Teaching Operating System Tool
Using
Windows CE

CSC400W

Dembaremba, Jealous Jason

Supervised by
Prof. Ken MacGregor

Department of Computer Science
University Of Cape Town
South Africa

October 10, 2003
Abstract

Computer Science as a field of study has always been a very practical field. It is never enough to have only the theoretical side without the complementing practical side of it. This is particularly true in the field of learning operating systems. However, the complexity of operating systems creates a barrier to having comprehensive practical experiments with them during the course period. To address this issue, this project aims to create a teaching tool using Microsoft Embedded operating system, Windows CE.

Educational tools have been looked at in the past and two examples that are widely known are Minix, Xinu. Another example is the PortOS tool developed at Cornell University. However, issues such as the need for dedicated machines and lack of emulators for windows platforms makes them unsuitable for environments that heavily rely on NT. Windows CE is chosen mainly for its small footprint and availability of an emulator. It also comes with a Platform Builder in which a complete operating system can be build. The architecture of Windows CE enhances flexibility as it is component based and thus components can be selected to suit specific needs.

There are two ways of building a Windows CE image; command line base or using the platform builder. The design settled for the platform builder approach as it is more user friendly and comes with more supporting tools in the building process. The implementation involved creating a simple wizard that guides the user in creating a platform specific to the parts of the OS internals.

The setting up of a custom environment that interfaces with the kernel internals worked exceptionally well. The component based architecture made customizing Windows CE very feasible and easy. Going through the stage of changing the algorithm in the scheduler had a lot of barriers. The mechanism used in Windows CE does not clearly distinguish the algorithm used from the rest of the other OS aspects. As a result one has to read the whole source code just to understand how a simple algorithm is implemented. For this reason working with the Windows CE source code was difficult and requires some support from Microsoft.

The project findings as a whole where that Windows CE can be used for teaching some aspects of an operating system such as understanding of how components and modules are built and put together to create a fully functional OS. The source code availability, though it is only a portion, supplements the understanding of how the OS is build. However, lower levels of implementation proved to be difficult. There is no clear distinction of the theory and mechanism in the source code. The source code for a particular implementation (say scheduling algorithm) tends to be spread over multiple source files as a result the students will end up reading almost the whole OS source code which complicates things.

However as future work, recommendations are that more effort is given to looking into
understanding the source files as in this project most time was spend in the first phase
as there was very little supporting documentation. Some academic licence issues need
to be addressed so that more support at very low levels can be made available otherwise
it will be very difficult to understand the underlying functionalities of the OS timeously.
Contents

List of Figures .................................................. iii
List of Tables ................................................... iv

1 INTRODUCTION .............................................. 1
  1.1 Project Overview ........................................... 1
    1.1.1 Motivation ........................................... 1
  1.2 Project Objectives and Approach .............................. 1
  1.3 Report Structure ........................................... 2

2 BACKGROUND ............................................... 3
  2.1 Teaching Operating System ................................ 3
  2.2 Related Work .............................................. 4
  2.3 Microsoft Windows CE ................................... 4
    2.3.1 What is Windows CE .................................. 4
    2.3.2 Windows CE Architecture .............................. 4
    2.3.3 Motivation for using Windows CE ........................ 5

3 DESIGN ................................................... 7
  3.1 Overview ................................................. 7
  3.2 How Windows CE Platform Builder Works .................... 8
    3.2.1 Platform Window Building ............................. 9
    3.2.2 Compiling and building an OS image .................. 13

4 IMPLEMENTATION ......................................... 14
  4.1 Implementation ........................................... 14
    4.1.1 Catalog Feature File ................................ 14
    4.1.2 IAKERNELBASE Folder ................................. 15
    4.1.3 Platform Builder Wizard ............................... 17
    4.1.4 Source Code Identification and Mapping ............... 17
    4.1.5 Integration of Developed Components .................. 18
  4.2 Testing ................................................ 22
    4.2.1 Phase 1: Environment Setting ........................ 22
    4.2.2 Phase 2: Using Windows CE as a Teaching Tool ......... 22
    4.2.3 Analysis and Evaluation ............................... 24
5 FUTURE WORK
5.1 Overview ................................................. 27
5.2 Challenges ............................................... 27

6 CONCLUSIONS ........................................... 29
A How To Use the Developed System .................. 31
B Cetest Result Debugging Output ................. 32
C Command Line Window Building .................. 34
List of Figures

3.1 Overview of the OS image building process .......................... 9
3.2 Screen Shots of Platform Wizard showing design walkthrough .... 12
4.1 A screenshot of the Catalog Feature File ................................. 15
4.2 IAKERNELBASE directory structure .................................. 16
4.3 An excerpt of the iakernelbase.bat file ................................. 17
4.4 An excerpt of the Private_Coreos.xml file showing component inclusion . 18
4.5 Private Directory Tree Structure ........................................ 19
4.6 Sample structure of a sources (left) and dirs (right) files .......... 20
4.7 A directory structure of the components implemented. ............ 20
4.8 Shows the relationship of the developed components to produce a platform. 21
4.9 Final design implementation of the wizard ............................ 24
4.10 Screenshot showing source code for a scheduler .................... 25
4.11 A possible case of thrashing as more process are spawned ......... 26
B.1 Osbech results for the Thread Yield Time performance ............ 33
List of Tables

3.1 Component and Module Selection ......................... 10
Acknowledgements

- I would like to thank James.Y.Wison and Aspi Havewala for assisting me in the initial stages of understanding how the platform builder works in general. Their book has also been very instrumental given that at some point I thought my project was a dead end.

- I would also like to thank my supervisor, Prof.K.MacGregor for the support in providing insightful ideas in trying to understand the Windows CE source code.
Chapter 1

INTRODUCTION

1.1 Project Overview

The project looks at the possibilities of making Microsoft Windows Embedded operating system as a desktop operating system that can be used for educational purposes in the Computer Science department of the University of Cape Town (UCT). The other section of the project looks at the feasibility of using Windows Embedded in the laboratories of the UCT. However this report concentrates more on the first aim and the rest of the report discusses the various areas revolving around the teaching operating system tool subject.

1.1.1 Motivation

The third year operating system (OS) course at the UCT takes a high level approach that looks at the theoretical approach to the basic concepts. Tutorials are done at an application layer and do not really look at the internals or the way the different principles and components of the operating system are put together in practice. There is need to have a module at honours level (4th year) that extends on the aspects not covered in the third year module. In particular, it aims to look at the core internals of an operating system.

The challenge is that, given the complexity of an operating system, there is need to come up with a practical approach that can fit in the semester period (20-25 lectures) that the department normally dedicate to each module. A cost effective approach need to be adopted which does not require additional resources or necessarily the need for a special laboratories only dedicated to the module.

1.2 Project Objectives and Approach

Windows CE is a commercial operating system that was not designed for educational purposes. Therefore the first goal of the project is to adapt Windows CE to suit an educational environment with more emphasis on very basic OS internals. These could be at least a simple shell, a kernel and basic Input-Output (IO) functionalities. The
second objective is to come up with practical demonstrations of the theory discussed in lectures such as synchronization - scheduling of threads and processes.

The final goal is that once Windows CE can be configured to allow for proper low level editing and compiling of the source code, tutorials can be developed at this layer for students to work with.

The design and implementation is done in two phases. Firstly, an easy to configure environment is created by designing and implementing a usable and flexible interface to the OS internals. The second phase will be to look at the possible tasks that can be done to the core internals of the OS to check for the feasibility of developing meaningful tutorials.

1.3 Report Structure

The rest of the report is structured as follows. The next section outlines the background of operating systems and the related work done in trying to create a teaching operating system tool. Section 3 looks at the guiding principles behind the system design and examine some alternatives. Section 4 describes implementation of the design and discusses the results. The report is wrapped up in sections 5 and 6. Some references and addition information can be found in the Appendices.
Chapter 2

BACKGROUND

2.1 Teaching Operating System

Teaching operating system internals has been a major topic of discussion for quite some time. A survey of operating system professors was conducted by Addison-Wesley and the general consensus was that they want to get students to be involved in programming projects [8]. A lot of work has been done to come up with a teaching tool that would allow students to have a strong feel of the core mechanisms of an operating system.

Gary Nutt clearly puts the whole picture of the problems in the preface of his book [8] which reads, "There are only a few widely used commercial operating systems. While studying these systems is valuable, there are practical barriers to experimenting with any of them in the classroom. First, commercial operating systems are very complex since they must offer full support to commercial applications. It is impractical to experiment with such complex software because it is sometimes difficult to see how specific issues are addressed within the software. Small changes to the code may have unpredictable effects on the behaviour of the overall OS. Second, the OS software sometimes has distinct proprietary value to the company that implemented it”. A further demand was on the need for operating system books that helps instructors deal with courses at a similar level, a separate exercise but very important.

Several other papers and articles shared the same sentiments on the need for a teaching operating system tool [11]. John O’Gorman talks of most operating system modules covering only the basic concepts at a very high level in his article [9]. He proposes exposing students to the internal operating system; the actual implementation of mechanisms and algorithms used to in an OS. The approach is, rather than saying a process control block contains such and such information and draw a block diagram, students can be presented with how it is actually declared in C. This approach ties in well with students’ developing experience of writing programs and makes it more immediate and real.
2.2 Related Work

Some work has been done to address the problems faced in the teaching of operating system internals. The two most widely known educational operating systems that are in use in some institutions are Minix and Xinu. Minix [12] famous for giving birth to Linux while Xinu is a Unix-like operating system tool of approximately 4000 lines of code. They both support most modern OS functionality, including message passing, TCP/IP, and an independent shell. However, the problem is that as stand-alone operating systems, rewriting and debugging the kernel often requires a dedicated machine. This would mean creating, possibly, a new laboratory facility just for one operating system module which is costly.

Another interesting educational tool developed at Cornell University is PortOS [3]. PortOS focuses on distributed computing on mobile computers and supports a wide number of platforms. It is developed as a portable layer that resides on top of an existing commercial operating system and provides a machine model to the systems programmer that closely mimics the underlying hardware. It however has a similar problem of requiring a dedicated machine for the debugging and recompiling of the OS.

The next section describes a possibly viable alternative that can be used with a few modifications in some areas, Microsoft Windows CE.

2.3 Microsoft Windows CE

2.3.1 What is Windows CE

Windows CE is a small modular operating system primarily for embedded systems and other handheld devices like PDAs. Other devices would include industrial controllers, phones, music players, televisions, game consoles, camcorders, organizers, and compact laptops [13].

2.3.2 Windows CE Architecture

Windows CE is a very component and module based operating system. This approach ensures a reliable, secure operating system with a very small footprint. It also allows system developers to fine tune the building of operating system images to the specific needs. Windows CE is modular at more than one level. The Graphics Windowing and Events Subsystem (GWES) is one example of a Windows CE module with most components. It is made up of a number of subcomponents, including menu, clipboard, caret, and user interface controls, each individually selectable at build time. The diagram below looks at an overview of the structuring of the OS [4]. Windows CE supports a variable of hardware devices and thus its emphasis is on building different types of OS images that support different devices. It also builds components and additional user applications compiled from source and may require the use of assemblers, compilers, and linkers to produce the final operating system image. Building of the system relies
heavily on environment variables to control the inclusion or exclusion of components from the final operating system image.

### 2.3.3 Motivation for using Windows CE

The availability of a Microsoft’s Shared Source agreement for Windows CE makes it a good start to look at using it as a tool to teach operating system internals. There are several other reasons for choosing Windows CE:

1. Windows CE is a simple OS with a very small footprint.

2. It comes with an emulator which eliminates the problem of dedicated machines.

3. In terms of overall scalability, it is possible to produce a build with nothing but Windows CEs preemptive multi tasked kernel such as the tiny kernel example that is shipped with it.

4. It comes with a Platform Builder IDE. This provides the build process with a graphical user interface including debugging tools. Although a full kernel image rebuild can take a long time, it is rebuilding an entire operating system in an IDE all integrated in one place.

5. The architecture is significantly flexible and it can allow the students/instructors to fine tune it to suit particular needs through selection of components and modules.

The general trend is moving towards distributed computing and embedded systems. There is a fundamental and quantifiable shift in the global computing landscape from desktop and centralized mainframe operating systems towards mobile systems. While this is not directly the objective of this project, basic understanding of Windows CE
would benefit the students in the one aspect of basic principles of an operating system. An integration of this understanding with the modern technology of designing operating systems for mobile devices would be the other aspect that would probably happen seamlessly given the same working environment. For these reasons, this project aim to achieve the project objectives using Windows CE. The next chapter looks at the approach and design adopted to achieve these objectives.
Chapter 3

DESIGN

3.1 Overview

The effective design steps and goals aim to address the following:

- creation of a basic platform (an environment in the Platform Builder where and OS can be configured and built),
- On top of the existing configuration tools, allow for configuration of the platform that is related to the internals of the OS (source code editing and project/module creation),
- build an operating system (OS) image,
- transfer the OS image to a target device (e.g. the Emulator) and
- debug the platform analyzing the effects of the OS configurations.

This section looks at how this can be achieved in an efficient manner.

As mentioned earlier in section 2.3.2, Windows CE allows one to build a whole new operating system image with options to include or exclude components. The process, theoretically sounds simple enough - at least (rightfully so) from an end-user’s point of view, however vast work is done in the background to ensure a successful completion of the steps mention above. It is at this level that the first design phase deal with - setting up of an environment that, in addition to simple inclusion and exclusion of high level applications, also allow for easy access of the internal core OS related source code. It also looks at the components and modules to include or exclude and integrating these changes correctly for a chosen type of operating system image. The environment should present the user with a logical organization of the required information (mainly the source code files) with a high level of relevancy to the chosen components or modules.

In Window CE, there are two ways in which an OS image can be built. A fine balance between ease of use and the degree of flexibility and functionality associated with each approach is also taken into consideration in the design.
Before going any further a few terms and concepts need to be put into perspective to better understand the sections to follow:

**Windows CE Platform Builder** is an Integrated Development Environment (IDE) for building customized embedded platforms based on the Microsoft Windows CE .NET operating system (OS).

**Platform Window OR simply a Platform** is an environment in a platform builder that a user can configure and create projects before building an OS image.

**Build Environment** At times interchangeably and loosely used as simply *environment* to describe a state of the development workspace, including the directory structure and environment variables, for a particular platform created.

**Component** The smallest individually selectable piece of functionality available to the tools, typically implemented as a static library. Components can be combined into modules.

**Module** is an executable piece of the operating system (e.g. .exe or .dll). Modules are created based on the features selected for the platform.

**Feature** is any logical grouping of OS components. Its functionality is a logical grouping of OS components that a user select. A Feature based OS provides greater configuration flexibility by allowing choices of only those pieces of the OS that provide the functionality that is needed at that point in time.

The last three terms can be summarized by looking at a *command processor*. It is a *feature* that could consist of a number of APIs spread across several OS *components* forming a CMD module within the OS. In addition, for it to be included in the OS image a specific *environment* variable called $SYSGEN_{CMD}$ need to be set. This essentially summarizes phase 1 of this project which is the process of environment setting and selection of components, modules and features for the intended OS image that satisfies the goals of this project.

### 3.2 How Windows CE Platform Builder Works

The process of building a Windows CE OS has three main stages:

- Platform Window Building
- Compiling and building an OS image
- booting or running the OS image on an Emulator

Figure 3.1 below illustrates the high level stages involved in the whole process with an additional stage of source code editing, representing the stage where a user performs the assigned work.
3.2.1 Platform Window Building

The platform window building (Step 1) is the most vital component of the whole process. The design will aim to:

- Strip components and ensure environment integrity
- Easy patching of the platform builder to incorporate our desired changes.
- Map the components to their respective source code and allow the users to edit using the Platform Builder IDE.

Component Stripping and Environment Setting

Of the three stages, the environment setup stage is the most important stage as it determines the integrity and success of building an OS image. The primary goal is to design an OS that is less complicated and an even smaller footprint than it is currently by removing some of the components and features that are not essential to the learning of the OS internals. This aims to reduce confusion to the students and help them concentrate more on the OS fundamentals. Building a full OS image was done and proved to take quite a long time - only to find out that a certain component has not been included and had to repeat the process again. Removing of some of the components is can reduce this building time of the OS image.

Table 3.1 lists some of the categories that features can be fit into and whether they
<table>
<thead>
<tr>
<th>Feature Category</th>
<th>Examples</th>
<th>Retained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application And Services Development</td>
<td>SOAP, Message Queuing</td>
<td>none</td>
</tr>
<tr>
<td>End User Applications</td>
<td>Active Sync, Games, Remote Desktop connection</td>
<td>File Viewers - Wordpad, Microsoft Word Viewer</td>
</tr>
<tr>
<td>Communicating Services and Networking</td>
<td>Servers, WAN, PAN related networking components</td>
<td>none at this stage (of interest could be LAN)</td>
</tr>
<tr>
<td>Core OS Services</td>
<td>kernel features, power management serial and parallel ports support</td>
<td>almost all</td>
</tr>
<tr>
<td>File System and Data Store</td>
<td>File and database replication, Registry Storage</td>
<td>almost all</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>fonts, multilingual support, internet client services, multimedia technologies</td>
<td>a selection of standard fonts</td>
</tr>
</tbody>
</table>

Table 3.1: Component and Module Selection

are excluded in the design or not. The listed items are not exhaustive but give the range and kind of features that are considered relevant or irrelevant. However these features can easily be added on at a later stage as the need arises.

To further enhance the project objectives, modifications need to be designed to ensure a suitable environment that provides a leeway to the internals of the core OS. Subsequent changes and building of the code need to be easy to increase productivity for students given the short semester time. The design aims to provide an easy way of selecting parts of the OS that a student wish to work on. It provides access to only the relevant parts of the source code for that chosen part.

Command-Line Based

Command line based involves creating an environment and building the OS image from the command prompt. Some feasibility tests were contacted (see Appendix for detailed steps taken) mixed results were obtained. There are three main stages required to build an image from command.

1. Initializing the variables and running Wince.bat. Firstly a few parameters have to be set such as where the windows CE root is for it to place the project to be built. A batch file, Wince.bat, is then run using the command, ”wince.bat x86 kernel EMULATOR” where kernel is the name of the project and target device being an Emulator for an x86 device. The wince.bat configures the environment variables (if they have not been set already by the user) to determine the components and modules required,

2. Generating or building the system using the SYSGEN.bat file. This file is called in directly by using the BldDemo tool with Syntax: ”BldDemo [clean — noclean] [rel — norel] [Cebuild.bat parameters]” where rel/norel means building a release (retail) version or non-release(debug) version of the OS.
3. the final stage is to perform a make image call using the \textit{makeimg} tool. This tool creates the final nk.bin image that can be transferred to another partition or a target device such as the emulator.

A utility was developed to do the first two steps in an automatic fashion - using a batch file called wincesetup.bat. For more information on command line based approach can be found in the Appendix with all the findings of each stage discussed.

\textbf{Platform Builder Based}

As for the platform builder, the process is all GUI driven. The IDE wraps up the underlying command line build. Selecting components from the catalog and through the wizard sets the underlying environment variable, which is then used by the build system to include the appropriate modules and components. A catalog provides information that is not readily available from the command line. It provides facilities to set properties for the features such as approximate size of the component, vendor, additional OS dependencies required among other useful utilities.

Having looked at these two approaches, the Platform Builder approach was adopted mainly because of the ease with which one can build an OS image. Moreover, it provides more tools to work with and integrates the source editing, other environment settings and building of the final image together significantly well. With the command-line based approach this is not possible. Environment settings are done in command prompt and one has to use a different editor to edit the code before coming back to the command again to build the OS.

After a few tests with the command line approach, it was realized that having compiled the image, booting it using the emulator as the target device is involving. Of the few quick tests that were done, none were successful. The reason being, in the platform builder, a few more device settings are done specific to the emulator before booting the image of which the equivalent procedures are not well known in the case of command-based. Overall, command line turned out to be a bit tedious although one can argue that if well mastered it can be more flexible that the platform builder simple because every variable can be defined completely from scratch whereas in the platform builder, the wizard does some of the work in the background that the user cannot change. Some developers actually prefer to build their operating system from the command line, which gives fine control over which parts of the operating system are to be built and can improve build times by building the components that have changed and then making the final operating system image [5].

Figure 3.2 looks at the current design implementation used by the platform builder to create a platform and the presentation of the source code in the workspace before building the final OS image. The design will follow a similar approach and in each figure the different relevant changes are laid out.

The process starts at stage 1 followed by stage 2 where options are selected. These
are envisaged as the sections of the OS that need to be worked with. Further information is presumed to be collected after this stage until the last stage is reached, stage 3. The expected final environment is shown by the final stage snapshot showing a hierarchy of files of folders and an editing window. Similarly, this could be files related to the options selected during the wizard walkthrough phase.

![Figure 3.2: Screen Shots of Platform Wizard showing design walkthrough](image)

1. Selecting target device.
2. Selecting type of platform to build.

Relevant source code presentation.

Figure 3.2: Screen Shots of Platform Wizard showing design walkthrough
3.2.2 Compiling and building an OS image

When all the customizing of the platform - source code editing etc. - is done, the next step will be to compile the changes and build the OS Image. At this stage, there are not many design issues as most of the work is handled by the platform builder. The only important setting to do is to perform a platform setting that builds a debug type of OS. There are two type of OS images that can be built namely debug and retail versions. The former is clearly for debugging purposes whereas the latter is meant to be a final image essentially ready for shipping. The debug version required version to build and more importantly to allow for kernel monitoring of the changes which would have been made.

The final stage would be to boot the image in the Emulator.

The next chapter looks at how this design was implemented in detail including the testing and evaluation of the outcomes.
Chapter 4

IMPLEMENTATION

4.1 Implementation

Various setup components were developed to implement the necessary changes that could allow for building and OS image. This phase involved various technologies such as XML, batch files, and Platform builder’s CEC editing tool for creating Catalog Feature files. Catalog Feature files (with the extension .cec) are files for defining components and modules, describing the metadata about these features and their relationships.

Phase 2 involved implementing some simple changes to the scheduler to demonstrate the kind of approach that can be taken to explore the internals of the kernel. An associated kernel test module is also implemented that is mostly at the application level of dealing with the kernel. It is primarily created to serve two purposes:

1. Testing the changes made to the kernel internals and the image’s behavior in general. It simply does some thread and process spawning. Most importantly it spawns multiple processes of an OS benchmarking tool - osbench.exe - with different parameters for testing different issues.

2. Demonstrating how a component can be developed and included as one of the many components and modules that make an OS.

4.1.1 Catalog Feature File

Before a Catalog Feature File could be defined, some environment variable analysis was done using the already built in environments that come with the platform builder. The main objective was to understand, in particular, the dependencies of the components and general understanding of how the platform builder functions. The core catalog feature file that is responsible for defining the core OS was called iabase.cec file. In this file, required components and modules such as the coredll (represents the core.dll components) are defined as well as the optional components and associated variable settings. The iabase.cec file defines two types of Core OS settings: one that defines a Headless (non graphical with only a shell) and one that is Display based (graphical with a desktop simulation). Both cases do not associate any stage with internals of the
The implementation, thus, looks as adding a third core OS definition that sits in between the headless and display based. This means it has very minimal components and modules as the headless but supports display features. This core OS definition is called **Headless_Display** based definition. The implementation used the **Display based** definition as point of reference, stripping, modifying and/or adding features following the design considerations stated in the design section. The reason for using the display based definition is that, it is relatively easy to remove components than adding to the Headless based definition. The Display based definition contains most, if not all, of the settings defined in the headless definition.

The definitions of the components and associated variables for the **Headless_Display** core OS definition need to map to some description that can handle these settings when creating an environment. This is basically a directory structure with relevant batch files that determines how the environment is set and consequently builds. Figure 4.1 shows the defined features and components for the **Headless_Display** and the description set to IAKERNELBASE. The next section explains the IAKERNELBASE directory structure that was defined to work with the **Headless_Display** definition.

### 4.1.2 IAKERNELBASE Folder

In the Public directory of the Windows CE sits the IAKERNELBASE directory that describes the environment settings and the way a system is generated. Two most im-
important files are the `iakernelbase.bat` and `cesysgen.bat` files.

**The `iakernelbase.bat`**

This file defines some variables specific to the `Headless Display` core OS definition and the platform under creation.

Figure 4.3 shows an excerpt of the `iakernelbase.bat` file. The variable `SYSGEN.CETEST` ensures that the cetest (see section 4.2.2) component is included in the OS image. The "if" statement defines the modules that can be considered during the building of the OS image. A special variable called DEPTREES holds the modules to compile and build such as dcom, viewers as listed in the extract. However, these modules are only built if this variable is enabled in the platform builder just before building an OS image.

**The `cesysgen.bat`**

The `cesysgen.bat` (batch) file is the system generator that is run during the phase of generating an OS image or simply the headers that makes the final OS image. This phase is called the `SYSGEN` phase that is then followed by the `MAKEIMG` (make image) phase. The `SYSGEN` phase involves the compiling and putting together of the components based on the variable settings. The `cesysgen.bat` file plays this role and checks for integrity of the settings. If the checking succeeds it calls the `sysgen.bat` file (a built in utility) that in turn calls the `makefiles` to compile the different modules accordingly.

This file basically was a modification of a cesysgen.bat file from the Display based description folder that is designed for building graphical based OS image. The changes made were mostly to do with removing or disabling calls that include some high level application components and modules such as per the design table 3.1. Other changes were meant for consistency with definitions made in the IAKERNELBASE settings in the iabase.cec file. This batch file is one of the most complicated files. The complication is mainly due to the need to accurately match the variable settings and component calls as they would have been set by the wizard and catalog feature file (the iabase.cec file).
4.1.3 Platform Builder Wizard

A simple wizard approach used traditionally by the platform builder was implemented (see figure 3.2). It was somehow involving as well, modifying the flow and type of content included in the wizard. However the whole process boiled down to XML and DTD files describing the metadata for the type of content and flow of the wizard. Besides defining the flow and content, this metadata links with the Catalog Feature file [section 4.1.1] registering the selected options and setting the initial or the required variables for a particular type of environment. It achieves this by using a Globally Unique IDentifier (GUID - A unique 128-bit value used to identify objects). The extract of the XML file (Private_Coreos.xml) in figure 4.4 shows calls to include components that have been defined in the iabase.cec file.

4.1.4 Source Code Identification and Mapping

Having set the different components described above, the next stage is to map the relevant kernel source files to the respective types of modules or components that they belong to. Figure 4.5 shows the some of the directories in PRIVATE directory of the Windows CE that contains the core source files. The implementation looks at grouping source files, taking into consideration dependency issues as well, into the following categories:

- COREDLL module - the basic operating system (OS) module that provides core functionality to other modules. It is a required in all platforms although not all of its components are required.
Figure 4.4: An excerpt of the Private_Coreos.xml file showing component inclusion

- Core Kernel
- Device Management
- File system and storage
- shell

Windows CE uses a mechanism of files called "dirs" and "sources" to build an operating system [7]. The dirs act as navigators around the PRIVATE directory until a sources file is reached that determines the source files to compile and other compilation information. The same technique was used to map relevant files to components and modules defined in the iabase.ccc file. These were carefully defined planted in the respective directories. A simple excerpt of a sources and dirs file is shown in figure 4.6. In the appendix a more detailed explanation of how the sources and dirs files work is provided.

### 4.1.5 Integration of Developed Components

The resultant implementation that is put together consists of utilities for running the initial setup and some directories containing files for mapping source code and related definitions. This integration forms a whole new subsystem that needs to be integrated with the Platform Builder using the setup utilities provided. Section A explains how this is done given a new Platform Builder Installation. Figure 4.7 shows the directory structure containing the components implemented whilst figure 4.8 shows the relationships of these components in producing a platform.
Figure 4.5: Private Directory Tree Structure
Copyright (c) Microsoft Corporation. All rights reserved.

This source code is licensed under Microsoft Shared Source License
Version 1.0 for Windows CE.
For a copy of the license visit
http://go.microsoft.com/fwlink/?LinkId=1223.

EXETYPE=StartUp
CDEFINES=-(CDEFINES) -Zi -I.\x56
LDEFINES=-subsystem:native -align:4096
/DEBUG /DEBUGTYPE:BOOTH,FIXUP/STACK:64000,64000

EXEBASE=0x8C040000
x86_SOURCES=\
..\x86\init.asm\
..\x86\sysinit.c\
..\x86\mdx86.c\
..\x86\fault.c\
..\x86\strings.c\
..\x86\exsup.c\

SOURCES

DIRS=
\kds\w\kernel\tools\schedioq\celog\no compr\kite\mapfile\nomapfile

OPTIONAL_DIRS=
buildexe

Figure 4.6: Sample structure of a sources (left) and dirs (right) files

Figure 4.7: A directory structure of the components implemented.
Figure 4.8: Shows the relationship of the developed components to produce a platform.

Since Phase 2 of the implementation is can also be regarded as essentially testing phase 1 as well, it is describe in more detail in the testing section 4.2.
4.2 Testing

"Testing? What's that? If it compiles, it is good, if it boots up it is perfect" - Linus Torvalds

4.2.1 Phase 1: Environment Setting

When some components are removed there is need to do a dependency check to ensure the integrity of the OS. The only way to test these configurations was to successfully build an operating system.

4.2.2 Phase 2: Using Windows CE as a Teaching Tool

Having verified the integrity of the new environment settings, the second phase comes into the picture whereby changes are made to the actual core kernel source code. There are two aspects to teaching an operating system; the policy of algorithms and the mechanism itself. There is a need to clearly isolate the algorithm from the mechanism given a piece of source code for particular module.

The scheduler was the chosen component of the OS to try and identify mechanisms that implements the policies. A very simple approach to the changes was done. Windows CE’s scheduler implements a priority-based, fixed-time-slice, algorithm to schedule the execution of threads [6]. The idea was then to implement a simple scheduling policy that uses a variable-time-slice approach. The algorithm simply states that: if a thread is preempted before it has finished execution its default time quantum assigned to it for the next round is increased by $x$ time. If a thread yields for an IO with still some time quantum to spare, its default quantum is reduced by $x$ time the next time it needs processor time. The thread priority also needs to be adjusted accordingly.

Three main expected outcomes were to

- Successfully compile the changes. A successful compilation also confirms an acceptable level of correctness in the implementation of phase 1.

- Secondly, a successful build of the image and booting on the target device, the Emulator, together with the expected varying of the time quantum checked through debugging statements inserted in the essential part of the code.

- Finally, analyze the outcomes of running the cetest module, looking at the changes in behaviour of the processes, particulary looking at how the threads are being yielded and the re-assignment time quantum. Furthermore, an analysis of the results from the benchmarking tool explained below is also taken into consideration.

Going through the stage of changing the algorithm in the scheduler had a lot of barriers. The mechanism used in Windows CE does not clearly distinguish the algorithm used from the rest of the other OS aspects. As a result one has to read the whole source code just to understand how a simple algorithm is implemented. For this reason working with the Windows CE source code was difficult and requires some support from Microsoft.
Cetest Component

A small application was developed to test the OS images that were built. The component
does not do anything particularly spectacular other that spawning processes such as the
Osbench (described below) and and a few threads. It was the base for the standard
test that could anticipate the expected behavior and sequence in which the process
and threads were being spawned. The source files can be found in the cepatch/APPS
directory (see 4.7). The implementation of this component is an abstraction of the tktest
shipped with the platform builder.

Osbench Tool

A benchmarking tool - Osbench program - was used to supplement the testing of the
changes performed in the scheduler. This tool enables us to collect timing samples
to measure the performance of the kernel by conducting scheduler performance-timing
tests. A scheduler performance-timing test measures how much time is required for a
basic kernel operation, such as a synchronization action. An example of a synchroniza-
tion action is scheduling a thread that is waiting on an event that has just been set
by another thread. Wherever appropriate, the test runs two sets of metrics: thread-to-
thread within a process and thread-to-thread across processes.

The osbench tool can be used to determine how long it takes to perform the follow-
ing tasks:

1. Acquire or release a critical section
2. Wait or signal an event
3. Create a semaphore or mutex
4. Yield a thread
5. System API call

The most useful benchmark for the test is the fourth option - time to yield a thread.
Although this tool was used in these image tests, the results are not reliable since the
tests where done using the Emulator. The Emulator makes direct use of the CPU and
system memory of the development workstation. Therefore, booting an OS image using
this target device generally degrades the performance and thus the times produced by
the osbench. These results will depend on the performance of the workstation at the
point of running the osbench. Generally from a performance perspective, application
code running in the Emulator, runs at 80 percent of the speed than it runs on hardware
[6].

- Firstly wanted to see how the OS will respond compile time wise to a few changes
  in the core kernel. Secondly, obviously have to boot the emulator as before the
  changes.
• Debugging statements inserted in the vital functions such as YieldThread, CreateThread, NextThread and so forth. In addition a Debug version of the OS is build to allow for easy debugging as well.

4.2.3 Analysis and Evaluation

Phase 1: Build Environment

The setting up of a custom environment to suit an environment that interfaces with the kernel internals worked exceptionally well. Quite a few dependency issues where encountered with some components, mostly in Graphical Windowing and Event Sub-system (GWES). As a result they ended up being retained in the new design. Figure shows the final design implementation from the wizard stage to the source code editing stage.

Phase 2: OS Teaching Tool Analysis

Coming up with a good task that deals with the internals of the OS proved to be very difficult. The main reason was that, at this level of development, there is need for support on the organization of the source code. With the scheduler that was looked at,
some of these issues could be addressed for instance, figuring out and understanding the places were the actual calls to thread/process creation etc, is done.

A fair picture of how the algorithm as whole was functioning could be deduced. However, one typical case that could not be deduced implementation wise was keeping track of the thread priorities. The time quantum could be made variable but to complete the algorithm, the priority of the threads had to be altered accordingly to prevent starvation. Furthermore, the way interrupts are passed into the scheduler could not be well understood. The end result is that students will end up having to go through a lot more different source files that expected which makes things more complicated. It was at this point that support, probably from Microsoft, was required.

Although a full implementation could not be achieved some experience was gained in the basic understanding of how the scheduler was implemented and some interesting debugging output was obtained using the minimal changes. Figure 4.11 shows some of the debugging output where a third process of an osbench is spawned and some kind of thrashing starts to occur. This could be due to the fact that after changing the time quantum, the priority was not modified accordingly as well. Useful trend could not be deduced from these debugging outputs and the osbench values (see Appendix B).
Figure 4.11: A possible case of thrashing as more processes are spawned
Chapter 5

FUTURE WORK

5.1 Overview

While the first phase was very successful, the big question really was in the second phase of the design - what is to be changed? The second phase did not get adequate time to explore although in that little time it did not seem very easy or feasible to work with Windows CE at a very low level. However it still stands as a good area to look into as feature work. Reasons mainly being more time will be given to just understanding the source code that configuring and setting up the whole environment that puts the OS together - problem already solved by the first phase of this project. The other reason is that, some pointers are already available as to where to start with associated anticipated problems. The one important area is of ironing out of some of the challenges mentioned in the problems section below. Clearly the platform builder comes as a big, easy to use interface for building an Operating System and that strong supports the idea of using it as a teaching tool.

5.2 Challenges

- Very little support on how the OS actually work. Some sections require the experts - person who actually wrote the portion of code to understand fully - of which under the licence it’s not really possible. One situation was during my few exchanges of mail, Havewala replied that he (rightfully so) could not answer my question simply because of the Microsoft Licence issue.

- difficult to work with some files missing - simply because Microsoft only ship a portion of the source code. Perhaps one solution would be to at least inform the institution (obviously under some licence) what one can do with the source code supplies and point out the areas that just won’t work if changed simply because they depend on some files that are supplied. That way, a lot of time is saved and probably will put the whole pieces together for the instructor looking at the code.

"Only a portion of the shell is available in the private code tree. MS has licensed part of it from other vendors and that code isn’t available. Therefore, you can’t modify the
CE shell.” - [10]

This is the typical problem that was mentioned earlier in section 2.1 by Gary Nutt. Therefore until some agreement of some sort is made between the software makers and the institutions, it is pretty much almost impossible to work with the core mechanisms of the Operating System in a meaningful way. In that case open source (linux, FreeBSD) can always be put on the table.
Chapter 6

CONCLUSIONS

The setting up of a custom environment that interfaces with the kernel internals worked exceptionally well. The component based architecture made customizing Windows CE very feasible and easy. Example modules or components that are of interest to the teaching of operating system such as Scheduling and Memory Management could be mapped easily with the relevant source code files presented to the user using the modified platform builder wizard and environment settings.

Windows CE is a versatile operating system. The combination of Microsoft’s Shared Source initiative and the Platform Builder IDE, allows for the possibility of using Windows CE to create a teaching tool for teaching operating systems internals. However the shared source is incomplete and complex making it difficult to achieve the all desired requirements.

However, there is also another form of learning that can be done using Windows CE. It clearly helps in understanding how different modules and components are put together. One interesting point to note is that the mechanism of using dirs and sources files is heavily used in Windows 2000 code structure.

Overall, the project called for a lot of individual research using the manuals [7] and [6] almost 99% of the time simply because the level and type of content was too sensitive to get support from places like the internet (such as the Microsoft’s ”public” Windows CE newsgroups [2]) or anywhere else. Nevertheless, one of the vital lessons learnt, is the understanding of how OS components are put together and seeing an interesting mechanism used to build a fully functional Operating System in an IDE. At the end of it all, a fairly concrete understanding of Microsoft Platform Builder 4.1 was gained.
Bibliography


Appendix A

How To Use the Developed System

A new installation of the Platform Builder needs to be updated with the components developed. The utilities in the CEPATCH directory described in previous sub sections should be run to achieve this. The simple steps to follow are:

1. Run command and make sure you are in the UTILS directory.

2. run the command "setup /path to the WINCE410 folder". Simply type setup for more detail. A confirmation is displayed for a successful setup otherwise an error is displayed.

3. Follow the one more instruction that follows.
Appendix B

Cetest Result Debugging Output

These screen shots are just shown here to show the kind of output that was possible to obtain from the core source code. Other than that they did not really make sense as they kept changing randomly despite trying to keep the conditions consistent.
Figure B.1: Osbech results for the Thread Yield Time performance
Appendix C

Command Line Window Building

DATE: 15 July 2003

------------------
Stage: Wince.bat -> ...

------------------
SETTING UP THE ENVIRONMENT USING Wince.bat

=> Initializing the variables AND running Wince.bat
- set _PROJECTROOT=<DRIVE:\>Wince400\Public\MyPlatform
- set _FLATRELEASEDIR=<DRIVE:\>Wince400\Release
- set _WINCEROOT=<DRIVE:\>Wince400
- Change DIR to %_WINCEROOT%\Public\Common\Oak\Misc.
  then run the wince.bat
  Wince.bat CPUFamily CPU CE MyProject MyPlatform
  i.e _TGTCPUTYPE, _TGTCPU, _TGTOS, _TGTPROJ, _TGTPLAT variables
  respectively e.g x86, i486, CE or NT, MyTinyKernel, CEC or EMULATOR
  respectively
NB: The _FLATRELEASEDIR pertains to the directory where the
released project source code and binary files will be placed.
The default setting is
%_WINCEROOT%\Public\%_TGTPROJ%\RelDir\%_TGTPLAT%Debug.

WHAT WE DID
* set _projectroot=c:\wince410\public\kernel
  (kernel folder none existent at this point)
* set _winceroot=c:\wince410
* set _flatreleasedir=c:\wince410\Release

* Ran the wince.bat in Public\common\oak\misc
  - wince.bat x86 kernel EMULATOR

NB: can now run a simple batch file to setup these environment:

  wincesetup.bat
  make sure the location of the file is in the path
GENERAL NOTES AND FINDINGS

# Building Information etc is supplied in the form of environment variables

=> system components, components developed by OEMS, what to include and what not to include etc.

# These environment variables are project specific via the project-batch file

e.g mykernelproject.bat.

=> Stored in _PROJECTROOT

=> The wince.bat runs this batchfile to setup the environment variables

# Base Config Components - suspect they deal with setting up of the core kernel found in file cesysgen.bat for the specific project.

Below is an excerpt from a cesysgen.bat for a tinykernel project.

--------------- TinyKernel Example -----------------------------

set CE_MODULES=coredll nk
set COREDLL_COMPONENTS=coremain lmem thunks coreimmstub
set NK_COMPONENTS=
set COREDLL_MESSAGEDIALOGBOXCUSTOMIZE_COMPONENT=messagedialogboxcustomize
if "%BSP_FPEMUL%"="1" set __SYSGEN_FPEMUL=1
if "%__SYSGEN_FPEMUL%"="1" set COREDLL_COMPONENTS=%COREDLL_COMPONENTS% fpemul
-------------------------------------------------------------------------

The other optional components chosen via the wizard are added as follows:

Optional components from core OS

------------------------------

if "%SYSGEN_FIBER%"="1" set COREDLL_COMPONENTS=%COREDLL_COMPONENTS% fiber
if "%SYSGEN_FMTMSG%"="1" set COREDLL_COMPONENTS=%COREDLL_COMPONENTS% fmtmsg
if "%SYSGEN_FMTRES%"="1" set COREDLL_COMPONENTS=%COREDLL_COMPONENTS% fmtres
if "%SYSGEN_CORESTRA%"="1" set COREDLL_COMPONENTS=%COREDLL_COMPONENTS% corestra
if "%SYSGEN_STDIO%"="1" set COREDLL_COMPONENTS=%COREDLL_COMPONENTS% coresiow
if "%SYSGEN_STDIOA%"="1" set COREDLL_COMPONENTS=%COREDLL_COMPONENTS% coresioa
if "%SYSGEN_STDIO%"="1" set SYSGEN_SERDEV=1
if "%SYSGEN_STDIOA%"="1" set SYSGEN_SERDEV=1
if "%SYSGEN_SERDEV%"="1" set COREDLL_COMPONENTS=%COREDLL_COMPONENTS% serdev
...etc ...

# Interesting is that in the Public\COMMON directory a sources.cmn file is defined as
WINCEOEM=1
WINCEPROJ=COMMON
WINCETREE=winceos
WARNISERROR=1
Then in the COREOS directory the sources.cmn file defines in pretty much the same way. So they seem to basically depend on each other. Thus when a blddemo is called, building COMMON will call winceos folder and thus compiling the source code in there provided the dirs file defines the subdirectories.

We want to believe that then the resulting compiled stuff should be automatically placed in the respective COMMON directories such as: TARGET => Contains the application and DLLs that make the Windows CE OS and components. The DLL and .exe files, with their associated .pdb and .map files, are contained in directories with CPU\Debug format. etc It is only for Windows CE OS components. It’s located in the public\common\oak directory. So COMMON contains and deals with the building of the Core Operating System modules # HLBASE => Support for Headless-based devices # IABASE => Support for display-based devices # _DEPTREES => To enable this variable in the platform builder do the following:
   go to: TOOLS -> OPTIONS -> BUILD TAB -> CHECK Enable Deptree Build