A Bare Machine Application Development Methodology

Gholam H. Khaksari*
Penn State Abington, Abington, PA 19001-3918 USA

Ramesh K. Karne† and Alexander L. Wijesinha†
Towson University, Towson, Maryland, 21252 USA

Abstract

Traditional software applications are developed based upon some type of operating system and its related tools and environments. These applications are designed and implemented to run on top of an existing operating system, kernel or Exokernel, making them dependent on the underlying platform. This paper presents a bare machine application development methodology for building software applications that are totally independent of any operating system platform or environment, and able to execute directly on Intel IA-32 based CPU hardware. First, we introduce the bare machine computing principles on which the methodology is based and compare the bare machine application development paradigm with the conventional approach for application development. Next, we present the methodology steps drawing on past experience gained by developing complex bare machine applications. Finally, we illustrate the use of this methodology to develop a bare machine soft-phone application and its elements: self-boot, loader, application logic, tasking structure, execution threads, application behavior model, object model and relations, and the memory map.

Key Words: Bare machine computing, operating systems, software development paradigms, software applications, soft-phones.

1 Introduction

Traditionally, computer software applications are developed and targeted for a commercial off the shelf (COTS) platform based on an operating system (OS) such as Microsoft Windows, Linux, UNIX, or some sort of kernel. Moreover, most OS platforms are augmented with a state of art development environment and tools to support application development and testing. These OS-based applications utilize rich OS services and development environment tools making them fully dependent on their platform. The chosen programming language also assumes a target OS that is realized through the compilation and linking process by substituting appropriate system calls or an API (application programming interface).

In this paper, we introduce a methodology for software application development based on the bare machine computing (BMC) paradigm [10], which completely eliminates the OS, kernels, and operating environments. The elimination of these layers gives software developers complete control over hardware resources and enables the development of BMC applications with many characteristics that are not present in conventional systems. These include low overhead and cost, optimal CPU tasking structures and strategies, protocol intertwining, and immunity against attacks targeting an OS. However, BMC application development presents a challenge in that most of the tools or features provided by an OS-based software development environment cannot be used when writing bare PC applications. Currently, only a minimal set of tools, program structures and drivers are available for bare machine application development.

Bare machine applications execute directly on Intel IA-32 based processor hardware, although other CPU architectures could also be used in principle. The applications have direct access to bare machine hardware enabling them to control and manage hardware resources. Many bare machine applications including Web servers [5-6], email servers [4], and VoIP soft-phones [12-13] have been developed. These applications execute on ordinary desktops or laptops (with no OS), and their underlying building blocks may be reused or adapted when creating new bare machine applications. In addition, one or more related applications can also be designed to run as a given set of desktop applications.

Bare machine programmers must have knowledge of computer-system software concepts and techniques in addition to being able to write, debug and test C/C++ applications. For example, they deal with device drivers and their usage, memory management, process control and management, hardware interrupts, and CPU scheduling when writing bare machine applications. Providing programmers with full privileges to access and control bare machine resources simplifies resource management and control, and allows them to develop applications that are much simpler and more efficient. However, the BMC approach could present a programmer with challenges when designing, coding, and testing applications since there is neither OS support nor an

* School of Information Sciences and Technology. E-mail: ghk10@psu.edu.
† Department of Computer and Information Sciences. E-mail: rkarne@towson.edu and awijesinha@towson.edu.

integrated development environment or tools available on a bare machine. However, the common hardware interfaces such as keyboard, display, device drivers, CPU, USB mass storage and memory are provided as APIs to C or C++ programs.

A bare machine application is designed and programmed as a self-supporting and self-executing program called an Application Object (AO) [8]. In addition to the application code, an AO contains all necessary operating structures and features to manage itself during execution. An AO is able to boot and load itself from USB persistent storage, and while executing it can perform its own memory management, CPU scheduling, multi-tasking, and I/O, and handle exceptions and interrupts. The single monolithic executable AO is created using the C/C++ programming language. An AO directly communicates through its hardware interfaces to access memory, CPU, I/O device and interrupts. Only one AO may be executing on a bare machine at any given time. An AO could contain a single application such as an Email server, or a set of applications such as an Email server, Web server, and VoIP soft-phone client.

Bare machine hardware interfaces are specific to the CPU architecture in use. Currently, we have chosen the Intel IA-32 platform family for the initial BMC research and development effort; hence the current bare machine application development is limited to use of the IA-32 CPU architecture and C++ programming language. The BMC methodology can also be applied to other families of CPU architectures, control devices and peripheral equipment. The bare machine hardware interfaces are developed using C and/or Assembly language, and are directly invoked from C++ programs [9].

By eliminating the OS layer, the BMC paradigm enables total autonomy in program execution and control. Bare machine programmers develop applications that are self-executing and self-controlled. Thus, these applications contain all necessary execution controls and also have full privileges to access and control bare machine hardware resources.

The rest of this paper is organized as follows. BMC principles are presented and the bare machine application development paradigm is contrasted with the traditional approach for application development in Section 2; a bare machine application development methodology is described in Section 3; the use of this methodology to design and implement a bare machine soft-phone application is illustrated in Section 4; related work is summarized in Section 5; and the conclusion is stated in Section 6.

2 Bare Machine Computing Principles

There are fundamental differences between OS-based computing and BMC. However, original computing approaches were a form of BMC. As it evolved over time, computing gradually became OS-dependent to make it more convenient for application programmers. Furthermore, OS evolution was also driven by a business model. As a result, popular OSs have grown exponentially in size and complexity reaching well over 70 million lines of code to support the requirements of a universal set of applications. This increase in OS size and complexity has made it harder to build secure systems. Many attacks target vulnerabilities in the underlying OS and enable attackers to gain administrative privileges after system compromise. In addition to being immune to OS-based attacks, BMC systems are more resistant to conventional attacks since AO code is statically bound and execution flow is a single thread of execution without interruption.

The BMC paradigm is based on writing software applications without use of any middleware or OS. Simplicity and efficiency are the two primary goals of this paradigm. Any trained software programmer could use it to create applications that are very simple and highly efficient, and also provide

- smaller code size
- high longevity
- pervasiveness
- complete ownership and control
- immunity to OS-related security vulnerabilities
- direct access to hardware with no OS intermediaries.

The rest of this section will enumerate and discuss the novel features of BMC that enable programmers to design and develop simple and efficient software applications.

2.1 Single Application Object

Bare machine applications are encapsulated in a single monolithic executable referred to as an Application Object (AO). An AO may include only a single application such as a soft-phone client, or it may contain a set of applications such as an email client, a web server and a soft-phone client. The object code for an AO is stored on a portable bootable device such as a USB flash drive. When a bare machine is booted from the portable boot device, the resident AO object code is loaded into the bare machine main memory and then executed. A simple non-graphical user interface allows commands to be entered by the user in order to select execution options or enter configuration parameters.

2.2 Single-Execution Operating Environment

A bare machine application is a single-execution operating environment as opposed to the multi-execution operating environments provided by a traditional OS-based system. This means that only a single AO can be loaded and executed on a bare machine at any given time. No additional user programs can be loaded into the bare machine memory while an AO is executing. This also implies that there is no other software including an operating system or any other system programs running on the bare machine other than the AO itself.

2.3 Self-Boot and Load

The BMC paradigm requires a portable bootable device such as a flash drive for booting and loading an AO application into

bare machine main memory during the power-up sequence. It should be noted that since the BMC approach does not use a hard disk, one may not be used as a bootable device. The bootable device must contain the AO application image as well as the necessary boot code and a program loader. During the bare machine power-up sequence using this bootable device, the bare machine BIOS loads the boot sector from the bootable device into internal memory. Next, the boot code loads the AO application from the boot device into internal memory and starts executing the application by jumping to the starting address for the application program contained within the AO. User authentication and authorization procedures/policies may be implemented and enforced before executing any of the tasks contained within an AO.

2.4 Self-Cleanup

Upon termination of an AO execution, all memory resident program code and data are erased from the bare machine's internal memory. The bare machine can be shut down after the user removes the bootable device.

2.5 Direct Access to Hardware

An AO has full and direct access to hardware resources and it alone is responsible for their proper utilization and management. AO applications can access the bare machine hardware resources via the provided hardware driver interfaces or the hardware Application Programming Interfaces (APIs). For example, by using the API for the Network Interface Card (NIC), an AO is able to receive and send IP network packets. Hardware APIs are also available for the AO to perform other necessary actions such as stopping and restarting the CPU, reading buffers from an audio device, writing buffers to an audio device, loading data into main memory, reading from the keyboard, and writing to the display device.

2.6 Static Memory-Binding

An AO application is statically bound to bare machine internal memory at compile time. There are no features or facilities that allow for any modification to the AO code when it is executing since there is no support for dynamic linking and binding. The entire AO object code is a single monolithic executable throughout its entire lifetime. Static binding makes it more difficult to attack a BMC system.

2.7 Self-Managed Dynamic Memory

With traditional OS-based software, an application is responsible for allocation and release of dynamic memory while the underlying OS is responsible for actual implementation of the dynamic memory service. Hence, traditional software applications do not have any control over how dynamic memory is actually managed and implemented. In contrast, an AO has complete control over dynamic memory and is responsible for allocation, release and all other aspects of dynamic memory management. The BMC paradigm treats dynamic memory as real memory, and avoids the need for complicated virtual memory management and paging techniques of an OS-based system. This results in a reduction of AO complexity since the AO programmer has control over defining and organizing main memory space. AO programmers can also restrict memory access rights. Real memory size limitations are not a major issue in bare machine computing since real memory is cheap and easy to expand. Additionally, due to its application-centric design, an AO has a very small code size and does not require additional memory during execution. The BMC paradigm does not allow dynamic memory to be shared. Instead, communication between tasks or between AO elements is achieved by using message passing.

2.8 No Concurrency Control

In BMC, concurrency control mechanisms are avoided. Each BMC application request is processed independently of all other requests, and its processing state is captured in its own memory. A request constitutes a single thread of execution that executes entirely before it returns control back to the main task. While a task is executing, no other tasks are allowed to preempt the running task. Therefore, concurrency control is not needed. An executing task suspends itself and returns control back to the main task upon completion. Such techniques are not practical in OS-based applications since the OS manages and controls program execution. In BMC, an AO manages and controls its own execution. A task is executed, suspended, or completed as predetermined by the AO programmer and all memory bindings are static. Execution threads are also designed and implemented by the AO programmer. In BMC, a program is divided up into individual logical partitions so that each partition can be executed as a single thread of execution. The AO programmer is responsible for execution control and for optimizing CPU utilization to achieve higher levels of program performance.

2.9 No Local Persistent Storage

BMC applications do not use a local hard disk for storage and retrieval of persistent application data. Thus, there is no need to swap modules in and out of main memory. Using only main memory allows for a simpler AO design and eliminates the need for long-term storage and protection of persistent program data. In cases where persistent program data is required, it may be stored in network files or on a flash drive. With technology advances enabling increases in capacity of USB-based devices, it is possible to handle much larger BMC applications and persistent data. This approach provides a form of removable persistent storage.

2.10 Task Self-Management

Each bare machine AO has its own specific requirements for task creation and management as well as for communication between tasks. BMC programmers can create, execute, manage and terminate tasks within an AO. A pool of task

structures is created at compile time, and at run time, the task pool is inserted into stack structures. To make a task ready for execution, it is removed from the top of the task stack and inserted into a circular list of tasks. The tasks in this list are selected for execution on a First Come First Serve (FCFS) basis. When a task is completed, it is returned to the stack and can be reused as needed.

In addition to application-specific tasks, each AO has two fixed tasks: Main and Receive. The Main task begins to execute immediately when the AO starts and it continues to execute until the AO is terminated. This task is responsible for management of all tasks in the circular list. The Receive task handles incoming messages from the network. The Main and the Receive tasks have the highest priority and they are treated differently from tasks in the circular list.

AO programmers create all tasks except for Main and Receive. An AO may contain and manage many types of tasks. For example, an AO used for communication may include HTTP, SMTP, and audio tasks. The AO programmer has total control over these tasks and can execute, suspend and terminate them as needed.

The BMC paradigm uses the following novel and simple tasking strategy that results in optimum performance:

if (a task has productive work to do)
continue executing without any interruption
else
suspend and return control back to the Main task

A task in the circular list will not execute until its delay time has expired or an event occurs that causes the task to resume immediately ignoring the remaining delay time. The task self-delay approach coupled with the event-based resume function achieves optimal CPU scheduling and enables better task control and performance compared to OS-based systems.

2.11 Single-Threaded Execution

BMC threads are very different from those used in OS-based systems. We define a thread as the program code used to completely process an event or a request without any interruption. For example, if a message arrives, all the necessary processing is done in a single thread within the Receive task. This includes the Ethernet and any higher layer protocol handling as well as the application logic if relevant. When processing a message, appropriate changes are made to relevant data structures and state variables. State transition diagrams are used to manage task flow and track the impact of new events within a process. The single thread of execution simplifies the event processing logic since it does not require any complex concurrency control. Also, the single thread of execution is very efficient since a process is allowed to do productive work without being interrupted. From a security viewpoint, it is difficult to interrupt the single thread of execution in an AO to take control of a BMC system. The AO programmer designs and implements the necessary state transition diagrams and data structures required by each application within the AO.

2.12 Limited Interrupts

Conventional OS-based systems are interrupt-driven. Interrupts occur frequently and the OS gains control of the system to process each interrupt. In BMC, limited interrupts are used that are handled by the AO itself. Such interrupts and the necessary actions are explicitly defined during AO development.

2.13 Single Buffer Access

In traditional OS-based computing, data copying is minimized by means of intricate mechanisms to bypass the operating system kernel and memory channels. In BMC, a single buffer can be accessed by different protocol handlers within an AO without any constraints as there is no distinction between kernel and user space. Appropriate circular lists and data buffers are defined once and manipulated using pointers. Thus, a single buffer suffices for data when handling a message through various protocols during a single thread of execution.

2.14 Restricted Open Ports

Applications running on traditional OS-based computer systems are associated with open ports implemented by message queues. Open ports are a security risk and could be used by attackers to gain access to a system. In BMC, as in conventional secure applications, only a port (or ports) specific to an application are recognized and there is no possibility of any other ports being inadvertently opened. For example, if a bare machine system is running an email server and a Web server, then only ports 25 and 80 respectively are implemented. It is also impossible for an AO to open additional ports while executing.

2.15 Loosely Coupled User Interfaces

In conventional systems, a sophisticated user interface and the application code are intertwined and tightly coupled with the underlying OS. Such applications are susceptible to OS changes and require that they be frequently ported to new platforms. In contrast, BMC application code and its user interface are completely self-supporting. An AO uses a simple non-GUI command line and menu-driven interfaces. Thus, there is no dependence on specific system elements that precludes transferring an AO to a different system. For example, an email client application looks essentially the same for a desktop, laptop, hand-held or cell phone device even though screen size and resolution may vary on the target device. Decoupling the user interface from the application's code and simplifying its design enables applications to run on a variety of devices with very minimal changes. This produces stable applications that have a smaller code size and are able to run on a variety of devices with no changes, thus making BMC

more natural to pervasive computing.

2.16 Tightly Coupled Device Drivers

Traditional OS-based device drivers cannot be ported and used by BMC applications. These drivers will not work since system calls are specific to the underlying OS platform. Hence, device drivers that are specific to the BMC paradigm must be developed. Bare machine device drivers are tightly coupled with AO code since their API is part of the AO itself. However, the need for BMC-specific device drivers poses a challenge to developing applications that can run on alternate systems. This problem is mitigated to some extent by the use of USB device drivers that are easier to work with in a BMC environment.

2.17 Self-Managed Exception and Error Handling

In BMC, all necessary processing code to handle application exceptions and error conditions is placed within the AO and does not rely on any external software components. This provides for tighter control of AO operations while executing on a bare machine. In order to properly handle exceptions and error conditions, AO programmers are required to have a thorough understanding of possible application exceptions and error conditions, and be able to write appropriate code to process them.

2.18 Strict Memory Limits and Access Controls

Every application within an AO is given a base and an upper bound for its internal memory boundaries. Application memory boundaries are strictly checked and enforced by the Intel IA-32 hardware architecture facilities. Moreover, use of code segments, data segments, and stack segments by application programs is strictly controlled. Additionally, dynamic linking and loading of programs is not available and it is impossible to alter memory limits at run time.

2.19 Intertwining Protocols

Traditional systems use a layered protocol stack whose modules are part of the OS. Each layer corresponds to an individual protocol, and inter-layer communication is typically limited to adjacent protocol layers. A layered approach simplifies protocol design, but it is limiting and inefficient especially when lower layer information needs to be communicated to an upper layer or the application itself. It also introduces additional overhead when handling packet payloads. In contrast, BMC uses an intertwined protocol design within the AO that facilitates inter-layer communication between all protocols and enables any protocol to interact with the application. Processing of protocol headers are still done by separate protocol handlers, but all protocol layers and the application share the same copy of the payload, so there is no unnecessary passing of data between layers. Furthermore, the upper layer protocols are intertwined with the application. For example, in the Web server AO, part of HTTP processing is coupled with the TCP and IP protocols and the Ethernet layer due to using the same Receive task. The RTP, UDP, IP and Ethernet protocols used by the VoIP soft-phone application are similarly intertwined. During recording, voice data packets are directly copied from the microphone buffer into the Ethernet card (NIC) buffer, thereby eliminating unnecessary copying overhead. The RTP, UDP, IP, and Ethernet headers are directly added to the front of the payload by each protocol handler. The streamlined BMC protocol design approach enables applications to pass data between protocols in a very efficient manner.

3 A Bare Machine Application Design Methodology

Traditional software applications are developed using a methodology such as the waterfall or the spiral model. These methodologies require that application programmers have a complete understanding of the application domain, but a minimal understanding of the underlying OS platform usually suffices. The OS-abstracted layers hide the underlying platform so that application programmers do not have to understand the details of the middle and hardware layers. Moreover, these OS-based development platforms provide an API for tasks such as process management and control, network programming, and I/O operations.

In contrast, the BMC methodology requires that application programmers have both an understanding of the application domain as well as full knowledge of the underlying hardware resources and their bare machine interfaces. It allows programmers to create small and compact applications by providing them with full access to manage and control hardware resources. In particular, BMC programmers have direct access to the bare machine via these hardware APIs, and are provided with programming controls, data structures and other simple tools to build complex applications. However, while process control and resource management are now more efficient, AO programmers have to learn details of the hardware APIs in order to build applications. The BMC methodology has been used to develop web server, email server, and peer-to-peer VoIP soft-phone applications that run with no OS support.

The AO concept is fundamental to the BMC methodology (Figure 1). An AO is a self-contained program with its own application-specific logic that can manage itself and the available hardware resources. The AO-based BMC methodology facilitates software reuse. Bare machine AOs contain many reusable components such as the boot and load program, Main and Receive tasks, task management and control, network protocols, and the memory map. Transferring an AO from one domain to another one could be as simple as adding application-specific code to an existing code base from another AO.

Creation of standard API sets for bare machine components has been investigated. Yet, providing shared services to diverse applications through APIs requires that bare machine component functionalities be isolated, and that they include independent execution and process controls. These requirements conflict with the BMC goal of avoiding shared services and process controls. Moreover, they lead to

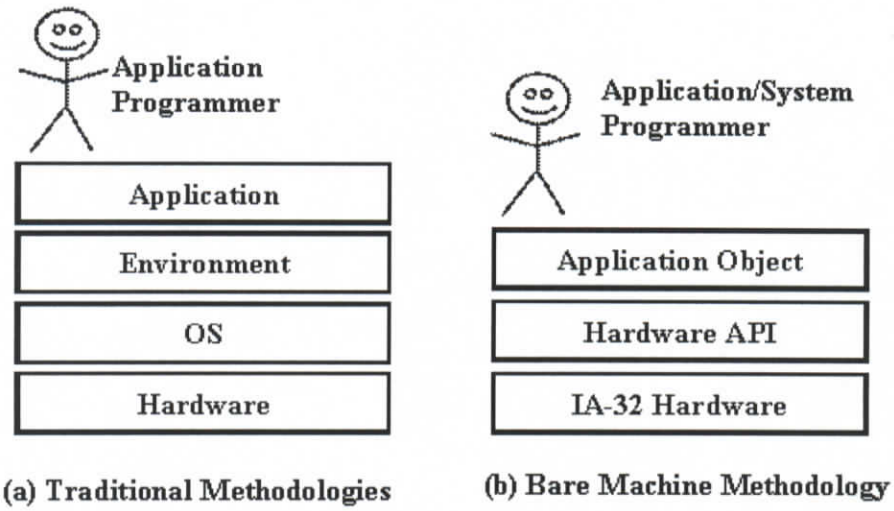

Figure 1: Bare machine computing application object

centralized resource management and controls found in a traditional OS. Avoiding a centralized OS is a key feature of the bare machine application development methodology and thus a restricted API containing only the essential elements for building an AO is provided.

The BMC methodology is derived from the Dispersed Operating System Computing (DOSC) and the Application Object (AO) concepts. Although it has several advantages, the BMC methodology presents programmers with many challenges when designing, coding, and testing applications. There is no OS support or software development environment, and minimal programmer tools are available. BMC compile, boot, load and execute phases are discussed next.

The code for a typical BMC application may be viewed as having the following components: (1) code shared by many applications that can be used as is or with minimal modifications; (2) code for hardware interfaces that is device specific; (3) code for a user interface that is device specific; (4) code for device drivers. Thus, the steps in compiling, linking, and executing BMC applications can be illustrated as shown in Figure 2.

BMC applications are written entirely in C++. The compiling environment uses batch files to compile and link the application. Visual Studio C++ compiler (batch mode), MASM 6.11 assembler, and Turbo assembler compilers are used to create executable modules. We have written batch files to do compilation and linking for boot and loader programs and the BMC application. All command files are executed in the root directory for the BMC application. For a bare VoIP softphone application, the code components are as shown in Table 1.

In order to load and execute the bare machine application, the boot program, the loader program, and the application are placed on a bootable device such as a flash drive. The boot program enables the BMC application to be executed after booting is completed. Standard PC boot procedures are followed and the PC is powered up with the boot device in the boot drive. During the boot process, the loader program will load an interface menu into internal memory for execution. Using this interface menu, the user can load the bare machine softphone application from the same boot device and then execute it.

The rest of this section presents the following key elements in bare machine application design and development using AOs: 1) design trade-offs, 2) design requirements, 3) reuse, 4) tasking structure, 5) execution threads, 6) application behavior modeling, 7) objects and their relations, and 8) memory map.

3.1 AO Design Trade-offs

AO programmers make these early design decisions and trade-offs with respect to an AO:

1) **Small vs. Large AO**: An AO is composed of one or more applications. For example, email and web browser applications can be bundled into a small AO. Adding a VoIP soft-phone application to this AO may unnecessarily complicate the design. Alternatively, if the VoIP soft-phone is not concurrently used with email and web browser applications, then two smaller AOs can be designed. Such trade-offs will allow for design of AOs that are much simpler and still meet application requirements.
2) **General vs. Specific AO:** Typical generalized OS-based software applications designs require the support of complex OS code. The supporting OS may also generate many processes and create additional overhead requiring large amounts of main memory. The BMC methodology, in contrast, promotes simple and specific application designs sufficient for meeting specific user requirements. For example, an AO programmer could create a specific self-supporting design for an email server to handle a

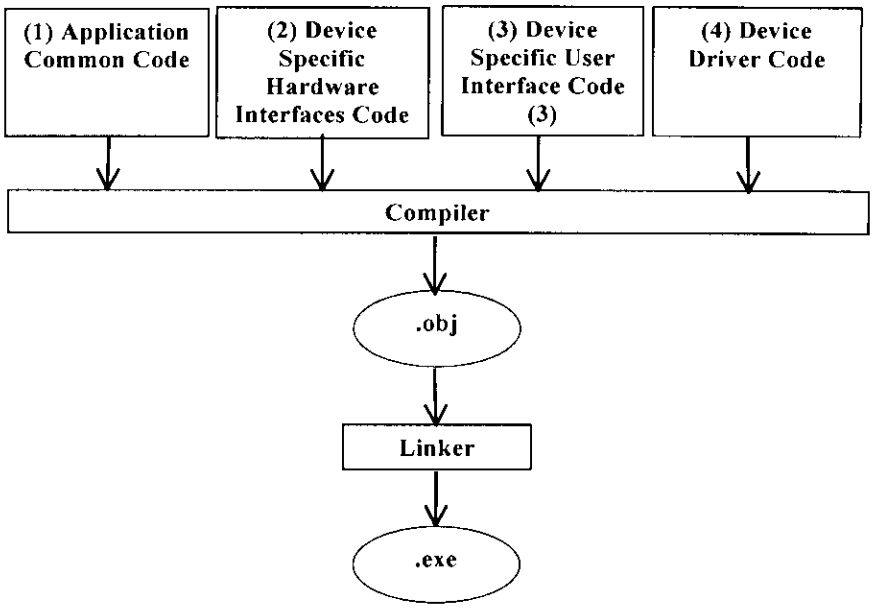

Figure 2: Compile, link, and execute in BMC

Table 1

\VoIPSec	*Root directory of the softphone*
\aes	*AES code*
\arp	*Address Resolution Protocol code*
\audio	*Audio card drivers code*
\bin	*Compiler and linker executables*
\dosclib	*DOSC object files*
\ethernet	*Ethernet protocol code*
\G711Codec	*G.711 codec code*
\hshake	*Handshake code*
\interfaces	*DOSC interface files*
\ip	*Internet Protocol code*
\jitter	*Jitter buffer code*
\MASS	*Assembler executable*
\memorymap	*DOSC memory map files*
\rp	*Record and playback code*
\rsa	*RSA key generation code*
\rtp	*Real-Time Transport Protocol code*
\sha1	*Sha1 code*
\tcp	*TCP code*
\udp	*UDP code*
\vsec	*VoIP security code*
\webserver	*C++ main and tasking code*

limited set of user IDs. By placing a limit on the user ID set, the programming logic for indexing email tables becomes simpler while still fulfilling the user requirements.

3) **Open vs. Closed AO:** Application design of traditional open systems involves careful evaluation in order to ensure that the necessary APIs meet all requirements. This results in a more complex application design that requires more development efforts and time. Alternatively, a closed-system AO may be sufficient and preferable for some application domains.
4) **Simple vs. Complex AO:** To be efficient, traditional methodologies utilize complex data structures and algorithms that in turn require more development time. A simpler and more efficient AO can be designed instead that uses more internal memory and simpler indexing techniques. In BMC methodology, more internal memory is available since only one AO is executing at any time.

3.2 AO Design Requirements

Careful identification of AO design requirements are critical for any bare machine application. During AO design, it is necessary to address the following elements:

1) **Main Memory Size:** This is the size of RAM on a personal computer for loading and executing an AO. An AO could use a small or large portion of the main memory.
2) **Static Memory Size:** This is the size of main memory used for storage of application code, static data, and the tasking stack. The application code size is the size of the AO executable image generated by the C++ compiler,

while the static data size is the total memory space required by the AO data structure. The tasking stack size is in the range of 8-64 KB for every task in an AO.

3) **Dynamic Memory Size:** This is the size of main memory for run-time allocation of data structures and temporary variables required by an AO. The remaining main memory, after setting aside space for static memory, is used as dynamic memory. The dynamic memory area is divided up into portions with each portion assigned to specific AO data structures requiring dynamic memory allocation.
4) **Number of Necessary Tasks:** An AO may be composed of many applications, and each application may have several tasks. As discussed earlier, the tasks required by an AO are placed into a task pool and managed by the AO during execution.
5) **AO Application Set:** The set of applications that are bundled into a single AO must be defined at design-time. Usually, the number of user applications ranges from between 1 to 10.
6) **User Interfaces and Interactions:** The user interactions with an AO could include system power-on and boot-up, program load and execution, program execution intercept, user input and output, program termination, and system power-down. The necessary user-AO interactions may be implemented by user interface elements including: menus, error screens, debugging screens, backup files and data logging.
7) **Persistent Storage:** BMC methodology does not require a hard disk. However, bare machine applications could use detachable persistent devices such as a flash or a CD drive as well as any available network storage devices. AO designers must define the persistent storage requirements. Also, before terminating an AO, there may be a need for backup of data from main memory onto a persistent storage device.
8) **I/O Devices and Drivers:** BMC uses native I/O device drivers that are tightly coupled with the AO enabling it to directly control device operations. Using native device drivers limits the proliferation of device types and drivers, and reduces the chances of bugs that may occur when using device driver code from different hardware vendors.

3.3 Reusing Existing AOs

There is no need to develop each new bare machine AO from the ground up; existing bare machine AOs can be adapted if needed, and reused. AOs contain many reusable components such as boot and load, Main and Receive tasks, tasking structure and controls, network protocols, and the memory map. Transforming an AO or its basic components from one domain to another related one is a relatively straightforward process. For example, many elements of the Web server AO are reused as is or with minor modifictions in the email server AO. A bare machine Web server provides services to web clients using the HTTP protocol via a related state transition model and flow controls. Likewise, an email server provides services to email clients using the SMTP protocol; thus, these two bare machine servers have many elements that are common or easily transformed from one server to the other. Yet, the bare machine web server and email server also have several features that are unique to each application. It is easy to combine two or more bare machine applications into a single bare machine application that is then encapsulated within a single AO.

3.4 Defining AO Tasking Structure

In addition to having their own specific tasks, all AOs share two common tasks: Main and Receive (Figure 3). The Main task is responsible for management of AO specific tasks, and the Receive task is responsible for checking and processing packets from the network. An AO may contain many applications, and each application may contain many task types (groups). Each task type may include one or more copies of the same tasks. A task performs a specific function. Task types are created at compile-time and are placed in their own specific Idle Task list.

Bare machine tasks have a very similar design making them reusable across different applications. Each task type has a specific stack data structure. AO programmers define the necessary task types and the maximum number of tasks per task type. Each task is implemented using a specific task object method, which is in essence a continuous while-loop processing application events. The loop implements the state transition diagram for the task to control the execution of application threads.

Task activation is triggered by an application event. To activate a task, it is taken from an appropriate Idle Task list and placed into the Active Task list to handle the event. The tasks in the Active Task list are processed in FCFS order. When a task is executing, it will run until it is suspended, or it completes, at which point control returns to the Main task. A suspended task is placed in the back of the Active Task list with some delay. When a task delay expires, or when an event enabling task resumption occurs, the task becomes ready to run again. When a task is completed, it is moved into the back of the Idle Task list.

The bare machine tasking structure described here is unique to BMC, and it is usable for all types of bare machine AO development. Pre-determined task execution and control decisions are made by the AO programmer based on the needs of the specific application. AO programmers control all aspects related to the scheduling, suspension, and resumption of tasks.

3.5 Identifying AO Threads of Execution

A bare machine thread of execution is very different from execution threads used in conventional OS-based systems. A bare machine thread of execution contains the required processing logic to handle a unique function. It has a starting point and an ending point that defines the boundaries of the associated processing code for the thread. A thread is executed without interruption from its starting point to its ending point.

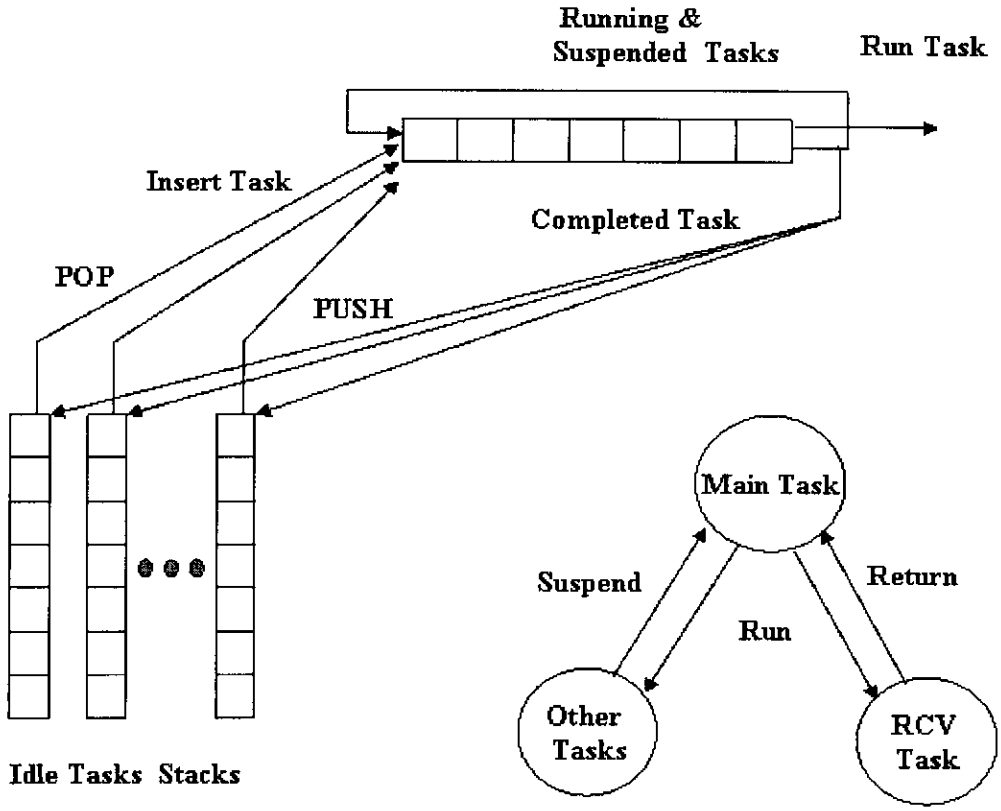

Figure 3: Bare machine tasking structure

Bare machine AOs contain the necessary task controls, whereas OS-based software, process and thread controls reside inside the OS. Thus, task scheduling and control is fundamentally different in BMC than in an OS-based system, since task execution and management is done within the AO itself. This eliminates the need for centralized resource and process management and the related multiplexing of hardware resources.

3.6 Modeling AO Behavior

A careful review of any bare machine AO reveals that application code is placed into two distinct sections: a) application behavior (state transition) logic; and b) application threads of execution. Application behavior logic determines its threads of execution.

In conventional software development methodology, application behavior is modeled at the application layer with no consideration for any ongoing execution controls inside the underlying OS. In contrast, in the BMC approach, application behavior is modeled to directly control and manage bare machine hardware resources. Application behavior can be modeled by copying and modifying the state transition diagrams for the AO. Alternatively, one may simply adapt AO code from an existing AO as needed.

The state transition diagram for a typical bare machine application has many state variables and processing states. A new event causes a transition from the current state to the next, and the state transition model drives the execution logic of the bare machine application. In essence, this logic determines when tasks are executed based on changes made to the state variables.

3.7 Identifying Objects and Relationships

At present, bare machine applications are implemented using the C++ programming language. The network protocols needed by a given application are implemented as objects that are part of the application. Every object must provide necessary interfaces for use by other objects. For example, as will be discussed in detail below, for the VoIP soft-phone AO, its Record and Play objects use the Audio object interfaces to control and manage microphone recording and speaker playback functions. A new AO is created by developing new specific objects in addition to reusing existing objects from other AOs when possible. Bare machine programmers must be experienced with object-oriented programming concepts and techniques in addition to having a thorough understanding of existing bare machine objects and their interfaces.

3.8 Defining AO Memory Map

The memory map for an AO specifies how main memory is used. This memory map is customized for each AO based on its static and dynamic memory requirements. It includes space for static data, stacks, and code as well as for dynamic data. Memory is allocated at compile-time and validated at run-time. An AO uses real memory and it thus does not require any virtual memory/paging and the associated complex logic of an OS-based system. AO programmers are responsible for defining boundaries for real and protected memory. Persistent or external memory, if needed, requires a USB or a network mass storage device. For example, interfaces to USB flash memory are available and can be included in an AO if needed.

4 An Example BMC Application: The Soft-Phone AO

We now illustrate how the BMC approach and its application development methodology may be used to design a specific bare machine application. We have chosen a peer-to-peer VoIP soft-phone client application as it exemplifies many of the unique BMC characteristics discussed in the previous sections.

The VoIP soft-phone application inputs digitized voice from a microphone, encapsulates encoded voice in network packets, and transmits the packets over the IP network. It also inputs voice packets from the IP network, decapsulates and decodes each voice packet and outputs voice data for playback to the speakers. Figure 4 shows the required architectural elements of the BMC soft-phone client in more detail.

Voice is input through the microphone, digitized at PCM rate and placed in the microphone buffer. Next, voice is compressed using a codec (G.711 in this case), and then RTP, UDP, IP, and ETH headers are added to form voice packets that are placed in the Ethernet send buffer for transmission on the IP network.

Conversely, when a voice packet is received, it is placed in the Ethernet receive buffer. After network protocol headers for ETH, IP, UDP and RTP are processed and removed, the voice payload is stored in the jitter buffer. Lastly, voice packets are decompressed and placed in the speaker buffer for playback.

The general requirement is to design, implement, and test a

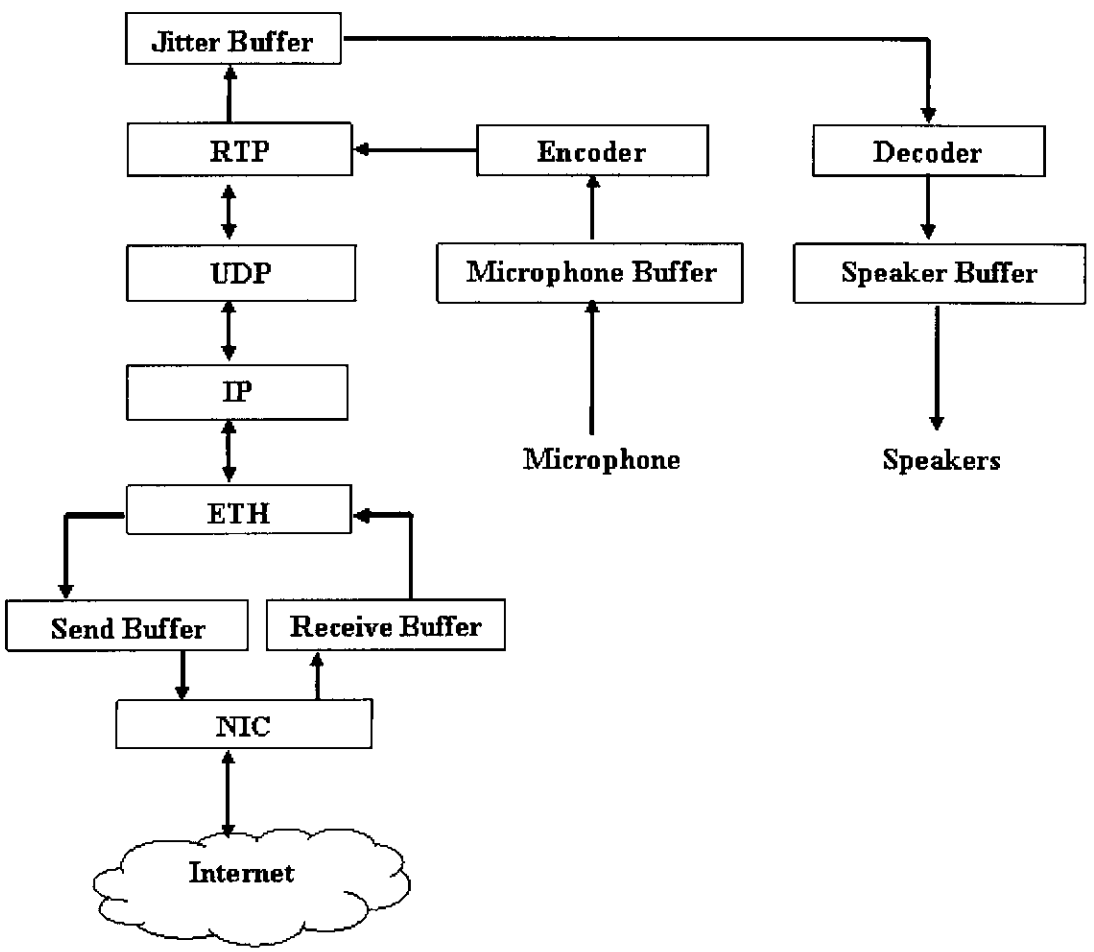

Figure 4: Soft-phone architectural elements

peer-to-peer VoIP soft-phone application that can run on an Intel IA-32 based processor without an operating system, while providing acceptable voice quality. It must be able to record, compress and transmit voice packets over the IP network, as well as receive, decompress, and playback voice packets from the IP network. Specific soft-phone requirements are:

1) Run on Intel IA-32 processors without an operating system
2) Operate in a peer-to-peer mode
3) Provide acceptable voice quality
4) Manage the microphone recording and speaker playback
5) Use a PCM codec to compress and decompress voice data
6) Use the BMC network protocols to send and receive voice packets
7) Communicate with an OS-based soft-phone
8) Operate on a LAN or on the Internet while maintaining acceptable voice quality.

4.1 Soft-Phone AO Design Trade-Offs

Based on the requirements of the soft-phone application, and the BMC approach discussed earlier, the design trade-off decisions are as follows:

- **Small vs. Large AO:** We will create a small AO containing a single soft-phone application.
- **General vs. Specific AO:** Initially, the BMC peer-to-peer soft-phone client is only required to provide basic VoIP communication capabilities and a specific AO suffices. Since call set-up is done only for a small number of bare machine clients in a customized peer-to-peer fashion, the SIP protocol is not needed [12-13]. Similarly, this version of the soft-phone provides limited security using AES encrypted voice and a simple handshake to exchange keys [11].
- **Open vs. Closed AO:** This soft-phone client is a closed AO and it is not required to provide any API.
- **Simple vs. Complex AO:** We will design a soft-phone client AO with minimal functionality. It uses simple buffer management.

4.2 Soft-phone AO Design Requirements

The design requirements for the soft-phone AO are as follows:

- **Main Memory Size:** The soft-phone AO is the only application within the AO and therefore it consumes all internal memory.
- **Static Memory Size:** The size of static memory for the soft-phone application is the size of the AO executable image generated by C++ compiler.
- **Dynamic Memory Size:** The static memory size is subtracted from the total internal memory size to determine the dynamic memory size for the soft-phone application. Dynamic memory area will be partitioned, with each partition assigned to a specific soft-phone AO data structure that requires dynamic memory.
- **Number of Necessary Tasks:** The soft-phone AO requires four separate tasks:

  1. **Main Task:** for management and scheduling of the other soft-phone tasks.
  2. **Handshake Task:** for establishment of a TCP/IP handshake and exchange of an AES key.
  3. **Receive Task:** for handling incoming voice packets.
  4. **Audio Task:** for management of microphone recording and speaker playback.

- **AO Application Set:** The soft-phone AO contains a single soft-phone application.
- **User Interfaces and Interactions:** The user interacts with the soft-phone AO through keyboard and menu selections, and simple text is output to the display.
- **Persistent Storage:** The soft-phone AO does not require any hard disk.
- **I/O Devices and Drivers:** The soft-phone AO will use existing device drivers for the audio card, keyboard, monitor, and the NIC card.

4.3 Reusing Existing AO

Existing bare machine AOs were reviewed in order to select a bare machine AO which best fits the design requirements of the soft-phone AO. It was found that the following elements of the Web server AO can be reused by the soft-phone application: boot and load program; main, receive and handshake tasks; ETH and IP protocols; keyboard and display drivers. In addition, the following new elements are needed for the soft-phone AO: codec; audio card driver; audio task; UDP and RTP protocols.

4.4 Soft-phone Tasking Structure

The soft-phone tasking structure and strategy enables optimal CPU scheduling, which has been shown to improve intrinsic jitter and delay, i.e., jitter and delay due to internal soft-phone processing [12-13]. Figure 5 shows the soft-phone tasking structure and relationships:

1) **Handshake Task:** responsible for the establishment and termination of the TCP/IP handshake used for call set up and exchange of an AES encryption key.
2) **Main Task:** responsible for management of the Receive and Audio tasks.
3) **Receive Task:** responsible for processing new network packets and transferring these packets from the NIC (Ethernet) buffer to the jitter buffer.
4) **Audio Task:** responsible for moving voice data from the microphone buffer to the NIC buffer, as well as moving voice packets from the jitter buffer to the speaker buffer.

The optimal tasking strategy for the soft-phone AO works as follows: The Handshake task will execute once during call

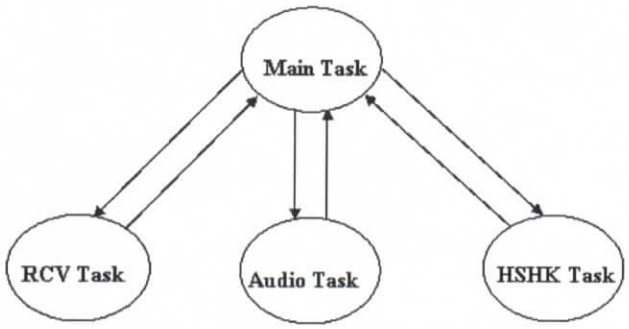

Figure 5: Soft-phone tasking structure

set up to exchange an AES key between two peer soft-phones before returning to the Main task. The Receive task will return to the main task after processing a newly arrived network packet. Similarly, the Audio task will return to the Main task after recording or playing voice data. Upon activation, each task is allowed unlimited CPU time. When a task is not doing useful work, it is suspended and returns control back to the Main task. When suspending itself, a task requests to be scheduled at a future time by specifying a delay. The Main task executes a task if and only if its requested delay has expired.

4.5 Soft-phone Threads of Execution

The processing performed within each soft-phone task is partitioned into different threads of execution for better design of the soft-phone logic and to facilitate its control. The threads of execution for the soft-phone tasks are as follows:

1) **Handshake Task:** two threads of execution; the first thread is used to act as a server to generate a random AES key, sign and send it to the other soft-phone client; and the second thread is used to act as a soft-phone client to receive, authenticate and save the AES key.
2) **Main Task:** requires one thread of execution to schedule, activate and execute other soft-phone tasks.
3) **Receive Task:** one thread of execution to check for arrival of new network packets, process the protocol headers, and place voice packets into the jitter buffer.
4) **Audio Task:** two threads of execution; the first thread (the recording) is responsible for moving recorded voice data from the microphone buffer to the NIC buffer; and the second thread (the playback) is responsible for moving received voice data from the jitter buffer to the speaker buffer.

4.6 Modeling Soft-phone AO Behavior

Figure 6 depicts the soft-phone AO behavior model including boot up, program load and execution, as well as tasking and threads. The user inserts the bootable flash-drive containing the soft-phone AO into the PC and reboots the machine. The user menu will then be displayed enabling selection of the soft-phone application.

After the soft-phone AO is loaded from flash-drive into internal memory, the user runs the soft-phone application and starts the Main task. The Main task checks for incoming network packets and starts the Receive task to process them. The Receive task does the Ethernet, IP, UDP, and RTP processing, and places the voice frames in the jitter buffer. The Main task starts the Audio task, which requests a frame from the jitter buffer, and does the G.711 decoding, and writes the decoded frame to the speaker buffer. It also checks for voice frames in the microphone buffer, does the G.711 encoding followed by the relevant protocol processing. It then writes the encapsulated voice data to the NIC buffer for transmission on the network. The coupling of recording and playback functions via the Audio task is done for simplicity. The AO may be easily modified to decouple these functions.

4.7 Soft-phone Objects and Relationship

The soft-phone C++ code uses a set of interacting objects. Every object maintains its own local state and provides interfaces for communication with the outside world. For example, the Audio object is responsible for managing the operations of the audio codec. It provides public methods that the soft-phone uses to setup the audio (G.711) codec, control its recording and playback operations, and read and write voice data.

The object diagram shown in Figure 7 illustrates the soft-phone objects and their relations. The objects are as follows:

- **G.711:** handles voice compression and decompression
- **Audio Object:** implements the audio codec, and is also concerned with managing the recording and playback functions
- **Jitter:** implements the jitter buffer and its operations
- **RP:** manages the recording and playback threads of the soft-phone
- **APPTASK:** implements the bare machine tasking structure
- **Hshake:** implements the handshake for exchange of the AES key during call set up between two client soft-phones
- **RTP:** implements the RTP protocol
- **Vsec:** deals with voice security
- **TCP, ARP, UDP, IP, ETH:** implement the respective network protocols
- **RSA:** implements the RSA algorithms
- **AES:** implements AES encryption and decryption algorithms

4.8 Soft-phone AO Memory Map

The memory map for the Web server AO is customized for use by the soft-phone AO based on its dynamic memory requirements (Figure 8). It includes partitions for dynamic allocation of jitter buffer elements, microphone and speaker buffers, and RSA and AES encryption algorithms.

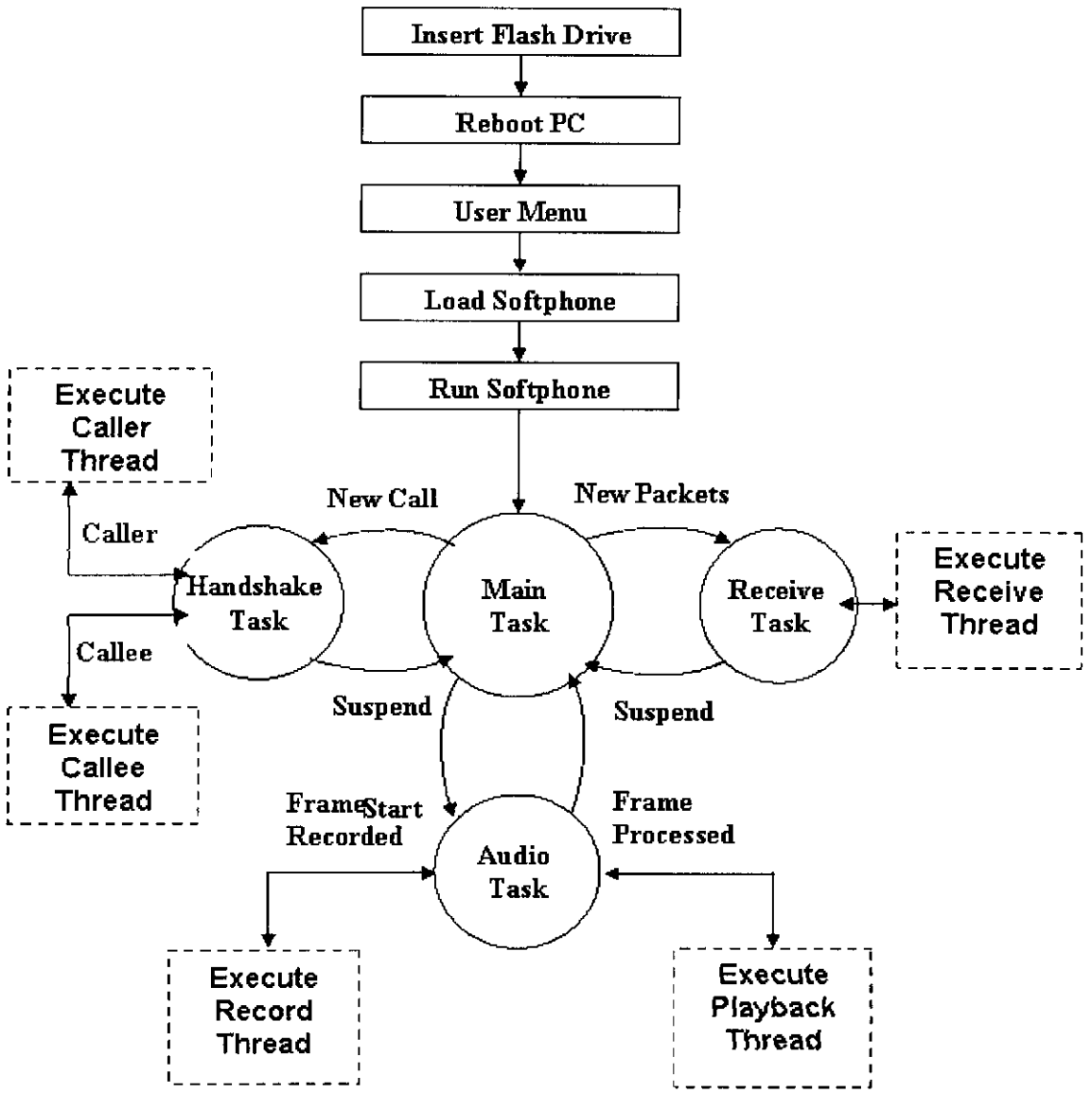

Figure 6: Soft-phone AO behavior model

5 Related Work

The BMC software application development methodology introduced in this paper is derived from the Dispersed Operating System Computing (DOSC) paradigm [10]. However, the focus of the DOSC paper was on building a Web server, it did not systematically identify the general characteristics of BMC or provide a methodology for BMC application development. Although our BMC methodology relies on the Application Object (AO) concept [8], the scope and applicability of this work go well beyond the basic ideas presented there. In the process of developing various AOs, techniques were devised to directly communicate with the hardware [9].

Many applications have been developed using the BMC methodology including a Web Server [5-7], a VoIP Soft-phone [12-13], and an Email Server [4]. In addition, numerous studies to investigate the performance of different BMC applications and protocols have been conducted, including [5, 6, 11, 13, 17].

All of these earlier studies focus on a single BMC application, and those that discuss design and implementation of a BMC application focus on that one application. In contrast, the software development methodology described in this paper is applicable to any (existing and future) BMC application. Furthermore, we have detailed the relevant design tradeoffs, requirements and steps, which have not been addressed in a general context in any of the earlier work. For

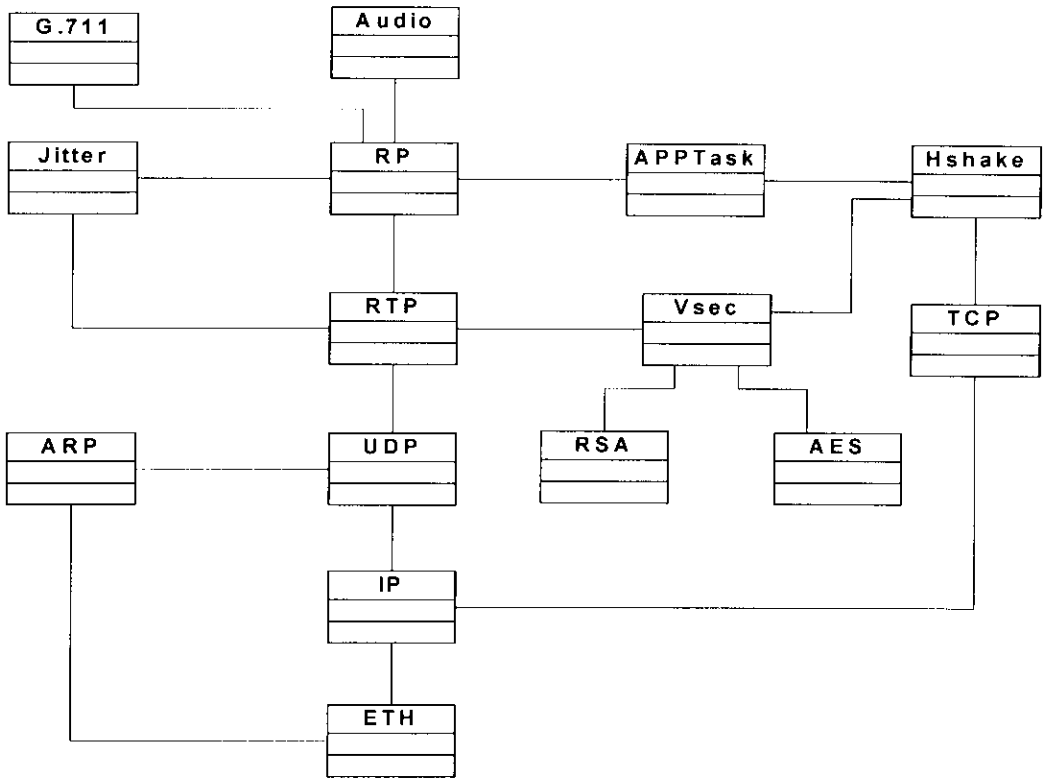

Figure 7: Soft-phone AO objects and relations

C++ type	Memory Constant	Memory Location	Description
const long	TASK_LIST	0x04d10000	
const long	TASK_STACK1	0x04d20000	
const long	TASK_STACK2	0x04d30000	
const long	JT_BaseAddr	0x04d60000	Jitter Table Base Address
const long	JB_BaseAddr	0x04d70000	Jitter Buffer Base Address
const long	Spkr_BDLPointer	0x04f50000	Speaker Buffer Pointer Table
const long	Mic_BDLPointer	0x04f60000	Microphone Buffer Pointer Table
const long	Spkr_BDLDataPointer	0x04f70000	Speaker Buffer Pointer
const long	Mic_BDLDataPointer	0x05170000	Microphone Buffer Pointer
const long	RSA_AREA	0x1100cb40	RSA Memory
const long	AES_AREA	0x1110cb40	AES Memory
const long	TCB_TRACE	0x0530000;	
const long	TCP_MSG_ADDR;	0x062608a0	
const long	FILE_ADDR	0x08100000	
const long	STACK_ADDR	0x10000000	

Figure 8: Soft-phone AO memory map

example, in [12] (and [13] to a limited extent), design and implementation issues relevant to a bare PC soft-phone are presented, but here we have discussed the soft-phone AO from the viewpoint of illustrating the general software application development methodology that is the focus of this paper. Likewise, in [5] and [7] design issues are discussed focusing only on the Web server application.

Many attempts have been made to eliminate OS abstractions, or provide a lean OS environment [1-3, 14-16]. While these approaches have the general goal of reducing OS overhead, none completely eliminate the OS, kernel or Exokernel layer, as is done in BMC. To the best of our knowledge, there are no applications today that run directly over the hardware without at least minimal support from a lean OS or minimal OS kernel (such as a customized Linux kernel). It should be noted that virtual machine (VM) systems that provide direct access to the hardware still require a base OS (or kernel) that provides this access. BMC differs from all related approaches in that it is characterized by a single application-centric monolithic executable (i.e., the AO) that is in complete control of the hardware. Moreover, there are no OS elements or other software present in the bare machine when an AO is running. In essence, in BMC, one and only one AO runs at any point in time on a bare machine.

6 Conclusion

This paper presented a novel bare machine application development methodology based on the bare machine computing (BMC) paradigm and the Application Object (AO) concept. Unlike traditional methodologies that rely on an OS platform, the BMC methodology enables the design and implementation of software applications that are totally independent of any OS, kernel or Exokernel platform. The new methodology requires AO programmers to manage hardware resources, while utilizing optimal tasking structures, intertwined lean protocols and bare machine-specific device drivers for application development. Many bare machine applications that execute directly on the bare Intel IA-32 based processor have been previously developed using this methodology. We described the BMC approach and compared it with traditional application development methodologies; detailed the steps involved in the bare machine application development methodology; and illustrated the use of this methodology by considering a bare machine soft-phone application. The BMC development methodology can be used to build applications that run on the hardware with no OS or kernel support.

Acknowledgment

The authors sincerely thank NSF for partial support of bare machine computing via SGER[1] and REU[2] grants. We also express our gratitude to the late Dr. Frank Anger, for his support of early BMC research.

[1] NSF SGER Grant CCR-0120155.

[2] NSF REU Grant 0353868.

References

[1] D. R. Engler and M.F. Kaashoek, "Exterminate all Operating System Abstractions," *Fifth Workshop on Hot Topics in Operating Systems*, USENIX, Orcas Island, WA, p. 78, May 1995.

[2] B. Ford, and R. Cox, "Vx:32: Lightweight User-Level Sandboxing on the x86," Proceedings of the *USENIX Annual Technical Conference*, USENIX, Boston, MA, June 2008.

[3] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullman, "Interface and Execution Models in the Fluke Kernel," *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, USENIX Technical Program, New Orleans, LA, pp. 101-115, February 1999.

[4] G. H. Ford, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi, "The Design and Implementation of a Bare PC Email Server," *33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009)*, Seattle, Washington, pp. 480-485, July 2009.

[5] L. He, R. K. Karne, and A. L. Wijesinha, "Design and Performance of a Bare PC Web Server," *International Journal of Computer and Appl.*, Acta Press, 15:100-112, June 2008.

[6] L. He, R. K. Karne, A. L. Wijesinha, and A. Emdadi, "A Study of Bare PC Web Server Performance for Workloads with Dynamic and Static Content," *The 11th IEEE International Conference on High Performance Computing and Communications (HPCC-09)*, Seoul, Korea, pp. 494-499, June 2009.

[7] L. He, R. K. Karne, A. Wijesinha, S. Girumala, and G. Khaksari, "Design Issues in a Bare PC Web Server," *Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'06)* pp. 165-170, 2006.

[8] R. K. Karne, "Application-Oriented Object Architecture: A Revolutionary Approach," 6th International Conference, HPC Asia 2002, Center for Development of Advanced Computing, Bangalore, Karnataka, India, December 2002.

[9] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "How to run C++ Applications on a Bare PC," SNPD 2005, *Proceedings of SNPD 2005, 6th ACIS International Conference*, IEEE, pp. 50-55, May 2005.

[10] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "DOSC: Dispersed Operating System Computing," *OOPSLA '05, 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications, Onward Track*, ACM, San Diego, CA, pp. 55-61, October 2005.

[11] G. H. Khaksari, A. L. Wijesinha, and R. K. Karne, "Secure VoIP using a Bare PC," *Proceeding of IEEE NTMS*, Cairo, Egypt, pp. 1-5, 2009.

[12] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, and Q. Yao, "A VoIP Softphone on a Bare PC," *Proceedings 2007 Worldcomp ESA*, Las Vegas, Nevada, USA, pp. 55-61, 2007.

[13] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, "A Peer-to-Peer Bare PC VoIP Application," *Proceedings of the IEEE Consumer and Communications and Networking Conference*, IEEE Press, Las Vegas, Nevada, pp. 803-807, January 2007.

[14] *The OS Kit Project*, School of Computing, University of Utah, Salt Lake, UT, June 2002, http://www.cs.utah.edu/flux/oskit.

[15] Tiny OS, *Tiny OS Open Technology Alliance*, University of California, Berkeley, CA, 2004, http://www.tinyos.net/

[16] T. Venton, M. Miller, R. Kalla, and A. Blanchard, "A Linux-Based Tool for Hardware Bring Up," Linux Development, and Manufacturing," *IBM Systems J.*, IBM, NY, 44(2):319-330, 2005.

[17] R. Yasinovskyy, A. L. Wijesinha, R. K. Karne, and G. Khaksari, "A Comparison of VoIP Performance on IPv6 and IPv4 Networks," The 7th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA-2009), Rabat, Morocco, May 10-13, 2009

Gholam H. Khaksari is an Assistant Professor in the Department of Information Sciences and Technology at Penn State University, Abington College. Prior to that, he spent more than two decades in the software development industry. Dr. Khaksari received his D.Sc. in Applied Information Technology from Towson University, M.S. in Computer Science from Johns Hopkins University, and a B.S. in Computer Science from Youngstown State University. His area of research includes VoIP, bare machine computing, database and expert systems.

Ramesh K. Karne is a Professor in the Department of Computer and Information Sciences at Towson University. He obtained his Ph.D. in Computer Science from George Mason University. Prior to that, he worked with IBM at many locations in hardware, software, and architecture development for mainframes. He also worked at the Institute of Systems Research at the University of Maryland, College Park as a research scientist. His research interests are in bare machine/dispersed operating system computing.

Alexander L. Wijesinha is an Associate Professor in the Department of Computer and Information Sciences at Towson University. He holds a Ph.D. in Computer Science from the University of Maryland Baltimore County, and both a M.S. in Computer Science and a Ph.D. in Mathematics from the University of Florida. He received his B.S. in Mathematics from the University of Colombo, Sri Lanka. His research interests are in computer networks including wireless networks, VoIP, network protocol adaptation for bare machines, network performance, and network security.