions like exp(), sin() and pow(), this library also provides matherr(): a modifiable math error handler invoked whenever a serious math-related error like an underflow or loss of precision is detected. By customizing this function, these situations can be handled in whatever way is appropriate for the target application.[12]

Libm also includes functions that take float parameters, instead of double. These extensions are named after their full precision equivalents, i.e. sinf() is the single precision version of the sin() function. The reduced precision functions have a considerable speed advantage over their IEEE-compliant double precision counterparts, which can put some floating point operations within reach of hardware that is too weak for full double precision computations.

6.2.3 Newlib Reentrancy Support

Newlib's C and math libraries are reentrant when properly integrated into a multithreaded environment. The implementation is not obvious at first glance, so the next paragraphs describe how it works. Once the details are known, it will be clear how to set it up properly in target system.[11]

6.2.3.1 Making errno Reentrant

Newlib encloses errno and several related values into a structure of type struct _reent, and redefines the symbol errno as a macro that references a global _reent* pointer named __impure_ptr. As a result, when a statement refers to the value of errno, it is actually doing an indirect structure lookup that resolves to the errno field in a data structure.

The code in Figure 6.2 describes in general how errno is modified under newlib. The code in Figure 6.3 is a common example of how to use errno in an ANSI C environment; because the reimplementation of errno is transparent to the application, this code works without modification under newlib. [11]

```
#define errno (*__errno())
extern int *__errno _PARAMS ((void));
#define _REENT _impure_ptr
#define _REENT_INIT(var) \
{ 0, &var.__sf[0], &var.__sf[1], &var.__sf[2], 0, "", 0, "C", \
0, NULL, NULL, 0, NULL, NULL, 0, NULL, { {0, NULL, "", \
{ 0,0,0,0,0,0,0,0}, 0, 1} } }
static struct _reent impure_data = _REENT_INIT (impure_data);
struct _reent * _impure_ptr = &impure_data;
int * __errno ()
{
return &_REENT->_errno;
}
```

Figure 6.2: How errno is modified under newlib.

```
fp = fopen( "myfile.txt", "rw" );
if( fp == NULL ) {
switch( errno ) {
case EACCES:
/* we don’t have permissions */
...
```

Figure 6.3: Using errno in an ANSI C environment.

6.2.3.2 Managing _reent Structures

Newlib declares one _reent structure and aims _impure_ptr at it during initialization, so everything starts out correct for situations where only one thread of execution will be in the library at a time. To provide the capability to have multiple library processing contexts, allocate multiple _reent structures, and move _impure_ptr between them during context switches.

The _reent structure also contains fields for the standard input (stdin), standard output (stdout), and standard error (stderr) descriptors. This allows each task to define its own set of streams for reading and writing data: tasks A and B could both use printf() simultaneously, with each task’s output going to different locations.[11]

6.2.3.3 Reentrancy In Memory Management

In order to permit multiple processing contexts in newlib’s malloc() implementation, __malloc_lock() and __malloc_unlock() functions must also be provided to protect the memory pool from corruption during simultaneous allocations. If an RTOS’s reentrant memory pool implementation is used for dynamic memory allocation, however, this heap protection is unnecessary--- the RTOS protects the heap itself.

6.2.3.4 Building Newlib

Building newlib for a supported target is a straightforward process that follows the conventions adhered to by most open source and Free Software projects.

The build process starts with the use of the commands shown in Figure 6.4. When the commands completed, the build process will have produced the files libc.a, libg.a (a debugging-enabled libc), and libm.a, in the directory /usr/local/<target-name>. If the target specified has several variants, the build process will produce multiple files, each with compilation settings specific to each variant. After linking one or more of these files with the target application, and it will provide a free C runtime environment.

Note - that if a "—prefix" option is provided when building the GNU cross compiler, then the same --prefix must be provided here for newlib. The default value for --prefix is /usr/local/.

```
$ tar xzvf newlib-1.9.0.tar.gz
$ mkdir build-newlib && cd build-newlib
$ ../newlib-1.9.0/configure --target=$TARGET --prefix=$PREFIX
$ make all install info install-info
```

Figure 6.4: Building newlib

Newlib's build process produces documentation, in the files libc.info and libm.info. By default these files go into /usr/local/info5, and they can be browsed using info, a documentation browser included with most Linux distributions.[11] Just change to the directory containing these files, and type:

```
$ info -f ./libc.info
```

Newlib's configuration script supports several options, not the least of which are the definition of the target system (what CPU and OS the library will run under), and where to put the files generated during the build.

The following are some of the --target specifications that newlib supports.

- m68k-coff
- h8300-coff
- sh-elf
- z8k-coff
- mn10300-elf
- i386-elf
- i386-coff

6.2.4 Newlib Configuration Points

Newlib's source code has a few configuration points, and unneeded stubs are choosen to be eliminated, to optimize for code size instead of speed, or to remove floating point support.[11]

To modify a configuration point, change its value in the Makefile generated by the configure command, before make command is typed. Look for the variable called CFLAGS_FOR_TARGET, and add flags there like:

-DINTEGER_ONLY
to build an integer-only library, or

-DPREFER_SIZE_OVER_SPEED
to enable a few small changes that reduce library code size.

You can also adjust the value of the CFLAGS setting, to affect the way the library is compiled. For example, if GNU C compiler is choosen the then use:

-Os (instead of -O2)
to tell it to optimize for code size over performance, or:

-O3 (instead of -O2)
to tell the compiler to optimize for raw performance over everything else.

Add:
-fomit-frame-pointer

Telling the compiler to not build stack frames for functions in the library that do not need them, which saves some space and boosts library performance.[11]

6.2.5 Porting Newlib to MicroC/OS-II

All of newlib's functionality sits on top an integration layer of seventeen stubs of code that supply capabilities that newlib cannot provide itself: low-level filesystem access, requests to enlarge its memory heap, getting the time of day, and various types of context management like process forking and killing. Newlib supplies templates for each of these stubs, which either return "not implemented", or fail silently.

The requirements for each stub are fully documented in newlib's libc.info file, in the section called Syscalls. The key to a successfully ported newlib is providing stubs that bridge the gap between the functionality newlib needs, and what the target system can provide.

6.2.5.1 _fork_r

Newlib calls upon this stub to do the work for the fork() system call, which in POSIX environments is used to create a clone of the current processing context. The semantics of the conventional fork() do not coexist peacefully with MicroC/OS-II's way of managing task creation and identification.

In fact, trying to implement fork() in MicroC/OS-II is probably a bad idea, because it raises task priority and synchronization issues that MicroC/OS-II already addresses quite well on its own. So, this stub is choosen to be left essentially unimplemented. The code is in Figure 6.5.

```
İnt
_fork_r ( struct _reent *ptr )
{
/* return "not supported" */
ptr->errno = ENOTSUP;
return -1;
}
```

Figure 6.5: Newlib _fork_r stub.

Take this approach for several other context management-related stubs, including _execve, _kill, _wait_r and _getpid_r.

6.2.5.2 _write_r & _read_r

These stubs are a bit more interesting to implement, because MicroC/OS-II does not provide any type of device driver or filesystem model. Newlib calls _write_r any time it requires to send data to a device, be it due to a write() call, printf() or fprintf(), or anything similar. The _reent parameter provides a place for the stub to communicate errors should they occur, and the file descriptor parameter, fd, tells the stub which device is being addressed. The remaining arguments supply a source data buffer and number of bytes to write.

The stub doesn’t need to write all the bytes that newlib asks it to, but if it doesn’t then newlib will simply invoke it again with the remaining data. So if the return value never eventually equals the number of bytes requested, newlib will misbehave.

Furthermore, newlib doesn’t call open() for file descriptors 0, 1, or 2, which means that the _write_r call is the first activity the stub will see on those streams. Stream zero is defined by convention to be the "standard input" stream, which newlib uses for the getc() and similar functions that don’t otherwise specify an input stream. Stream number one is "standard output", the destination of printf() and puts(). Stream number two refers to standard error", the destination conventionally reserved for messages of grave importance.

Implementation of _write_r, starts by defining a simple "device operations" table, with function pointers for all the kinds of activities a stream-like device driver to support is expected as well. The structure for this table is shown in Figure 6.6.

```
Typedef struct {
const char *name;
int (*open_r )( struct _reent *r, const char *path,
int flags, int mode );
int (*close_r )( struct _reent *r, int fd );
long (*write_r )( struct _reent *r, int fd,
const char *ptr, int len );
long (*read_r )( struct _reent *r, int fd,
char *ptr, int len );
} devoptab_t;
```

Figure 6.6: Newlib devopttab_t structure.

Each device driver will supply its own operations table:

```
/* devoptab for an example stream device called "com1" */
const devoptab_t devoptab_com1 = { "com1",
com1_open_r,
com1_close_r,
com1_write_r,
com1_read_r };
```

Figure 6.7: Newlib primitive stream device declaration.

Each driver provides its own implementations of open_r, close_r, write_r and read_r functions to handle device initialization and shutdown, and data movement to and from the physical device hardware. In the sample declaration above, these functions are named com1_open_r(), etc.

Somewhere in the application, all the devoptab_t declarations gathered up into on place, sorted by file descriptor:

```
const devoptab_t *devoptab_list[] = {
&dotab_com1, /* standard input */
&dotab_com1, /* standard output */
&dotab_com1, /* standard error */
&dotab_com2, /* another device */
... , /* and so on... */
0 /* terminates the list */
};
```
Figure 6.8: Newlib primitive device driver definitions.

With all of that done, the _write_r stub is straightforward to implement because all it has to do is map a file descriptor to the proper set of device operations. Figure 6.9 shows how to do this.

```
Long
_write_r ( struct _reent *ptr,
int fd,
const void *buf,
size_t cnt )
{
return devoptab_list[fd].write_r( ptr, fd, buf, cnt );
}
```
Figure 6.9: Newlib _write_r stub.

The _read_r stub is identical, except that it calls the driver’s read_r method. The devoptab_t strategy leaves device drivers free to use whatever MicroC/OS-II services they need in order to manage reentrancy, mutual exclusion and performance issues.

6.2.5.3 _open_r

This stub translates a device or file "name" to a file descriptor. With the exception of the standard input, standard output and standard error devices, this function can also be used to provide advance notice of an impending write() or read() request. Continuing with our approach utilizing device operation tables, the _open_r stub can be very simple. The code is shown in Figure 6.10.

```
İnt
_open_r ( struct _reent *ptr,
const char *file,
int flags,
int mode )
{
int which_devoptab = 0;
int fd = -1;
/* search for "file" in dotab_list[].name */
do {
if( strcmp( devoptab_list[which_devoptab].name, file ) == 0 ) {
fd = which_devoptab;
break;
}
} while( devoptab_list[which_devoptab++] );
/* if we found the requested file/device,
then invoke the device's open_r() method */
if( fd != -1 ) devoptab_list[fd].open_r( ptr, file, flags, mode );
/* it doesn't exist! */
else ptr->errno = ENODEV;
return fd;
}
```

Figure 6.10: Newlib _open_r stub.

6.2.5.4 _close_r

This stub is almost a clone of _write_r and _read_r, as shown in Figure 6.11.

```
Long
_close_r ( struct _reent *ptr,
int fd )
{
return devoptab_list[fd].close_r( ptr, fd );
}
```
Figure 6.11: Newlib _read_r stub.

6.2.5.5 _sbrk_r

Newlib’s memory allocator will only ask for incremental chunks of memory, a benign artifact of its UNIX heritage. Assuming a reserved a heap memory area using a character array called _heap, the _sbrk_r stub would look like the code in Figure 6.12.

```
unsigned char _heap[HEAPSIZE];
caddr_t _sbrk_r ( int incr )
{
static unsigned char *heap_end;
unsigned char *prev_heap_end;
/* initialize */
if( heap_end == 0 ) heap_end = heap;
prev_heap_end = heap_end;
if( heap_end + incr - heap > HEAPSIZE ) {
/* heap overflow--- announce on stderr */
write( 2, "Heap overflow!\n", 15 );
abort();
}
heap_end += incr;
return (caddr_t) prev_heap_end;
}
```
Figure 6.12: Newlib _sbrk_r stub.

Each time malloc() calls _sbrk_r the heap end grows by incr bytes. When it encounters the end of the allocated heap space (which hopefully never occurs), the stub sends a message to the standard error stream, then forcibly terminates the program. Another approach to a heap overflow would be to return NULL, and let the application find a way to muddle through on its own.

6.2.5.6 _malloc_lock & _malloc_unlock

Newlib's memory management routines like malloc() call these functions when they need to manipulate the memory heap. By implementing mutual exclusion in them, newlib's memory management becomes code reentrant or at least thread safe.

Portions of newlib's memory management code are recursive, so following sequence of invocations is often seen in response to a malloc() function call:

__malloc_lock, __malloc_lock, __malloc_unlock, __malloc_unlock

The tricky part here is that, the second __malloc_lock will cause itself to wait for a lock that it already holds from the first __malloc_lock. There are two ways to solve this problem. The first is to simply punt, and reimplement malloc() in its entirety using MicroC/OS-II's reentrant memory pool API. The second option is to really implement a working __malloc_lock and __malloc_unlock.

Figure 6.13 is an example of how to use MicroC/OS-II memory pools to implement malloc(). In this code, each allocation request consumes one block from the memory pool, whether the allocation needs that much space or not. Furthermore, if the allocation size exceeds the block size then the request fails, because MicroC/OS-II's memory block manager does not permit this.

```
/* number of bytes per allocation */
#define HEAPBLKSIZE 64
/* number of allocations available */
#define HEAPBLKS 1024
/* our heap */
OS_MEM *heap;
unsigned char heapmem[HEAPBLKS * HEAPBLKSIZE];
void *malloc ( size_t size )
{
INT8U err = OS_NO_ERR;
void *alloc = 0;
/* initialize, if necessary */
OS_ENTER_CRITICAL();
if( !heap )
heap = OSMemCreate( heapmem, HEAPBLKS,
HEAPBLKSIZE, &err );
OS_EXIT_CRITICAL();
if( heap && err == OS_NO_ERR ) {
/* if the request fits the heap block length,
then make the allocation from the heap */
if( size <= HEAPBLKLEN )
alloc = OSMemGet( heap, &err );
/* otherwise, we’re sunk */
else err = OS_MEM_NO_FREE_BLKS;
}
/* deny the allocation on errors */
if( err != OS_NO_ERR )
alloc = 0;
return alloc;
}
```

Figure 6.13: Newlib implementing malloc() with a memory pool.

Using MicroC/OS-II’s memory pools eliminates fragmentation worries and makes malloc() reentrant, but wastes memory if the pool’s block size doesn’t match up with the typical allocation request. Some of the waste may be reduced by providing buffer pools of several different sizes (perhaps corresponding to the sizes of data structures which is known to be frequently allocating memory for), but this approach is hardly generic.

For situations where a range of allocation sizes are needed, or the size of the largest potential allocation request is unknown, use newlib’s memory allocator and implement __malloc_lock and __malloc_unlock functions. Figure 6.14 shows how to do that.

```
/* semaphore to protect the heap */
static OS_EVENT *heapsem;
/* id of the task that is
currently manipulating the heap */
static int lockid;
/* number of times
__malloc_lock has recursed */
static int locks;
void
__malloc_lock ( struct _reent *_r )
{
OS_TCB tcb;
OS_SEM_DATA semdata;
INT8U err;
int id;
/* use our priority as a task id */
OSTaskQuery( OS_PRIO_SELF, &tcb );
id = tcb.OSTCBPrio;
/* see if we own the heap already */
OSSemQuery( heapsem, &semdata );
if( semdata.OSEventGrp && id == lockid ) {
/* we do; just count the recursion */
locks++;
}
else {
/* wait on the other task to yield the
heap, then claim ownership of it */
OSSemPend( heapsem, 0, &err );
lockid = id;
```

Figure 6.14: Newlib __malloc_lock and __malloc_unlock functions.

```
}
return;
}
void
__malloc_unlock ( struct _reent *_r )
{
/* release the heap once the number of
locks == the number of unlocks */
if( (--locks) == 0 ) {
lockid = -1;
OSSemPost( heapsem );
}
}
```

Figure 6.14: Continues.

6.2.5.7 _env_lock & _ env_unlock

These stubs protect the application's environment memory space, similar to what __malloc_lock and __malloc_unlock do for heap space. They are related to newlib's setenv() and getenv() functions; if environment variables are not used, they can be ignored or the strategy used for heap memory protection can be duplicated.

6.2.5.8 _exit

This stub forcibly terminates the application in response to the exit() or system() functions. There are several possiblities here, from allowing a watchdog timeout, to passing control to some kind of secondary application, to simulating a powerup reset in software.

This technique does not restore all of the target CPU's registers and peripherals to their powerup states, so application code can not depend on initial values for proper operation. In particular, device drivers cannot enable device interrupts prior to clearing any pending interrupt requests, or a spurious interrupt will result.

6.2.5.9 _stat_r, _fstat_r, _link_r, _unlink_r, and _lseek_r

These stubs implement newlib’s stat(), fstat(), link(), unlink() and lseek() functions. These functions all involve files, so they’re of little importance when the target environment lacks an underlying filesystem.

For _stat_r and _fstat_r, just tell the caller that the requested file or descriptor is a character device. This code is shown in Figure 6.15.

```
İnt
_stat_r ( struct _reent *_r, const char *file,
struct stat *pstat )
{
pstat->st_mode = S_IFCHR;
return 0;
}
int
_fstat_r ( struct _reent *_r, int fd, struct stat *pstat )
{
pstat->st_mode = S_IFCHR;
return 0;
}
```

Figure 6.15: Newlib _stat_r and _fstat_r stubs.

For _link_r and _unlink_r, claim that the operation always fails. See Figure 6.16.

```
İnt
_link_r ( struct _reent *_r, const char *oldname,
const char *newname )
{
r->errno = EMLINK;
return -1;
}

int
_unlink_r ( struct _reent *_r, const char *name )
{
r->errno = EMLINK;
return -1;
}
```

Figure 6.16: Newlib _link_r and _unlink_r stubs.

For _lseek_r, pretend that the request is always successful. See Figure 6.17.

```
off_t
_lseek_r( struct _reent *_r, int fd,
off_t pos, int whence )
{
return 0;
}
```

Figure 6.17: Newlib _lseek_r stub.

6.2.5.10 getpid

This function returns the context's process id, which can be emulated by using the MicroC/OS-II‘s OSTaskQuery() function. The code is in Figure 6.18.

```
int getpid ( void )
{
OS_TCB tcb;
INT8U err;
int id;
/* use our priority as a task id */
OSTaskQuery( OS_PRIO_SELF, &tcb );
id = tcb.OSTCBPrio;
return id;
}
```

Figure 6.18: Newlib getpid stub.

6.2.5.11 _times_r

This stub returns various time measurements for the current context. MicroC/OS-II doesn’t keep statistics on a task’s run time, so this function can be left unimplemented as shown in Figure 6.19.

```
İnt
_times_r ( struct _reent *r, struct tms *tmsbuf )
{
return -1;
}
```

Figure 6.19: Newlib _times_r stub.

6.3 Use of Libepc Library

The library provides processor and PC hardware initialization (including the interrupt descripter table), console I/O, timer access, sound, heap management, and more.

Unlike the functions in libc, those in libepc never use the ROM BIOS or expect an operating system. In some cases, functions in libepc duplicate services found in the standard C run-time library (libc). The corresponding libc fountains assume a desktop (rather than embedded) application environment, and reference objects in other libc modules that cause the linker to include a large amount of unnecessary or inappropriate code. The alternative implementations of those functions provided by libepc are self contained and eleminate this problem.

It should be noted that in order to cause linker to use the libepc versions instead of those in libc, libepc.a should be infront of the libc.a in the library list. And sometimes even this solution does not prevent some link time problems from happening due to tight relation between linker scripts and the library declarations due to use of object references declared in the linker scripts. In this project the rest of the object references required are, that not provided by libepc, just included for avoiding from this link time problems.

The following data types are defined in libepc.h and appear in the function descriptions given.

```
typedef  int                         BOOL;
typedef  unsigned char               BYTE8
typedef  unsigned short int          WORD16;
typedef  unsigned long int           DWORD32;
typedef  unsigned long long int      QWORD64;
typedef  signed long int             FIXED32;
typedef  signed long long int        FIXED64;
typedef  void                        (*ISR)(void);
```

Figure 6.20: Data types defined in “libepc.h”.

6.3.1 Libepc Library Memory Layout And Initialization

The IBM-PC partitions the address space into three regions: “conventional memory” (0-640KB), “reserved” memory (640KB to 1 MB), and “extended” memory (above 1MB).[10]

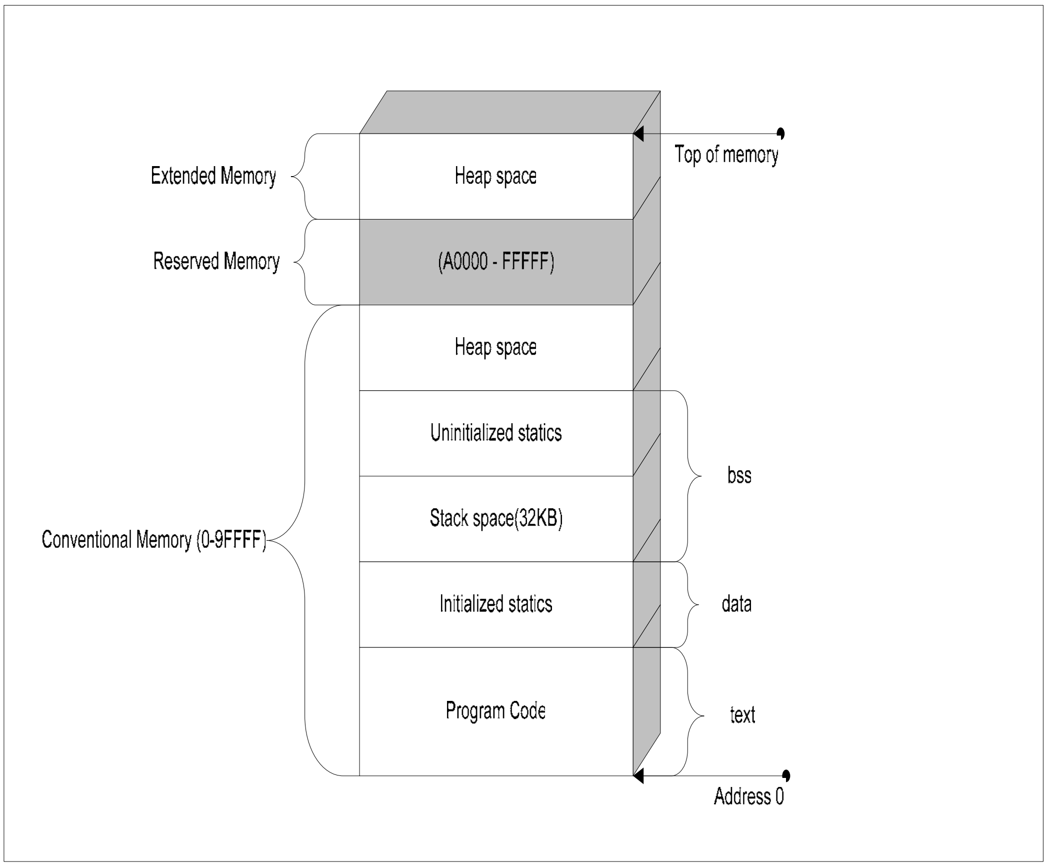


Figure 6.21: Libepc Library IBM-PC memory layout.

Conventional memory is subdivided into three major areas the code space (known as text), initialized 'data', and uninitialized data (called 'bss'). The bss contains all uninitialized static objects and a program stack of 32KB. Any remaining conventional memory and all of extended memory are used by the heap.

Execution of embedded application begins at address zero. Interrupts remain disabled throughout the initialization process, which ends with a call to function main. Initialization performs the following tasks:

1. Switches the processor into protected mode, establishes a flat memory model with all segments starting at address zero, and sets all segment sizes to 4GB. (init-cpu.asm)
2. Sets all uninitialized statics within the bss to zeros, and optionally copies the contents of any ROM data into RAM.(int-crt.c)
3. Initializes the 8259 Programmable Interrupt Controller (init8259.c)

4. Initializes the 8253 Programmable Timer to provide DRAM refresh and a 100-tick-per-second interrupt.(init8253.c)
5. Creates and initializes an Interrupt Descriptor Table (init-idt.c)

The Figure 6.22 outlines the functionality provided by the libepc library by giving the libepc.h header file content.

```
/* ============================================================ */
/* File: LIBEPC.H                                          */
/*                                                         */
/* Copyright (C) 2001, Daniel W. Lewis and Prentice-Hall   */
/*                                                         */
/* Purpose: Various #defines, structures, and function
      */
/* prototypes needed to use the corresponding library LIBEPC.A.
      */
/*                                                         */
/* Designed for use with the DJGPP port of the GNU C/C++   */
/* protected mode 386 compiler.                            */
/*                                                         */
/* Modification History:                                   */
/*                                                         */
/* ============================================================ */

#ifndef _LIBEPC_H_     /* Avoid multiple inclusions */
#define     _LIBEPC_H_

/* ------------------------------------------------------------ */
/* A few datatypes to make the operand size more obvious.  */
/* ------------------------------------------------------------ */
typedef int            BOOL  ;
typedef unsigned char        BYTE8 ;
typedef unsigned short int   WORD16     ;
typedef unsigned long int    DWORD32    ;
typedef unsigned long long int     QWORD64     ;
typedef signed long int      FIXED32    ;      /* 16.16 Fixed-
Point */
typedef signed long long int FIXED64    ;      /* 32.32 Fixed-
Point */
typedef void                 (*ISR)(void) ;    /* Pointer to an
ISR */

/* ------------------------------------------------------------ */
/* Constants for use with datatype BOOL (above).           */
/* ------------------------------------------------------------ */
#ifndef TRUE
#define     TRUE        1
#endif
#ifndef FALSE
#define     FALSE       0
#endif
```

Figure 6.22: The libepc.h header file used by the adaptation code.

```
/* -------------------------------------------------------------
*/
/* Macros to extract the LSByte and MSByte of a WORD16 value
      */
/* -------------------------------------------------------------
*/
#define     LSB(u)            ((u) & 0xFF)
#define     MSB(u)            ((u) >> 8)

/* -------------------------------------------------------------
*/
/* Returns number of elements in an array. (Use in for loops.)
      */
/* -------------------------------------------------------------
*/
#define     ENTRIES(a)  (sizeof(a)/sizeof(a[0]))

/* -------------------------------------------------------------
*/
/* Declaration prefix to hide an object from the linker.   */
/* -------------------------------------------------------------
*/
#define     PRIVATE           static

/* -------------------------------------------------------------
*/
/* Define a NULL pointer.                            */
/* -------------------------------------------------------------
*/
#ifndef NULL
#define     NULL        ((void *) 0)
#endif

/* -------------------------------------------------------------
*/
/* 386 instructions needed when writing ISR's. Note that IRET
      */
/* pops the pointer to the stack frame that was established by
      */
/* code that the compiler generates at every function entry.
      */
/* -------------------------------------------------------------
*/
#define     PUSHCS            __asm__ __volatile__ ("PUSHL %CS")
;
#define     PUSHF       __asm__ __volatile__ ("PUSHFL")
#define     POPF        __asm__ __volatile__ ("POPFL")
#define     STI         __asm__ __volatile__ ("STI")
#define     CLI         __asm__ __volatile__ ("CLI")
#define     PUSHA       __asm__ __volatile__ ("PUSHAL")
#define     POPA        __asm__ __volatile__ ("POPAL")
#define     ENTER       __asm__ __volatile__ ("ENTER $0,$0")
#define     LEAVE       __asm__ __volatile__ ("LEAVE")
#define     IRET        __asm__ __volatile__ ("IRET")
```

Figure 6.22: Continues.

```
/* ------------------------------------------------------------
*/
/* Support for functions implemented in IO.ASM             */
/* ------------------------------------------------------- */
void        outportb(WORD16, BYTE8) ;
BYTE8       inportb(WORD16) ;
void        exit(int) ;

/* ------------------------------------------------------------
*/
/* Support for functions implemented in INIT-CRT.C         */
/* ------------------------------------------------------------
*/
void *            LastMemoryAddress(void) ;

/* ------------------------------------------------------------
*/
/* Support for functions implemented in INIT-IDT.C         */
/* ------------------------------------------------------------
*/
#define           IRQ_TICK    0
#define           IRQ_KYBD    1
#define           IRQ_COM2_COM4     3
#define           IRQ_COM1_COM3     4
#define           IRQ_FLOPPY  6
#define           IRQ_PAR_PORT      7
#define           IRQ_RTC           8
#define           IRQ_PS2_MOUSE     12
#define           IRQ_HARD_DISK     14

int         IRQ2INT(int irq) ;
ISR         GetISR(int int_numb) ;
void        SetISR(int int_numb, ISR isr) ;

/* ------------------------------------------------------------
*/
/* Support for functions implemented in KEYBOARD.C         */
/* ------------------------------------------------------------
*/
BYTE8       GetScanCode(void) ;
BOOL        ScanCodeRdy(void) ;
BOOL        SetsKybdState(BYTE8) ;
WORD16            ScanCode2Ascii(BYTE8 code) ;

/* ------------------------------------------------------------
*/
/* Support for functions implemented in SPEAKER.C          */
/* ------------------------------------------------------------
*/
void        Sound(int hertz) ;
/* ------------------------------------------------------------
*/
/* Support for functions implemented in CYCLES.ASM         */
/* ------------------------------------------------------------
*/
QWORD64           CPU_Clock_Cycles(void) ;
```

Figure 6.22: Continues.

```
/* -----------------------------------------------------------------
*/
/* Support for functions implemented in TIMER.C            */
/* -----------------------------------------------------------------
*/
DWORD32           Milliseconds(void) ;
DWORD32           Now_Plus(int seconds) ;

/* -----------------------------------------------------------------
*/
/* Support for functions implemented in DISPLAY.C          */
/* -----------------------------------------------------------------
*/
WORD16 *    Cell(int row, int col) ;
void        ClearScreen(BYTE8 attb) ;
char *            FormatUnsigned
            (char *bfr, unsigned val, int base, int width, char
fill) ;
int         GetCursorCol(void) ;
int         GetCursorRow(void) ;
void        PutAttb(BYTE8 attb, int cells) ;
void        PutChar(char ch) ;
void        PutCharAt(char ch, int row, int col) ;
void        PutString(char *string) ;
void        PutUnsigned(unsigned val, int base, int width) ;
void        SetCursorPosition(int row, int col) ;
void        SetCursorVisible(BOOL visible) ;
char *            Unsigned2Ascii(char *bfr, unsigned val, int
base) ;

/* -----------------------------------------------------------------
*/
/* Support for functions implemented in WINDOW.C           */
/* -----------------------------------------------------------------
*/
typedef struct ROWCOL
      {
      int   first ;
      int   last ;
      int   cursor ;
      } ROWCOL ;

typedef struct WINDOW
      {
      ROWCOL      row ;
      ROWCOL      col ;
      char  title[1] ;
      } WINDOW ;

WINDOW *    WindowCreate(char *title, int row_first, int
row_last,
                  int col_first, int col_last) ;
void        WindowErase(WINDOW *w) ;
void        WindowPutChar(WINDOW *w, char ch) ;
void        WindowPutString(WINDOW *w, char *str) ;
```
Figure 6.22: Continues.

```
void        WindowSelect(WINDOW *w) ;
void        WindowSetCursor(WINDOW *w, int row, int col) ;

/* -------------------------------------------------------------
*/
/* Support for functions implemented in HEAP.C             */
/* -------------------------------------------------------------
*/
void *            malloc(long unsigned int) ;
void        free(void *) ;
/* -------------------------------------------------------------
*/
/* Support for functions implemented in QUEUE.C            */
/* -------------------------------------------------------------
*/
typedef     struct QUEUE
      {
      int   item_size ;
      int   max_items ;
      int   item_count ;
      int   nq_index ;
      int   dq_index ;
      char  bfr[0] ;
      } QUEUE ;

QUEUE *           QueueCreate(int numb_items, int item_size) ;
BOOL        QueueInsert(QUEUE *q, void *data) ;
BOOL        QueueRemove(QUEUE *q, void *data) ;

/* -------------------------------------------------------------
*/
/* Support for functions implemented in FIXEDPT.ASM        */
/* -------------------------------------------------------------
*/
#define FIXED32_ONE           65536L
#define FIXED32_PI            205887L
#define FIXED32_2PI           411775L
#define FIXED32_E       178144L
#define FIXED32_ROOT2         74804L
#define FIXED32_ROOT3         113512L
#define FIXED32_GOLDEN        106039L
#define FLOAT32(x)            ((FIXED32) ((x) << 16))
#define TRUNC32(x)            ((int) ((x) >> 16))
#define ROUND32(x)            ((int) (((x) + 0x8000) >> 16))
FIXED32           Product32(FIXED32 multiplier, FIXED32
multiplicand) ;
FIXED32           Quotient32(FIXED32 dividend, FIXED32 divisor) ;
FIXED32           Inverse32(FIXED32 n) ;
FIXED32           Sqrt32(FIXED32 n) ;
FIXED64           Product64(FIXED64 multiplier, FIXED64
multiplicand) ;


#endif
```

Figure 6.22: Continues.

CHAPTER 7

VIRTUAL MACHINE RUN-TIME ENVIRONMENT

7.1 Overview

The execution of the virtual machine starts with the bootup procedure and MicroC/OS-II initializes a task which contains the virtual machine startup section. And the rest is performed by the virtual machine code re-used from the KVM. KVM make use of the functions that it needs during its execution from the MicroC/OS-II where needed. The Figure 7.1 illustrates the run-time environment of the virtual machine.[13]

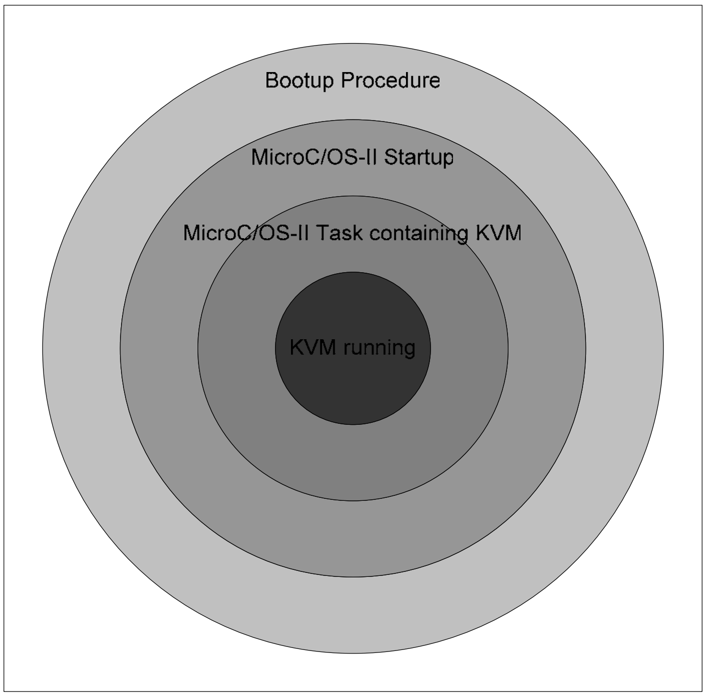


Figure 7.1: Layered Container for KVM

7.2 Preparing the KVM for the run-time

The procedure of preparing a bootable KVM composed of the following steps.

7.2.1 Preparing "embedded.bin" Binary File

The virtual machine executable code is organized into an binary file by going through the following steps. The source code is built by using the make utility provided by the DJGPP development environment. The makefile shown in Figure 7.2 includes the compiler settings and linker commands to build the embedded.bin file.

```
#
# Configuration: Release
#
OUTDIR=Release
OUTFILE=
CFG_INC=-IuCOS-Libc/include -Iucos_ii/ -Ilibepc/ -IuCOS-Libc/time
-IVmCommon/h/ -IVmExtra/h/ -IuCOSPlatform/h/
CFG_LIB=
CFG_OBJ=
COMMON_OBJ=$(OUTDIR)/cache.o $(OUTDIR)/class.o \
      $(OUTDIR)/collector.o $(OUTDIR)/e_rem_pio2.o
$(OUTDIR)/e_sqrt.o \
      $(OUTDIR)/events.o $(OUTDIR)/execute.o $(OUTDIR)/fields.o \
      $(OUTDIR)/fp_bytecodes.o $(OUTDIR)/fp_math.o
$(OUTDIR)/frame.o \
      $(OUTDIR)/garbage.o $(OUTDIR)/global.o $(OUTDIR)/hashtable.o
\
      $(OUTDIR)/inflate.o $(OUTDIR)/interpret.o $(OUTDIR)/jar.o \
      $(OUTDIR)/k_cos.o $(OUTDIR)/k_rem_pio2.o $(OUTDIR)/k_sin.o \
      $(OUTDIR)/k_tan.o $(OUTDIR)/loader.o $(OUTDIR)/loaderFile.o
\
      $(OUTDIR)/log.o $(OUTDIR)/main.o $(OUTDIR)/native.o \
      $(OUTDIR)/nativeCore.o $(OUTDIR)/nativeFunctionTableUCOS.o \
      $(OUTDIR)/pool.o $(OUTDIR)/profiling.o $(OUTDIR)/property.o
\
      $(OUTDIR)/ROMJavaUCOS.o $(OUTDIR)/runtime2_md.o \
      $(OUTDIR)/runtime_md.o $(OUTDIR)/s_ceil.o
$(OUTDIR)/s_copysign.o \
      $(OUTDIR)/s_cos.o $(OUTDIR)/s_fabs.o $(OUTDIR)/s_floor.o \
      $(OUTDIR)/s_scalbn.o $(OUTDIR)/s_sin.o $(OUTDIR)/s_tan.o \
      $(OUTDIR)/stackmap.o $(OUTDIR)/StartJVM.o $(OUTDIR)/thread.o
\
      $(OUTDIR)/verifier.o $(OUTDIR)/verifierUtil.o
$(OUTDIR)/w_sqrt.o
OBJ=$(COMMON_OBJ) $(CFG_OBJ)
ALL_OBJ=$(OUTDIR)/cache.o $(OUTDIR)/class.o \
      $(OUTDIR)/collector.o $(OUTDIR)/e_rem_pio2.o
$(OUTDIR)/e_sqrt.o \
      $(OUTDIR)/events.o $(OUTDIR)/execute.o $(OUTDIR)/fields.o \
      $(OUTDIR)/fp_bytecodes.o $(OUTDIR)/fp_math.o
```

Figure 7.2: Makefile used to compile and link the embedded.bin binary.

```
$(OUTDIR)/frame.o \
      $(OUTDIR)/garbage.o $(OUTDIR)/global.o
$(OUTDIR)/hashtable.o \
      $(OUTDIR)/inflate.o $(OUTDIR)/interpret.o $(OUTDIR)/jar.o \
      $(OUTDIR)/k_cos.o $(OUTDIR)/k_rem_pio2.o $(OUTDIR)/k_sin.o
\
      $(OUTDIR)/k_tan.o $(OUTDIR)/loader.o $(OUTDIR)/loaderFile.o
\
      $(OUTDIR)/log.o $(OUTDIR)/main.o $(OUTDIR)/native.o \
      $(OUTDIR)/nativeCore.o $(OUTDIR)/nativeFunctionTableUCOS.o
\
      $(OUTDIR)/pool.o $(OUTDIR)/profiling.o $(OUTDIR)/property.o
\
      $(OUTDIR)/ROMJavaUCOS.o $(OUTDIR)/runtime2_md.o \
      $(OUTDIR)/runtime_md.o $(OUTDIR)/s_ceil.o
$(OUTDIR)/s_copysign.o \
      $(OUTDIR)/s_cos.o $(OUTDIR)/s_fabs.o $(OUTDIR)/s_floor.o \
      $(OUTDIR)/s_scalbn.o $(OUTDIR)/s_sin.o $(OUTDIR)/s_tan.o \
      $(OUTDIR)/stackmap.o $(OUTDIR)/StartJVM.o
$(OUTDIR)/thread.o \
      $(OUTDIR)/verifier.o $(OUTDIR)/verifierUtil.o
$(OUTDIR)/w_sqrt.o

COMPILE=gcc -c -Os -o "$(OUTDIR)/$(*F).o" $(CFG_INC) "$<"
LINK=ld $(ALL_OBJ) -Tlink.cmd -ustart -Map Link.map

# Pattern rules
$(OUTDIR)/%.o : VmCommon/src/%.c
      $(COMPILE)

$(OUTDIR)/%.o : uCOSPlatform/src/%.c
      $(COMPILE)

$(OUTDIR)/%.o : VmExtra/src/%.c
      $(COMPILE)

$(OUTDIR)/%.o : uCOSPlatform/JavaROMImage/%.c
      $(COMPILE)

$(OUTDIR)/%.o : VmExtra/src/fp/%.c
      $(COMPILE)

# Build rules
all: $(OUTDIR) $(OBJ)
      $(LINK)

$(OUTDIR):
      mkdir -p "$(OUTDIR)"

# Rebuild this project
rebuild: cleanall all
```

Figure 7.2: Continues.

```
# Clean this project
clean: $(OUTFILE)
      del $(OUTDIR)\*.o

# Clean this project and all dependencies
cleanall: clean
```

Figure 7.2: Continues.

The compiler command provided in the make file just includes the optimize for size flag “-Os” while compling the KVM source code.

```
COMPILE=gcc -c -Os -o "$(OUTDIR)/$(*F).o" $(CFG_INC) "$<"
```

The linker command provided in the makefile includes the flags for

“-Tlink.cmd” : linker script file for application memory layout specification.

“-ustart” : application entry point for starting the execution of the application.

“-Map link.map” : a map file including the details about the object files and object references of the application.

```
LINK=ld $(ALL_OBJ) -Tlink.cmd -ustart -Map Link.map
```

The linker script content listed in Figure 7.3 includes the application memory usage details complying with an IBM-PC compatible machine.[14]

```
/* ---------------------------------------------------------- */
/* Script file for linker ld.  Designed for use on an IBM-PC.   */
/* ---------------------------------------------------------- */
GROUP ("ucos_ii/ucos_ii.a", "Libepc/libepc.a","uCOS-
Libc/libc.a","uCOS-Libc/libm.a")
OUTPUT_FORMAT("binary")
OUTPUT("embedded.bin")
EXTERN(start)
ENTRY(start)
MEMORY {
   /* ------------------------------------------------------- */
   /* The loader starts execution of the application at         */
   /* physical address zero, so .start must be placed in        */
   /* conventional memory, thus '>conventional'. You may put    */
   /* one or more of the remaining sections in extended memory  */
   /* by replacing their respective '>conventional' directives  */
   /* with a '>extended' directive where it appears below. If   */
   /* you do, however, the application will not run on a        */
   /* machine with no extended memory.                          */
   /* ------------------------------------------------------- */
   conventional : ORIGIN = 0x00000000, LENGTH = 640*1024
       reserved : ORIGIN = 0x000A0000, LENGTH = 384*1024
       extended : ORIGIN = 0x00100000, LENGTH = 4095*1024*1024
}

SECTIONS {
   .text 0x00000000 : {
       text_frst = . ;
       *(.start)
       *(.init)
       *(.text)
       PROVIDE(__exit = _exit);
       text_last = . - 1 ;
   } >conventional

   .data : {
       data_frst = . ;
       *(.data)
       data_last = . - 1 ;
   } >conventional

   .bss : {
       bss_frst = . ;
       *(.bss)
       *(COMMON)
       bss_last = . - 1 ;
   } >conventional
```

Figure 7.3: Linker script used while linking the embedded.bin binary.

```
   /* ---------------------------------------------------------
*/
   /* The stack sits just above the bss.  It's size is set here
*/
   /* so that it can be changed without recompiling the code.
*/
   /* ---------------------------------------------------------
*/
   stack_frst = bss_last + 1 ;
   stack_last = bss_last + 128*1024 ;

   /* ---------------------------------------------------------
*/
   /* The heap starts just above the stack and gets the rest.
*/
   /* ---------------------------------------------------------
*/
   heap_frst = stack_last + 1 ;
      _end = heap_frst;
}
```

Figure 7.3: Continues.

The text section includes two sections prior to declare the rest of the text(code) section which is critical for the application run-time initialization. These sections are *(.start) and *(.init) sections which are declared in the init-cpu.asm (part of libepc library). Since these sections includes the code for system initialization and protected mode switching, these two sections should be kept in the first 64KB of the memory. Because, in real-mode, system can not address a code section beyond the 64KB code section and unforutnately system starts in real-mode. That becomes a serious problem esspecially when the code section exceeds 64KB. The rest of the linker script is mapping with the memory model defined in section 6.3.1.[15]

The embedded.bin file is generated by two step procedure detailed above.

7.2.2 Preparing The Bootable KVM

There are two steps two prepare the bootable KVM diskette.

1. Copy the built embedded.bin to a formatted 1'44 empty diskette
2. Run "copyboot a: "in the command line where "bootload.bin" file is in the same directory.

In the previous sections embedded.bin file related issues covered in detail and in the following section “bootload.bin” which is a boot loader is covered in detail.

CHAPTER 8

CONCLUSION

The Java Virtual Machine implementation over the MicroC/OS-II real-time operating system is an effort to prove the availability of java in resource constraint target platforms. This thesis work shows that it is possible to a run a virtual machine with a memory footprint as small as 150 KB. Its run-time performance is acceptable for a soft real-time application. During the realization of thesis the stubs required to port the K Virtual Machine or any other portable java virtual machine is also outlined from the target real-time operating system point of view and from the K Virtual Machine point of view. The configuration parameters required to be specified to port K Virtual Machine on the MicroC/OS-II operating system over an x86 platform are identified and used appropriately on purpose.

Also the development environment candidates to build KVM over MicroC/OS-II is evaluated. The development environment candidates are Microsoft Visual C++ 6.0 development environment and DJGPP development enviornment.

The first candidate evaluated was the Microsoft Visual C++ 6.0 development environment. An executable file with “exe“ extension is populated as an output of this environement but it was lack of booting capability. So it was not suitable for the aim of the project.

Another development environment which provides some more customization over the generated output is DJGPP environment. DJGPP environment is the port of GNU port of the GCC compiler and utilities under MS-DOS. Even if the DJGPP tools provide more flexibility over the executable binary image generation process, the conventional dependency to the standard c run-time libraries of DJGPP environment makes executable binary image a garbage of un-used objects and does not allow to customize the application memory layout as expected by the bootloader. But with the use of customization capabilities development environment is converted into an appropriate set of tools suitable for convenient executable generation.

The evaluation of development environments designated the effective way to go further to complete the K Virtual Machine implementation. The next step taken is preparing the libraries required by the K Virtual Machine for its proper execution. The Newlib is choosen as the adaptable c run-time library source because of its portability features.

Porting the Newlib required to build a cross compiler tool set whose host platform is an x86 based MS-DOS system and the target is an coff-i386 formatted executable binary. After configuring and compiling the cross compiler from the source code of the DJGPP binutils and gcc packages, it becomes possible to build the customized libc.a from the Newlib source tree. From that point on the K Virtual Machine code becomes, nevertheless, compilable.

The process of preparing the java application code deployable on to a K Virtual Machine with the briefed configuration above is not also a straight forward operation. As any ordinary java application deployment, it starts with the java compilation by using the legacy java compiler provided by Sun Corporation. And the further stages are dedicated for preparing this java application embedded into this K Virtual Machine. So the resultant virtual machine will be dedicated to run this application only.

The next phase is applying the preverification to the class files of application generated by the java compiler in the previous step. The preverification modifies the content of class file into an convenient format acceptable by the run-time verifier of the K Virtual Machine.

After the completion of preverification, Java Code Compact tool takes the application specific and other required system class files and populates two C files from this content which is plugable into the K Virtual Machine at the build-time.

Another tool which is very useful and a replacement for JavaFilter application is the proguard tool.[5] This tool plays an important role for optimizing the class file set required by the application. The size of the C files generated by Java Code Compact becomes approximately five times smaller in size when the class files provided to Java Code Compact is filtered with Proguard.

After the K Virtual Machine is built with the C files plugged, it becomes ready to execute the application for which it is deployed by booting the target x86 platform.

The work realized as part of this thesis preparation is a starting point for improving the availability of java for resource constraint environments. There are a number of developmental fields to enhance the proposed architecture.

Since the proposal of thesis is organized to keep the I/O requirements of K Virtual Machine at minimum, the provided source may be taken as a baseline and some more I/O capability may be added as part of improvement activities. As one of the capabilities of K Virtual Machine, Java Application Manager is left out of realized K Virtual Machine port with the use of configuration options. If the support for the I/O functions is improved then the JAM capability may be enabled as a functionality.

Another enhancement for the K Virtual Machine may be the development environment improvement. Development environment customizations may be implemented for the development environments which does not supporting the appropriate K Virtual Machine development and deployment.

The supporting libraries may be subject to improvement with better performance and with a wider range of functionality. A better Newlib port may be built for better performance. It should also be planned to insert the libepc library functionality into the Newlib build for improving the portability.

Some other Newlib builds should be generated for supporting different platforms without requiring any source code modification on the K Virtual Machine source code as well as the MicroC/OS-II source code to improve portabillity.

The procedure of preparing the pluggable C files from the java application may be improved by combining all or some of the tools required for this process. Java Code Compact tool may be extended to include the optimization features for decreasing the size of C files. Another alternative may be combining the preverifier tool with the class set optimization tool which will provide the input expected by the Java Code Compact tool.

K Virtual Machine may be optimized by using the cocurrency services provided by the MicroC/OS-II real-time operating system. One alternative may be seperating the garbage collector and bytecode interpreter into distinct tasks and make use of the message queues and other multitasking features provided to improve the concurrency.

Another virtual machine implementation may be choosen to port within boundries of the given architecture. Another improvement stragtegy may be developing a virtual machine from scratch, but that should be a team work rather than a individual effort, when the complexity of virtual machine implementation is taken into consideration.

The K Virtual Machine port may be improved by one of the strategies highlighted above to satisfy the Real-Time Java Specification which is becoming a standard to comply for the tool and application developing society.

Finally, Java Virtual Machine implementations for embedded platforms has been seen to be one of the most challenging development area due to popularity of java. There are also efforts to design java aware hardware architectures to increase the performance of applications. The era of java looks like to be competitor of the computers in Von Neuman architecture with their new hardware components to handle software problems like garbage collection instead of leaving all the responsibility of executing the applications to the components as central processing unit and main memory only.

REFERENCES

[1] T.Lindholm and F.Yellin. *The Java(tm) Virtual Machine Specification Second Edition*. Addison Wesley. 1999.

[2] R.Stark, J.Schmid and E.Börger. *The Java(tm) and the Java(tm) Virtual Machine Definition, Verification, Validation*. Springer. 2001.

[3] Sun Microsystems, Inc. “The Java HotSpot virtual machine”. http://java.sun.com/products/hotspot/, 2001.

[4] Sun Developer Network. “K Virtual Machine(KVM) Whitepaper”. http://java.sun.com/products/cldc/wp/. 2005.

[5] E.Lafortune. “Proguard 3.3.1 Users Manual”, http:// proguard. sourceforge.net/manual/. 2005.

[6] Sun Microsystems, Inc. “Connected Limited Device Configuration, Specification 1.0, Java 2 Platform Micro Edition”. Sun Microsystems Inc. 2000.

[7] Sun Microsystems, Inc. “K Virtual Machine Porting Guide, Java 2 Platform Micro Edition”. Sun Microsystems. 2000.

[8] T.Wilkinson. “KAFFE, A Virtual Machine to run Java Code”. http://www.kaffe.org. 2000.

[9] J.J.Labrosse. *MicroC/OS-II The Real-Time Kernel, Second Edition*. CMP Books. 2002.

[10] D.W.Lewis. *Fundamentals of embedded software: where C and assembly meet*. Prentice Hall. 2002.

[11] Redhat Corporation. “Redhat Newlib C Library: libc”. http://sources.redhat.com/newlib/libc.html. 2005.

[12] Redhat Corporation. “Redhat Newlib C Library: libm”. http://sources.redhat.com/newlib/libm.html. 2005.

[13] J.Gereau. “Porting MicroC/OS-II to the X86 (PM)”. http://www.exposecorp.com/embedded/portx86p-2.htm. 2005.

[14] D.W.Lewis. “Errata&Updates for Text: Fundamentals of Embedded Software”. http://www.cse.scu.edu/~dlewis/book1 /Errata1stPrinting.htm. 2005.

[15] Usenet discussion group. “OSD: PC bootstrap”. http://my.execpc .com /~geezer/osd/boot/. 2005.