TOWSON UNIVERSITY
OFFICE OF GRADUATE STUDIES

BARE MACHINE COMPUTING ON ARM DEVICES

by

Alexander Peter

A Dissertation

Presented to the faculty of

Towson University

in partial fulfillment

of the requirements for the degree of

Doctor of Science in Information Technology

Department of Computer & Information Sciences

Towson University
Towson, M aryland 21252

(May, 2013)

TOWSON UNIVERSITY
OFFICE OF GRADUATE STUDIES

DISSERTATION APPROVAL PAGE

This is to certify that the dissertation prepared by Alexander Peter entitled "BARE MACHINE COMPUTING ON ARM DEVICES" has been approved by the thesis committee as satisfactorily completing the dissertation requirements for the degree of Doctor of Science in Information Technology.

Dr. Ramesh K. Karne Chair, Thesis Committee	Signature	4-19-2013 Date
Dr. Alexander L. Wijesinha Committee Member	Signature	4/19/13 Date
Dr. Sungchul Hong Committee Member	Signature	4/19/2013 Date
Dr. Ziying Tang Committee Member	Signature	4/19/2013 Date
Dr Janet Delany Dean of Graduate Studies	Signature	4/29/13 Date

ACKNOWLEDGEMENTS

I would like to express my appreciation to all those who have supported my efforts to complete this dissertation. I am greatly appreciative of my research committee Dr. Ramesh K. Karne (chair), Dr. Alexander Wijesinha, Dr. Sungchul Hong, and Dr. Ziying Tang for supporting this research. I am especially thankful to Dr. Karne and Dr. Wijesinha for investing long hours in the lab mentoring, troubleshooting, framing research methods and advising me throughout this research. Also, I am thankful to my family, friends and colleagues who have motivated and heartily supported my academic goals during the long period of my doctoral study especially my father.

I also would like to thank Dr. Chao Lu, Chair of the Department of Computer and Information Sciences, and Towson University for facilitating this research. Gratitude is also given to the late Frank Anger (National Science Foundation) for his support of the Application oriented Object Architecture, which evolved into Bare Machine Computing research and this dissertation.

ABSTRACT

BARE MACHINE COMPUTING ON ARM DEVICES

Alexander Peter

This thesis extends on-going Bare Machine Computing (BMC) research at Towson University. BMC applications run on a bare machine without any operating system, kernel, or other centralized support. Many complex applications were designed and implemented on x86 processor using BMC concept. This research consists of applying BMC concept to pervasive devices and their underlying architecture. In particular, it investigates implementing bare machine computing on ARM processors, which are commonly used in many systems including: smart phones, GPS, sensor networks, tablets, gaming and multimedia electronics.

Applying BMC concept to pervasive devices poses many challenges. Some of these are addressed in this dissertation. To investigate these issues, a graphics application running on x86 PC is transformed to run on an ARM device. A transformation methodology is developed to map x86 applications to ARM processor. This process addresses many design issues including: boot program, loader, device drivers for ARM and its development environment. It was discovered that ARM architectures are more BMC friendly than x86 processors as they are most commonly used in handheld devices and embedded systems. The transformation process identified that majority of the code can be mapped to run on ARM with minor modifications. It also notes that it is possible to write parts of the program that can run on pervasive devices.

The dissertation further explored construction of a simple temperature sensor device to run on bare ARM board with buzzer, graphics and touch screen. This study resulted in

understanding the BMC application development process on ARM processors. It also discovered that it is possible to write software for one type of architecture and transform it to run on another with minimal code changes. That is, it addresses the heterogeneity that exists in today's pervasive devices and proposes a BMC paradigm to solve their design issues. The dissertation concludes that it is possible to write common software that runs on many pervasive devices.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SECTION I

MOTIVATION

We have developed a variety of bare PC applications that run on a machine without using any operating system (OS), or kernel, or any embedded system. The bare PC architecture is simple, application-centric, extensible, and lean and independent of any operating environment or execution platform. At present, it is based on a single programming language C/C++ with some low-level interface code written in C or assembly language. The bare PC applications are based on bare machine computing (BMC) paradigm running on x86 processor/CPU architecture.

As the world turns its attention to mobile and pervasive devices, this motivates to develop BMC applications that run on bare ARM and other processors. When x86 and ARM devices are made bare, one can hope to develop BMC applications that run across many pervasive devices.

SECTION II

RELATED WORK

The BMC concept also known as dispersed operating system computing (DOSC) [45] enables computer applications to run on a bare machine or a bare PC. The BMC concept originated from an Application Oriented Object Architecture (AOA) as cited in [42]. The AGO run on a bare PC without the support of any OS or kernel. The AGO provides direct hardware communication interfaces to AO programmers thus eliminating all the abstraction layers introduced by OSs and their environments. Direct BMC hardware interfaces for C/C++ applications are described in [44]. These interfaces enable program load, screen display, mouse and keyboard access, process management, network/communication, multimedia and other controls.

There has been considerable research and significant advances in the areas of graphics and multimedia. In [46], an OpenGL-based scalable parallel rending framework that provides a graphics API was discussed. In [40], two user interfaces for interactive control of dynamically-simulated character using embedded system platforms were demonstrated. A lean mapping graphics interfaces that uses a method for real-time filtering of specular highlights in bump and normal maps was described in [35]. All such approaches require conventional OS-based platform support. A comprehensive low-level graphics design and implementation was described in [31]. However, these graphics interfaces use DOS (Microsoft Disk Operating System) primitives and interrupt 21h, which require DOS environment. In bare PC applications, only required interrupts are used and included with the application. At present, there appears to be no direct hardware API for multimedia applications that can run on a bare PC with no OS support. In the

areas of code transformation and translation, many innovative techniques have been proposed. For example, Intel x86 programs can be translated from binary code to ARM and Alfa binaries with reasonable code densities and quality [30, 57]. The hardware abstraction layer concept is used to implement the Java virtual machine directly on hardware in an embedded system (as extensions of standard interpreters and hardware objects that interface directly with the JVM) [36]. In SoulPad [41], an auto-configuring OS with a suspended virtual machine is created on a small portable device at boot-time, giving users access to their personal environment and previously running computations. In BulkCompiler [54], a simple compiler layer is provided that works with ISA primitives and software algorithms to drive instruction-group formation, and to transform code to exploit groups. Our work differs from previous code transformation/translation approaches in that we are transforming the same bare code so that it can run on different CPU architectures.

Computer applications are typically written in high level programming languages, which at compile and link time require OS calls in some form. These calls enable applications to access hardware resources at run time. In contrast, bare machine applications eliminate the OS and run directly on the hardware. Previously developed bare machine applications include web servers [33]; email servers and clients [24, 38]; SIP servers [1]; applications secured via IPsec [37]; VoIP softphone clients [25]; VoIP Agent [2]; and split protocol servers [9]. These applications are based on the BMC paradigm, which does not use any OS, kernel, or embedded system calls. Bare applications directly communicate with the hardware using its own bare interfaces to the hardware.

The BMC approach differs from previous code transformation/translation approaches in that it is based on eliminating OS or kernel dependent code in applications. This approach still has a dependency on the underlying architecture. This dependency could be eliminated in the future by developing a generic hardware API for various architectures, and eventually including these interfaces in the hardware. However, until the hardware API is standardized, the BMC approach requires that the programmer write both application and systems code as an integral part of an application.

This research takes the BMC paradigm to another dimension where pervasive devices can be made bare. When devices are made bare, applications can be pervasive across many devices and computer architectures. Someday, CPU manufacturers may integrate the hardware API cited here right into hardware thus facilitating programs to communicate directly to hardware without a need for any sort of middleware.

SECTION III

INTRODUCTION

BMC applications referenced here [26] focus on x86 architecture and bare PC system. This dissertation extends this work and investigates applicability of BMC on pervasive devices. In particular, it develops steps to implement BMC concept on ARM processors. These popular ARM processors are used in mobile phones, sensor devices, and other control applications. These applications typically use an operating system (OS), lean kernel, or they are part of an embedded system [32]. We describe a methodology to develop bare ARM applications, where the application directly communicates to hardware without any middleware or OS. The BMC paradigm allows an application programmer to have sole control of the application and its execution environment. When computing devices or hardware systems are bare, they could become ownerless, pervasive, adaptable, and re-configurable to suit the needs of an application. The bare hardware or system can be used by any user anywhere without concern for sharing resources. A portable device such as a USB flash drive can be used to carry the bare machine application and run it on any bare machine. The BMC paradigm enables computing to be polarized on applications rather than computing environments.

There are architectural differences between one processor to another. The x86 processors are popular for desktop environment. Similarly ARM processors are commonly used in handheld devices. Most of these devices today use common applications such as web browsers, emails, text messaging and so on. Today, these applications are unique to a given platform. An email application written for ARM device will not run on a x86 desktop PC. There are different web browsers that work on different

platforms. How can we homogenize application programs that run across pervasive devices? In order to accomplish this, one need to learn writing application code that is independent of any computing environment or execution platform. When OS or kernel is eliminated, most of the platform issues are eliminated but the compiled code is still dependent on underlying CPU architecture.

This research approaches to address the above problems in multi-facet way. A reasonably complex graphics application is written that runs on a bare PC. This application is written such that majority of the code is independent of its computing environment. The compiled code still has some affinity with the underlying CPU architecture. This application is transformed to run on ARM processor with minimal changes in the code. The design and research issues are studied to conduct this transformation. The ARM Development Board (ADB) is used to run this graphics application, where it uses U-BOOT to get control of the CPU and rest of the code is independent of any OS, kernel, or embedded system. A graphics API is developed that is shown to be generic and common to many pervasive devices.

A temperature sensor device application is written for ARM to study the applicability of BMC on ARM processors. This application runs on the ARM Development Board with its own buzzer, touch screen and user interfaces that does not use any OS or kernel interfaces. These interfaces are written in low level C interfaces that directly communicate to hardware. The experiments indicate that it is possible to write low level hardware interfaces that directly work with application programs. There is no centralized program running in the machine other than the application itself. The temperature sensor device application demonstrated the feasibility of BMC concept in ARM processors.

The ARM development process involves deep understanding of many ARM interfaces including: USB (Universal Serial Bus), UART (Universal Asynchronous Receiver/Transmitter), GPIO (General Purpose I/O), SPI (Serial Peripheral Interface), IC2 (Inter-Integrated Circuit) and SDIO(Secure Digital I/O). It requires thorough knowledge of product specifications and data sheets. In low level coding, there is a need to understand device pin configurations that are relevant to ARM architecture [20, 47, 55, 56]. The development platform and testing environment is also dependent on the ARM processor and its architecture. Enormous efforts are spent in learning and making the system work with the BMC paradigm.

The intricacies of making ARM applications bare and strategies to transform x86 application to bare ARM along with many bare ARM interfaces and device drivers were developed for graphics and sensor device applications as shown in Appendix A. This research shows greater potential for making ARM applications bare and even greater potential for transforming bare applications from x86 to ARM.

SECTION IV

BARE MACHINE COMPUTING

Background

Bare Machine Computing (BMC) first invented by Dr. Karne at Towson University is motivated by the unique concept to make a computing box bare and carry the software application in a portable device such as a flash-drive. Bare machine applications use the BMC or dispersed operating system concept [45]. That is, there is no operating system (OS) or centralized kernel running in the machine. Instead, the application is written in C++ and runs as an application object (AO) [43] by using its own interfaces to the hardware and device drivers.

When computing hardware is made bare, the bare hardware can be used to run any given application on the fly. There is no need to protect the bare box as it does not have any valuable resources. The bare box simply has memory, CPU, user interface (input/output) and a network interface. All persistent data is stored externally on a mass storage device or on the network. This BMC approach is applicable to any pervasive device including: desktop, laptop, hand-held, or any other electronic equipment.

The BMC concept is not an embedded approach as it is applicable to generic computing. It is not a mini-OS as there is no centralized program running in the machine when an application set is running on a bare machine. The software application(s) can be modeled as a single monolithic executable as an application object (AO), which is based on a single end-user application or a set of end-user applications that are required at a given time. For example, a web browsing is an end-user application. One can design an AO that consists of simply a web browser, or a composite of applications such as

email, messaging, graphics, database, spreadsheet and word processing. The computers carry their applications as AOs on a flash-drive. In this computing, the information technology world is application centric rather than environment dependent as AOs can be run on any bare machines.

One can develop AOs based on end-user applications and they can be made tailored to suit individual needs. The AO is written in a single programming language and compiled into a given bare machine code. These AOs can be run on any bare computers which have no particular ownership. For example, one can walk into an organization and use their computer by using their own AO, without harming the bare computer. This computing paradigm will change the way we do business today and make the computing devices standard and universal. The AOs can be open ended application domains. The computer programming languages will become standard and extensible as the applications grow. There will be a greater need for standardizing hardware, software and interfaces for bare machines.

The bare machine computing applications are small, end-user centric and application centric. Once an AO programmer is trained, it is easy to write AOs, as it requires a single programming language expertise and only the AOs domain knowledge, and there is no need to know other computing platforms or environments. The AO programmer is in total control of a given application set's design and its execution order. No centralized OS or kernel is involved in its execution. As the AO controls the application aspect as well as the execution aspect; these applications will be inherently more secure. The AOs are quite suitable for peer-to-peer secure communications. In fact, the bare to bare communication reaches the ultimate security one can achieve in peer-to-peer

communication, as it avoids all the system related vulnerabilities by making the device bare. The AOs can be tailored on the fly for a given set of users or group of users. The communication and security protocols can also be tailored to provide more secure communications. The computing aspect of bare machines becomes standard and limited, but the network and security aspect of computing becomes more open for efficient protocols that are suitable to end users.

Other Related Work

The enormous growth in computing hardware and software has created unmanageable electronic dump without any reuse of hardware and software. Software applications, operating systems, tools and gadgets come and go on a daily basis without serving their useful life cycle. The OS code sizes have reached close to one hundred million lines of code resulting in rapid upgrades, version releases, errors and security flaws.

Many researchers now focus on coping with small kernels and lean operating systems or dedicated applications. While the BMC concept resembles approaches that reduce OS overhead and/or use lean kernels such as OS abstraction [7, 8, 16], Exokernel OS [6, 10, 15], Also in IO-Lite [53], Palacio [30], Libra [22], bare-metal Linux [17, 49], OS Kit[50], Factored OS [18] and TinyOS [51], Fast and flexible networking [27], there are significant differences such as the lack of a centralized code that manages system resources and the absence of a standard TCP/IP protocol stack. In essence, the AO itself manages the CPU and memory, and contains lean versions of the necessary protocols. Protocol intertwining is a form of cross-layer design. Further details on bare PC applications and bare machine computing (BMC) can be found in [26].

SECTION V

MAKING ARM PROCESSOR BOARD BARE

The bare machine computing applications developed in the past use x86 Bare Machine Computing (BMC) architecture. In this model, it uses bare boot program, loader and direct Hardware Application Interfaces (HAPI) [44]. In addition, it uses Visual Studio based compiler and linker modules to compile applications without using any default libraries (option: NODEFAULTLIB) [52]. In order to apply BMC concepts to ARM board in a similar fashion, there is a need for techniques and tools for developing bare ARM applications. The following sections describe the development process for ARM board applications.

The development environment for the bare machine application is shown in Figure 1. The Application Development Board (ADB) is connected to a Windows laptop using an RS232/UART cable. No OS is loaded in the ADB. The laptop is used to load, test, and debug the application. The temperature sensor device is plugged into the ADB. The ADB consists of a power switch and reset button in addition to the other components in Figure 1. The power switch is used to boot the application and the reset button is used to restart the application.

Development Environment

As noted in Appendix C, the C/C++ bare application is developed under the “Eclipse” IDE, which is run on the laptop. We used the RS232/UART Telnet/Hyper-Terminal on this machine to communicate with the ADB. Figure 1 shows an ADB diagram with the bare memory map. The map contains bareperipheral.bin (application executable) and the

images area. The images are used to display some icons in the LCD (bare graphics). The application is loaded at 0xc0008000 and the images data is loaded at 0xc5000000.

Eclipse IDE is used as the development environment for writing the ARM code on a laptop or desktop system. The code for x86 was generated using batch files and Visual Studio.

Development Board

In order to have full control over the development process and to also make the system bare, an ARM development board (OK6410 from Samsung) is used. This board comes with built-in (conventional) systems including Android, Win-CE and Embedded Linux. It has 256 MB main memory, 2 GB mass storage, and 667 MHz clock speed. A SanDisk 2GB SD card is used for secondary storage [47].

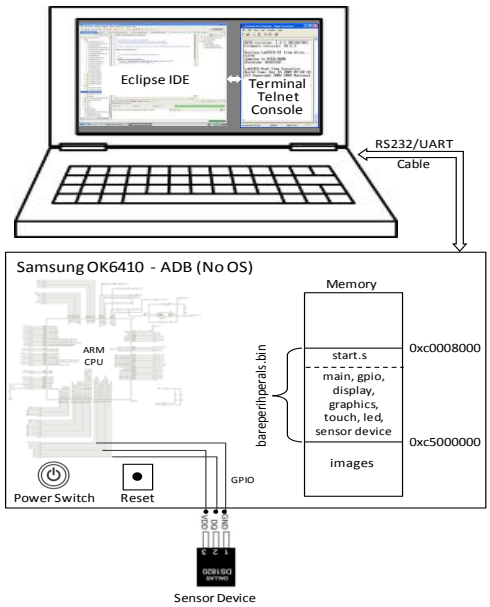

Figure 1. Bare ARM Application Development Environment

Boot and Load

We use the U-Boot tool, which is a universal boot loader [19]; it also provides free source code. The interactive commands provided by the tool can be used to boot, load,

and execute programs. We use minimal U-Boot functionality including CPU Register/Stack setup (before call to an application), setup UART console, setup clock/PLL setting, and memory initialization. Although the boot/load process could also be made bare (similar to what is done in x86 bare application development), this has not been done for the bare graphics application. After the executable is created using the Eclipse IDE, it is loaded using U-Boot's "loadb" command. Usually, an application is loaded at a specified address (e.g. 0x8000).

U-Boot 1.1.6 is used to boot the ADB. When the system is booted, it displays a user menu. We exit this menu using option (e) so that no other software is loaded in the ADB (bare ADB). In the U-Boot shell, the command (#dnw c0008000 bareperipheral.bin) is used to load the bin file. The command (#dnw c5000000 imagefiles) is used to load image files. Once all the code and data are loaded, the command (#go c0008000) is used to run the bare application as described in Appendix D.

Execution

After the executable is loaded into memory, it is executed by invoking U-Boot's "go" command. The "go" command will load this specific start address in the PC register and start executing from this address [5, 19]. In an x86 bare PC application, boot, load, and execute are all part of the application object (AO), which is controlled by the programmer [26].

When the application starts, it first executes the start.s code. This is a small ARM assembly code segment [32] that functions as a bare boot program. The assembly code then jumps to main(), which is the starting point of the C/C++ application in the binary file.

SECTION VI

BARE PC GRAPHICS ON X86

In order to apply BMC principles to ARM devices or pervasive devices in general, we need a sample application that can be made to run on a variety of architectures. This process is achieved in three stages. In the first stage, a sample graphics application is chosen which requires many aspects of computing resources. This application runs on a bare PC (x86 architecture), and displays some standard graphics images. In the second stage, this graphics application is transformed to run on an ARM device. During this stage, many ARM interfaces are designed which can be used for other applications. These bare ARM interfaces are furthermore used in the third stage, where a temperature sensor application is built that runs on the ARM architecture without any need for centralized OS or kernel. This section provides bare PC graphics and its detailed design. Section VII describes the transformation process of graphics application onto an ARM processor and Section VIII provides a temperature sensor application with comprehensive BMC interfaces.

Current multimedia and graphics applications are built on graphics, video and audio drivers that are accessible through some platform such as Windows or Linux. In handheld devices, multimedia software is embedded in the devices to allow graphics capabilities. These embedded systems rely on some lean OS or kernel. In most cases, multimedia or graphics applications are dependent on the device platform. Modern multimedia applications use a graphics processing unit (GPU) along with video memory to provide parallel processing power before rendering to screen or storage. The video card’s processing power and technological advancements in hardware pave the way for

new software architectures that exploit the capabilities of modern systems. For example, since video cards provide gigabytes of low cost memory, paging and virtual memory are unnecessary, and multiple address spaces can be avoided by using a single monolithic executable code with real memory [39]. Today's high definition video cards can stream well over 60 frames per second, and are even over-clocked to higher speeds for 3D simulation using the Wiggle effect [14].

However, the above technological trends and techniques are platform-dependent and are not easily ported from one environment to another. Our research considers the design and implementation of a bare graphics architecture that is self-contained and does not require any operating system, kernel or environment to run. It is written in C/C++ and accesses video memory directly from its application program.

Graphics Architecture

The BMC x86 Graphics architecture differs in many aspects from that of conventional or embedded graphics systems as the interfaces are directly accessible by the AO programmer. A given interface executes without any interruption as a single thread of execution. The AO programmer can control activation, suspension and resumption of this thread at program time. Figure 2 illustrates graphics architecture using x86 processor-based systems.

An application programmer writes a graphics application (e.g., animation, visualization) application object (AO) in C++ or C using the direct hardware graphics interfaces (API). These interfaces are provided by the application graphics object (AGO). The AGO implements high level application logic if needed and sets up parameter passing for shared memory. The AGO invokes "C" language interfaces (using

extern “C” {}) to invoke C calls from C++. The C calls in turn will invoke assembly calls for a given graphics interface. The assembly call then calls a graphics API software interrupt (int 0xfa). The AO, AGO, C, assembly calls have full access through a memory interface to read or write data in shared memory. This is accomplished by using a MEMDataSel selector that allows access to shared memory in real and protected modes using zero base select value. All of this code is executed in protected mode.

The software interrupt above is an interrupt gate that takes the call to real mode. The graphics interfaces are implemented in assembly code that run in real mode. These interfaces in real mode have access to video memory as shown in Figure 2. PC BIOS interrupts are also used to control video memory and graphics modes. Interrupt descriptor table (IDT), global descriptor table (GDT), local descriptor table (LDT), task state segment (TSS), boot, loader, interrupt service routines are all part of an AO. The AO is a self-contained, self-managed and self-executable module. The AO programmer has sole control of the facilities that are needed to run a given application.

The graphics architecture views the screen as just a multi-dimensional array of pixels that can be represented using vector-based mathematical models. This differs from conventional approaches that deal with each pixel every time the graphic changes.

The BMC graphics interfaces reduce complexity by providing a pointer to video memory that dynamically binds the interfaces at the hardware level to the video memory buffer. In this approach, there is no need to synchronize the GPU with the CPU functions as done in a conventional system [14].

The AGO architectural design is classified into eight broader categories:

- *Application Program (AO) Protected Mode:* actual user program main() executes here
- *Application Graphics Object (AGO):* software and hardware API used by AO
- *C-Programming Interface:* used as a gateway between C++ and ASM language abstracts
- *PC Assembly Interface:* used to interact with real-mode shared memory
- *Software Interrupt*: Interrupt to bridge between real and protected mode dynamically
- *Interrupt Gate to Real Mode:* used in real-mode to interact directly with the Hardware Video Memory
- *Graphics Operations Real Mode:* used to invoke low-level graphics primitives including screen access, screen framework, font/symbol, and other attributes.

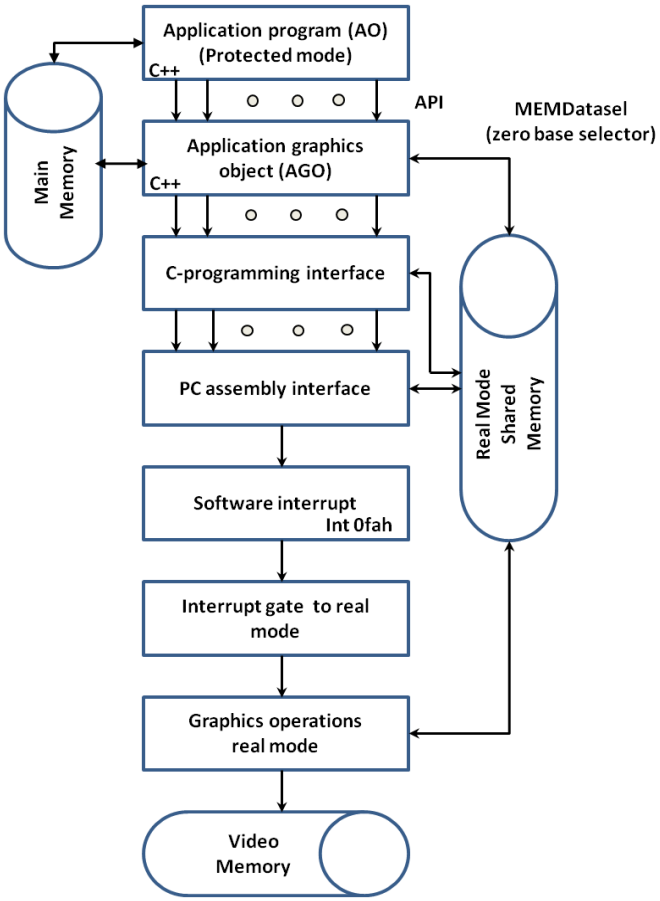

Figure 2. BMC x86 Graphics Architecture

The lean graphics architecture has many novel characteristics. The interfaces developed here can be directly invoked from standard C or C++ programs and are fully controlled by the programmer. The architecture is very generic and can be implemented on any pervasive device. As this approach uses only video memory instead of graphics card interfaces, a given graphics application can be ported easily for many types of devices. The current AGO does not use any graphics accelerators and hardware support, bare interfaces to audio, video and graphics cards can be written if necessary to resolve any performance issues. When more sophisticated bare graphics and multimedia becomes feasible, users can carry their own USB with a Web client and browse the Web on any bare device without carrying any hardware and with no reliance on any OS, kernel or

environment. In addition to being convenient for users, this graphics architecture eliminates overhead and may be easier to secure due to its simplicity.

Design, Implementation and Interfaces

A Bare PC graphics system is designed to perform graphics functions, such as drawing geometric figures with fraction (fractal) primitives, displaying text characters, and performing other attributes such as color, pixel, line, circle and displaying a bitmap image.

The following graphics functions are implemented using standard C and Intel assembly language. They are based on the design and implementation principles in [31] and modified to work with a bare PC system as described below:

- *Screen access:* clear screen, set the entire screen to a color or attribute, save the screen image in memory, and restore a saved screen image
- *Screen framework:* set a shape screen area to a given color or attribute, save and retrieve a screen area in video memory
- *Font/Symbols:* based on Vector / ASCII
- *Images/video:* based on pixel and compression
- *Shapes:* based on Vector/Pixel Mathematical Algorithms
- *Attributes:* set the current drawing color, set the current fill color, set the current shading attribute, set the current text color, set the current text font, set the current line type (continuous, dotted, dashed, etc.), and set the current drawing thickness
- *Image transformation:* scale, rotate, translate, and clip image
- *Bit operations for performance:* BIT Shifters; XOR, OR, NOT and AND bitwise operations.

In order to illustrate our implementation, we describe five basic direct graphics APIs:

(1) draw_pixel(): This interface takes x and y coordinates of a given pixel and computes its video memory location. The video memory address, location of a pixel, color of a pixel and its "opcode" are stored in shared memory. As illustrated in Figure 2, this interface goes from protected mode to real mode to the graphics operations code. It then obtains the pixel parameters from shared memory and places the pixel in the video memory. After displaying the pixel on the screen, it will return to its AO. The entire API process is executed as a single thread of execution.

(2**)** draw_line(): This is simply drawing many pixels to draw a line using Bresenham's algorithm [13].

(3) draw_box(): This API uses draw_line() API repeatedly to plot a box.

(4) draw_circle(): The circle is implemented without using the sine and/or cosine functions. It uses the algorithm described in [29] and uses draw_pixel() API.

(5) draw_bitmap(): First, the bitmap file is read from the removable device (USB) during the program load and stored in main memory. Second, the bitmap file header is parsed for size, color and image data offset location parameters as shown in Figure 4. Third, the display mode is setup to match minimum color palette requirements for the image as shown on Figure 3. Fourth, the video memory address, pointer to the bitmap image data and its opcodes are stored in shared memory as illustrated in Figure 2. Finally, the AGO will copy the image data from shared memory to video memory for display as shown in Figure 5. After placing the bitmap on the screen, it will return to its AO.

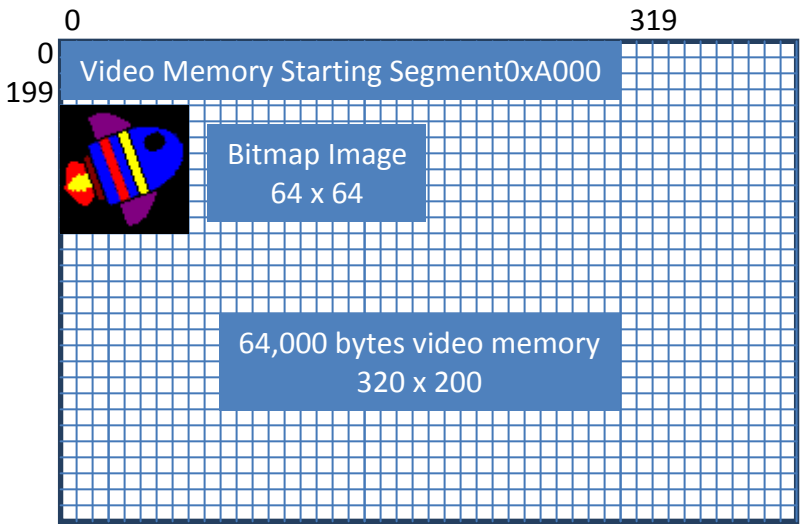

Figure 3. Video Memory Layout

Video memory is a contiguous linear addressing model, which differs from the x and y coordinates of the computer screen. To plot a pixel, the offset is calculated from the beginning of the video memory as follows: y coordinate multiplied by the total width of the screen and the x coordinate added to it.

In the example shown in Figure 3, we use the VGA Mode 0x13 with screen dimension of 320 pixels in width and 200 pixels in height. This translates to 0 to 319 on the x axis (width) and 0 to 199 on the y axis (height). The top left corner starts at coordinate (0, 0). Each pixel represents 8 bits (1 byte). Thus, the memory needed to store images of this size (320x200) is 64,000 bytes.

Figure 4 shows the Bitmap Image structure. It consists of several components that are described below.

The File Header in Figure 4 contains the FileType which starts with 4D42h ("BM"). FileSize is the Size of the image file in bytes. Reserved fields are used for future enhancements, with default values set to 0. The BitmapOffset stores the starting position of image data in bytes. The total size of the File Header is 14-bytes.

"Size" is the size of this header in bytes, and Width and Height are the Image width and height in pixels. A plane is the number of color planes and BitsPerPixel is the number of bits per pixel. The total size of the Bitmap Header is 40-bytes.

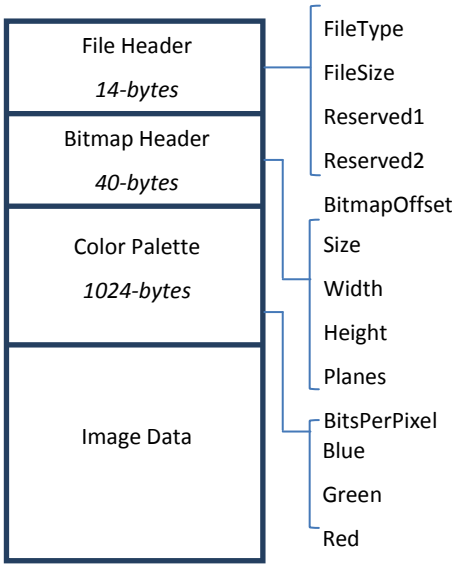

Figure 4. Bitmap Image Format

The BMC Color Palette specifies the red, green, and blue values of each pixel in the bitmap data by storing a single value used as an index into the color palette. In the newer versions of the BMP standard, the Color Palette and Image Data are merged together. In our example, we are using the Image Data directly, since we have pre-defined our palette to 256-colors. The total size of the Color Palette is 1024-bytes.

The following pseudo-code in Figure 5 is used to display a 64x64 bitmap to video memory and since video memory is linear, the following calculation is used:

$$\sum_{199}^{screen\_height=0}(y\_axis*319)+x\_axis$$

```
void draw_bitmap(AO, AGO Object Reference)
Initialize Memory Locations;
Initialize Variables;

Loop While screen_height < 200
{
 For (y=0; y < image_width; y++) {
     For (x=0; x < image_height; x++){
       //AGO Implementation, copy to Video Memory

            Video_Memory_Pointer [x+y*320]=
                   Shared_Memory_Pointer[x+y*64] } }
}
```

Figure 5. Bitmap to Video Memory Pseudo-Code

Testing and Demonstration

The testing was conducted on a standard VGA graphics card and VESA enabled BIOS on VGA Mode 13, with 320-by-200 pixel resolution in 256-Colors. These graphics were tested on Dell Optiplex GX260 PC with 512 MB memory.

AGO Object	*Response Time (Microseconds)*
draw_pixel() graph on figure 6	2.25
draw_line() graph on figure 7	83.25
draw_ circle() graph on figure 8	250
draw_bitmap() graph on figure 9	5

Table 1. BMC Graphics – Preliminary Response Time

The preliminary response time in Table 1 was conducted using the system time; it is an end-to-end measurement, which includes other components such as the VGA hardware device and other display intermediaries.

Below, we illustrate five basic primitives as described in Section VI. Each API is a direct hardware interface available to the AO programmer that can be invoked directly from C or C++ code.

Pixels

Using the draw_pixel() AGO API, 5000 pixels were plotted as shown in Figure 6. The x, y coordinates and color were chosen randomly. Preliminary performance tests shows a response time of 2.25 microseconds.

Figure 6. BMC Graphics - Random Pixels and Colors

Line

Using the draw_line() AGO API, 5000 lines were rendered as shown in Figure 7. The x1, x2, y1, y2 coordinates and color where chosen randomly. The draw_circle API is a direct hardware interface inherited from the draw_pixel() AGO. Preliminary performance tests shows a response time of 83.25 microseconds.

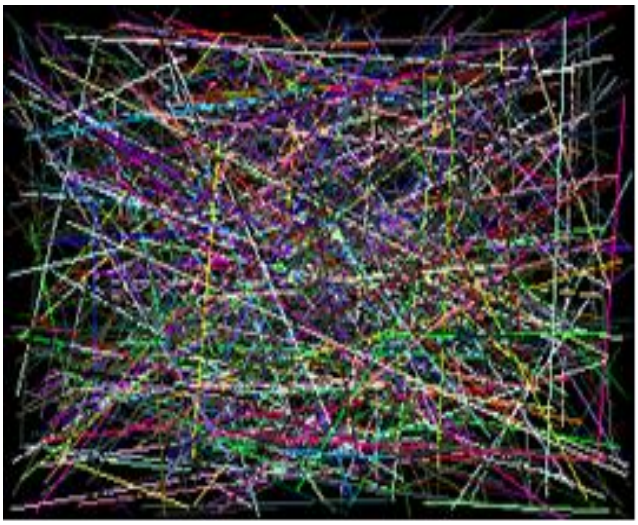

Figure 7. BMC Graphics - Random Lines and Colors

Circle

Using the draw_circle() AGO API, 5,000 circles were rendered as shown in Figure 8. The x, y coordinates, radius size and color were chosen randomly. The draw_circle API is a direct hardware interface inherited from the draw_pixel() AGO. Preliminary performance tests shows a response time of 250 microseconds.

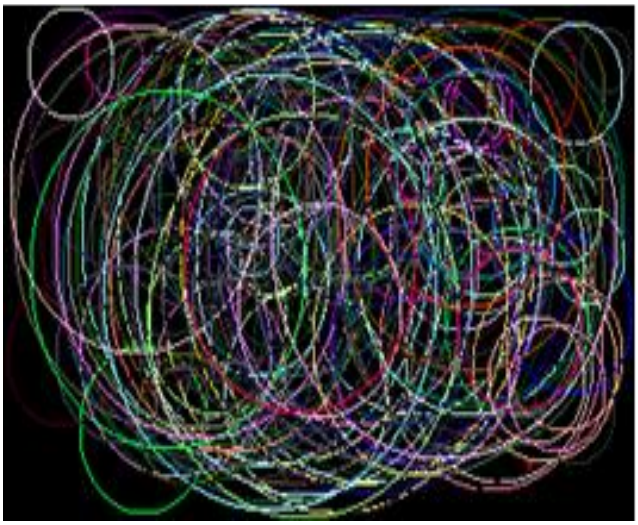

Figure 8. BMC Graphics - Circle with Random Size and Color

Bitmap

The bitmap used in this example is a 64 by 64 256-color bitmap with 8 bits per pixel, the file format is Windows RGB-encoded BMP format uncompressed. For a 256-color bitmap, there is a 54-byte header and a 1024-byte palette table in addition to the actual bitmap data.

Using the draw_bitmap() AGO API, the bitmap image was loaded into video memory directly for display. The bitmap was loaded three times with different orientation to show rotation and animation. This can be achieved by changing the pixel loading order while copying to memory as shown on Figure 9. Preliminary performance tests shows a response time of 5 microseconds.

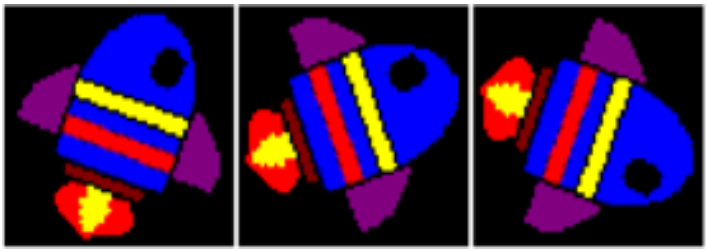

Figure 9. BMC Graphics – Bitmap

We described detailed knowledge that is required to build BMC Graphics application using C++ programming language. The AGO C++ API illustrated some of the fundamental graphic elements and its functionality. The fundamental problem with many computing systems is the layer of abstraction and common denomination that the OS puts on the graphics system. With Bare PC Graphics, the programmer has direct access to the video graphics device thus access to full set of features with significant performance improvement and programmers' freedom to innovate and be creative in organizing and deploying an application from the ground up. Now, this graphics application written for x86 is ready to be transformed to ARM processor as described in the next section.

SECTION VII

TRANSFORMING BARE PC TO ARM DEVICE

We consider the problem of transforming a Bare PC x86 application that does not use any form of operating system (OS) or kernel to run on an ARM processor. Bare PC applications have low overhead since they run directly on the underlying hardware, which makes them a good match for low-power mobile devices. They are also suited for pervasive computing in view of their ability to run without any additional support or software. While a variety of bare PC applications have been developed previously, they run on devices with x86 CPUs. Transforming a bare PC application to run on the ARM architecture is a first step towards investigating their potential for mobile and pervasive computing.

Bare applications are self-contained and independent of any OS, kernel, or execution environment. They are written to self-manage the hardware assuming a CPU architecture based on x86 (the hardware API, tasking mechanisms, and drivers are therefore x86-specific). Transforming an x86-based bare PC application to run on the ARM architecture is non-trivial since there is no OS or kernel support of any kind to act as an intermediary on behalf of the application. By understanding how to write bare applications in a general manner, the transformation process can be simplified. It will then be possible to run the same bare application code (for example, a bare VoIP client, a bare lightweight Web browser, or a bare sensor application) on a variety of devices with minimal code changes.

We solve the transformation problem from x86 to the ARM architecture for a particular application. Specifically, we transform a graphics application that runs on an x86-based bare PC so that it will run on a device with an ARM processor. The

transformation is achieved by making minimal changes to the existing x86-based bare PC graphics application. We also measure the time to run the same bare graphics application on an x86-based PC, an ARM development board, DOSBox emulator, and the QEMU-VM simulator.

Architectural Comparison

We now examine the main differences between the x86 and ARM architectures that impact the transformation process. In an x86 bare application (such as a graphics application [4]), a bootable USB and PC BIOS are used to boot the PC. The boot process starts in real mode (address space 1MB) in an x86 system. In protected mode for x86, the address space is 4GB. There is no notion of real or protected mode in an ARM system. Instead, it has several mode types in system mode, and it also has a user mode. These modes are used by an OS to provide protection to applications and the system [55].

The user display in x86 is controlled by video memory (0x B8000 for text mode): a user places the data in memory and it is immediately displayed on the screen. In the ARM architecture, a low-level driver is needed to display information on the screen. In x86, mass storage can be provided by hard disk (HD), solid state disk (SDD), optical disk (CD/DVD) or USB. In ARM, mass storage is provided by SD (Secure Digital non-volatile memory card), NAND Flash, NOR Flash, and USB; however they are not similar to a PCI (peripheral component interconnect) bus in x86. In x86, Intel's Host Controller Hub (ICH) provides interfaces to many I/O devices. In ARM, each I/O device has a controller that is directly connected to the CPU and provides GPIO (general purpose I/O with 187 pins) to the programmer [47].

Table 2 summarizes the comparison. It should be noted that there are many other differences (and similarities) in the x86 and ARM architectures that are not considered here.

Characteristics	*x86*	*ARM*
Architecture	CISC	RISC
Development	Visual Studio, C/C++	Eclipse CDT, GNU C/C++
Boot	BIOS	X-Loader, U-boot
Address Space	Real(20), Protected(32)	User(32), System(32)
Display	Video Memory	Driver
User Input	Keyboard/Mouse	Touch Screen
Mass Storage	HD, SDD, USB	SD, USB

Table 2. Architecture Comparison

C/C++ compilers and linkers are available for both x86 and ARM devices. As expected, the assembly language and CPU instructions are different in these devices. Thus, any assembly level code/instructions used in x86 must be re-written for ARM processor with respect to its differences in attributes as shown in Table 2.

Transformation Process

A high-level methodology for transforming a small graphics application from x86 to ARM is shown in Figure 10. This approach can be adapted to develop a general transformation methodology in the future for other types of bare machine applications. Making ARM processor board bare is described in Section V including the development environment, developed board, boot/load process, and execution environment.

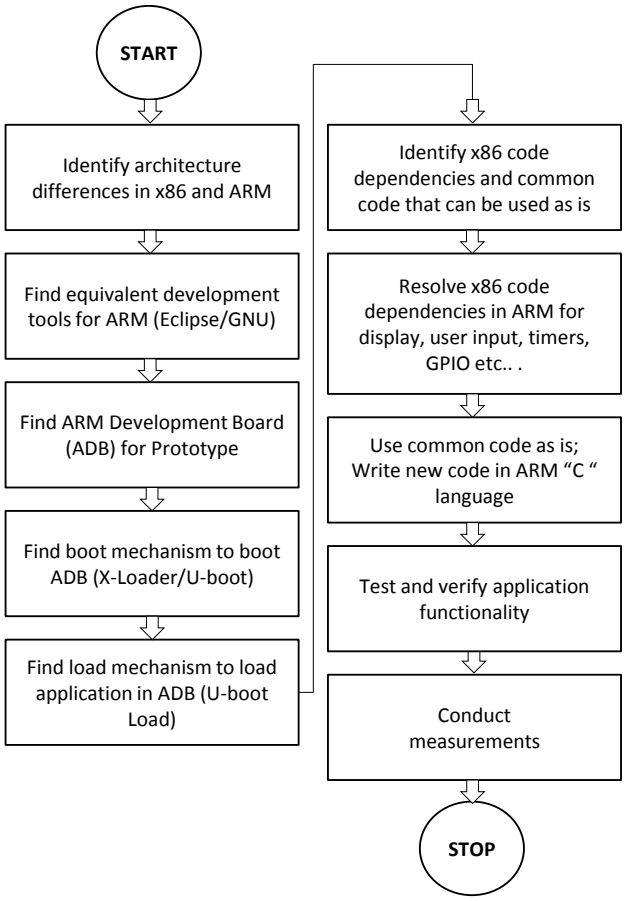

Figure 10. Transformation Process

Except for the boot and load process, the graphics application code transformed from x86 is bare code that does not depend on any OS or kernel. The following subsections provide details of the application transformation process.

The overall flow for transforming the bare x86-based application to an ARM application was shown in Figure 10. The key requirement is that the application should have the same functionality when running on an x86 bare PC and an ARM device.

Graphics Application on x86

The bare PC graphics application as discussed in Section VI is designed to perform graphics functions, such as drawing geometric figures with fractal primitives, displaying text characters, pixels, lines, circles, and bitmap images, and handling attributes such as

color. The basic graphics functions implemented using standard C and Intel assembly language include screen access, screen framework, font/symbols, image/video, shapes, attributes, image transformation and bit operations for performance as published in [4].

The number of lines of C code for the application is 615 (executable lines) and the number of lines of assembly code is 766 (including comments). The number of lines of assembly code for all hardware interfaces in a bare PC is 1969 (including comments). We only use about 10% of the hardware interfaces for the graphics application. The boot and load code are not counted in this measurement.

x86 Code Dependencies

The dependencies specific to x86 that were found in the bare PC graphics application code are due to data types, memory addresses, type casting, pointers, device addresses and device interfaces. For example, video memory is used to display graphics, whereas a bare driver is used to write to the screen in ARM. It would be possible to reduce such dependencies if the original code is written with the intention of transforming it to run on a different architecture.

Figure 11 shows code analysis that illustrates the transformation process. In this example, 38% of the code is reused from x86, 34% is newly written, and 28% is modified. The modifications are due to the dependencies identified above.

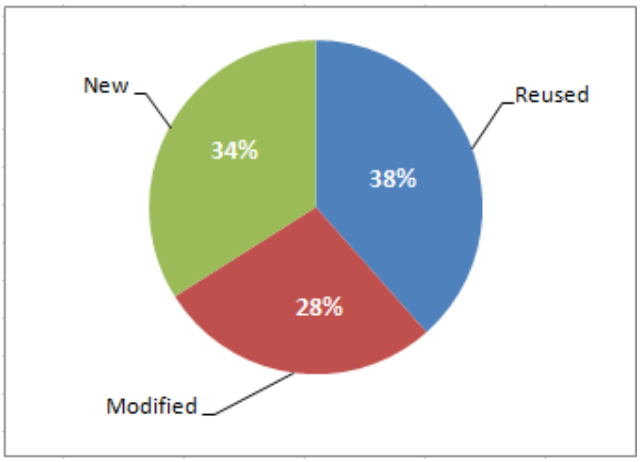

Figure 11. Code transformation x86 to ARM

The application in x86 uses its own boot, load, and execute written in assembly, which consists of about 1969 lines (including comments); the equivalent in ARM is U-boot. The assembly code used in bare (video memory access and controls) is about 766 lines (including comments), which is similar to the new code written for ARM. This code is 288 lines written in C. The total number of lines of C code in x86 is 615 compared to 580 lines for the ARM bare application. As expected, ARM code is smaller in size and appears to be more robust than the code in x86. In general, ARM is more suited for this type of bare machine application than x86.

x86 Implementation Dependencies

As noted above, there are many x86-specific dependencies in the bare graphics application code. However, since most of these are trivial, except device interfaces in x86 versus ARM, we only discuss the latter. In effect, we will illustrate the implementation differences when drawing a pixel on the screen, which serves as a building block for accomplishing most of the graphics API.

The bare PC graphics implementation in x86 is shown in Figure 12. The graphics application requires setup modes for controlling the graphics card. This is done by simply

using BIOS interrupts. Once the graphics mode and parameters are initialized, subsequent graphics operations use direct hardware interfaces to store graphics data in video memory. The bare PC does not have privileged and user modes, so all operations are done in user mode only. Since BIOS interrupts only work in real mode, all graphics control operations use an interrupt gate mechanism to go from protected mode to real mode. This switch is transparent to the AO programmer since it is encapsulated in the direct hardware interfaces used by bare PC applications.

An AO can also use software interrupts to reach certain hardware facilities in the bare PC system as shown in Figure 12. As bare PC applications do not use any graphics drivers, they rely on the direct hardware interfaces thus avoiding the need for any intermediary software in the system. We also use some shared memory in a bare PC application to facilitate direct communication between an AO and the underlying hardware. For example, a timer interrupt will update shared memory in real mode and the timer value can be accessed by an AO using a direct shared memory interface.

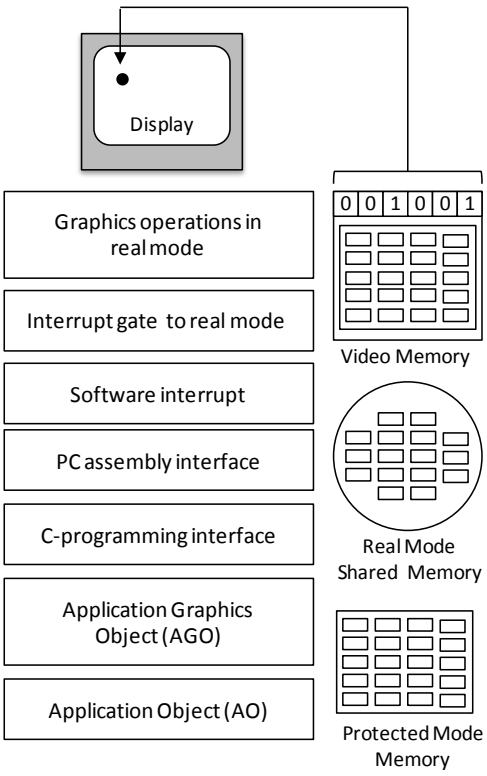

Figure 12. x86
Bare Graphics Implementation

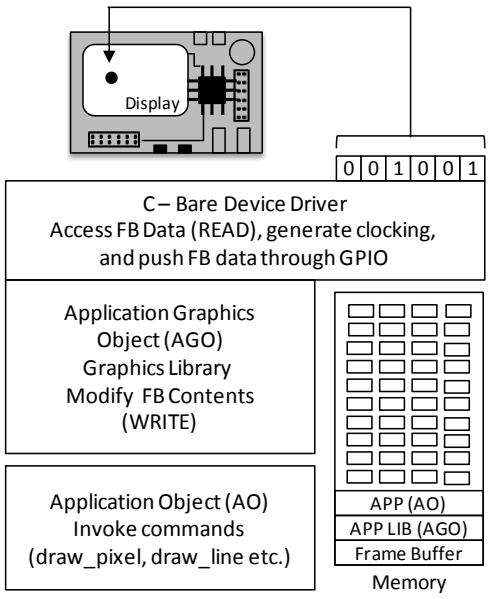

Figure 13. ARM
Bare Graphics Implementation

ARM Implementation Dependencies

The bare graphics implementation on ARM is shown in Figure 13. Figure 12 and Figure 13 taken together illustrate the similarities of the AO and AGO (Application Graphics Object) implementations in the BMC paradigm. At startup, the whole screen is blank and the LCD backlight is off. Since the display driver is written in C and it is not using any external or third party code or libraries, all necessary registers are defined and the macros are setup for controlling the lines through GPIO (General Purpose Input/Output). First, the application executes the bare display driver. Via registers and macros, it initializes the vertical and horizontal cycles with precise timing requirements as specified on the LCD Controller data sheet and turns on the LCD backlight. Second, the application initializes the Frame Buffer (FB), which is a location in memory to store

the display data. Third, the bare application calls the draw_pixel() function with three parameters: the y, x coordinates and color of a given pixel. Fourth, the bare graphics interface modifies the first four bytes in the FB with data according to the given color. Fifth, the application draws whole FB frame on the screen. It goes in a loop, reading all data from FB and generating the necessary signals through GPIO for CLK (Clock), VSYNC (Vertical Synchronization), HSYNC (Horizontal Synchronization) and DATA read from FB [47, 56].

Testing and Measurements

The testing environment consists of an x86 bare PC, bare ARM development board, DOSBox and QEMU-VM simulator. For x86, the testing was conducted on a Dell Latitude D620 laptop with a standard VGA graphics card and VESA enabled BIOS on VGA Mode 13, with 320-by-200 pixel resolution in 256-Colors, and 1.83 GHz clock speed. For ARM, the testing was conducted on an ARM development board: Samsung S3C6410/OK6410 micro controller with 677 MHz, and 4.3'' WXCAT43-TG3 TFT-LCD panel [56].

For DOSBox, the graphics application code was written in Turbo C (similar to x86 and ARM). This code uses BIOS calls for graphics. The bare PC graphics application can also be run using the QEMU-VM simulator. Here, the same application code as for x86 is run on the simulator, which runs on Microsoft Windows XP.

In Figure 14, the testing/execution environment for the measurements is shown. The left laptop shows the display of images for a bare PC application. The right laptop shows the same images on a QEMU-VM simulator running on Windows. The ARM development board is shown with wires illustrating the connections. The bottom three

screens show lines, circles, and images captured from the ARM display. It can be seen that these images look the same in all testing/execution environments.

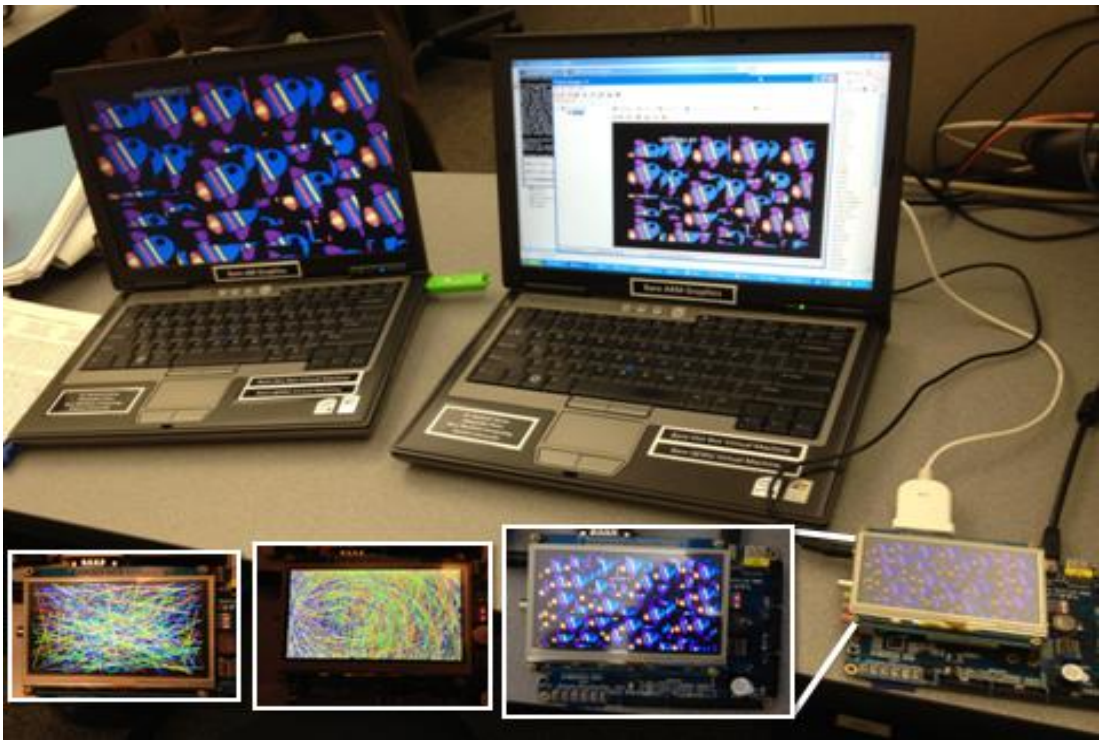

Figure 14. Executing and Testing Environments

For timing measurements, four graphics functions were used: draw_pixel(), draw_line(), draw_circle(), and draw_bitmap(). In an x86 bare PC, each pixel requires three bytes: the first two bytes indicate the location/screen size, and the third byte has the color information. These three bytes are formed in video memory directly for the desired number of pixels. Since video memory is linear, the location is calculated by multiplying the y coordinate by the total screen width, and adding the x coordinate. The draw_pixel() function forms the 3 bytes data in video memory. When drawing a large number of pixels, we used random coordinates and colors. In the ARM implementation, we used the same draw_pixel() interface. However, since video memory is not accessible in the ARM development board, we put the pixel data in main memory and pushed into the LCD. This

process is done completely in software instead of using a hardware display controller. The code written for the bare ARM device to control the LCD includes GPIO initialization, clocking, vertical sync, horizontal sync and data enable. It is independent of any operating system or environment.

Figure 15 shows the time to draw 500,000 to 75,000,000 pixels for x86 bare PC, ARM, DOSBox, and QEMU-VM. It can be seen that the x86 bare PC outperforms the other systems. Since the bare ARM development board uses a software display driver and its clock rate is 2.74 times slower than x86 bare PC, it runs slowest. The DOSBox performs better than the ARM device since it uses BIOS calls, the hardware display controller, and has no OS overhead. QEMU-VM performs better than the ARM device since it is based on the x86 bare PC code, and it simulates the bare PC environment. However, the overhead in QEMU-VMis much larger than for the DOSBox and for the x86 bare PC. The comparison merely demonstrates the basic performance of a bare application. A more detailed study is needed to account for differences in clock speed, instruction set and graphics controller implementation.

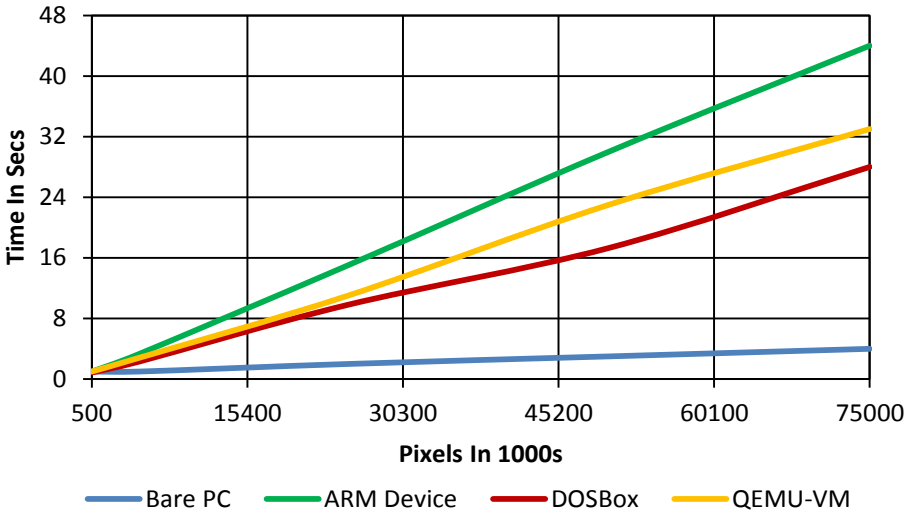

Figure 15. Time to Display Pixels

Figures 16-18 shows the time to draw 50,000 to 3,000,000 lines, 50,000 to 400,000 circles, and 1000 to 26,000 images respectively. In all cases, the performance of an x86 bare PC, ARM device, DOSBox and QEMU-VM simulator are essentially the same as for drawing pixels.

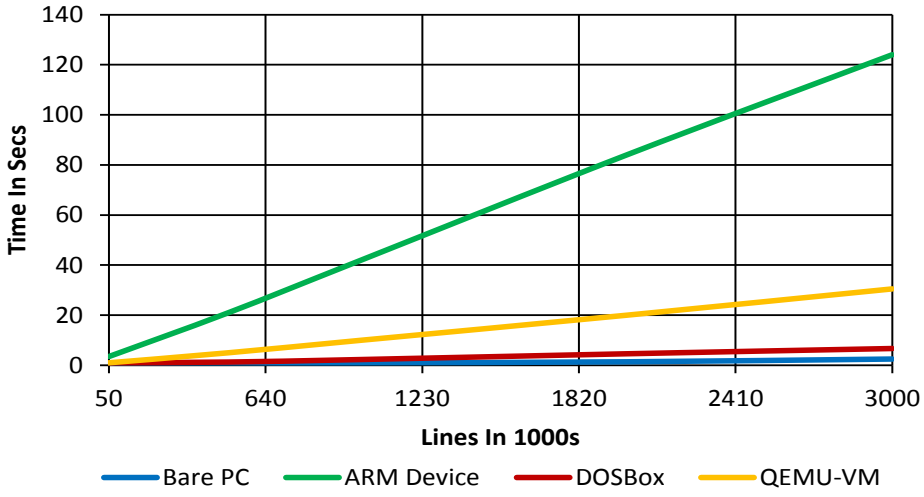

Figure 16. Time to Display Lines

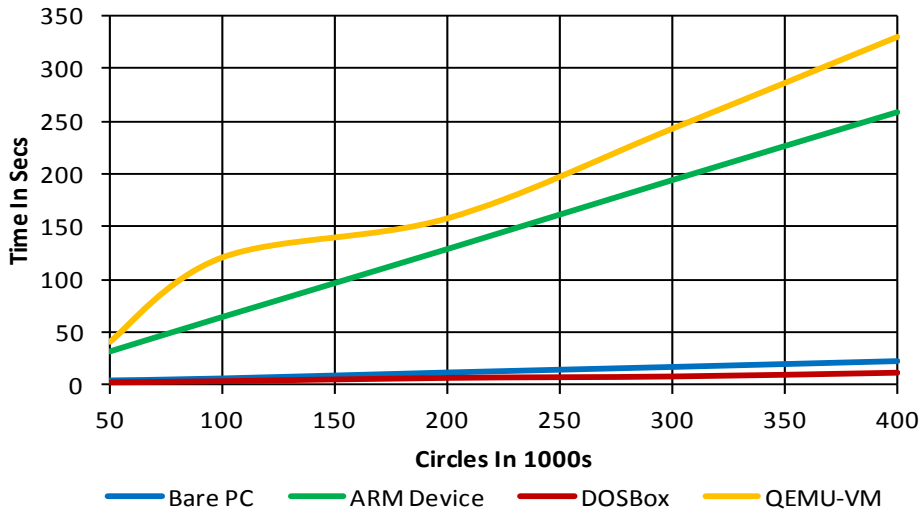

Figure 17. Time to Display Circles

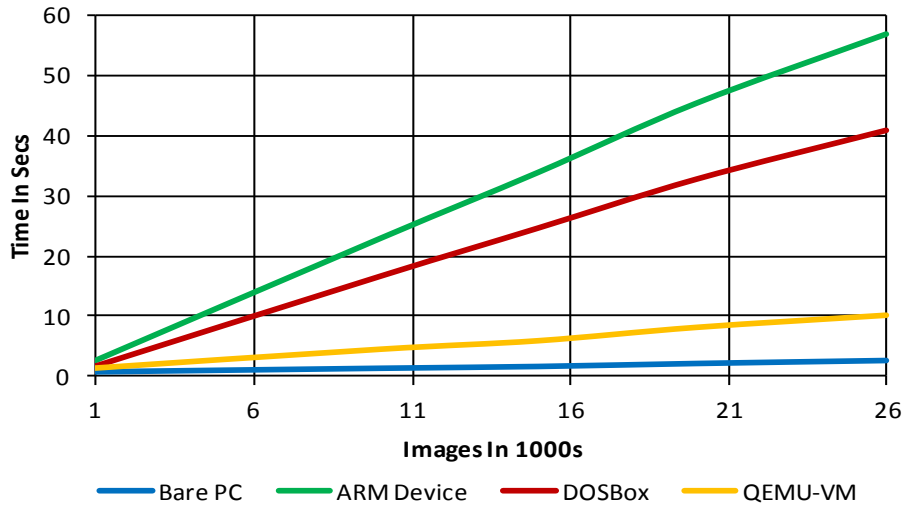

Figure 18. Time to Display Images

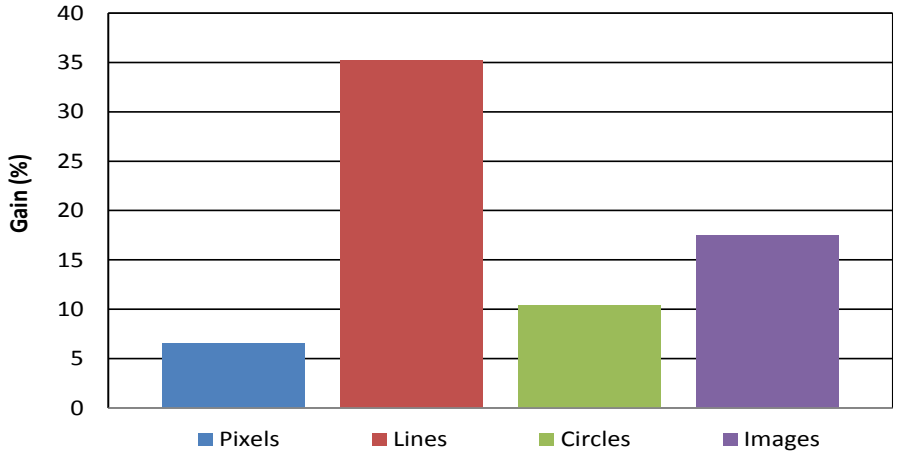

Figure 19. x86 Bare PC gain over a Bare ARM Device

The individual performance gains for the ARM development board and the x86 bare PC are shown in Figure 19. The performance gains for drawing pixels, lines circles, and images range from 1–11, 3.5–60, 8–11 and 0.2–27.8 respectively. These results indicate that the average gain for the x86 bare PC ranges from 6.5–35. Notice that the gain for drawing lines is much higher than for the other graphics. The likely reason is that drawing lines requires more calculations for computing the slope, which is faster in the x86 bare PC due to its clock speed and complex instructions for long operations.

As a first step towards investigating this potential, we transformed an x86-based bare PC graphics application to run on the popular ARM architecture. We identified differences in the x86 and ARM architecture relevant to the transformation, and outlined a transformation methodology. The graphics application code can be classified into three categories: code that can be used as is, code that is modified, and new code that is written for ARM. We resolved and implemented architectural differences to make the ARM code bare except for boot and load. The transformed bare application was tested using an ARM development board. We also did timing studies to measure bare application performance when running on the different architectures and demonstrated that the x86 bare PC performs better than the ARM device, DOSBox, and QEMU-VM simulator. The testing and measurement results demonstrated the feasibility of code transformation from x86 to ARM processor. The experiences learned in the transformation and writing bare graphics applications can also be applied to other platforms and applications. The bare ARM interfaces developed are used to design a bare temperature sensor device application as described in Section VIII.

SECTION VIII

BMC SENSOR DEVICE APPLICATION

A temperature sensor device application [3] is chosen to study the feasibility of BMC on a reasonably complex application that involves many interfaces that are common in ARM applications. This section will illustrate design and implementation of this application in detail.

ARM Development Board (ADB)

To design and implement the temperature sensor device application, an ADB [47] is used, which has most of the interfaces that are relevant to our study. Universal Boot Loader (U-Boot) [19] is used to load and run the bare application. The U-Boot is the only tool needed to bootstrap and load the application. The bare machine sensor device application is independent of any operating system, lean kernel, or embedded system.

In order to build a bare machine temperature sensor device application, it requires several components as shown in Figure 20. A brief functional description of this application is as follows. The application is loaded from the SD Card interface (Label 9) into memory using U-Boot. A temperature sensor (Label 1) is plugged into the ADB to monitor the current room temperature with a sensing granularity of 900 microseconds. As the temperature changes, its value (Label 8) and a graphic image (Label 7) is displayed on an LCD screen (Label 2). An output console is used to debug the inner working of the ADB and temperature (Label 3). An LED light (Label 4) provides a visual indication of temperature sensor operation. A buzzer (Label 5) is used to trigger an auditory alert based on a pre-defined temperature threshold. Also, the LCD screen color is changed to red to provide a visual alert, with a button icon to disable the buzzer via a touch screen

interface (Label 6). The ADB is used to illustrate the development of a bare temperature sensor device application without any middleware or kernel.

Now, the bare application program senses the temperature and displays the output on the LCD screen as shown in Figure 20. Testing was performed to verify that all functions in the application including buzzer, LED display, touch screen, and sensor device output worked as intended. The buzzer alarm activates when the temperature reaches a threshold value set in the application. This alarm can be disabled by a touch screen button as shown in Figure 20.

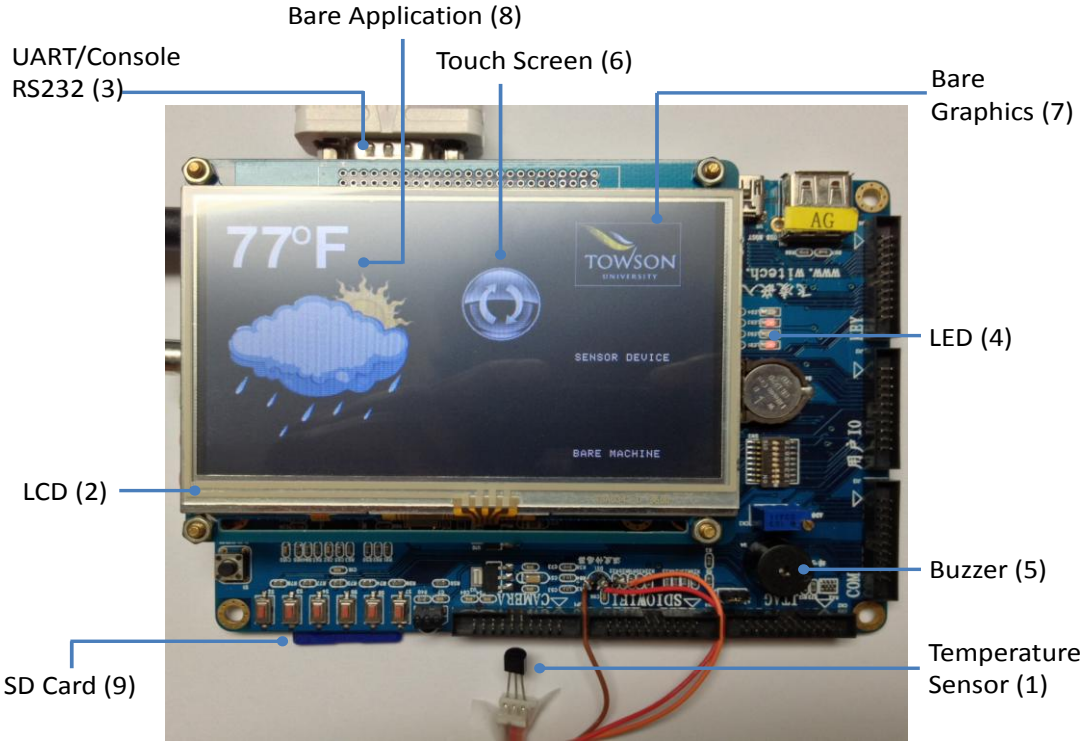

Figure 20. ARM Development Board (ADB)

Methodology

A general high-level methodology for building bare machine sensor device applications is shown in Figure 21. This methodology was used to construct the bare sensor ARM application. To build any bare sensor ARM application, it is necessary to

understand the various ARM architecture interfaces and protocols such as GPIO (General Purpose Input and Output), UART, USB, SPI, IC2 and SDIO. This involves both system and application level programming as when building any bare application. In particular, internal details, pin configurations, and other low-level information obtained from the product datasheet are needed to construct the appropriate bare interfaces for the ARM sensor application. Also, an appropriate development and hardware execution environment needs to be chosen to build, test and debug bare applications since there is no OS support. In some cases, an integrated development environment may help to speed up the bare application development process. Based on the type of sensor device, appropriate ARM interfaces (GPIO, UART, USB, etc.) must be selected to implement a given application. Furthermore, a generic ARM API must be developed to make the applications robust. In addition, an appropriate compiler and linker for the programming language must be available to create the bare application modules. Sensor application testing and validation are especially challenging in a bare environment, specifics are presented in Appendix B.

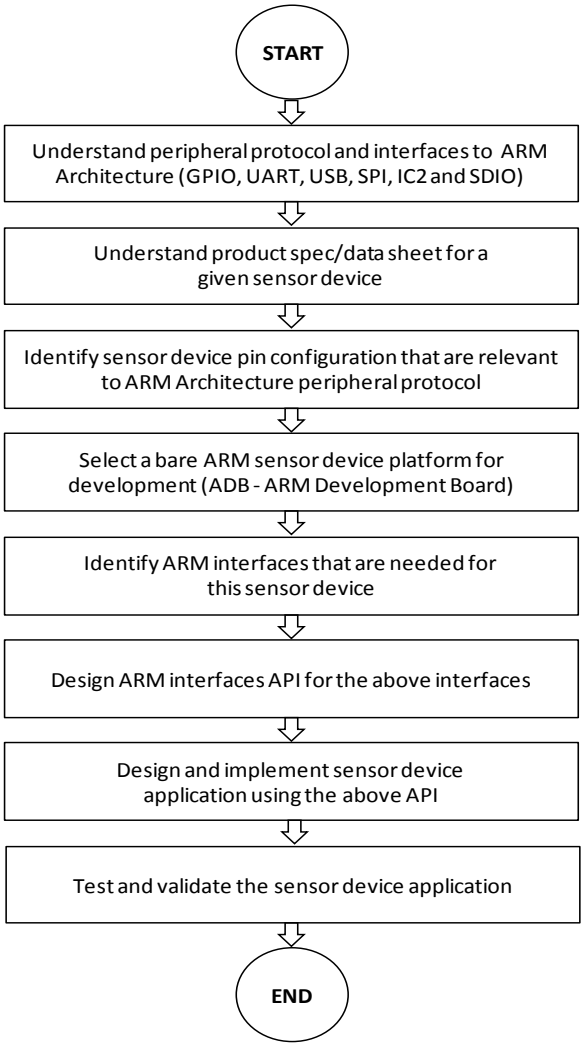

Figure 21. Bare Machine Sensor Development Methodology

Details of the bare sensor application that was built using this methodology are as follows. We selected a temperature sensor device, specifically the model DS18B20U 1-Wire Digital Thermometer [20], which has three leads (GND, DQ and VDD). The Samsung OK6410 ARM Development Board was used as the development platform. There is an OS/kernel included in this board. The sensor device is plugged into ARM's GPIO interface. The GPIO API is needed to initialize and communicate with the GPIO controller. A timer facility on the ADB is used to sample the sensor device since it requires a timer API for this application. Similarly, it is necessary to construct an API for

the RS232/UART (console debugging), LCD display (user interface), LED Controller (monitoring sensor device), buzzer (alert), and touch screen (to disable alerts).

While this methodology is specific to a temperature sensor for ARM device, it can be modified easily for other applications. Developing bare machine applications poses many challenges as indicated in Appendix A and B; since the programmer has to define, architect and implement all the necessary hardware interfaces needed for the application. There is no middleware or OS/kernel of any kind to support the application during execution. This is different from developing conventional OS-based applications.

Design, Implementation and Interfaces

This section describes the interfaces required, its design and implementation for the bare temperature sensor device application.

Interfaces

Figure 22 identifies the interfaces needed for developing a bare temperature sensor device application. This figure illustrates how the software API and external peripherals communicate with the internal hardware. In this application, the software API was designed and implemented for Touch-Screen, LCD Display, UART/RS232, Timer, LED, Buzzer, and Thermometer Sensor Device. The figure also indicates the actual interfaces designed and implemented for each class of API (e.g. LCD display (init(), clock_cycle(), write_frame()). We designed and implemented 20 software APIs for this application.

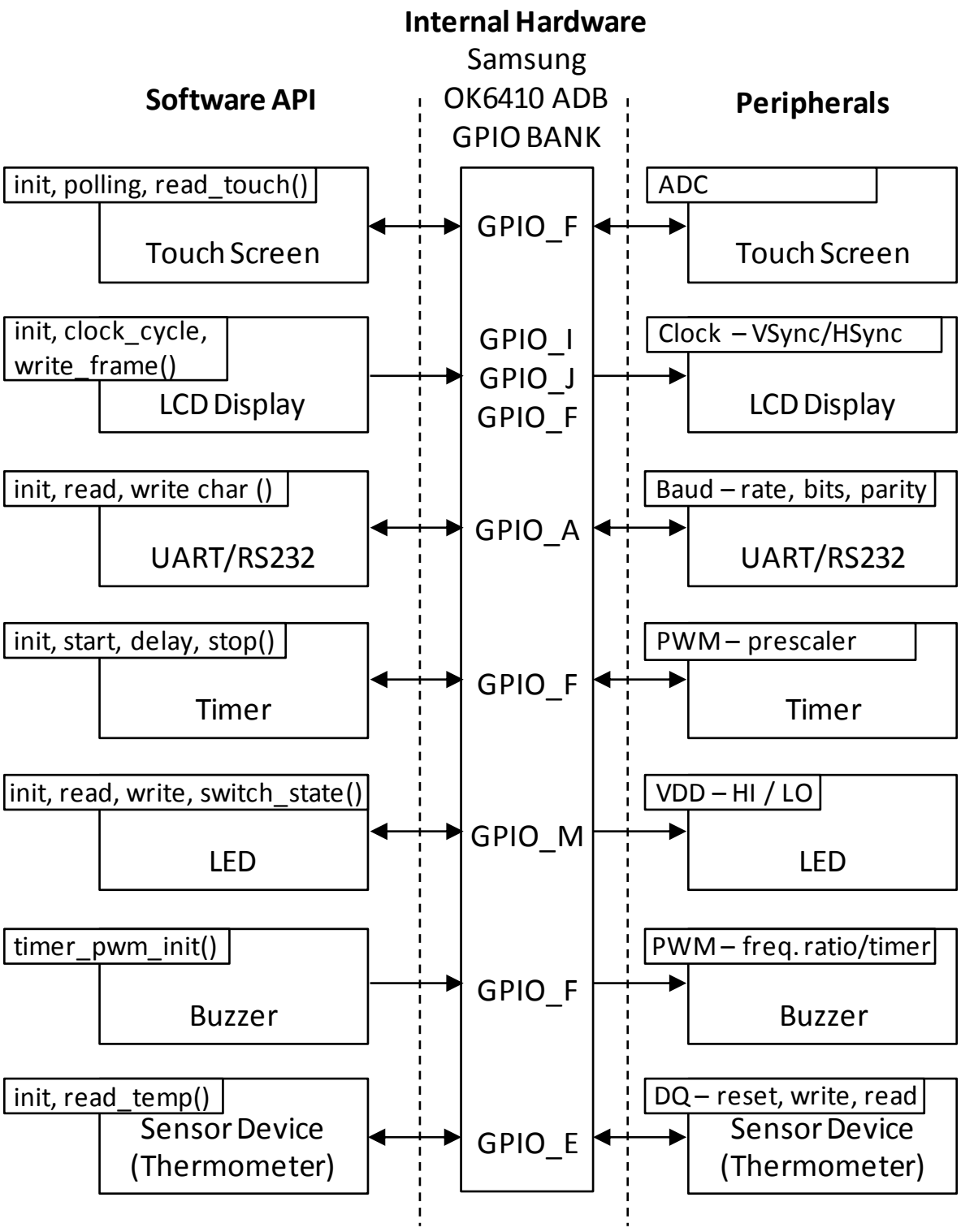

Figure 22. System Interfaces

The GPIO object is the key element of this application. It provides the internal implementation for the above software APIs. Sample code snippets to illustrate the GPIO API implementation are given in Figure 23. The interfaces use macros as defined in the #define statements. It can be seen that these interfaces directly manipulate GPIO facilities. We used GPIO banks A, E, F, I, J and M consisting of a total of 63 (8, 5, 16, 16, 12, and 6) pins. The connectivity of these pins and the bank registers are obtained from the SoC User's Manual [47].

```
#define GPIO_BASE 0x7F008000
#define GPIO_PORT_BASE(port) (port+GPIO_BASE)
#define GPIO_CON_BASE(port) (GPIO_PORT_BASE(port))
#define GPIO_DAT_BASE(port)
(GPIO_PORT_BASE(port)+0x4)
#define GPIO_PUD_BASE(port)
(GPIO_PORT_BASE(port)+0x8)
#define GPIOREG(address) ((*((volatile unsigned*)address)))

static void conf_bank( GPIO_BANK bank, unsigned
                                value, unsigned mask ) {
  GPIOREG(GPIO_CON_BASE(bank)) =
            (GPIOREG(GPIO_CON_BASE(bank))
                & ( ~mask )) | ( value & mask );  }

static void conf_pud_bank( GPIO_BANK bank,
                  unsigned value, unsigned mask )    {
  GPIOREG(GPIO_PUD_BASE(bank)) =
            (GPIOREG(GPIO_PUD_BASE(bank))
                & ( ~mask )) | ( value & mask );}

static void write_pin( GPIO_BANK bank, unsigned pin,
                                unsigned value )     {
  GPIOREG(GPIO_DAT_BASE(bank)) =
       (GPIOREG(GPIO_DAT_BASE(bank))
            & (~(1U<<pin))) | ((1U == value)<<pin); }

static unsigned read_pin( GPIO_BANK bank,
                                unsigned pin )       {
  return ( (GPIOREG(GPIO_DAT_BASE(bank))
                                & ((1U<<pin))) != 0);  }
```

Figure 23. GPIO Code Snippets

Details of the sample GPIO interface read_pin() in Figure 23 are as follows. The GPIO_BASE definition is a pointer to the starting address of the GPIO banks. There are three basic GPIO registers, which are crucial to the development of the GPIO API. The GPIO Configuration Register is used to initialize the pin for configuring it as an input or an output. The GPIO_CON_BASE provides the definition for this register. The GPIO Data Register is used to load or store the data. The GPIO_DAT_BASE provides the definition for this register. The GPIO Pull-up Register is used to provide an impedance match to the connection. The GPIO_PUD_BASE provides the definition for this register. In the bare sensor application, all these GPIO resources are directly accessed and managed by the programmer (and not by other middleware) i.e., the programmer needs to control the hardware operation of these registers in the application.

Design

The design and implementation of LED API init(), read(), write(), switch_state() operations are illustrated in Figure 24. The init() method uses the GPIO object to configure the state and status of the GPIO_M bank. Here, the state can be HI or LOW, the status is set to Output, and the switch_state() API is used to change the LED state to on/off. Also, the read() API is used to get the state of the LED, and the write() API is used to set the state of the LED. Note that the LED API uses the GPIO object interfaces such as gpio_driver_open(), conf_pud_bank(), and conf_bank().

```
void init( void )
{
 GpioDriver *gpio = gpio_driver_open();
 //disable pullup/down for GPM0,1,2,3 - fields 2bit wide
gpio->conf_pud_bank( GPIO_M, 0x00 , 0xFF );
//GPM0,1,2,3 as Output - fields 4bit wide
gpio->conf_bank( GPIO_M, 0x1111, 0xFFFF );
//set pin states to HI => disable led's
gpio->write( GPIO_M, 0xF, 0xF );
}

void switch_state( LED_NUM led )
{
  GpioDriver *gpio = gpio_driver_open();
  unsigned state = gpio->read_pin( GPIO_M, led );
  gpio->write_pin( GPIO_M, led, !state );
}

unsigned read(LED_NUM led )
{
  GpioDriver *gpio = gpio_driver_open();
 return gpio->read_pin( GPIO_M, led);
}

void write(LED_NUM led, LED_STATE state )
{
  GpioDriver *gpio = gpio_driver_open();
gpio->write_pin( GPIO_M, led, state );
}
```

Figure 24. LED Code Snippets

The read_pin operation takes two parameters bank and pin. For example if the parameters are E and 5 respectively, data is read from pin 5 in bank E, and the value is returned. The bit shifting shown in the code is needed to select an appropriate pin from the bank. The other interfaces are similar to this.

Implementation

The interfaces and sample code snippets shown in Figures 23 and 24 illustrate the simplicity of programming a BMC application. These interfaces are self-contained and do not require any other system libraries, kernel, or OS. The application programmer

controls all aspects of program development as described in [26]. In BMC, a program consists of a single monolithic executable referred to as an application object AO [43]. It is possible to have one or more end user applications programmed as a single AO.

Source File	*# Lines of Code*	*Header File*	*# Lines of Code*
start.s	14	n/a	n/a
main.c	62	n/a	n/a
gpio.c	136	gpio.h	177
display_sw.c	265	display_sw.h	29
gfx.c	634	gfx.h	81
adc_touch.c	159	adc_touch.h	14
led.c	78	led.h	44
thermo1820.c	161	thermo1820.h	10

Table 3. ARM Bare Code Size

We implemented the bare sensor device application in the ARM GNU C/C++ language under the Eclipse development environment. The application program consists of code for main, gpio, display, gfx (graphics), adc_touch (touch screen), led, thermo1820 (thermometer) as shown in Table 3. The source and header files for these functions along with their code sizes are shown in the table. The number of lines of source code including comments is 1509, and the header file code is 355 lines. This is the complete code required to execute the bare application at run time. There are no additional system calls or libraries used at run time.

OS vs. Bare Machine Interface Characteristics

In an OS-based system, the GPIO interfaces are more complex and involve many components. The GPIO.C (Linux) file hierarchy is shown in Figure 25. The large number of levels in the hierarchy and its interdependencies reflect the complexity in an OS environment. Numerous header files, which call other header files in the hierarchy, are also required. In the figure, nodes indicate the header file or an implementation file, and the links indicate the connectivity between different files. This tree has 24 required nodes: IRQ, Thread, Flags, Checks, File System, I/O Control Lock, Bit Operation, Type Checks, Procedure Trace, and 29 required inter-dependent links. In addition, it also has some optional nodes and links. An OS-based system has to cover all possible uses of an interface for generic application development. In this case, GPIO.C provides an API for any type of application that runs on an ARM processor under Linux.

In contrast, a bare machine sensor application only requires the GPIO interfaces in Figure 20, which are specific to this application. The GPIO.C file for such an application is shown in Figure 26. There is no hierarchy (i.e., the BMC model is flat) and the layers of complexity in the OS environment are avoided. The BMC paradigm thus allows applications to be completely autonomous (self-controlled, self-managed, and self-executable) by removing the centralized control in an OS-based approach.

An ARM application for the temperature sensor device that runs on Windows CE was used as the starting point for developing the bare application. It consists of three source files and their respective header files: TEM_AppDlg, TEM_App, and stdafx. The application does not include graphics, alarm, and touch screen. When these files are compiled in a bare system, where the system files (OS libraries) are excluded, there are

33 errors. These errors are related to Windows CE system calls, definitions, and some constants. During compilation, the system calls for Windows CE and related libraries are included to form an executable that runs on an ARM processor. In an OS environment, an application programmer cannot directly access the underlying ARM facilities. In the BMC system, the hardware interfaces as shown in Figure 22 and the ARM facilities in Figures 23 and 24 are directly accessed and managed by the bare ARM application programmer. This is the main distinction between an OS-based and a bare application.

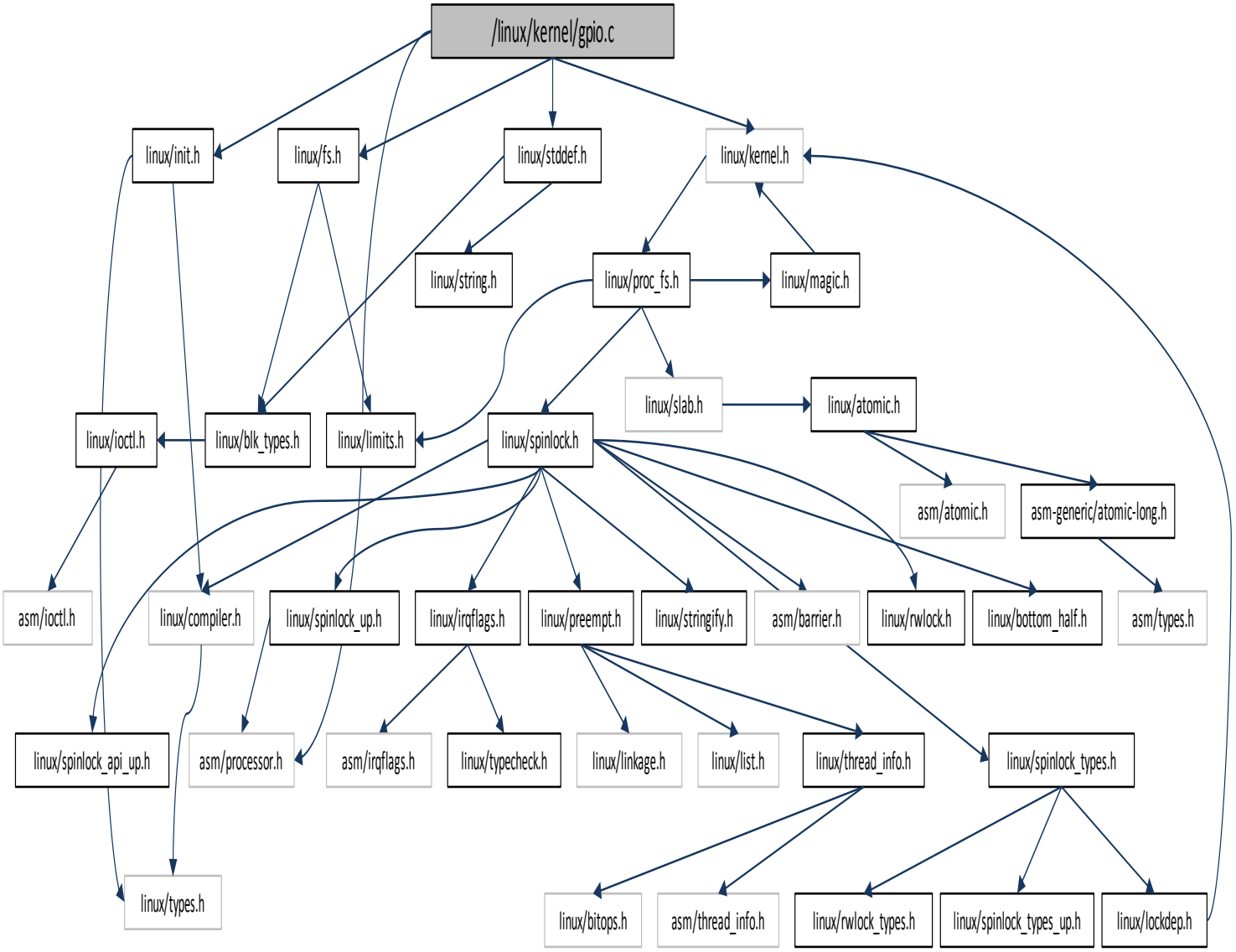

Figure 25. Linux GPIO Interface Structure

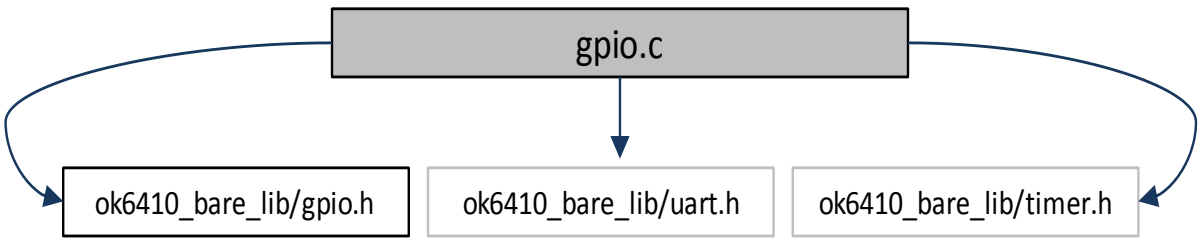

Figure 26. Bare Machine GPIO Interface Structure

Functional Operation and Testing

The development environment for the bare machine sensor device application and its description are illustrated in Section V. The bare temperature sensor device application tested for its functional operations as illustrated in Section VIII. A temperature sensor device has three pins. The DQ pin is attached to GPIO pin and VDD/GND pins are attached to the power source on the ADB. The ADB is connected to a laptop through RS232 interface cable. The laptop is used for debugging and loading application in addition to development.

A terminal window is opened in the laptop which is listening on the COM port connected to ADB. When power is turned on to ADB, using the terminal window, U-Boot is loaded to the ADB. After this, a U-Boot displays a menu with list of options. An exit to command line option is picked so that kernel load is bypassed. The bare sensor device application is loaded into ADB memory using U-Boot commands. Using the "go" command, the application is executed. The temperature is sensed by the application and displayed on the screen. A temperature sensor threshold is programmed into the application which triggers a buzzer and also a LED. Both were tested for proper operation. As the temperature changes, the application updates the screen. Touch screen is tested to disable the buzzer. This testing was performed many times to assure the proper operation and accuracy of the temperature sensor device. All the interfaces and code written for this application does not use any other software and kernel other than U-Boot to load and start the application.

A similar temperature sensor device application is also available that runs on Windows CE, which is also tested and compared with the bare sensor device application. Figure 27

shows the output screen for Windows CE application. The ARM application that runs on Windows CE has 333 lines of source code (excluding graphics, touch screen, alarm, system header files, and system call code libraries). The bare ARM application has 1864 lines of code including the application and interfaces as shown in Table 3. The Windows CE application code is small as it does not include all functions and the additional code it needs during run time.

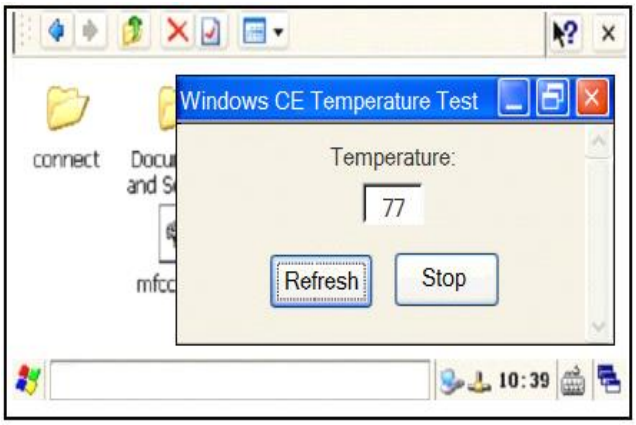

Figure 27. Windows CE temperature application

A methodology to develop the sensor application is described in this section. In particular, the bare machine interfaces to build sensor applications on ARM were identified, and internal design/implementation details of the relevant hardware API is presented. Functional operation and the bare machine aspects of ARM are also discussed. This development methodology can be adapted to build other bare sensor devices, and bare applications that are portable across a variety of mobile and pervasive devices.

SECTION IX

SIGNIFICANT CONTRIBUTIONS

This dissertation has many significant contributions and broader impact in software engineering and information technology. The investigations undertaken here infer many contributions as follows:

- The graphics API developed indicates that one can create a generic API that is applicable to many pervasive devices and their underlying architectures (e.g. a graphics code written for x86 can be used in ARM board).
- The device driver code developed for temperature sensor can be used for other sensor applications.
- The bare machine computing concept is applicable to any pervasive device (this dissertation demonstrated x86 and ARM).
- When pervasive devices are made bare, then they become ownerless, anyone can use your smart phone with their application (detachable storage that can boot, load and run a user's application).
- When direct hardware interfaces become generic, they can be implemented in the CPU and then software can access them directly similar to CPU instructions.
- When common software is identified for applications, which is independent of platform and CPU architecture, then immense productivity in software can be achieved.

SECTION X

SUMMARY

In order to demonstrate the applicability of bare machine computing on pervasive devices, this dissertation takes many investigative stages. Initially, a complex graphics application is written based on BMC concept that runs on an x86-based bare PC. This application is written in C/C++ programming language. A generic application graphics object is developed with API that illustrates some of the fundamental graphics elements and their functionality. The preliminary performance data indicates applicability of bare machine graphics for complex graphics applications. The direct hardware graphics API developed can be used in a variety of pervasive devices to achieve common graphics operations. The graphics API developed provides simplicity, eliminates abstraction layers, and self-contained as a single monolithic executable. In this approach, the programmer has direct access to the video graphics device and complete control of all hardware resources enabling autonomy with performance advantages due to elimination of system overhead.

As a next stage towards investigating this potential for a pervasive device, we transformed the x86-based bare PC graphics application to run on a bare ARM device. Key differences between the x86 and ARM architecture relevant to the transformation are identified. A methodology is described to transform the x86-based bare graphics application to run on the ARM architecture. Comparison of some timing measurements are presented for drawing graphics functions using the same bare application on an x86 bare PC, ARM development board, DOSBox, and QEMU-VM simulator. This approach

can serve as a basis to design bare machine applications in future so that they can run on a variety of devices with minimal code changes.

Finally, a novel bare machine temperature sensor application is developed. This application, which does not use any OS, kernel, or embedded system, has a direct API to communicate with and control the hardware. A methodology to develop bare sensor application for an ARM device is also given. In particular, the bare machine interfaces to build sensor applications on ARM devices are identified, and internal design/implementation details of the relevant hardware API are presented. The development methodology formulated in this stage can be adapted to build other bare sensor ARM applications, and bare applications that are portable across a variety of mobile and pervasive devices with ARM processors.

The above stages summarize this dissertation. Further research in this area can be pursued to investigate transformation to other pervasive devices, writing software that is independent of any platform and underlying architecture.

APPENDIX A

BARE DEVICE DRIVERS

GPIO Driver

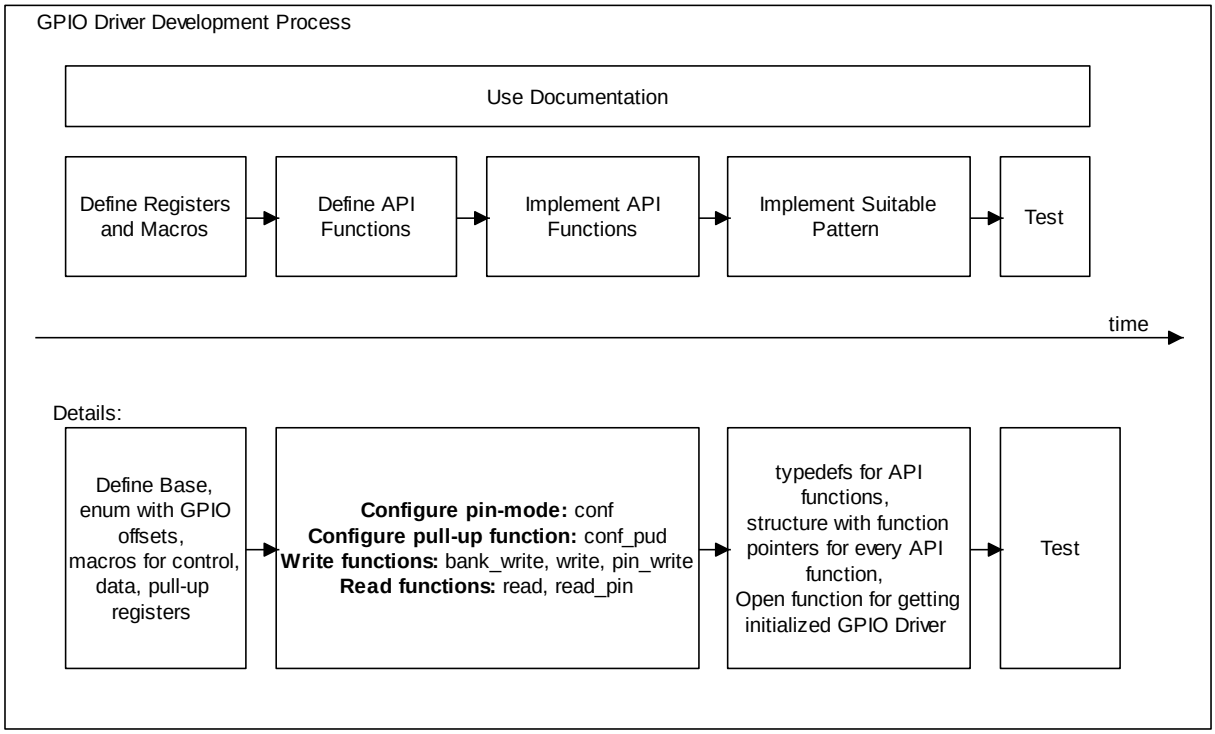

Figure 28. GPIO Driver Development Process

First thing that needs to be prepared is the definitions of the all necessary GPIO register – this can be found in the SoC user's guide – S3C6410 in this case [47].

Syntax for defining register is the following:

```
#define REGISTER_NAME (*((volatile unsigned *)REGISTER_ADDRES))
```

This is the most common pattern. This means we've defined 32-bit register (unsigned is 32-bit on ARM platform) with REGISTER_ADDRES, and we'll use REGISTER_NAME for use, definitely much more readable and comfortable than typing: `(*((volatile unsigned *)0x….))` every time.

Writing:

```
REGISTER_NAME = 123,
```

Reading:

```
unsigned value;
value = REGISTER_NAME;
```

We may notice repeating patterns very often, especially then there is many GPIO ports. In S3C6410 SoC there are 17 GPIO ports. After checking the SoC's user guide we may specify the following:

- GPIO base address: `0x7F008000`
- Port offsets for the every bank can be specified in the easily readable enum:

```
/** \enum GPIO_BANK
 * Bank definitions for GPIO.
 */
Typedef enum GPIO_BANK
{
GPIO_A=0x000,
GPIO_B=0x020,
GPIO_C=0x040,
GPIO_D=0x060,
GPIO_E=0x080,
GPIO_F=0x0A0,
GPIO_G=0x0C0,
GPIO_H=0x0E0,
GPIO_I=0x100,
GPIO_J=0x120,
GPIO_O=0x140,
GPIO_P=0x160,
GPIO_Q=0x180,
GPIO_K=0x800,
GPIO_L=0x810,
GPIO_M=0x820,
GPIO_N=0x830
}GPIO_BANK;
```

Now, macros for getting the addresses of the each GPIO register type:

For calculation of port base, usable in other macros:

```
#define GPIO_PORT_BASE(port) (port+GPIO_BASE)
```

For control register:

```
#define GPIO_CON_BASE(port) (GPIO_PORT_BASE(port))
```

For data register:

```
#define GPIO_DAT_BASE(port) (GPIO_PORT_BASE(port)+0x4)
```

For pull up register:

```
#define GPIO_PUD_BASE(port) (GPIO_PORT_BASE(port)+0x8)
```

Defined macros are just raw addresses; they can be easily treated as a register to write by using following macro:

```
#define GPIOREG(address) ((*((volatile unsigned *)address)))
```

Example usage of GPIO register – write some *value* to bank G:

```
GPIOREG(GPIO_DAT_BASE(GPIO_G))=value;
```

Of course this can be done in different way, for example GPIO_xxx_BASE can be register without another macro:

```
#define GPIO_DAT_BASE(port)
        ((*((volatile unsigned *)(GPIO_PORT_BASE(port)+0x4) )))
```

There are different approaches; however we're keeping the first idea for the rest of the analysis.

Based on what we already done, we can easily write common functions for GPIO, for example:

Write GPIO bank:

```
Static void gpio_write(GPIO_BANK bank,unsigned value,unsigned mask)
{
GPIOREG(GPIO_DAT_BASE(bank))=(GPIOREG(GPIO_DAT_BASE(bank))&
                                        (~mask))|(value&mask);
}
```

Read GPIO bank:

```
Static unsigned gpio_read_bank(GPIO_BANK bank)
{
  return GPIOREG(GPIO_DAT_BASE(bank));
}
```

Based on the analysis we've done, we know that following functions are needed for whole GPIO software support we need:

- Configure function – for setting the operation mode of pins

```
static void gpio_conf( GPIO_BANK bank, unsigned value,
                       unsigned mask );
```

- Configure pull-up function

```
static void gpio_conf_pud( GPIO_BANK bank, unsigned value,
                       unsigned mask );
```

- Write data to whole bank (set pin output of the whole bank by one function call)

```
static void gpio_bank_force_write( GPIO_BANK bank,
                       unsigned mask_value );
```

- Write data only to chosen pins of the bank

```
static void gpio_write( GPIO_BANK bank, unsigned value,
                       unsigned mask );
```

- Write data only to selected pin

```
static void gpio_pin_write( GPIO_BANK bank, unsigned pin,
                       unsigned value );
```

- Read the data from whole bank

```
static unsigned gpio_read_pin( GPIO_BANK bank, unsigned pin );
```

- Read the data from one pin of the chosen bank

```
static unsigned gpio_read_bank( GPIO_BANK bank );
```

Note: Detailed implementation is very simple and not really important at the moment.

Now, when functions are ready, and we're sure that taken and returned parameters suit all needs, it is recommended to create *typedefs* for all of the function, for example:

```
typedef void (*GPIO_CONF_BANK)( GPIO_BANK bank, unsigned value,
              unsigned mask );
typedef void (*GPIO_CONF_PUD_BANK)( GPIO_BANK bank, unsigned value,
              unsigned mask );
typedef void (*GPIO_FORCE_WRITE)( GPIO_BANK bank, unsigned value );
typedef void (*GPIO_WRITE)( GPIO_BANK bank, unsigned value,
              unsigned mask );
typedef void (*GPIO_PIN_WRITE)( GPIO_BANK bank, unsigned pin,
              unsigned value );
typedef unsigned (*GPIO_READ_PIN)( GPIO_BANK bank, unsigned pin );
typedef unsigned (*GPIO_READ_BANK)( GPIO_BANK bank );
```

Now it's easy to create a structure that we call the GpioDriver. This structure will consist of function pointers to every GPIO-related function.

Here's an example, including comments:

```
/** \struct GpioDriver
 *
 */
typedef struct GpioDriver
{
    /** Function for configuration of given GPIO_BANK
     *
     */
    GPIO_CONF_BANK conf_bank;

    /** Function for configuration pull-up resistors of given
      *  GPIO_BANK
     */
    GPIO_CONF_PUD_BANK conf_pud_bank;

    /** Function for writing \a value to the given \a bank
     * This function is quicker (compared to GPIO_WRITE function)
     * when modifying whole bank.
     * \note this function does not preserve any previously set pins.
     */
    GPIO_FORCE_WRITE write_forced;

    /** Function for calling Read-write-modify pattern at \a bank
     * Modifies bits only at given \a mask.
     */
    GPIO_WRITE write;

    /** Function for setting \a value at \a pin.
     *
     */
    GPIO_PIN_WRITE write_pin;

    /** This function returns state of a \a pin
```

```
     *
     */
    GPIO_READ_PIN read_pin;

    /** This function returns state of a \a bank
     *
     */
    GPIO_READ_BANK read_bank;
} GpioDriver;
```

Function that returns pointer to our driver is also necessary, declaration may look like this:

```
/* Use this function to open GPIO driver.
 * GpioDriver* that provides access to GPIO control functions.
 */
inline GpioDriver* gpio_driver_open();
```

Now the question is, how and where the GpioDriver structure should be initialized? Because of our approach and platform limitation it was decided to have only one static driver defined in the .c file. Based on what we already have, it can be easily initialized in the following way:

```
/** create and initialize static GPIO driver with proper function
  * pointers */
static GpioDriver gpio_driver =
                { gpio_conf, gpio_conf_pud, gpio_bank_force_write,
                  gpio_write, gpio_pin_write, gpio_read_pin,
                  gpio_read_bank };
```

Now, initialized gpio_driver structure can be returned in the gpio_driver_open function:

```
GpioDriver* gpio_driver_open()
{ return &gpio_driver; }
```

Sample usage:

Open the driver:

```
GpioDriver *gpio = gpio_driver_open();
```

Disable pullup/pulldown resistors in GPIO M bank for 0, 1, 2, and 3 pins (2 bit wide):

```
gpio->conf_pud_bank(GPIO_M,0x00,0xFF);
```

Configure GPM 0,1,2,3 as Output (fields 4bit wide):

```
gpio->conf_bank(GPIO_M,0x1111,0xFFFF);
```

Set pin states to HI:

```
gpio->write(GPIO_M,0xF,0xF);
```

GPIO Driver design and development is DONE!

Note: This kind of pattern with the structure that consists of function pointers is very common for developing drivers for SPI, I2C, LED, UART and many more peripherals.

Timer Driver

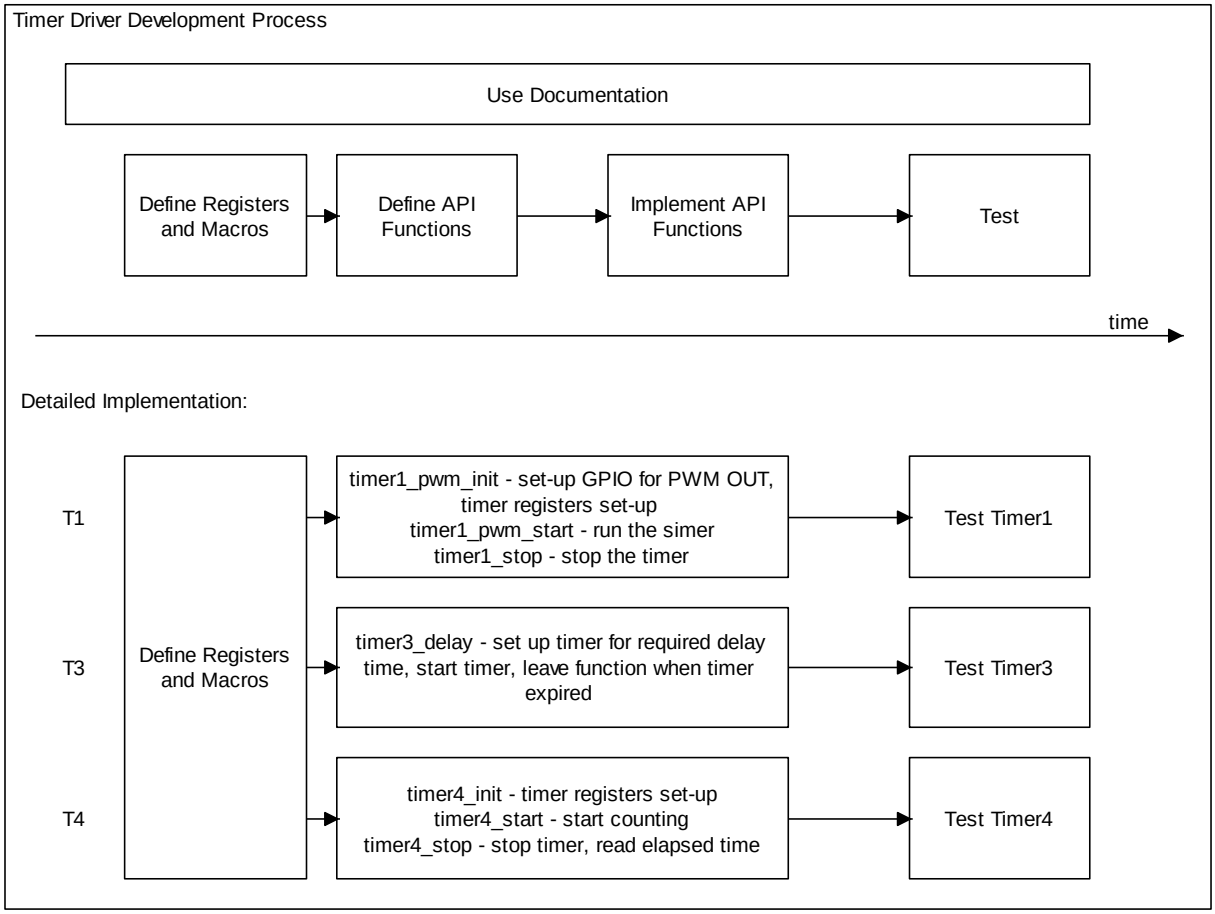

Figure 29. Timer Driver Development Process

Timers are the indispensable peripherals in the almost every system, their application is very wide. Typically they are used for: time measurement, task switching (scheduler), delays, Pulse Width Modulated (PWM) signals generation etc.

S3C6410 [47] provides five 32-bit timers. Three are used in:

- Timer 1 for PWM signal generation
- Timer 3 for simple delays
- Timer 4 for time measurements

Software for Timer usage is not developed in the pattern explained in GPIO Driver section; however it can be easily adjusted to the same pattern.

Timer 1

To use T1 for PWM signal generations, functions are available:

```
void timer1_pwm_init( unsigned char prescaler, TIMER_MUX mux );
```

This function initializes necessary timer registers, given values are prescaler and TIMER_MUX to set up frequency of generated output PWM signal.

GPIO driver is used to set up GPIO F pin 15 as PWM_TOUT1 (PWM Timer1 Output).

```
GpioDriver *gpio = gpio_driver_open();
gpio->conf_bank( GPIO_F, (0x2u<<30), 0xC0000000 );
```

Next step is to set up Timer Configuration Register TCFG0, TCFG1 for desired prescaler and TIMER_MUX (the divider of the prescaler).

Role of the prescaler is to divide input peripheral clock PCLK (66MHz in this case).

```
void timer1_pwm_start(PWM_RATIO ratio)
```

This function start generating PWM signal, PWM_RATIO describes the fill ratio of the PWM signal.

Before timer is started Timer Compare Buffer (TCMPB) and Timer Count Buffer (TCNTB) registers are filled with the given values of PWM_RATIO, this needs to be done twice because this timer is double-buffered.

Last step is to disable manual update bit and set the timer start bit to get the PWM output on the GPIO F15 pin.

```
void timer1_stop(void)
```

This function simply stops the timer by setting the timer start bit to zero.

Timer 3

This timer has only one function:

```
void timer3_delay(unsigned us);
```

Given parameter us is the microsecond amount desired to lock the program in the place when this function is called.

TCFG0 and TCFG1 registers are set up for prescalers and TIMER_MUX. These are set up to the lowest possible values so the timer has the maximum possible resolution.

Timer clock is calculated in the following way:

TCLK = PCLK / (prescaler+1) / MUX = 66MHz / (1+1) / 1 = 33MHz => 1 tick = .030*us*

As we can see, on this system timer clock tick takes 30 nanoseconds, and this is the timer highest possible timer resolution in this case.

Given *us* value is recalculated from the time base to the clock ticks and loaded into Timer Counter Buffer (TCNTB) register.

Timer is down-counting. When it reaches 0 interrupt can occur. For this simple delay case it is sufficient when the timer interrupt is enabled only on the timer peripheral level. This means Timer3 Interrupt Bit will be raised by hardware when timer reaches 0. Due to this approach, all we need is to poll the Timer3 interrupt bit in the while loop when it's on. This means desired time elapsed, so we can simply leave the function.

Timer 4

This timer API consists of the following functions:

```
void timer4_init(void);
```

Timer initialization done In the similar way than for Timer 3 within the differences:

- No maximum precision is needed, instead is better to set higher prescalers to have the possibility that longer time can be easily measured (timer is 32-bit, so it can "measure" 2^{32} (TIMER_MAX) of clock ticks, now it depends how much time takes one clock tick.

- Interrupt bit is not set up
- TIMER_MAX value is loaded into TCNT register

```
void timer4_start(void)
```

This function simply sets the start bit to 1 so the timer starts the down-counting

```
int timer4_stop(void)
```

This function stops the timer and returns measured time in microseconds.

First the TCNTO (Timer Counter Observation Register) is read to get the current *value* of the down-counting timer.

Timer is stopped and reloaded again with the maximum value for next use.

Last thing is to perform subtraction (TIMER_MAX – value). This gives the amount of clock ticks that elapsed since timer was started. Now, knowing the actual resolution of the timer set in the init function it is easy to recalculate the result and return it in microseconds.

UART Driver

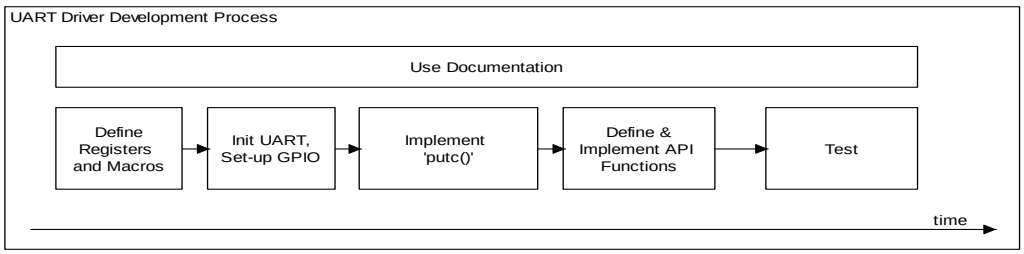

Figure 30. UART Driver Development Process

UART is used only for sending the bytes to the console so setup is easy [32].

Following steps need to be taken:

- Prepare necessary register and macro definitions
- Set-up UART peripheral (baud rate, start and stop bits, parity error)
- Set-up necessary GPIO pins to operate as pins for UART
- Implement “putc” function in a following way (simplest possible approach):
  - Function takes one char as input
  - When entered it waits until transmit buffer register will be empty
  - Previous condition is met, write character into transmit hold register (UART will take care of generate proper signal and send it through serial line)
  - (optional) When char is equal to new line (‘\n’) put additional caret return by additional recursive invoke of same function with the caret return sign ‘\r’.
- When putc function is implemented, printf-like functions can be easily implemented, in the most basic version this function may just take const char *

(simple null-ended string) and invoke putc function on every character until NULL character is found.

For debugging and printing data to the console some functions have been developed:

```
inline void my_putc(const char c)
```

This inline function prints one character on the screen. Waits until “transmit buffer empty” bit is set by UART, if it is set then writes given character to the UART transmit buffer.

```
void my_print(const char *str)
```

This function iterates over every character until NULL sign is found, calls my_putc for every character.

```
char *int_to_hex_str2(int val), char *int_to_dec(int val)
inline void print_int_16(int val), inline void print_int_10(int val)
```

These functions take the integer value and convert it into printable hexadecimal or decimal string format and prints directly on the screen.

Temperature Sensor Driver

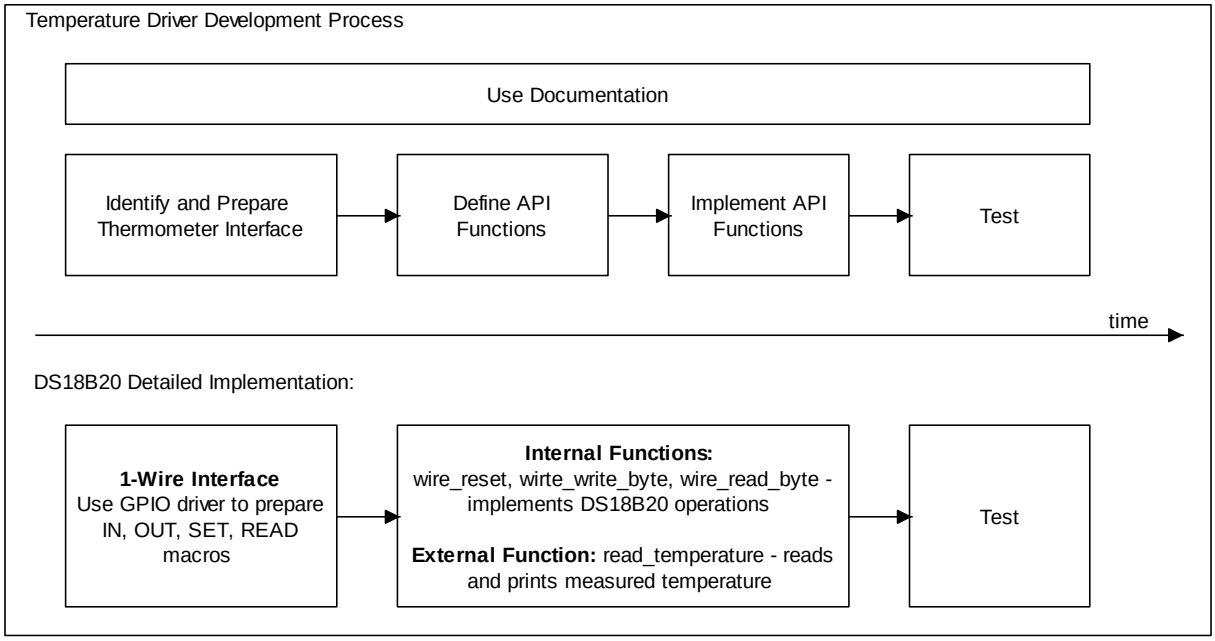

Figure 31. Temperature Driver Development Process

External temperature sensor DS18B20 [20] is connected to the OK6410 board.

In general have use of external peripheral, following initial steps need to be taken:

- Check the OK6410 schematic [47] and found the corresponding pins for the device.
- Check the thermometer datasheet for interface and pin description.
- Set up pins in the System on Chip (SoC) properly.

For the initial part in case of OK6410+DS18B20 we have:

- Thermometer uses only one-pin connection to the board, and that is GPIO E pin 0.
- Thermometer uses 1-Wire interface that requires manual interfacing setup.
- Pin can be set up by using GPIO Driver.

Now it is necessary to get to know how the device operates, what are the timings, what sequences are required to get the temperature? Everything is explained in the datasheet [20].

GPIO Driver and few additional helpful macros

We know we'll have to use the GPIO Driver to generate correct signal sequence for the temperature sensor.

Few additional macros will help with less-typing, and keep the code more readable:

```
#define GPIO_IN(drv) ( drv->conf_bank(GPIO_E, 0x0, 0xF) )
#define GPIO_OUT(drv) ( drv->conf_bank(GPIO_E, 0x1, 0xF) )
#define GPIO_SET(drv, state) ( drv->write_pin(GPIO_E, 0, state) )
#define GPIO_READ(drv) ( drv->read_pin(GPIO_E, 0) )
```

This macros provide: setting GPIOE0 as IN, setting GPIOE0 as out, setting GPIOE0 state to High or Low, Reading current GPIOE0 value

Note: Macros can be written also in the following way (and many other ways), so the GpioDriver pointer does not need to be passed:

```
#define GPIO_IN (gpio_driver_open()->conf_bank(GPIO_E, 0x0, 0xF) )
```

DS18B20 API

Based on the knowledge from the documentation, temperature sensor API cosists of the following functions:

```
unsigned char wire_reset(void);
void wire_write_byte(unsigned char data);
unsigned char wire_read_byte(void);
void read_temperature(void);
```

When the helpful macros are defined, all that is needed to be done is to write correct sequence of controlling GPIO Pins and providing delays (via timer3_delay function) when required.

Following sequence needs to be performed to obtain temperature:

- Perform reset on the wire via: **wire_reset**
- Write start temperature conversion command: **wire_write_byte: 0xCC, 0X44**
- Wait for required time until temperature is ready: **timer3_delay**
- Read the bytes from the SCRATCHPAD register: **wire_read_byte**
- Take bytes responsible for temperature (0 and 1) and convert them into human-readable temperature format, for example in Fahrenheit:

```
temperature=(int)((9./5.)*(((byte[1]<<8)| byte[0]) /10)+32);
```

- Measured temperature can be i.e. printed on the console:

```
my_print("\nTemperature [C]: ");
print_int_10(temperatureF);
```

Buzzer Driver

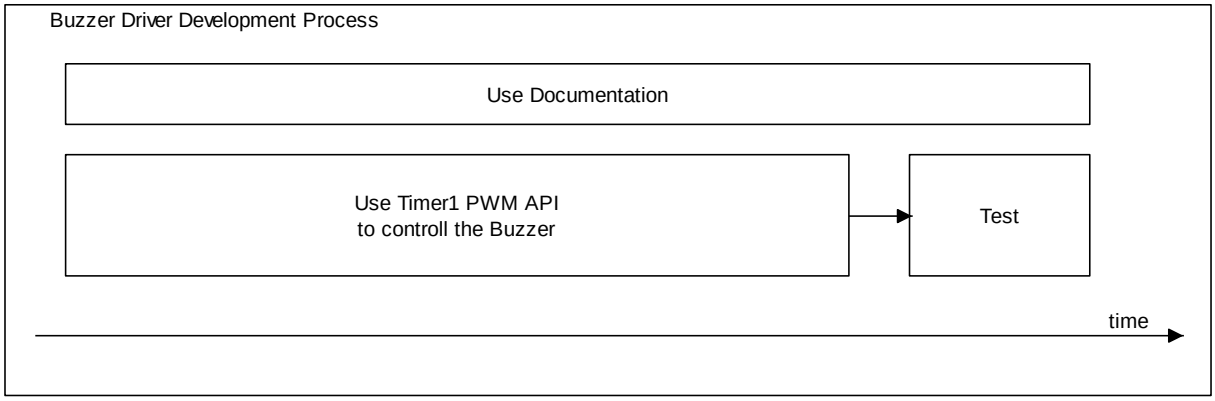

Figure 32. Buzzer Driver Development Process

Typically buzzer takes the PWM signal as input to "buzz."

- By checking the OK6410 board [47], we notice input to the Buzzer comes from GPIOF 15.
- Buzzer is connected to the T1PWMOUT pin by hardware designer, so now by setting up TIMER 1 properly we can get the device operational.
- By setting pre-scalers in timer1_pwm_init function tone of the buzzer can be controlled.
- By setting PWM_RATIO in the timer1_pwm_start loudness of the sound can be controlled.

For more details about TIMER 1 PWM control, please refer to the Timer Driver section.

LED Driver

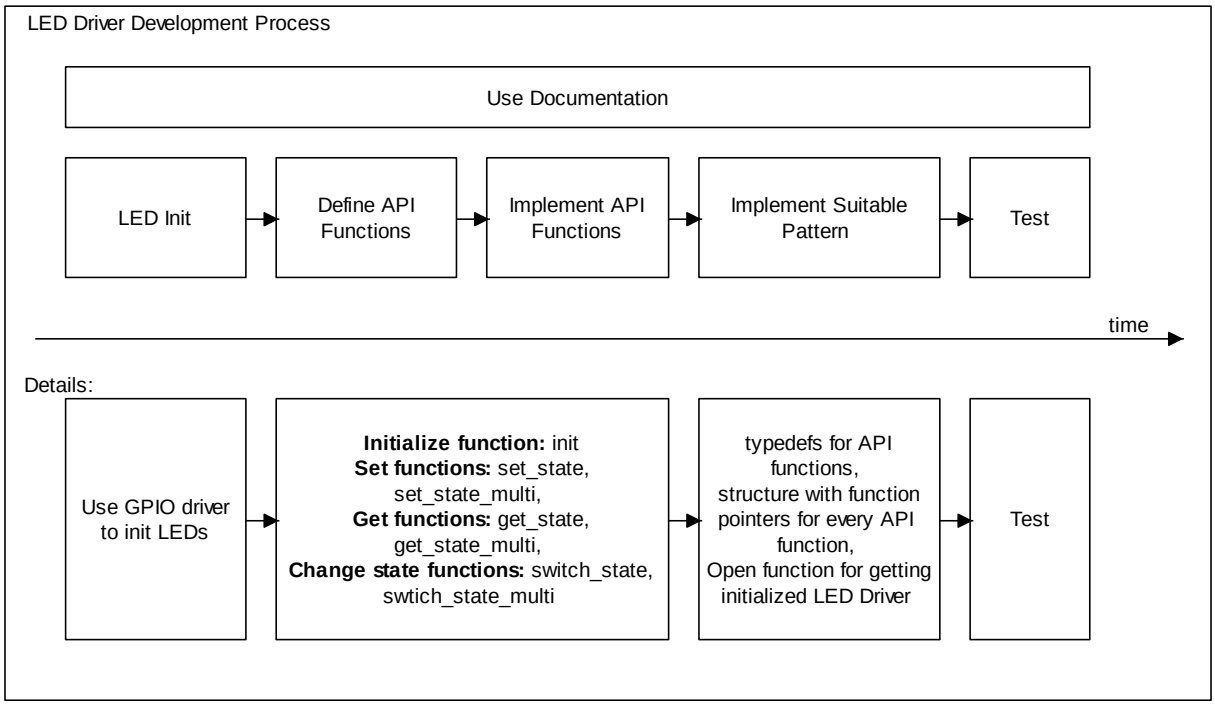

Figure 33. LED Driver Development process.

LED can be found on almost all embedded boards, as well as on OK6410.

- OK6410 board provides information that 4 LEDs are available to control.
- OK6410 schematic shows the LEDs connected to the GPIOM pins 0-4
- LEDs are controlled directly by the GPIO pins.

Basic GPIO LED Control

Most primitive possibility is to just drive corresponding GPIO pin to HI or LOW to get the LED on or off.

To do so, already developed GPIO Driver can be used in the following way:

Open the driver:

```
GpioDriver*gpio=gpio_driver_open();
```

Disable pullup/down for GPM 0, 1, 2 and 3:

```
gpio->conf_pud_bank(GPIO_M,0x00,0xFF);
```

Set GPM 0, 1, 2, and 3 as Output:

```
gpio->conf_bank(GPIO_M,0x1111,0xFFFF);
```

Set pin states to HI (disable LEDs):

```
gpio->write(GPIO_M,0xF,0xF);
```

Now the write_pin from GPIO API can be used to turn on or off LED, but we want to do this in nice and well organized way, so it is recommended to use the pattern described in the GPIO Driver.

Code above will be used for the led_driver_init function.

Whole API for the LED Driver may look like this:

Initialization function, already described:

```
void led_driver_init(void);
```

Set the *state* of the selected *led*

```
void led_set_state(LED_NUM led,LED_STATE state);
```

Set the of the selected *led_mask*according to *state_mask*

```
void led_set_multi(unsigned led_mask,unsigned state_mask);
```

Get the state of the selected *led*

```
unsigned led_get_state(LED_NUM led);
```

Get state of all LEDs in the “mask” format.

```
unsigned led_get_state_all(void);
```

Switch the state of the *led* (if is on, turn in off, if it’s off turn in on)

```
void led_switch(LED_NUM led);
```

Switch the state of LEDs selected in *led_mask*

```
void led_switch_multi(unsigned led_mask);
```

Enums for the LEDs are defined to have code easily readable:

For selecting the LED:

```
typedef enum LED_NUM
```

```
{
    LED_0=0,
    LED_1,
    LED_2,
    LED_3
}LED_NUM;
```

For Setting the LED State:

```
typedef enum LED_STATE
{
    LED_ON=0,
    LED_OFF=1
}LED_STATE;
```

Now in the same way as in GPIO Driver.

Typedefs for LED API:

```
typedef void(*LED_INIT_FUNC)(void);
typedef void(*LED_SET_FUNC)(LED_NUM led,LED_STATE state );
typedef void(*LED_SET_MULTI_FUNC)(unsigned led_mask,
            unsigned state_mask);
typedef unsigned(*LED_GET_FUNC)(LED_NUM led );
typedef unsigned(*LED_GET_ALL_FUNC)(void);
typedef void(*LED_SWITCH_FUNC)(LED_NUM led );
typedef void(*LED_SWITCH_MULTI_FUNC)(unsigned led_mask);
```

LedDriver structure that contains all LED API function pointers:

```
/** \structLedDriver
 *
 */
typedef struct LedDriver
{
    LED_INIT_FUNC init;
    LED_SET_FUNC set;
    LED_SET_MULTI_FUNC set_multi;
    LED_GET_FUNC get;
    LED_GET_ALL_FUNC get_all;
    LED_SWITCH_FUNC swith;
    LED_SWITCH_MULTI_FUNC switch_multi;
}LedDriver;
```

Getter for the driver:

```
LedDriver* led_driver_open(void);
```

LedDriver static initialization in the .c file:

```
static LedDriver led_driver={
    led_driver_init,
    led_set_state,
    led_set_multi,
    led_get_state,
    led_get_state_all,
    led_switch,
    led_switch_multi
};
```

Returning the initialized driver:

```
LedDriver* led_driver_open(void)
{
    return &led_driver;
}
```

Sample usage – MOD16 Counter:

In this example LedDriver is opened, LEDs are turned off, and then LED driver sets the state of all LEDs depending on the value of the led_counter. The led_counter&0xF operation makes sure that only 4-bit value is provided to the LED.

```
unsigned led_counter=0;
LedDriver *led_driver=0;
led_driver=led_driver_open();
led_driver->init();
led_driver->set_multi(0xF,0x0);

while(1)
{
led_driver->set_multi(0xF,led_counter&0xF);
++led_counter;
timer3_delay( 1000*1000 );
}
```

LCD Driver

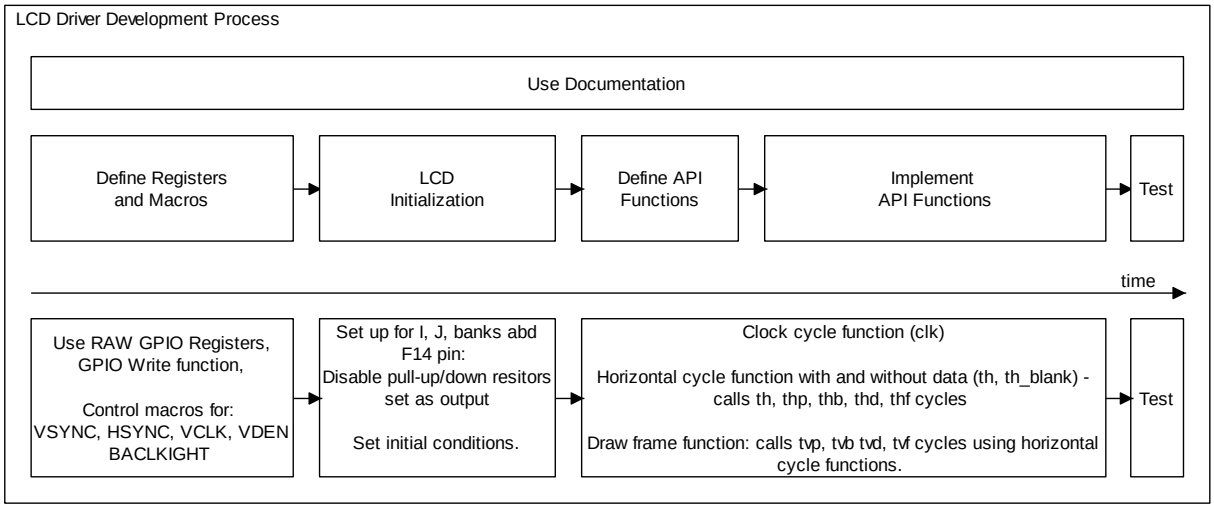

Figure 34. LCD Driver Development Process

Define Registers and Macros

Please note that in this special case, previously described GPIO driver won't be used. This is because code execution performance is absolutely critical in this case, so code for controlling needs to be highly optimized as per the datasheet [56].

GPIO Registers definitions used in the LCD driver:

```
#define GPICON          (*((volatile unsigned *)0x7F008100))
#define GPIDAT          (*((volatile unsigned *)0x7F008104))
#define GPIPUD          (*((volatile unsigned *)0x7F008108))
#define GPICONSLP       (*((volatile unsigned *)0x7F00810C))
#define GPIPUDSLP       (*((volatile unsigned *)0x7F008110))
#define GPJCON          (*((volatile unsigned *)0x7F008120))
#define GPJDAT          (*((volatile unsigned *)0x7F008124))
#define GPJPUD          (*((volatile unsigned *)0x7F008128))
#define GPJCONSLP       (*((volatile unsigned *)0x7F00812C))
#define GPJPUDSLP       (*((volatile unsigned *)0x7F008130))
#define GPFCON          (*((volatile unsigned *)0x7F0080A0))
#define GPFDAT          (*((volatile unsigned *)0x7F0080A4))
#define GPFPUD          (*((volatile unsigned *)0x7F0080A8))
#define GPFCONSLP       (*((volatile unsigned *)0x7F0080AC))
#define GPFPUDSLP       (*((volatile unsigned *)0x7F0081B0))
```

Control macros (data write, VSYNC, HSYNC, CLK, data enable, backlight control):

Data write macros:

```
#define GPIO_WRITEH(d)   ( GPIDAT=(d>>8))
#define GPIO_WRITEL(d)   ( GPJDAT=(0x000000FF&d)|(0xFFFFFF00&GPJDAT))
```

Vertical sync control macros:

```
VSYNC output level to H
#define VSYNC_H          ( GPJDAT |= (1<<9))
VSYNC output level to L
#define VSYNC_L          ( GPJDAT &= (~(1<<9)))
```

Horizontal sync control macros:

```
HSYNC output level to H
#define HSYNC_H          ( GPJDAT |= (1<<8))
HSYNC output level to L
#define HSYNC_L          ( GPJDAT &= (~(1<<8)))
```

Clock control macros:

```
VCLK output level to H
#define VCLK_H           ( GPJDAT |= (1<<11))
VCLK output level to L
#define VCLK_L           ( GPJDAT &= (~(1<<11)))
```

Data enable control macros:

```
DE (data enable) output level to H
#define VDEN_H           ( GPJDAT |= (1<<10))
DE (data enable) output level to L
#define VDEN_L           ( GPJDAT &= (~(1<<10)))
```

Backlight control macros:

```
turn the backlight on
#define BACKLIGHT_ON (GPFDAT |= (1<<14))
turn the backlight off
#define BACKLIGHT_OFF (GPFDAT &= ~(1<<14))
```

Initialization:

```
Void sw_display_init( unsigned line)
```

This function:

- Sets Pin 14 of bank F pin as output
- Sets all pins of banks I and J as output
- Disables pull up/down resistors for all above.
- Turns on the backlight
- Set clock line to low
- Set vsync line to high
- Set hsync line to high

Define and Implement API Functions

Clock cycle function

This function will generate requested *clk_amount* clock cycles.

```
inline void clk( int clk_amount )
```

Horizontal cycle functions

This function generate one horizontal cycle and draws one line on the screen.

Line parameter helps to define which line is drawing.

```
void gfx_th( unsigned line)
```

Internally this function has the loop that runs for the amount of horizontal pixel amount of the screen and puts each pixel data for the provided line.

The line and the loop index identifies the pixel of the screen so it’s easy to provide a table with the color data for the pixels.

This functions will generate one horizontal cycle without drawing.

```
void th_blank( void )
```

Draw Frame Function

This function draws one full frame on the screen.

```
void sw_draw_gfx_frame( void )
```

It calls the required th_blank sequences and calls gfx_th function for the amount of lines (vertical size) that the LCD screen has.

Touch Screen Driver

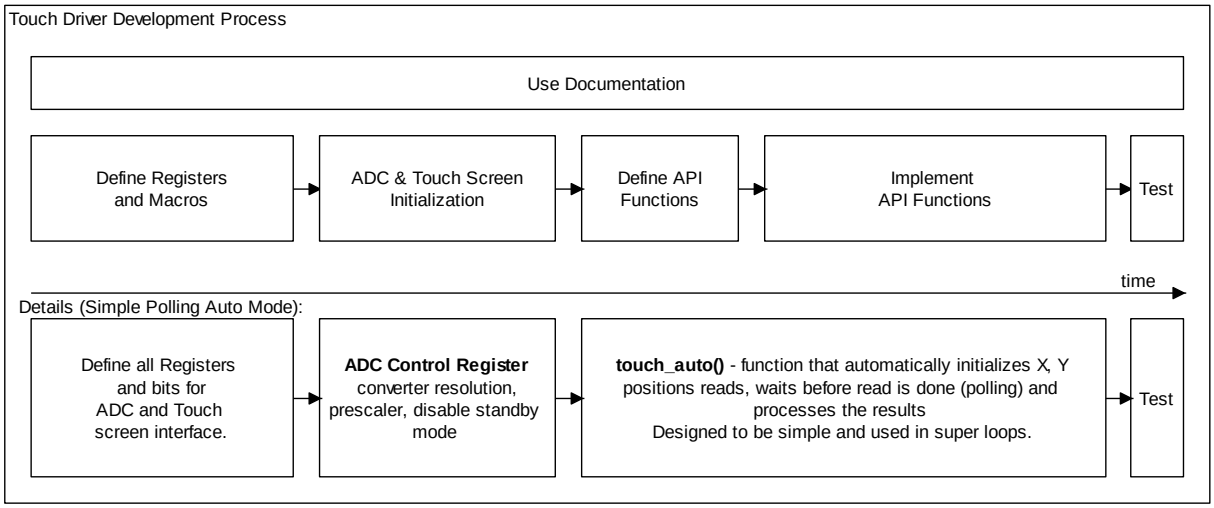

Figure 35. Touch Driver Development Process

Difficulty of writing a driver for Touch Screen may strongly vary depends on the amount of features that the driver should provide.

We're explaining a basic touch-driver that works in the following way:

- Touch event occurs on the screen
- Driver reads the ADC measurements
- Driver converts the ADC measurements to the X-Y position coordinates in pixel
- Driver processes the results

Define Registers and Macros

For the ADC & Touch screen section we need to define all ADC and Touch screen interface registers, and define all used bits as per the datasheet [56].

Note: These registers can be use to handle "pure" ADC operations like converting voltage attached to some input, but this case is simple and not very interesting in comparison to Touch screen support so it's out of scope of this document.

Following registers can be defined for the ADC:

ADC Control register:

```
#define ADCCON      (*((volatile unsigned int *)0x7E00B000))
```

ADC Touch Screen Control Register:

```
#define ADCTSC      (*((volatile unsigned int *)0x7E00B004))
```

Start or Interval Delay Register:

```
#define ADCDLY      (*((volatile unsigned int *)0x7E00B008))
```

Conversion Data Registers (Read Only):

```
#define ADCDAT0     (*((volatile unsigned int *)0x7E00B00C))
#define ADCDAT1     (*((volatile unsigned int *)0x7E00B010))
```

Stylus Up or Down Interrupt Register:

```
#define ADCUPDN     (*((volatile unsigned int *)0x7E00B014))
```

Clear interrupt registers:

```
#define ADCCLRINT   (*((volatile unsigned int *)0x7E00B018))
#define ADCCLRINTPNDNUP (*((volatile unsigned int *)0x7E00B020))
```

Following bits can be defined for the ADC/Touch screen:

```
/* ADCCON Register Bits */
#define S3C_ADCCON_RESSEL_10BIT         (0x0<<16)
#define S3C_ADCCON_RESSEL_12BIT         (0x1<<16)
#define S3C_ADCCON_ECFLG                (1<<15)
#define S3C_ADCCON_PRSCEN               (1<<14)
#define S3C_ADCCON_PRSCVL(x)            (((x)&0xFF)<<6)
#define S3C_ADCCON_PRSCVLMASK           (0xFF<<6)
#define S3C_ADCCON_SELMUX(x)            (((x)&0x7)<<3)
#define S3C_ADCCON_SELMUX_1(x)          (((x)&0xF)<<0)
#define S3C_ADCCON_MUXMASK              (0x7<<3)
#define S3C_ADCCON_RESSEL_10BIT_1       (0x0<<3)
#define S3C_ADCCON_RESSEL_12BIT_1       (0x1<<3)
#define S3C_ADCCON_STDBM                (1<<2)
#define S3C_ADCCON_READ_START           (1<<1)
#define S3C_ADCCON_ENABLE_START         (1<<0)
#define S3C_ADCCON_STARTMASK            (0x3<<0)
```

```
/* ADCTSC Register Bits ADC Touch Screen*/
#define S3C_ADCTSC_UD_SEN         (1<<8)
#define S3C_ADCTSC_YM_SEN               (1<<7)
#define S3C_ADCTSC_YP_SEN         (1<<6)
#define S3C_ADCTSC_XM_SEN         (1<<5)
#define S3C_ADCTSC_XP_SEN         (1<<4)
#define S3C_ADCTSC_PULL_UP_DISABLE      (1<<3)
#define S3C_ADCTSC_AUTO_PST       (1<<2)
#define S3C_ADCTSC_XY_PST(x)      (((x)&0x3)<<0)
/* ADCDAT0 Bits */
#define S3C_ADCDAT0_UPDOWN        (1<<15)
#define S3C_ADCDAT0_AUTO_PST      (1<<14)
#define S3C_ADCDAT0_XY_PST        (0x3<<12)
#define S3C_ADCDAT0_XPDATA_MASK   (0x03FF)
#define S3C_ADCDAT0_XPDATA_MASK_12BIT   (0x0FFF)
/* ADCDAT1 Bits */
#define S3C_ADCDAT1_UPDOWN              (1<<15)
#define S3C_ADCDAT1_AUTO_PST            (1<<14)
#define S3C_ADCDAT1_XY_PST              (0x3<<12)
#define S3C_ADCDAT1_YPDATA_MASK         (0x03FF)
#define S3C_ADCDAT1_YPDATA_MASK_12BIT   (0x0FFF)
```

General Information about ADC and Touch Screen:

Because of many possible operating modes for ADC and touch screen, this chapter provides overview information and possibilities of setting up and using the touch screen.

ADC & Touch Screen Initialization:

touch_init function can be defined to provide initialization code of the touch support. This function need to set the:

ADC Control Register

- A/D Converter resolution (10 or 12-bits)
- A/D Converterprescaler settings – A/D converters have limits on their conversion speeds. For example A/D converter integrated into S3C6410 can operate at maximum 5MHz clock speed and conversion

takes 5 cycles, so theoretically it can convert up to 1 million samples per second.

- A/D operates in the peripheral clock domain (PCLK) so prescaler needs to be set respectively to the PLCK. For example when PCLK operates at 50MHz, prescaler needs to be enabled and it's value set at least to 9, that will give the division factor = 10 so the A/D operational speed will be 5MHz.
- End of Conversion flag is useful to check if the new conversion results are ready.

Touch Screen Control Register

In this register it is required to setup:

- XM, XP, YM, YP control inputs,
- disable/enable pull-up resistors
- X/Y measurement mode (i.e. wait for interrupt mode) and type of conversion (manual, automatic)

Define and Implement API function:

Read conversion results function

This function should provide capability to read the ADC conversion results.In case manual measurements are chosen it should also initialize the A/D conversion from the required inputs.

ADC Interrupt Polling function or ADC Interrupt Routine Function

Depends on the solution and previously chosen measurement mode and software requirements two types of "waiting" for touch event are possible

- Polling – constantly check the bit that indicates the screen have been touched – this is very simple to test and use, but obviously locks the software for the time until touch event will occur
- ISR – in this type of handling, an registered ADC interrupt service routine will be called when touch event occurs. This mode additionally requires of Vectored Interrupt Controller (VIC) setup, but it's typically used and does not lock the software execution.

No matter the choice, next steps are similar: when touch event occurs, handler needs to call previously defined *read conversion results function* to obtain the X/Y position data, clear up interrupt statuses and call the provided *callback function* that i.e. prints the indicated results in the console, draw something on the display etc. Of course any action can be performed directly in this function but using a callback is the common usage that gives more flexibility.

Touch Screen simple Polling Auto Mode Example:

ADC Touch Screen Initialization

ADC/Touch screen peripheral provides "Auto conversion mode" functionality. To use it, ADC control register needs to be initialized in the following way:

- Set the proper ADC conversion frequency (1MHz) by enabling prescaler and setting proper value to it. A/D converter frequency is calculated in the following way: $F_{A/D}$=PCLK/(PRESCALER_VAL+1), so for our system:

  $F_{A/D}$=66MHz/(65+1)=1MHz

```
ADCCON = S3C_ADCCON_PRSCEN | S3C_ADCCON_PRSCVL(65);
```

- Set A/D resolution do 12-bit

```
ADCCON |= S3C_ADCCON_RESSEL_12BIT;
```

- Disable Standby Mode:

```
ADCCON &=~S3C_ADCCON_STDBM;
```

Define & Implement Auto Mode converting function

This function operates in the following steps:

```
void touch_loop_auto(void)
{
unsigned int x=0;
unsigned int y=0;
//STEP 1 Enable AutoXY conversion mode
    ADCTSC = S3C_ADCTSC_AUTO_PST;
//STEP 2 Start A/D conversion
    ADCCON |= S3C_ADCCON_ENABLE_START;
//STEP 3 Wait until conversion for X and Y is done ("wait for
  interrupt mode" bits are set in the registers.)
while((ADCDAT0 &(S3C_ADCDAT0_XY_PST | S3C_ADCDAT0_AUTO_PST))
&&(ADCDAT1 &(S3C_ADCDAT1_XY_PST | S3C_ADCDAT1_AUTO_PST))
&&(ADCTSC & S3C_ADCTSC_XY_PST(3)))
{
}
//STEP 4 Read and store the data
x= ADCDAT0 & S3C_ADCDAT0_XPDATA_MASK_12BIT;//for 12bit
y= ADCDAT1 & S3C_ADCDAT1_YPDATA_MASK_12BIT;//for 12bit
//STEP 5 Clear "wait for interrupt mode" and clear the
  interrupt statuses
    ADCTSC &=~S3C_ADCTSC_XY_PST(3);//clr wait for interrupt
    ADCCLRINT =1;
    ADCCLRINTPNDNUP =1;
//STEP 6 Process the x, y coordinates as your choice i.e. print
  the results on the console
print_int_16(x);
my_putc(' ');
print_int_16(y);
endl();
}
```

Note: This function is desired to run sequentially-in infinite loop. If we move the pen on the screen, the function will constantly print the X Y results in the console.

APPENDIX B

DEBUGGING AND MONITORING

Debugging and Reverse Engineering

Bare applications are well know from being hard to debug, however there are still some ways for debugging even on quite simple set-ups.

One of the solutions might be to initialize UART, implement simple printf-like functions, send data to PC and observe results on terminal application. This is common and simple solution but not always suitable. When it comes to time-critical operation sending bytes of data through UART even on simplest possible implementation is slow in comparison to raw ARM core instruction/code processing speed.

Second solution that is helpful, especially in situation when we don't want to touch the executing code is to monitor external signals that are coming out of the processor chip and check if they are correct. This kind of approach requires additional hardware such as interface signal analyzer and oscilloscope. For most cases quite simple oscilloscopes are completely enough to analyze the all possible signals, but even that kind of models are still large and expensive. Dedicated signal analyzers have more limited functionality (mostly to monitor signals from GPIO and serial interfaces such as SPI, UART, I2C etc.), but they are very cheap in comparison to oscilloscopes.

Last but not least and very common recommended solution is to use dedicated JTAG. Modern JTAGs typically use Ethernet or USB interface to connect with PC and dedicated interface to connect with CPU core. JTAGs are very useful when debugging software, they can halt CPU core any time and read register and data accessible by CPU core at the moment, so they are suitable to obtain and verify if the program flow is proper, registers

are initialized with good values. They are also very helpful with reverse engineering. Depending on the features JTAG devices can be very cheap, but have limited core types support and limited functionality. Cheaper models may not support erasing of NAND/NOR flash memories, which makes it very limited for non-microcontroller CPUs (i.e. SEGGER J-Link JTAG[44] supports flash memories from ARM7 and Cortex M-3 based CPUs but does not support NAND flash i.e. on Samsung S3C6410 ARM11 based CPU). Expensive, but full-blown JTAGs like those provided by Lauterbach company, provides multiple core support, NAND/NOR flash support, and even own IDE for debugging, however they can be afforded by medium and large companies. Cheaper JTAGs instead of having own IDE and debugging interface use OpenOCD or popular GNU Debugger setup (GDB + GDBServer).

Implementing a bare printf function

To implement bare printf function:

- Set-up UART peripheral (baud rate, start and stop bits, parity error)
- Implement “putc” function in a following way:
    - function takes one char
    - when entered it waits until transmit buffer register will be empty
    - previous condition is met, write character into transmit hold register (UART will take care of generate proper signal and send it through serial line)
    - (optional) when char is equal to new line (‘\n’) put additional caret return by additional recursive invoke of same function with the caret return sign ‘\r’

- When putc function is implemented, we need to implement print function, in the most basic version this function may just take const char * and invoke putc function on every character until NULL character is found.

Bare Application Debugging

Debugging bare metal application on OK6410 board [47] (Samsung S3C6410 CPU with ARM1176-JZF Core) with Segger J-Link JTAG [48] is presented in this section as an example.

JTAG Setup

To setup JTAG:

- Download and install J-Link ARM drivers on PC. Before the installation, please make sure that J-Link GDB server is installed.
- Power up CPU/Target board, connect JTAG to PC via USB and to CPU via dedicated connector.
- Launch SEGGER J-Link GDB Server from toolset installed together with J-Link drivers
- J-Link GDB Server will detect target CPU Core, voltage power etc.
- From the log output we may read information like: “Listening on TCP/IP port 2331”, this means GDB Server is ready and waits for connection.
- JTAG setup is ready

GDB setup and run

While on desktop debuggers are used directly on the executing machine, it will be extremely difficult to not only setup but also use GDB itself. GDB Server + GDB provides suitable solution for cross-development of embedded software.

GDB (GNU Debugger) from currently installed toolchain can be used to connect with the GDB Server that is running on J-Link. To do so, we can connect to already running GDB Server manually and set up everything by invoking few commands, but instead of this, GDB “startup script” feature can be used.

In the directory where GDB is located (arm-none-eabi-gdb.exe in case of Sourcery G++ Lite), “.gdbinit” file needs to be created. File with that name, when exists, is automatically executed when GDB starts it work.

Contents of .gdbinit are dependent on the JTAG and CPU. If JTAG supports GDB Server feature, reading JTAG dedicated user guide is recommended. for J-Link + ARM11 core, simplest startup may look like this:

```
# connect do GDB Server on localhost at tcp port 2331
target remote localhost:2331
# Set JTAG speed to 30 kHz
monitor speed 30
# Set GDBServer to big endian
monitor endian big
# Reset the chip to get to a known state.
monitor reset
# Set auto JTAG speed
monitor speed auto
# Setup GDB FOR FASTER DOWNLOADS
set remote memory-write-packet-size 1024
set remote memory-write-packet-size fixed
```

Now GDB is ready to run. After invoking “arm-none-eabi-gdb” command, J-Link GDB Server will show GDB status as “Connected to 127.0.0.1” which means everything is up and running, and GDB is successfully attached to core. Issuing “monitor” command on GDB prompt will show possible commands.

Monitor Sample GDB session

GDB is ready to go, CPU is in reset because “monitor reset” command was invoked during startup.

For simplest debug session, following commands may be typed:

```
monitor go – run target CPU
monitor halt – halt target CPU
monitor regs – display all CPU Register contents
monitor step – step by one instruction
monitor MemU32 <address > - reads contents from given addres, for
example monitor MemU32 0x0 will read current reset vector
monitor WriteU32 <address><value> - write value at the given address
```

Reverse Engineering Example

Sometimes, for example when documentation is incomplete, it’s necessary to perform reverse engineering. In such situation JTAG can be very useful. Example in this section will demonstrate how to reverse engineer via J-Link JTAG when Linux is running.

For purpose of this example, let’s say we would like to check S3C6410 Display Controller (DC) registers Linux X-Window system is running.

Since Linux is using Memory Management Unit (MMU) virtual addresses of Display Controller are different to phisycal ones. S3C6410 peripheral addresses can be found in the linux source code under the following path: arch/arm/plat-s3c/map-base.h.

This file contains:

```
#define S3C_ADDR_BASE (0xF4000000)
#define S3C_VA_LCD    S3C_ADDR(0x00600000)
```

So, the DC virtual address base is 0xF4600000. Specific registers can be checked in regs-lcd.h file, or directly from S3C6410 datasheet (recommended). Please note that datasheet contains physical register addresses, so their base needs to be changed. Physical DC address base is 0x77100000, but it this case “visible” address is 0xF4600000 and this

base address needs to be used in the GDB command prompt when Linux is running, so for example VIDTCON0 register have address 0x77100010, which means under Linux it will have virtual address 0xF4600010.

Console output of reverse engineering session of DC register with the use of GDB+GDBServer+J-Link is listed below:

```
(gdb) monitor reset
Resetting target
(gdb) monitor go
<wait until Linux boots>
(gdb) monitor halt
(gdb) monitor memU32 0xF4600010
Reading from address 0xF4600010 (Data = 0x00010109)
(gdb) monitor memU32 0xF4600000
Reading from address 0xF4600000 (Data = 0x00000353)
(gdb) monitor memU32 0xF4600004
Reading from address 0xF4600004 (Data = 0x0042C060)
(gdb) monitor memU32 0xF4600008
Reading from address 0xF4600008 (Data = 0x00000000)
(gdb) monitor memU32 0xF4600010
Reading from address 0xF4600010 (Data = 0x00010109)
(gdb) monitor memU32 0xF4600014
Reading from address 0xF4600014 (Data = 0x00010128
```

In this case registers can be easily read and DC settings can be easily read and verified.

APPENDIX C

ECLIPSE IDE, TOOLCHAINAND START-UP CODE

Eclipse and Toolchain setup

There are many commercial/open source IDE and toolchains, we used the Open Source Eclipse IDE [21] to ease the development and free Lite version of SourceryCodeBench [34] for the toolchain.

Environment setup steps

1. Download and install SourceryCodeBench Lite Edition

   http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/

   Note: Adding toolchain into PATH is recommended. This will help to detect the toolchain automatically by Eclipse IDE.

2. Download Eclipse ide for C/C++ Developers (Indigo 3.6 is assumed)

   http://www.eclipse.org/downloads/

   Note: Other version of Eclipse can be used as well, but additional plug-insmaybe required.

3. Download GNU ARM Eclipse Plug-in [28]. This will greatly help with the development for ARM from here - http://sourceforge.net/projects/gnuarmeclipse/

4. Launch the Eclipse and install the GNUARM plugin.

   When in workspace, choose Help->Install New software. Choose Add->Archive and select previously downloaded archive with the GNUARM plugin, or just type in the “Work with” section the following: jar:file:/*path_to_plugin_file*/*zip_name*!/

When GNU ARM C/C++ Development Support is discovered, we may select the plugin and finish the installation. Additional Eclipse restart may be required.

5. Next, we may proceed with creating the project. In Project Explorer window, press Right mouse button and Choose New->C Project->Makefile Project->Empty Project. Now thanks to the GNUARM plugin, selecting of toolchain is possible. ARM Windows GCC (Sourcery G++ Lite) was used during the LCD Driver development. Choose the project name and whole environment setup procedure is completed.

Additional tools

When developing under multiple systems (i.e. under Windows and Linux), the shell commands are different. There are free Coreutils – GNU core utilities for linux tools available [11]. Also, installing Coreutils on windows (GnuWin [12]) is recommended to obtain ability of using commands likels, pwd, rm etc., on MS Windows OS.

To use Coreutils for Windows:

- Download GnuWincoreutils package i.e. from SourceForge web page[12]
- Install, during installation add GnuWin32\bin directory to your system PATH, or after installation update PATH manually (i.e. for default directory this might be C:\Program Files\GnuWin32\bin)
- Linux-like commands are ready to use, run cmd and type i.e. ls to check that they are working, if not, check PATH variable, if it's not updated – re-log to system.
- One great advantage is that i.e. in Makefile we can use "rm" in clean section. Issuing "make clean" in the project directory will simply work on Windows and Linux when Makefile will be updated in the following way:

The Makefile

For quick and easy build of code, by using *make* tool from toolchain Makefile is written for this project.

Variable definitions for the current toolchain:

```
TRGT = arm-none-eabi-
CC = $(TRGT)gcc
CP = $(TRGT)objcopy
AS= $(TRGT)as
BIN = $(CP) -O binary
HEX = $(CP) -O ihex
```

Project name variable:

```
PROJECT=b2
```

Sources variables:

```
SRC = ./main.c
ASRC = ./start.s
```

Optimization level definition:

```
OPT = -O2
```

CPU specific variables:

```
MCU = arm1176jzf-s
MARCH = -march=armv6
MCPU = -mcpu=$(MCU)
```

Linker script file:

```
LDSCRIPT_ROM = arm1176.ld
```

Compiled objects variables:

```
OBJS = $(ASRC:.s=.o) $(SRC:.c=.o)
```

Flags for building assembly and C files

```
ASFLAGS = $(MCPU) $(MARCH) -Wall
CPFLAGS = $(MCPU) $(OPT) –Wall
```

Flags for building the elf file:

```
LDFLAGS_ROM = $(MCPU) -nostartfiles -T$(LDSCRIPT_ROM) -Wl,
 -Map=$(PROJECT).map,--cref,--no-warn-mismatch
```

The definition of what will happen when make all command is issued:

```
all: ROM
```

where:

```
ROM: $(OBJS) $(PROJECT).elf $(PROJECT).hex $(PROJECT).bin
```

Compile the .c files to .o files with previously set flags:

```
%o : %c
    $(CC) $(CPFLAGS) $< -o $@
```

Compile the .s files to .o files with previously set flags:

```
%o : %s
    $(AS) $(ASFLAGS) $< -o $@
```

Generate elf file from compiled .o files with the given linker flags

```
%.elf: $(OBJS)
    $(CC) $(OBJS) $(LDFLAGS_ROM) -o $@
```

Generate hex file:

```
%hex: %elf
    $(HEX) $< $@
```

Generate bin file:

```
%bin: %elf
    $(BIN) $< $@
```

Assembly startup code (start.s)

Simple assembly code is required because code does not rely on any extra libraries. Only one reason of this is to enter the main() function in the code, because normally programs are starting at _start in the main function, and without default libs there is no such routine.

```
.section.text,"ax"    /* GNU assembler directive to
                       * append to end of the text subsection */
.code 32              /* selects the 32-bit ARM instruction set */
.extern main

_start
mov   r0, #0          /* No arguments */
mov   r1, #0          /* No arguments */
ldr   r2, =main
mov   lr, pc
bx    r2              /* branch to the main() function */
```

APPENDIX D

RUNNING APPLICATION ON ADB

It is important to be able to quickly check the code during development. One of the possibilities to run the test is to use U-Boot. Running compiled binary via U-Boot[19] is the following:

1. Build the project. Binary .bin file is generated during the build of this project.
2. Download the binary into RAM at specified address. This can be done i.e. via dnw command or tftp.
3. Use go command to begin executing of the binary. For example if binary is loaded at address 0xC0008000, just type "*go C0008000*".

   The "go" command will cause to load specified address into PC register (program counter) and start executing from this address.

   Program begins execution of the first instruction in the _start section of start.s assembly file compiled in this project.

REFERENCES

[1] A. Alexander, A. L. Wijesinha, and R. Karne, “A Study of Bare PC SIP Server Performance,” 5th International Conference on Systems and Networks Communications (ICSNC), pp. 392 – 397, August 2010.

[2] A. Alexander, A. L. Wijesinha, and R. Karne, "Implementing a VOIP Server and a User Agent on a Bare PC," The Second International Conference on Future Computational Technologies and Applications, Future Computing, Portugal, Lisbon, pp. 8 – 13, November 2010.

[3] A. Peter, R. K. Karne, and A. L. Wijesinha, "A Bare Machine Sensor Application for an ARM Processor," in Proc. 2013 IEEE Electro/Information Technology Conference, Rapid City, SD, USA, May 9-11, 2013.

[4] A. Peter, R. K. Karne, A. L. Wijesinha and P. Appiah-Kubi, “The Design and Implementation of Bare PC Graphics,” 7th International Multi-Conference on Computing in the Global Information Technology (ICCGI) pp. 315-320, 2012.

[5] A. Peter, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi, "Transforming a Bare PC Application to Run on an ARM Device," in Proc. 2013 IEEE Southeastcon Conference, Jacksonville, FL, USA, April 4-7, 2013.

[6] B. Chen, Thesis: Multiprocessing with the Exokernel Operating System. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 2000.

[7] B. Ford, and R. Cox, Vx32: “Lightweight User-level Sandboxing on the x86”, USENIX Annual Technical Conference, USENIX, Boston, MA, June 2008.

[8] 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.

[9] B. Rawal, R. Karne, and A. L. Wijesinha. “Mini Web Server Clusters for HTTP Request Splitting,” 13th International Confrence on High Performance Computing and Comunication (HPCC), 2011.

[10] C. Coffing, An x86 Protected Mode Virtual Machine Monitor for the MIT Exokernel. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1999.

[11] Coreutils – GNU core utilities for Linux- http://www.gnu.org/software/coreutils/

[12] Coreutils for Windows (GnuWin) on SourceForge - http://sourceforge.net/projects/gnuwin32/files/coreutils/5.3.0/

[13] D. Brackeen, Developing Games in Java. Berkeley, CA: New Riders Games, pp. 63-70, 2003.

[14] D. Salomon, The Computer Graphics Manual, Ithaca, NY: Springer-Verlag Publisher, pp. 200 – 240, 2011.

[15] D. Engler, "The Exokernel Operating System Architecture," Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Ph.D. Thesis, 1998.

[16] D. R. Engler and M.F. Kaashoek, “Exterminate all operating system abstractions,” Fifth Workshop on Hot Topics in Operating Systems, USENIX, p. 78, Orcas Island, WA, May 1995.

[17] D. Simon , C. Cifuentes , D. Cleal , J. Daniels , D. White, "Java on the Bare Metal of Wireless Sensor Devices: The Squawk Java Virtual Machine," Proceedings of the 2nd International Conference on Virtual execution environments, pp. 78-88, 2006.

[18] D.Wentzlaff and A. Agarwal, “Factored operating systems (fos): the case for a scalable operating system for multicores,” ACM SIGOPS Operating Systems Review, Volume 43, Issue 2, pp. 76-85, April 2009.

[19] Das U-Boot Bootloader - http://www.denx.de/wiki/U-Boot/WebHome

[20] DS18B20 1-Wire Digital Thermometer with Programmable Resolution Datasheet. Dallas Semiconductor Corporation, 2005.

[21] Eclipse Project – http://www.eclipse.org

[22] G. Ammons, J. Appayoo, M. Butrico, D. Silva, D. Grove, K. Kawachiva, O. Krieger, B.Rosenburg, E. Hensbergen, R.W.isniewski, “Libra: A Library Operating System for a JVM in aVirtualized Execution Environment,” VEE ’07: Proceedings of the3rd International Conference on Virtual Execution Environments, June 2007.

[23] G. Ford, R. Karne, A. L. Wijesinha, and P. Appiah-Kubi, “The Performance of a Bare Machine Email Server,” 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2009.

[24] G. Ford, R.K. Karne, A.L. Wijesinha, and P. Appiah-Kubi. “The design and implementation of a Bare PC email server,” COMPSAC‟09, 33rd IEEE International Computer and Applications Conference, pp. 480-485, July 2009.

[25] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, "A Peer-to-Peer Bare PC VoIP Application," IEEE Consumer and Communications and Networking Conference (CCNC), pp. 803-807, 2007.

[26] G. H. Khaksari, R. K. Karne and A. L. Wijesinha. A Bare Machine Application Development Methodology, International Journal of Computer Applications (IJCA), Vol. 19, No.1, March 2012, p10-25.

[27] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt and T. Pinckney. "Fast and flexible application-level networking on exokernel system," ACM Transactions on Computer Systems (TOCS), Volume 20, Issue 1, pp. 49-83, February 2002.

[28] GNU ARM Eclipse Plug-in - http://sourceforge.net/projects/gnuarmeclipse/

[29] J. E. Frith. "Fast Circle Algorithm," http://www.tutego.de/aufgaben/j/insel/additives/base/fcircle.txt, Copyright (c) 1996 James E. Frith, Email: jfrith@compumedia.com [Retrieved: March, 2011].

[30] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, R. Brightwell, "Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing," Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), April, 2010.

[31] J. Sanchez and C. Maria, Computer Animation Programming Methods & Techniques, McGraw-Hill, 1995.

[32] L. Edwards. "Embedded System Design on a Shoestring," Boston: Newnes, 2003.

[33] L. He, R. K. Karne, and A. L. Wijesinha, “Design and Performance of a bare PC Web Server,” International Journal of Computer and Applications, vol. 15, pp. 100-112, June 2008.

[34] Mentor Graphics CodeSourcery – http://www.mentor.com/embedded-software/codesourcery

[35] M. Olana, and D. Baker, “Lean Mapping,” Proceedings of 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, pp. 181-188, February 2010.

[36] M. Schoeberl, S. Korsholm, T. Kalibera, and A.P. Ravn, “A Hardware Abstraction Layer in Java,” ACM Transactions on Embedded Computing Systems (TECS), v.10 n.4, pp.1-40, 2011.

[37] N. Kazemi, A. L. Wijesinha, and R. Karne, "Evaluation of IPsec Overhead for VoIP using a Bare PC," 2nd International Conference on Computer Engineering and Technology (TCCET), vol. 2, pp. 586 – 589, April 2010.

[38] P. Appiah-Kubi, A. L. Wijesinha, and R. K. Karne. “The Design and Performance of a Bare PC Webmail Server,” 12th IEEE International Conference on High Performance Computing and Communications (AHPCN), pp. 521-526, 2010.

[39] P. Kovach and J. Richter, Inside Direct3D, Redmond, WA: Microsoft Press, 2000.

[40] P. Zhao and M. Van de Panne, “User interfaces for interactive control of physics-based 3D characters,” Proceeding of the 2005 symposium on Interactive 3D graphics and games, pp. 87 – 94, April 2005.

[41] R. Cáceres, C. Carter, C. Narayanaswami and M. Raghunath, “Reincarnating PCs with Portable SoulPads,” IBM T.J. Watson Research Center, New York.

[42] Karne, R.K., Object-oriented Computer Architectures for New Generation of Applications, Computer Architecture News, December 1995, Vol. 23, No. 5, pp. 8-19.

[43] R. K. Karne, "Application-oriented Object Architecture: A Revolutionary Approach," In 6th International Conference, HPC Asia, Poster presentation, December 2002.

[44] R. K. Karne, K. Venkatasamy and T. Ahmed, “How to run C++ applications on a bare PC,” 6th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel / Distributed Computing (SNPD), pp. 50 – 55, 2005.

[45] R. K. Karne, K.V. Jaganathan, T. Ahmed, and N. Rosa, “DOSC: Dispersed Operating System Computing,” OOPSLA, 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Onward Track, Sandiego, CA, pp. 55-61, October 2005.

[46] S. Eilemann, M. Makhinya and R. Pajarola. “Equalizer: A Scalable parallel rending framework,” IEEE Transactions on Visualization and Computer Graphics, pp. 436 – 452, June 2008.

[47] S3C6410X/OK6410 RISC Microprocessor Rev 1.20 User's Manual. Samsung Electronics Inc., 2009

[48] SEGGER - http://www.segger.com/index.html

[49] T. Venton, M. Miller, R. Kalla, and A. Blanchard, “A Linux-based tool for hardware bring up, Linux development, and manufacturing,” IBM Systems Journal, Vol. 44 (2), pp. 319-330, IBM, NY, 2005.

[50] "The OS Kit Project," School of Computing, University of Utah, Salt Lake City, UT, June 2002, http://www.cs.utah.edu/flux/oskit [Retrieved: May, 2009].

[51] "Tiny OS," Tiny OS Open Technology Alliance, University of California, Berkeley, CA, 2004, http://www.tinyos.net/. Last Accessed Aug. 2012.

[52] U. Okafor, R. K. Karne, A. L. Wijesinha and B. Rawal Transforming SQLITE to Run on a Bare PC, In Proceedings of the 7th International Conference on Software Paradigm Trends, pages 311-314, Rome, Italy, July 2012.

[53] V. S. Pai, P. Druschel, and Zwaenepoel. "IO-Lite: A Unified I/O Buffering and Caching System," ACM Transactions on Computer Systems, Vol.18 (1), pp. 37-66, ACM, February 2000.

[54] W. Ahn, S. Qi, M. Nicolaides and J. Torrellas, "BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support," MICRO'09, pp. 133-144, 2009.

[55] W. Hohl. "ARM Assembly Language. Fundamentals and Techniques," Florida: CRC Press, 2009.

[56] WXCAT43-TG3 4.3'' TFT-LCD with Touch Panel Module Product Specification Document. WANXIN IMAGE Inc., Taiwan 2008.

[57] Y-S Hwang, T-Y Lin, and R-G Chang, "DisIRer: Converting a Retargetable Compiler into a Multiplatform Binary Translator," ACM Transactions on Architecture and Code Optimization (TACO), Volume 7, Issue 4, p18-1:18-33, 2010.

CURRICULUM VITA

NAME: ALEXANDER PETER

PERMANENT ADDRESS: 701 JACKSON RD, SILVER SPRING, MD 20904

PROGRAM OF STUDY: INFORMATION TECHNOLOGY

DEGREE AND DATE TO BE CONFERRED: DOCTOR OF SCIENCE, MAY 2013

Secondary Education: Master of Science in Software Engineering
University of Maryland University College (UMUC), Maryland, U.S.A 2001

Collegiate Institutions	Attended Dates	Degree	Date of Degree
Towson University Maryland, U.S.A.	Aug2007- May 2013	Doctor of Science (Information Technology)	May 2013
UMUC Maryland, U.S.A	Jun1999- May 2001	Master of Science (Software Engineering)	May 2001
Strayer University Virginia, U.S.A	Mar 1997- May 1999	Bachelor of Science (Computer Information System)	Sep 1999

Major: Information Technology

Research Interest:

Application-Oriented Architecture, Bare Machine Computing System-on-Chip, Sensor Networks, Embedded Systems, Human-Computer Interaction, Pervasive Computing, Data mining/data engineering, Signal & image processing, FPGA applications, Organic electronic, Micro-Electro-Mechanical systems (MEMS), Biosensors and Biomedical applications

Professional publications:

[1] A. Peter, R. K. Karne, and A. L. Wijesinha, "A Bare Machine Sensor Application for an ARM Processor," *in Proc. 2013 IEEE Electro/Information Technology Conference*, Rapid City, SD, USA, May 9-11, 2013.
[2] A. Peter, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi, "Transforming a Bare PC Application to Run on an ARM Device," in *Proc. 2013 IEEE Southeastcon Conference*, Jacksonville, FL, USA, April 4-7, 2013.
[3] A. Peter, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi,"The Design and Implementation of Bare PC Graphics," *The Seventh International Multi-Conference on Computing in the Global Information Technology (ICCGI)*, Venice, Italy, June 24-29, 2012.

Professional positions held:

2006-2013	Principal Software Engineer AOL Inc. Dulles, Virginia, USA
20003-2006	Software Engineer / Systems Analyst Nextel Communications Reston, Virginia, USA

Research experience:

2009-2013	Doctoral Research: Bare Machine Computing on ARM Devices. Applications: Bare IDE, Device Drivers, Graphics and Sensor Devices. Demonstrated Graphics/Text Application, Display, Temperature, Buzzer, LED, Touch Screen sensors and WiFi. Department of Computer and Information Science Towson University, Towson, MD Advisor: Ramesh Karne Ph.D. This dissertation proposes a platform for Bare Machine Computing on ARM Devices.
1999-2001	Master's Research: Legacy Software Migration Department of Computer Science University of Maryland (UMUC), College Park, MD Advisor: Hasan Sayani Ph.D. This research proposes using cross-compilers and compiler tool chains to identify legacy code for issues such as the Y2K bug.

References:

Dr. Ramesh K. Karne, Professor and Dissertation Chair
Department of Computer and Information Sciences
Towson University, Towson, MD 21252
rkarne@towson.edu
(410) 704-3955

Dr. Hasan Sayani, Professor and Program Director
Department of Computer Information System
University of Maryland University College, MD 20783
hasan.sayani@umuc.edu
(240) 684-2020