MILWAUKEE SCHOOL OF ENGINEERING

ELECTRICAL ENGINEERING AND COMPUTER SCIENCE DEPARTMENT

CS-400 SENIOR DESIGN I

SEGA® DREAMCAST™ AND MICROSOFT® WINDOWS® CE GAME DEVELOPMENT

<u>Technology Report</u>

Submitted to: Prof. Barnekow

Submitted by: Hai Bui
Adam Lindsay
Brett Morien
Nathan Schultz

Date Submitted: January 14, 2000

TABLE OF CONTENTS

SEGA DREAMCAST

The Dreamcast was Sega's answer to Sony's Playstation and Nintendo's N64 system. Sega had a failing run in the area of cartridge-based home game consoles. Sega decided that what they needed was a system that not only could perform the same as the Playstation and the N64, but also to keep the system a valuable console in the future. Sega's Dreamcast has the opportunity to do as its designers wanted. The Dreamcast has combined several elements to ensure that it will remain a force in the home console area. These elements will be discussed; their advantages and disadvantages will also be explored.

One of Sega's biggest down falls in the earlier cartridge era was that in order to develop for the Sega console one would have to learn a new language and technology. Sega has side stepped this issue in the Dreamcast by using tools that are currently being taught and used by many people in the programming field. Sega has the Dreamcast using the Windows CE operating system for development. The Dreamcast still requires a special SDK, but the SDK does not require a person to learn a programming language that they are unfamiliar with. The SDK's main purpose is to setup Windows CE to handle the programming for the Dreamcast and its components. Another advantage that Sega gained from using the Windows CE toolkit is that the Dreamcast can now use DirectX. DirectX is used all over the programming industry and with Sega using the Windows CE toolkit and DirectX a programmer does not have to worry about the interface between the hardware and the software.

The size of the required Windows CE platform is also an added benefit to the Sega Corporation. Below in Figure 1 one can see how the programs are layer for their use.

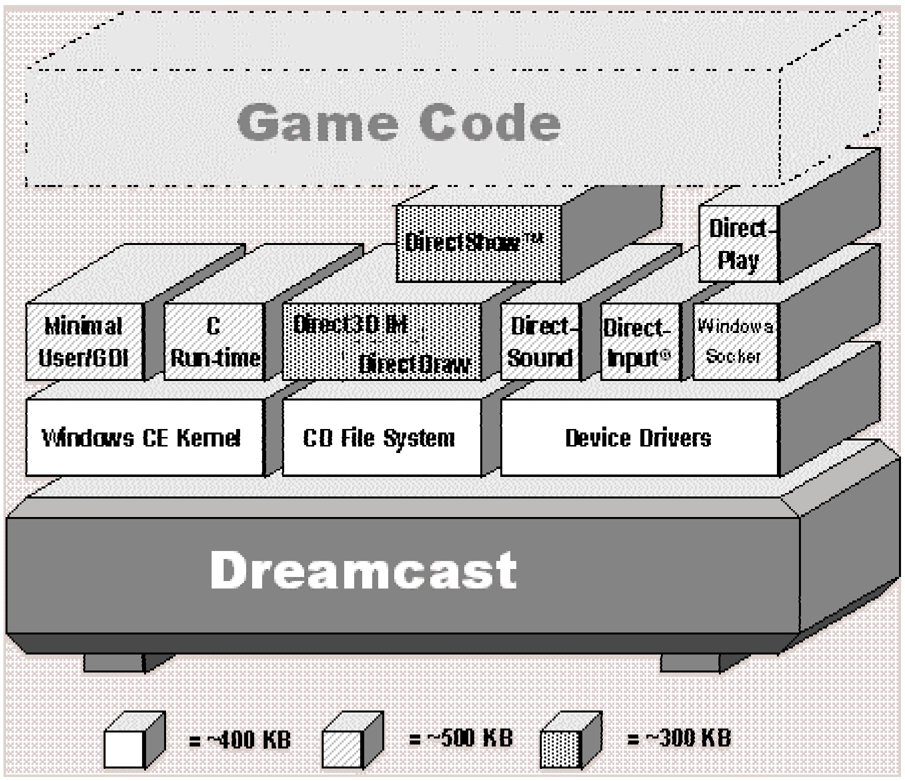

Figure 1: Dreamcast code layers and sizes

One can see from the above figure (Figure 1) that the main operating system of the Dreamcast is only a small percentage of the disk space. If one does not count the game code the operating system takes less than 5 MB of room on a gigabyte CD.

Developers for the Sega Dreamcast also see some benefits from Sega's use of the Windows CE system. Since, the programming for Windows CE for the Dreamcast is similar to what one might use in programming for Windows 9x. One can easily port games from the Windows CE standard to the Windows 9x standard. This is shown by a Figure taken from the Microsoft web page on developing for the Sega Dreamcast (Figure 2).

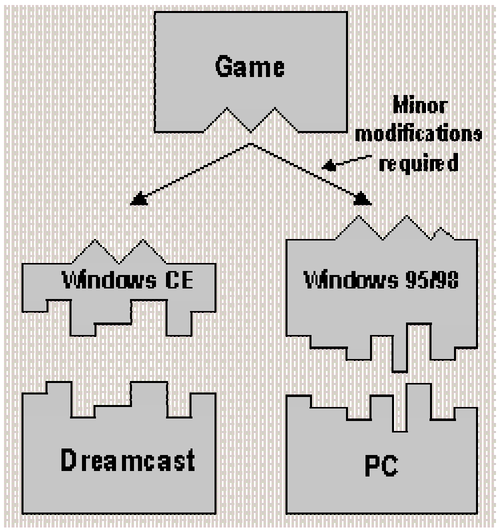

Figure 2: The portability of the program from the Dreamcast to the PC

Sega also has an advantage using its GigaROM CD. The GigaROM was originally developed to help keep its software from being pirated by users and to stop CD switching associated with PC and Playstation games. This choice made for more advantages than initially thought. With the GigaROM CD Sega was able to place the entire Windows CE platform on the CD rather than into the console. The fact that the entire console platform could be placed onto the CD added yet another advantage as now the Sega Dreamcast would no longer have to worry about version conflicts. Windows CE can be continually updated or upgraded as well as DirectX; this will no longer be a problem since the platform is contained on the GigaROM CD. The console downloads the platform from the CD thus the Dreamcast does not know, nor care what version of Windows CE or DirectX is running as long as it can be downloaded into its memory. The layers of the GigaROM CD can be seen below in Figure 3 as well as how they interact with the Dreamcast.

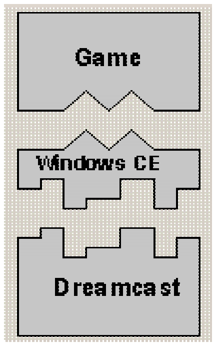

Figure 3: The interactive of the GigaROM CD and the Dreamcast Console

One may better understand the flexibility of the Dreamcast if they knew what was contained in the Dreamcast. The Dreamcast console includes a 200MHz Hitachi Sb microprocessor and VideoLogic PowerVR graphics chip capable of rendering more than 1 million polygons per second1. The sound chip is the Yamaha AICA, which uses the ARM7DI as its sound controller1. These microprocessors main function is to handle graphics, sound and system resources of the Dreamcast and its GigaROM CD. In the aspect of memory the Dreamcast has planned for the future as its present games don't require the full value of the memory provided. The system has 16MB of main memory, 8MB of video memory, and 2MB of sound memory1.

The Dreamcast also carries a benefit from the earlier mentioned Gigabyte Rom or GD-ROM. Unlike other consoles that either use cartridges (like the Nintendo 64) or standard CD-ROMs (like the Sony PlayStation), Sega designed the proprietary storage format for the Dreamcast called GD-ROM (Gigabyte Disk ROM). These discs can store both standard CD-ROM data and game data in a high-density band. The GD-ROM format allows Dreamcast titles to store more data than standard CD-ROMs without having to employ the much more expensive DVD drives.1

One critical point that should be mentioned is that, in events that are time critical, the Sega Dreamcast does have some flaws. The Windows CE platform covers important issues for the Dreamcast programmer such as the ones listed below:

- Virtual memory management, which all but eliminates memory fragmentation[2]
- Memory protection, which prevents games from crashing the operating system[2]
- CD file management, including asynchronous file loading and file enumeration using wildcard[2]
- Dynamic link libraries, which allow both code and data to be loaded, linked and unloaded at run time and under program control[2]
- Multithreaded event synchronization, which allows threads to lie dormant, consuming no CPU time, until an event occurs. Events can be signaled by other threads or by the operating system.[2]

However, some of the overhead associated with the Windows CE platform does not allow real time events to occur. To solve this problem many programmers will program the section of time critical code in the assembly language for the Hitachi Sb microprocessor. The assembly code runs faster and can be added inline using the Dreamcast SDK, solving the normal overhead time problem that can be associated with the Windows CE operating platform. Programmers have also in cases chosen the route of either rewriting part or all of the DirectX components in order to increase the efficiency of the game code.

To develop for the Dreamcast it not only requires Windows CE and the Dreamcast SDK as explained above, but also a development box referred to as a set5. The set5 development box is basically a Sega Dreamcast with a SCSI adapter attached. The set5 contains all the components as described above as well as SCSI adapter and other added ports to help the developer debug their program. In Figure 4, one can see what the development box looks like a mid-tower computer with added components.

If one could look at the front of the set5 one could see that the development box also includes 4 game ports for Dreamcast controllers and the Sega Dreamcast CD ROM bay. The set5 is a full emulator or debugging tool for the Dreamcast. This tool replicates every aspect of the Dreamcast.

More information on the Dreamcast can be found on the following website:

http://marcus.mangakai.org/dc/

[1] http://msdn.microsoft.com/library/periodic/period99/msft/msj/0799/directx/directx.htm
[2] http://msdn.microsoft.com/cetools/platform/backgrnd.asp

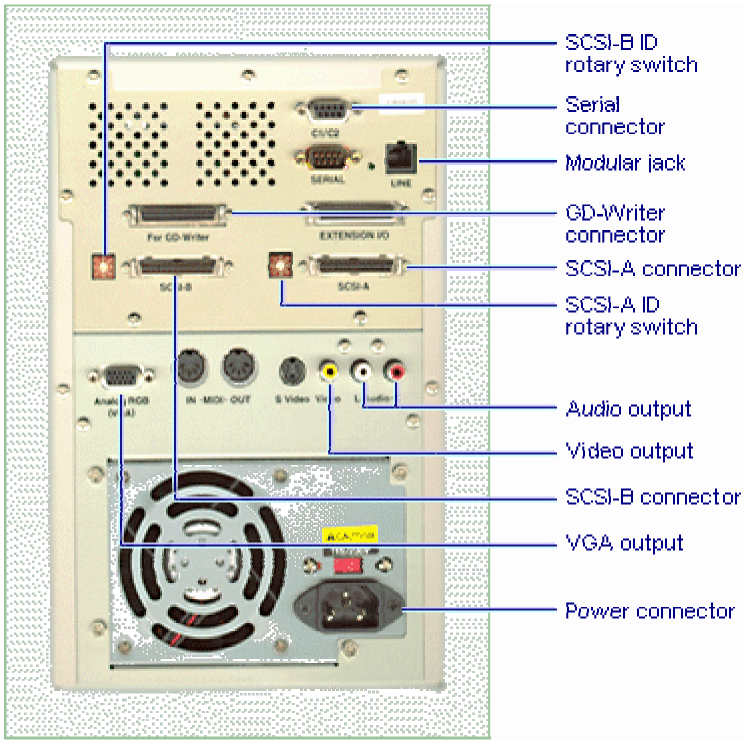

Figure 4: The Back of the Set5 Development Box

Currently, this development tool is the only device that the Windows CE Group is missing to complete its project as expected. If the Windows CE Group fails to obtain this device then a contingency plan will be used.

THE LIGHT GUN

Ever since the release of DuckHunt for the original Nintendo system, the light gun has been a significant part of consol gaming.

To understand the light gun, one must first understand input devices in general. The Dreamcast supports several types of devices, including:

- Light guns
- Game controllers
- Keyboards
- Visual Memory units
- Vibration devices
- Microphones

An input device is a device that receives user input, such as a joystick, gamepad, or light gun. These types of devices are handled through the DirectInput interface API. A non-input device is a device that does not receive user input, such as a Visual Memory timer, liquid crystal display (LCD), or external flash storage. These types of devices are handled through the Maple bus interface API.

Light gun, are viewed by the system as two distinct devices. For example, the input device connects through the DirectInput interface, which handles input from the trigger and buttons, and the non-input device connects through the ILGun interface, which handles the positioning of the gun on the screen.

Special APIs were developed for each non-input device type. For example, a vibration device connects to Dreamcast through the IVib device interface, while a light gun connects through the ILGun device interface. Each device interface is accessed through the Maple functions. The following list explains the general process for accessing a non*input device interface through the Maple APIs:

- The MapleEnumerateDevices function enumerates the devices that are connected to the Maple bus.
- Using the provided MapleEnumDeviceCallback callback function, MapleEnumerateDevices provides a pointer to a MAPLEDEVICEINSTANCE structure. One structure exists for each enumerated device.
- Each MAPLEDEVICEINSTANCE structure contains information about an enumerated device, such as the device type and a GUID. You pass the GUID to MapleCreateDevice.
- The MapleCreateDevice function instantiates an interface to the device and returns a ppIUnknown pointer.
- You pass ppIUnknown to QueryInterface, which then returns a pointer to the appropriate device interface, such as IVib and ILGun.
- You use the pointer provided by QueryInterface to access the functions of the device interface. For example, if an IVib interface is returned, you can begin accessing IVib::GetVibArbitraryWaveform and IVib::GetVibInfo.

The Sega Dreamcast currently has two light guns available for it:

The first light gun available for the Sega Dreamcast, pictured to the right, gives the user a controller imbedded in the light gun. This design features rapid fire, dual trigger, and auto-reload. **Figure 4A: Sega Dreamcast Light Gun 1 →**	
The second light gun, pictured to the right, provides a sleeker style. It also has an auto-reload feature. A better design gives this gun better accuracy. **Figure 4B: Sega Dreamcast Light Gun 2 →**	

Despite the differences in the above light guns, the system views them in the same way. To the system, the light gun is comprised of two separate devices: one input and one non-input. The input device is considered a joystick with a trigger and six buttons. You connect to it through the DirectInput APIs. The non-input device displays the light beam on the screen. You connect to it through the special ILGun device interface, which is accessed through the Maple APIs. The ILGun interface can be called at any time, but the recommended approach is to call it when the trigger is pressed. To identify the light beam's screen location, the gun's sensor must be calibrated by using ILGun::Calibrate.

Functional Coding Samples:

CLightgun.cpp. This file details the function definitions required to initialize the light gun input device:

```
/******************************************************************
Copyright (c) 1999 Microsoft Corporation

Module Name:
CLightgun.cpp

Abstract:
Member functions for the CLightgun class.
******************************************************************/
// **** Include Files ********************************************
#include "Lightgun.hpp"
/*******************************************************************
Function:
CLightgun::CLightgun

Description:
Constructor for CLightgun Class.

Arguments:
GUID        guidLightgun - GUID of the Lightgun device
CController *pcont - Controller to which the Lightgun device is
attached

Return Value:
None
******************************************************************/
CLightgun::CLightgun(GUID guidLightgun)
{
m_guidLightgun = guidLightgun;
m_plgunintf    = NULL;
}
/*******************************************************************
Function:
CLightgun::~CLightgun

Description:
Destructor for CLightgun class.

Arguments:
None

Return Value:
None
******************************************************************/
```

```
CLightgun::~CLightgun()
{
}

/*********************************************************************
Function:
CLightgun::Initialize

Description:
Initializes the CLightgun object.

Arguments:
None

Return Value:
TRUE on success, FALSE on failure.
*********************************************************************/
BOOL CLightgun::Initialize()
{
IUnknown *pIUnknown;

g_errLast = MapleCreateDevice(&m_guidLightgun, &pIUnknown);
if (CheckError(TEXT("Create Maple Device")))
return FALSE;

pIUnknown*>QueryInterface(IID_ILGun, (PVOID*)&m_plgunintf);
pIUnknown*>Release();

if (m_plgunintf == NULL)
return FALSE;

// Note, this event must be manually reset.
m_hEvent = CreateEvent(NULL, TRUE, TRUE, NULL);

return TRUE;
}
```

CLightgun.hpp. This file details the class definitions and function declarations for the light gun input device implementation:

```
/*******************************************************************
Copyright (c) 1999 Microsoft Corporation
Module Name:
    CLightgun.hpp

Abstract:
   CLightgun class declaration
*******************************************************************/
// **** Global Variables ********************************************
enum ELightgunCommands
{
    ecmdReadLoc,
};

// **** Classes *****************************************************
class CController;

class CLightgun
{
public:
    CLightgun(GUID LightgunGuid);
    ~CLightgun();

    // Initialize the Lightgun object
    BOOL Initialize(void);

    HANDLE m_hEvent;

    // LGun interface
    PLGUN m_plgunintf;

    // Last position the gun was fired at
    LGUN_POSITION m_lgunpos;

private:
    // The GUID of the Lightgun device
    GUID m_guidLightgun;
};
```

COM

Along the evolution of the Windows operating system, Microsoft decided that there needed to be better methods of inter-process communication. They 'borrowed' the concept of OLE (Object Linking and Embedding) from the Wang Corporation and incorporated it into Windows 3.11. It was based on the vision of embedding objects from one application into the workings of another application. This allowed a person to, for example, insert a bitmap and an Excel chart into a Word application and make changes without having to go back to the original program. The idea of OLE was expanded into a larger system of shared components and evolved into COM.

COM stands for Component Object Model. What this means is that an application can be built out of smaller components that, as a side benefit, are completely reusable by any application. This is accomplished by building components with interfaces. The interfaces are really not much more than a structure of function pointers. When a component is created, it implements an interface by linking these pointers to functions within itself. What this means, then, is that any application written around this interface can be used in the same way. The permutations become endless at that point.

An interface is much like a system contract. There are certain things that are promised by the interface, and there are things that are expected by applications using it. In order for a system like this to be feasible, interfaces must never change once they have been published. If this were allowed, then entire applications would be rendered useless upon change. Even if the change were so simple as to add another method, the interface structure would be changed and it would be impossible to implement without code change. If code needed to be changed that often, we would call it Linux.

One final key feature to COM architecture under a Windows environment is the extension called DCOM (Distributed Component Object Model). This allows COM objects to be instantiated on a separate computer connected to the same network. This has advantages in the areas of distributed computing, some publish subscribe architectures, etc.

DIRECTX

OVERVIEW

The version of DirectX that will be used in this project is 6.1. This includes up-to-date versions of DirectDraw, DirectSound, Direct3D, DirectInput, DirectPlay, and DirectMusic. The specialized version of DirectX for Dreamcast does not include support for DirectMusic and support for the other components is limited.

The first thing that happens after creating a DirectX object is finding the devices that are on the system and exploring their capabilities. This is accomplished in a two-step process. First, an enumerating function is used that calls a callback function for each device found. After the devices are enumerated, a device is chosen by the application or by the user. A function is called (GetCaps) that retrieves a capability structure with information about the device. This procedure applies to all of the DirectX objects.

DirectX is based off of COM technology. DirectX is composed of the following components: DirectDraw, DirectSound, Direct3D, DirectMusic, DirectPlay, DirectShow and DirectInput. Each of these components are implemented using a series of interfaces and have their own subparts.

DIRECTDRAW

The DirectDraw system's primary canvas is a surface. Surfaces are created, and bitmapped data is "blitted" to the surface. These surfaces can be swapped and moved between video memory and system memory and represent different on and off screen regions. DirectDraw allows the user quick access to the video hardware without having to go through the Windows' GDI system. This allows for tremendous increases in speed since the Windows GDI was not designed for high speed applications. The other advantage of DirectDraw is the abstract approach to the display that it takes. It doesn't require the user to be knowledgeable of the specific way a hardware manufacturer implements its card or even what card is at the other end.

DIRECTSOUND

The primary purpose of DirectSound is to provide low-latency access to the audio hardware and allow easy mixing of sounds. In the old days, the user had to select their sound driver from a menu before the game could start. DirectSound allows for the programmer to write for the DirectX interface instead of attempting to include every known device driver.

The DirectSound system uses sound buffers for most of its work. A sound buffer is either a segment of memory holding a sound clip or a circular buffer with streaming data in it. DirectSound allow direct access to the audio hardware without having to deal with the latency of using the Windows multimedia system. DirectSound allows for mixing of multiple audio tracks and for hardware mixing on some sound devices. Finally, DirectSound comes with built in three-dimensional sound support that is rather easy to work with.

DIRECT3D

Direct3D is a much more sophisticated beast. This area has been the focus of a large portion of the development effort at Microsoft, and it will be the primary focus in this project. Direct3D is an extension of the DirectDraw system. It allows support for hardware and software rendering of 3D graphics. There is a common set of capabilities included (such as fog, alpha blending, translations) that are passed to the graphics hardware if the capabilities exist; otherwise, they are emulated using software. Since these decisions are made internally, the programmer has no concern about what video cards are installed on the target machine.

Direct3D comes in two flavors: immediate mode and retained mode. Retained mode is a high level abstraction of the 3D API designed for rapid development with a sacrifice made to run-time performance. Immediate mode is designed for lower level access to video hardware, yet remaining device-independent in code. This is ideal for developers porting existing games or those who demand higher performance, even if it takes a bit longer to develop.

Direct3D provides support for translation and rotation effects, lighting, shading, clipping, texturing, depth buffering, and transparency effects. Direct3D also has the ability to remain device independent. Since some hardware supports features that others don't, a robust software emulation layer exists that will fill in the blanks.

Since the group is on a limited time scale and the capabilities of the Dreamcast are so great, retained mode will most likely be the method of choice. Retained mode works by the usage of faces. A face represents a single polygon and hold information about texture, material, surface normal, color and topology. These faces are fed into the Direct3D system, and they are brought to the display.

Direct3D retained mode also includes support for light, mesh, viewport, and texture structures for higher extractions from the hardware. These structures can be manipulated in a high level way to created the desired effects.

DIRECTINPUT

The concept of DirectInput is that it allows many devices to be attached to a game and be treated roughly the same way. In the old days, the user of a game would have to inform the game of the hardware attached to the game, such as a joystick, and calibrate it before the game can start. DirectInput allows the game to detect attached devices and extract common calibrations from the operating system. As a side note, DirectInput has native support for forced feedback devices.

DirectInput allows for an abstraction of the attached devices for simplicity. For instance, a mouse device can be treated the same as a light gun device. This sort of architecture allows game programmers to avoid doubling up on unnecessary work. For example, with a light gun, an event is fired that announces that the trigger has been pressed. The application then can poll for the position of the gun on the screen and get screen coordinates back without having to deal with counting scan lines and timing.

DIRECTPLAY

DirectPlay allows the user to have a common interface for gameplay between separate consoles or computers. The DirectPlay interface masks the specifics of the type of network in order to make connecting computers seamless. This allows the programmer to develop the game to be multiplayer without specifying how the connection is to be made and building it into code manually.

DirectPlay is centered about the concept of a lobby where players converse before a game, and an administrator can set rules for the upcoming game. After that, DirectPlay takes care of keeping the gamestate synchronized between stations.

Dreamcast supports DirectPlay4 and uses it to connect the Dreamcast to the Internet. The user has the ability to buy a modem that connects the Dreamcast to the Internet through the phone lines. This is the primary means of communication for DirectPlay at this time. This group has no plans to incorporate DirectPlay into the game.

WINDOWS CE

OVERVIEW

Microsoft Windows CE is a compact, scalable operating system (OS) that is designed for a variety of embedded systems and products. Designed and written from scratch for hardware with limited resources, Windows CE supports a multithreaded, multitasking, fully preemptive OS environment. Its modular design enables embedded systems developers and application developers to customize it for a variety of products, such as consumer electronic devices, specialized industrial controllers, and embedded communications devices.

Windows CE supports various hardware peripherals, devices, and networking systems. These include keyboards, mouse devices, touch panels, serial ports, Ethernet connections, modems, universal serial bus(USB) devices, audio devices, parallel ports, printer devices, and storage devices, such as PC Cards. Windows CE supports the following technologies:

- Real-time processing for managing time-critical responses
- a variety of serial and network communication technologies, including USB support
- Mobile Channels, which provides Web services for Windows CE users
- Automation and other methods of interprocess communication

Additionally, Windows CE supports more than 1,000 common Microsoft Win32 APIs and several additional programming interfaces that one can use to develop applications. These interfaces include:

- Component Object Model (COM)
- Microsoft Foundation Classes (MFC)
- Microsoft ActiveX controls
- Microsoft Active Template Library (ATL)

It is important to note that only the common APIs and programming interfaces are implemented for Windows CE. The developer has to verify whether a Win32 API is supported by Windows CE or not. In addition, Windows CE only supports one-level deep directory structure. It does not support security, current directory, nor handle inheritance. Therefore, the majority of Microsoft Win32 APIs parameters must be set to NULL or 0 when utilized for Windows CE developments.

WINDOWS CE ARCHITECTURE

OVERVIEW

Windows CE is built from a number of discrete modules, each providing specific functionality. Several of these modules are divided into components. Components enable Windows CE to become very compact (less than 200 KB of ROM), using only the minimum ROM, RAM, and other hardware resources that are required to run a device.

The Windows CE OS contains four modules that provide the most critical features of the operating system: the kernel; the object store; the Graphics, Windowing, and Events Subsystem (GWES); and communications. Windows CE also contains additional, optional modules that support such tasks as managing installable device drivers and supporting COM.

KERNEL

The kernel is the core of the OS, and is represented by the Coredll module. It provides the base operating system functionality that must be present on all devices. The kernel is responsible for memory management, process management, and certain required file management functions. It manages virtual memory, scheduling, multitasking, multithreading, and exception handling.

Most components of the Coredll module are required for any configuration of Windows CE. There are some optional kernel components, however, that are needed only when one includes such operating system features as telephony, multimedia, and graphics device interface (GDI) graphics.

OBJECT STORE

The Filesys module supports the Windows CE object store API functions. The following table shows the types of persistent storage that the object store supports.

Type of storage	Description
File system	Contains application and data files
System registry	Stores the system configuration and any other information that an application must access quickly
Windows CE database	Provides structured storage

The object store offers an alternative to storing user data and application data in files or in the registry. These various object store components can be selected or omitted during the operating system build process to include only those features that are required.

GWES

GWES is the graphical user interface between a user, the application, and the OS. GWES handles user input by translating keystrokes, stylus movements, and control selections into messages that convey information to applications and the OS. GWES handles output to the user by creating and managing the windows, graphics, and text that are displayed on display devices and printers.

Central to GWES is the window. All applications need windows in order to receive messages from the OS, even those applications created for devices that lack graphical displays. GWES provides controls, menus, dialog boxes, and resources for devices that require a graphical display. It also provides the GDI, which controls the display of text and graphics.

COMMUNICATIONS

The communications component provides support for the following communications hardware and data protocols:

- Serial I/O support
- Remote Access Service (RAS)
- Transmission Control Protocol/Internet Protocol (TCP/IP)
- Local Area Network (LAN)
- Telephony API (TAPI)
- Wireless Services for Windows CE

OPTIONAL COMPONENTS

In addition to the primary modules just described, other operating system modules are available. These include modules and components in the following categories:

- Device manager and installable device drivers
- Multimedia (sound) support module
- COM support module
- Windows CE Shell module

Each module or component provided in Windows CE supports a group of related API functions that are available to the developer.

WINDOWS CE FOR DREAMCAST

Windows CE for Dreamcast is an operating system designed to support high-performance, platform-independent game development with built-in Internet support on Sega's Dreamcast game console. Windows CE for Dreamcast consists of customized and optimized derivatives of Windows CE and DirectX, offering compatibility with the Win32 and DirectX APIs currently supported under Windows 9x.

Because Dreamcast is compatible with desktop versions of Windows and DirectX, games written for Windows CE can be ported to and from desktop versions of Windows. This compatibility enables cross-platform development, eliminating the need to understand every detail of the underlying hardware. Windows CE also provides high performance, a comprehensive set of APIs, Internet connectivity, and the Visual Studio integrated development environment (IDE).

Windows CE for Dreamcast is based on Windows CE, which was designed and written from scratch to provide high performance on inexpensive hardware. Windows CE for Dreamcast supports a subset of the Win32 API, but Windows CE for Dreamcast and Windows 9x share no code.

Windows CE for Dreamcast is smaller, faster, and cleaner than Windows 9x in many ways and for many reasons, including the following:

- Windows 9x contains a substantial amount of 16-bit code in order to maintain compatibility with older software. Windows CE for Dreamcast is 32-bit only and contains no legacy code, having been written from scratch.
- The Graphical Device Interface (GDI) supported by Windows 9x is largely absent from Windows CE for Dreamcast. All screen output is performed using DirectDraw surfaces.
- All Windows CE for Dreamcast games run in full-screen exclusive mode, as do most Windows 9x games. This requirement allows numerous Windows functions to be discarded, including multiple overlapping windows, window frames, and focus management.
- Windows CE for Dreamcast does not support a standard library of user interface objects, such as buttons, menus, scroll bars, and list boxes.
- There is no Windows desktop, no mouse driver, and no mouse cursor.
- With the exception of a small amount of optional backup RAM, Windows CE for Dreamcast does not support write-able storage media or printers.
- Windows CE for Dreamcast supports a single virtual address space, which is shared by all processes. This streamlines interprocess communications. Games are prevented from corrupting one another by using page protections.

<u>WHY WINDOWS CE FOR DREAMCAST</u>

Windows CE for Dreamcast is designed to achieve hardware independence for game development. a Windows CE for Dreamcast game can be ported to any platform that supports the Win32 and DirectX APIs, including platforms that did not yet exist when the game was written. The resulting reduction in cross-platform development costs, together with the productivity gains associated with using an operating system to provide basic services, are key benefits of developing games for Windows CE for Dreamcast. Because Windows CE for Dreamcast is Windows-compatible, game programmers will be able to move freely between consoles and PCs for the first time.

A software-based platform, Windows CE offers hardware independence. A Windows CE game can be ported to desktop versions of Windows that support the DirectX APIs. By using familiar APIs and development tools, experienced Windows game developers can develop games for Windows CE almost immediately. The inclusion of DirectX in Windows CE gives the developer a wide range of features, from animation and 3-D sound to a wide choice of input devices and Internet communications.

Even if a developer chooses to substitute custom components for all or part of DirectX, he or she can still benefit from the following services offered by the Windows CE kernel and file system:

- Virtual memory management, which all but eliminates memory fragmentation.
- Memory protection, which prevents games from crashing the operating system.
- File management, including asynchronous file loading and file enumeration using wildcards.
- Dynamic-link libraries (DLLs), which allow both the code and the data to be loaded, linked, and unloaded at run time under program control.
- Multithreaded event synchronization, which allows threads to lie dormant, consumes no CPU time, until an event occurs. Events can be signaled by other threads or by the operating system.

As Windows CE for Dreamcast is a streamlined version of Windows CE, it provides the performance and small memory demands necessary for game console hardware. It also supports extensive development tools and a comprehensive set of APIs, including the full-featured Visual Studio integrated development environment, as well as a version of DirectX that is optimized for Dreamcast.

PORTING ISSUES IN WINDOWS CE FOR DREAMCAST

The streamlined design of Windows CE for Dreamcast presents the following issues when porting a game from Windows 9x to Dreamcast:

- Windows CE for Dreamcast implements only a subset of the Windows CE and DirectX APIs. For example, the DirectInput game pad support is reduced because the properties of the Dreamcast game pad are known.
- Dreamcast has less RAM than most personal computers.
- Dreamcast has no hard disk and insufficient backup RAM for storing complex saved games.
- Windows CE for Dreamcast does not support mouse input or cursor objects. The standard Dreamcast input device is the game pad, and any cursors must be supplied by the game.
- Windows CE for Dreamcast does not support standard Windows user interface objects, such as dialog boxes.
- Dreamcast requires 3-D textures to be square, and their sizes must be powers of 2.
- Windows 95 supports ASCII text strings, while Windows CE supports both Unicode and ASCII.
- Windows CE does not support the Data Access Object API, which some games use to maintain databases containing statistics or other information.
- Dreamcast does not use an Intel processor. Any x86 assembly code must be rewritten.

When porting a game from Windows CE for Dreamcast to Windows 9x, there following issues also have to be considered:

- Personal computers may or may not have 3-D graphics hardware.
- Personal computers do not have Q-Sound hardware, which significantly increases the CPU resources required to support 3-D sound.
- Game pads are not standard equipment on personal computers. Keyboard input can be added as an option.
- One must convert code that uses Dreamcast flash storage to use the standard Windows file system.
- Any SH4 assembly code must be rewritten for an Intel processor.
- Many current personal computers are running on ASCII code, and have not yet supported Unicode programs.

According to such issues and limitations, porting a game from Windows CE for Dreamcast to Windows 9x is more achievable. There are 3-D graphics and 3-D sound hardware available for the personal computer upgrade, the keyboard can be used to substitute the game pad, the Windows file system and ASCII code can still be utilized by functions that perform conversion available in Windows CE Platform Development kit.

WINDOWS CE FOR DREAMCAST ARCHITECTURE

Windows CE includes discrete modules, and each one provides specific functionality. Several of these modules are divided into components, which enables Windows CE to become very compact. As a result, a minimum amount of ROM, RAM, and other hardware resources are required to run a device.

The Dreamcast OS includes the following modules:

- Windows CE for Dreamcast Kernel
- Object Store
- Persistent Storage and Physical Memory Usage
- Communications Interface
- Graphics Device Interface
- MIDI Interface
- Timing Interfaces
- C Run-Time Library

KERNEL

The Windows CE kernel is the core OS of the Dreamcast. It includes support for memory management, process management, exception handling, multitasking, and multithreading. All Dreamcast games run in a fully preemptive, multitasking environment in protected memory spaces. Windows CE supports Unicode and TrueType fonts, which allow applications to be internationalized.

The Windows CE kernel uses dynamic-link libraries (DLLs) to optimize the use of available memory.

PROCESSES AND THREADS

Windows CE supports up to 32 simultaneous processes. Each process is a single instance of an application and can create multiple threads of execution. The total number of threads is limited only by available physical memory. Threads can be synchronized within a process and between multiple processes (interprocess synchronization).

INTERRUPT HANDLING

To provide efficient processing of interrupts, Windows CE splits interrupt handling into two parts: an interrupt service routine (ISR) and an interrupt service thread (IST). The ISR launches the IST that is responsible for handling the event. Then, the ISR returns, and the system responds to the next interrupt. The division of interrupt handling allows the ISR to be very small and very fast, features that minimize interrupt latencies and speed up interrupt processing.

MEMORY ARCHITECTURE

Windows CE optimizes memory management for game development by using virtual memory and memory-mapped files.

The Hitachi SH-4 CPU for the Dreamcast system uses 32-bit memory addressing, allowing the SH-4 to address up to 4 GB of virtual memory. The kernel divides the memory into the following three sections:

- The kernel reserves a 2 GB address space.
- a 1 GB address space is subdivided into thirty-two 32 MB slots, one for each process. (a maximum of 32 processes can run concurrently.)
- The final 1 GB address space is shared among all processes and is used for large data blocks, such as memory-mapped files.

OBJECT STORE

The object store offers an alternative to storing user data and application data in files or in the registry. These various object store components can be selected or omitted during the operating system build process to include only required features.

PERSISTENT STORAGE AND PHYSICAL MEMORY USAGE

Since a Dreamcast game unit does not have a hard disk, physical memory plays a different role on a Dreamcast system than it does on a desktop computer. To maximize predictable timing, an entire game, its DLLs, and the operating system are loaded into RAM from the GD-ROM when the user starts the game. Only the boot ROM is stored in ROM.

The system registry is not persistent. Applications running on Windows CE can access and modify information in the registry with standard functions. To permanently store changes to the registry, its contents must be saved to an external flash memory device. Flash is an optional device that preserves data if power is lost. It plugs into the Dreamcast game unit.

COMMUNICATIONS INTERFACE

The communications interface supports serial communications, Internet client applications, and Remote Access Service (RAS). Windows CE supports the following options for serial I/O and networking through the Internet:

- High-level Internet networking support (WinInet), using the HTTP protocol and the Dreamcast modem.
- Low-level Internet networking support using a subset of Winsock version 1.1.
- a RAS client.
- Point-to-Point Protocol (PPP) for serial link and modem communications.

SERIAL COMMUNICATIONS

Serial I/O is used when there is a direct one-to-one connection between two devices. Transferring information over a serial cable connection is similar to reading from or writing to a file, and uses some of the same functions.

NETWORK COMMUNICATIONS

Windows CE supports a network stack that is accessible only through the Winsock interface. WinInet also uses Winsock internally and handles the details of setting up and managing socket connections. Windows CE supports a RAS client, which is a multi-protocol router used to connect remote devices. The Windows CE version of RAS supports only one point-to-point connection at a time. At the base of the network stack, Windows CE supports data-link layers for serial-link networks and local area networks (LANs). Dreamcast connects to a network through a built-in modem. Windows CE uses PPP to support serial communications.

GRAPHICS DEVICE INTERFACE

The Graphics Device Interface (GDI) module includes the low-level graphics functionality needed to display text and support DirectX graphics. The GDI provides the following low-level functions that support the higher-level graphics features of DirectX:

- Loading and initializing fonts
- Text output
- Loading bitmaps for use in textures
- Managing palettes

Other necessary user interface capabilities have been replaced in Windows CE by DirectDraw.

The GDI manages Windows messages, system timers, and a set of functions for manipulating rectangles. It also renders text and performs software bit block transfer (blit) operations. Hardware blits are performed using DirectDraw. Color displays with a color depth of up to 32 bits per pixel (BPP) are supported, as well as memory and display device contexts (DCs).

MIDI INTERFACE

The MIDI interface supports the playing of high-quality music. Windows CE defines a minimum MIDI configuration of a General MIDI System for playing, not recording, music. The MIDI API shares the 2 MB of dedicated sound memory on Dreamcast, with the requirements of DirectSound.

TIMING INTERFACES

The timing interfaces support dates, times, and accurate timing activities by using high-resolution or multimedia timers. The Windows CE timing APIs support system time, the date, file times, and high-resolution timing for accurate time measurement within a game. For specific application timing, a timer object can set up its own thread to interrupt the system at specified time intervals.

C RUN-TIME LIBRARY

This Windows CE Toolkit 2.0 for Dreamcast (the toolkit) includes the complete Windows CE C run-time library. The functions include general math, trigonometry, random number generation, string manipulation, character conversion, and memory manipulation. The **sin** and **cos** functions declared in the FloatMathLib.h header and FloatMath.lib files are special, high-speed hardware implementations on the Hitachi SH-4. The other functions are declared in the Stdlib.h header file.

WINDOWS CE FOR DREAMCAST CORE SYSTEM

PROCESSES AND THREADS

All Windows CE–based applications consist of a *process* and one or more *threads*. a process is a single instance of a running application. A thread is the basic unit to which the Windows CE operating system (OS) allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread. Windows CE for Dreamcast supports up to 32 simultaneous processes.

In a multithreaded process, each thread is allocated a slot in the Thread Local Storage (TLS), which is a method to store thread-specific data. The operating system stores all dynamic-link libraries (DLL), the stack, the heap, the application code, and the process data section in the slot assigned to the process. DLLs are loaded at the top of the slot, followed by the stack, the heap, and the executable file (.exe). The bottom 64 KB of the slot is always left free.

Each process can create multiple threads of execution. A thread is a single execution path within a process. Each process creates a primary thread. As each thread belongs to a particular process, a process and its threads share the same memory space. The total number of allowable threads is limited only by available physical memory.

Each thread operates independently from its process. However, a thread often needs to be managed by the process that owns it. Windows CE supports thread synchronization by providing a set of wait objects. These objects stop a thread until a change in the wait object signals the thread to proceed. Supported wait objects include:

- Critical-section objects.
- Named and unnamed event objects.
- Named mutual exclusion (mutex) objects.
- Thread objects.

Windows CE implements thread synchronization with a minimum of processor resources by using the kernel to handle thread-related tasks, such as scheduling, synchronization, and resource management. A game does not need to perform thread-management functions such as polling for process or thread completion.

When a process is created, Windows CE allocates a 64-slot array for each running process. When a DLL attaches to a process, the DLL calls the **TlsAlloc** function, which looks through the array to find a free slot. The function then marks the slot "in use" and returns an index value to the newly assigned slot. If no slots are available, the function returns –1. Individual threads cannot call **TlsAlloc**. Only a process or DLL can call the function and it must do so before creating the threads that will use the TLS slot.

To create a process, call the CreateProcess function. The *lpApplicationName* parameter must specify the name of the module to execute. Windows CE does not support passing NULL for *lpApplicationName*.

To terminate a process, call the **TerminateProcess** function. Processes do not have exit codes and cannot terminate themselves. TerminateProcess cannot be used to terminate a protected server library for processes contained within it.

To measure the performance for the application, use the GetThreadTimes function, which returns the total time a process has taken to perform a task.

To create a thread, call the CreateThread function. Resources used by a thread can be freed when it is no longer needed by calling the ExitThread function. Calling ExitThread for the primary thread causes the application to terminate.

Note: A process terminates if its primary thread is terminated, even if there are other active threads for the process. It also terminates if a related secondary thread generates an exception that is not handled.

CREATING A THREAD LOCAL STORAGE

Thread local storage (TLS) is the method by which each thread in a multithreaded process allocates a location in which to store thread-specific data. There are several situations in which one may want a thread to access unique data. For example, a racing game may implement instances of the same thread for each racer. The DLL that provides the functions for various racers' movements can use TLS to save data about the current speed or position for each racer.

TLS uses a TLS array to save thread-specific data. When a process is created, Windows CE allocates a 64-slot array for each running process. Only a process or DLL can call the function and it must do so before creating the threads that will use the TLS slot. Once a slot has been assigned, each thread can access its unique data by calling the **TlsSetValue** function to store data in the TLS slot, or the **TlsGetValue** function to retrieve data from the slot.

The following table describes the TLS functions that are supported by Windows CE.

Function	Description
TlsAlloc	Allocates a TLS index. The index is available to any thread in the process for storing and retrieving thread-specific values. One must store this index in global memory, where all threads can retrieve its value.
TlsFree	Releases the TLS index, making it available for reuse.
TlsGetValue	Retrieves the value that is pointed to by the TLS index.
TlsSetValue	Stores a value in the slot that is pointed to by the TLS index

CREATING A PROCESS

To start a process from within another process, call the **CreateProcess** function, which loads a new application into memory and creates a new process with at least one new thread.

The following code example shows the CreateProcess function prototype.

```
BOOL CreateProcess(LPCTSTR lpApplicationName,
LPTSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpProcessAttributes,
LPSECURITY_ATTRIBUTES lpThreadAttributes,
BOOL bInheritHandles, DWORD dwCreationFlags, LPVOID lpEnvironment,
LPCTSTR lpCurrentDirectory, LPSTARTUPINFO lpStartupInfo,
LPPROCESS_INFORMATION lpProcessInformation );
```

Because Windows CE does not support security or current directories and does not handle inheritance, the majority of the parameters must be set to NULL or 0. The following code example shows how the function prototype would look when all nonsupported features are taken into consideration.

```
BOOL CreateProcess(LPCTSTR lpApplicationName,
LPTSTR lpCommandLine, NULL, NULL, FALSE,
DWORD dwCreationFlags, NULL, NULL, NULL,
LPPROCESS_INFORMATION lpProcessInformation );
```

The first parameter, *lpApplicationName,* must contain a pointer to the name of the application to start. Windows CE does not support passing NULL for *lpApplicationName* and looks for the application in the following directories, in the following order:

- The path specified in *lpApplicationName*, if one is listed.
- An OEM-specified search path.
- The Windows directory (\Windows).
- The root directory in the object store (\).

The *lpCommandLine* parameter specifies the command line to pass to the new process. The command line must be passed as a Unicode string. The *dwCreationFlags* parameter specifies the initial state of the process after loading. The following table describes all of the supported flags.

Flag	**Description**
0	Creates a standard process.
CREATE_SUSPENDED	Creates a process with a suspended primary thread.
DEBUG_PROCESS	Creates a process to be debugged by the calling process.
DEBUG_ONLY_THIS_PROCESS	Creates a process to be debugged by the calling process, but doesn't debug any child processes that are launched by the process being debugged. This flag must be used in conjunction with DEBUG_PROCESS.
CREATE_NEW_CONSOLE	Creates a new console.

The last parameter used by **CreateProcess** is *lpProcessInformation*. This parameter points to the **PROCESS_INFORMATION** structure, which contains data about the new process. The parameter can also be set to NULL.

If the process cannot run, **CreateProcess** returns FALSE. For more information about the failure, call the GetLastError function.

TERMINATING A PROCESS

The most common way to terminate a process is to have it return from a **WinMain** function call. One can also terminate a process by having the primary thread of the process call the **ExitThread** function. a Windows CE process automatically terminates if its primary thread is terminated, even if there are other active threads in existence for the process. **ExitThread** returns the exit code of the process. One can determine the exit code of a process by calling the **GetExitCodeProcess** function. Specify the handle to the process, which one can obtain by calling the **CreateProcess** or **OpenProcess** function; the function returns the exit code. If the process is still running, the function returns the STILL_ACTIVE termination status.

There are also other, less common, ways of terminating a process:

- Use interprocess synchronization to instruct the process to terminate itself.
- If the process has a message queue, send a WM_CLOSE message to the main window of the process. An application might not close if it does not receive this message and might display a message box.
- Use the **TerminateProcess** function, which does not notify any attached DLLs that the process is terminating. This method should be used as a last resort.

Note a process immediately terminates if a related secondary thread generates an unhandled exception. This is a change in behavior from Windows CE version 2.10 or earlier.

CREATING A THREAD

To create a thread, call the **CreateThread** function. The following code example shows the **CreateThread** function prototype.

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
DWORD dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId );
```

Because Windows CE does not support the *lpThreadAttributes* and *dwStackSize* parameters, these parameters must be set to NULL or 0. The following table describes the remaining **CreateThread** parameters.

Parameter	**Description**
lpStartAddress	Points to the start of the thread routine
lpParameter	Specifies an application-defined value that is passed to the thread routine
dwCreationFlags	Set to 0 or CREATE_SUSPENDED
lpThreadId	Points to a **DWORD** that receives the new thread's identifier

If **CreateThread** is successful, it returns the handle to the new thread and the thread identifier. One can also retrieve the thread identifier by calling the **GetCurrentThreadId** function from within the thread. In Windows CE, the value returned in G**etCurrentThreadId** is the actual thread handle. a handle to the thread can be retrieved by calling the **GetCurrentThread** function. This function returns a pseudo-handle to the thread that is valid only while in the thread. If CREATE_SUSPENDED is specified in the *dwCreationFlags* parameter, the thread is created in a suspended state and must be resumed with a call to the **ResumeThread** function.

TERMINATING A THREAD

To terminate a thread, call the **`ExitThread`** function**.** The following code example shows the **ExitThread** function prototype.

`VOID ExitThread( DWORD` *`dwExitCode`* **`);`**

Parameter	**Description**
DwExitCode	Specifies the exit code for the calling thread. Use the **GetExitCodeThread** function to retrieve a thread's exit code.

Calling **ExitThread** for the primary thread causes the application to terminate.

THREAD SCHEDULING

When the operating system creates a new process, it also creates at least one thread and assigns that thread a priority level. Processes are not assigned a priority class, so preemption is based solely on priority level.

Threads with a higher priority run first. Threads with the same priority run in a round-robin fashion—when a thread has stopped running, all other threads of the same priority run before the original thread can continue. Threads at a lower priority do not run until all threads with a higher priority have either finished or have been blocked. If one thread is running and a thread of higher priority is unblocked, the lower-priority thread is immediately suspended and the higher-priority thread is scheduled.

Threads run for a specific slice of time—called a quantum—which has a default value of 25 milliseconds. An OEM can specify a different quantum. If, after the quantum has elapsed, the thread has not relinquished its time slice and is not time-critical, it is suspended and another thread is scheduled to run. Threads having a priority level of THREAD_PRIORITY_TIME_CRITICAL cannot be preempted except by an interrupt service routine (ISR).

Threads at level 0 do not share time slices—instead, they run until they finish or yield due to a blocking function, such as **WaitForSingleObject**. Threads at a lower priority do not run until all threads with a higher priority have finished. All threads are created with a default priority of level 3, THREAD_PRIORITY_NORMAL.

The highest levels of priority: 0 and 1, are used for real-time processing and device drivers. Levels 2, 3 (default), and 4 are used for kernel threads and normal applications. Levels 5, 6, and 7 are used for applications that can always be preempted by other applications.

The following table shows these priority level values:

Priority	**Value**
0 (highest)	THREAD_PRIORITY_TIME_CRITICAL
1	THREAD_PRIORITY_HIGHEST
2	THREAD_PRIORITY_ABOVE_NORMAL
3 (default)	THREAD_PRIORITY_NORMAL
4	THREAD_PRIORITY_BELOW_NORMAL
5	THREAD_PRIORITY_LOWEST
6	THREAD_PRIORITY_ABOVE_IDLE
7 (lowest)	THREAD_PRIORITY_IDLE

For the most part, thread priorities are fixed and do not change. However, there is one exception, called *priority inversion*. If a low-priority thread is using a resource that a high-priority thread is waiting to use, the kernel temporarily boosts the priority of the low-priority thread until it releases the resource that is required by the higher-priority thread.

PROCESSES AND THREADS SYNCHRONIZATION

To coordinate multiple threads, wait functions and synchronization objects (passed to a wait function) can be used.

The wait functions to use include:

- **WaitForSingleObject**
- **WaitForMultipleObjects**
- **MsgWaitForMultipleObjects**

A wait function does not return until its specified criteria are met. The type of wait function determines the set of criteria used. When a wait function is called, it checks if the wait criteria have been met. If not, the calling thread enters an efficient wait state, which consumes very little CPU time.

The types of synchronization objects contain:

- Critical-Section
- Event
- Mutex
- Thread

The state of a synchronization object is either signaled, which can allow the wait function to return, or non-signaled, which can prevent the function from returning.

WAIT FUNCTIONS

Windows CE supports two types of wait functions: single-object and multiple-object. The single-object function is **WaitForSingleObject**. The multiple-object functions are **WaitForMultipleObjects** and **MsgWaitForMultipleObjects**.

WaitForSingleObject requires a handle to each synchronization object. This function returns when one of the following occurs:

- The specified synchronization object is set to the signaled state.
- The state of a synchronization object is either signaled, which can allow the wait function to return, or non-signaled, which can prevent the function from returning.
- The time-out interval elapses.

To specify that the wait does not time out, set the time-out interval to INFINITE.

WaitForMultipleObjects and **MsgWaitForMultipleObjects** enable the calling thread to specify an array that contains one or more synchronization object handles. These functions return when one of the following occurs:

The state of any one of the specified objects is set to be signaled, or the states of all objects are set to be signaled.   In the function call, the developer control whether one or all of the states are required to trigger a return. The time-out interval elapses.

To specify that the wait does not time out, set the time-out interval to INFINITE.

The following code example shows two objects as parameters in the function call to **WaitForMultipleObjects**, which does not return until one of the objects is set to be signaled. The **CreateEvent** function creates two event objects.

```
int index;
DWORD dwEvent;
TCHAR szError[100];
HANDLE hEvents[2];

for (index = 0; index < 2; ++index)
{
  if (!(hEvents[index] = CreateEvent (
                            NULL,       // No security attributes
                            FALSE,      // Autoreset event object
                            FALSE,      // Initial state is nonsignaled.
                            NULL)))     // Unnamed object
  {
    swprintf (szError, TEXT("CreateEvent error: %d\n"), GetLastError());
  }
}

dwEvent = WaitForMultipleObjects (
                            2,          // Number of objects in array
                            hEvents,    // Array of objects
                            FALSE,      // Wait for any.
                            INFINITE);  // Indefinite wait
switch (dwEvent)
{
  case WAIT_OBJECT_0 + 0:
    break;

  case WAIT_OBJECT_0 + 1:
    break;

  default:
    swprintf (szError, TEXT("Wait error: %d\n"), GetLastError());
}
```

CRITICAL SECTION SYNCHRONIZATION OBJECTS

When multiple threads have shared access to the same data, the threads can interfere with one another. a critical section object protects a section of code from being accessed by more than one thread. A critical section is limited, however, to only one process or DLL and cannot be shared with other processes.

Critical sections work by having a thread call the **EnterCriticalSection** function to indicate that it has entered a critical section of code. If another thread calls **EnterCriticalSection** and references the same critical section object, it is blocked until the first thread calls the **LeaveCriticalSection** function. A critical section can protect more than one section of code as long as each section of code is protected by the same critical section object.

To use a critical section, a **CRITICAL_SECTION** structure must be declared. Because other critical section functions require a pointer to this structure, be sure to allocate it within the scope of all functions that are using the critical section. Then, create a handle to the critical section object by calling the **InitializeCriticalSection** function.

To request ownership of a critical section, call **EnterCriticalSection**; to release ownership, call **LeaveCriticalSection**. When a critical section is no longer used, call the **DeleteCriticalSection** function to release the system resources that were allocated when the critical section is intialized.

The following code example shows the prototype for the critical section functions. Notice that they all require a pointer to the **CRITICAL_SECTION** structure.

```
void InitializeCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
void EnterCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
void LeaveCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
void DeleteCriticalSection (LPCRITICAL_SECTION lpCriticalSection);
```

The following code example shows how a thread initializes, enters, and leaves a critical section. This example uses the **try-finally** structured exception-handling syntax to ensure that the thread calls **LeaveCriticalSection** to release the critical section object.

```
void CriticalSectionExample (void)
{
  CRITICAL_SECTION csMyCriticalSection;

  InitializeCriticalSection (&csMyCriticalSection);

  __try
  {
    EnterCriticalSection (&csMyCriticalSection);

    // Code to access the shared resource goes here.
  }
  __finally
  {
    // Release ownership of the critical section
    LeaveCriticalSection (&csMyCriticalSection);
  }
} // End of CriticalSectionExample code
```

EVENT SYNCHRONIZATION OBJECTS

An event synchronization object allows one thread to notify another thread that an event has occurred. A thread uses the **CreateEvent** function to create an event object. The creating thread specifies the initial state of the object and whether it is a manual-reset or auto-reset event object. The creating thread can also specify a name for the event object. Threads in other processes can open a handle to an existing event object by specifying its name in a call to **CreateEvent**.

Windows CE uses event objects to tell a thread when to perform its task or to indicate that a particular event has occurred. For example, a thread that writes to a buffer sets the event object to the signaled state when it has finished writing. Setting an event object to notify the thread that its task is finished allows the thread to start performing other tasks immediately.

The following code example shows how an application uses event objects to prevent several threads from reading from a shared memory buffer while a master thread is writing to that buffer. The master thread uses **CreateEvent** to create a manual-reset event object. It resets the event object to its non-signaled state when it is writing to the buffer, and then sets the object to its signaled state when it has finished. The master thread then creates several reader threads and an auto-reset event object for each thread. Each reader thread sets its event object to its signaled state when it is not reading from the buffer.

```
#define NUMTHREADS 4

HANDLE hGlobalWriteEvent;

void CreateEventsAndThreads ()
{
  int index;
  DWORD dwIDThread;
  HANDLE hThread,
         hReadEvents[NUMTHREADS];

  hGlobalWriteEvent = CreateEvent (
                            NULL,         //No security attributes
                            TRUE,         //Manual-reset event
                            TRUE,         //Initial state is signaled.
                            TEXT("WriteEvent"));
                                          // Object name
  if (!hGlobalWriteEvent)
  {
    // CreateEvent failed. Insert code here for error handling.
    // ...
  }

  for (index = 0; index < NUMTHREADS; ++index)
  {
    hReadEvents[index] = CreateEvent (
                            NULL,         //No security attributes
                            FALSE,        //Autoreset event
                            TRUE,         //Initial state is signaled.
                            NULL);        //Object not named

    if (!hReadEvents[index])
    {
      // CreateEvent failed. Insert code here for error handling.
      // ...
    }

    hThread = CreateThread (
                NULL,                   // Thread security attributes
                0,                      // Initial thread stack size
                (LPTHREAD_START_ROUTINE) ThreadFunction,
                                        // Pointer to thread function
                &hReadEvents[index],    // Argument for new thread
                0,                      // Creation flags
                &dwIDThread);           // Returned thread identifier

    if (!hThread)
    {
      // CreateThread failed. Insert code here for error handling.
      // ...
    }
  }
}
```

The following code example shows how to use the **ResetEvent** function to reset the state of *hGlobalWriteEvent*, an application-defined global variable, to its non-signaled state before the master thread writes to the shared buffer. This operation blocks the reader threads from starting a read operation. The master thread then uses **WaitForMultipleObjects** to wait for all reader threads to finish any current read operations. When **WaitForMultipleObjects** returns, the master thread can safely write to the buffer. After it has finished writing, the master thread sets *hGlobalWriteEvent* and all the reader-thread events to signaled. The reader threads can then resume read operations.

```
VOID WriteToBuffer (HANDLE hReadEvents[NUMTHREADS])
{
  int index;
  DWORD dwResult;
  TCHAR szError[200];

  if (!ResetEvent (hGlobalWriteEvent))
  {
    // ResetEvent failed. Insert code here for error handling.
    // ...
  }

  dwResult = WaitForMultipleObjects (
                        NUMTHREADS,     //Number of handles in array
                        hReadEvents,    //Array of read-event handles
                        TRUE,           //Wait until all are signaled.
                        INFINITE);      //Indefinite wait

  switch (dwResult)
  {
    case WAIT_OBJECT_0:
      // Write to the shared buffer.
      break;

    default:
      // Error occurred.
      wsprintf (szError, TEXT("Wait error: %d\n"), GetLastError ());
  }

  if (!SetEvent (hGlobalWriteEvent))
  {
    // SetEvent failed. Insert code here for error handling.
    // ...
  }

  for (index = 0; index < NUMTHREADS; index++)
  {
    if (!SetEvent (hReadEvents[index]))
    {
      // SetEvent failed. Insert code here for error handling.
      // ...
    }
  }
}
```

MUTEX SYNCHRONIZATION OBJECTS

A mutex object is a synchronization object whose state is set to signaled when it is not owned by any thread and non-signaled when it is owned. Its name comes from its usefulness in coordinating mutually exclusive access to a shared resource. Only one thread at a time can own a mutex object. For example, to prevent two threads from writing to shared memory at the same time, each thread waits for ownership of a mutex object before running the code that accesses the memory. After writing to the shared memory, the thread releases the mutex object.

A thread uses the **CreateMutex** function to create a mutex object. The creating thread can request immediate ownership of the mutex object and can also specify a name for the mutex object. Threads in other processes can open a handle to an existing mutex object by specifying its name in a call to **CreateMutex**.

Any thread with a handle to a mutex object can use one of the wait functions to request ownership of the mutex object. If the mutex object is owned by another thread, the wait function blocks the requesting thread until the owning thread releases the mutex object with the **ReleaseMutex** function. The return value of the wait function indicates the reason that the function returned.

Once a thread owns a mutex, it can specify that mutex in repeated calls to one of the wait functions without blocking it, thereby preventing a thread from blocking itself while waiting for a mutex that it already owns. To release ownership, the thread must call **ReleaseMutex** once for each time that the mutex satisfies the conditions of a wait function.

If a thread terminates without releasing ownership of a mutex object, the mutex object is considered abandoned. A waiting thread can acquire ownership of an abandoned mutex object, but the wait function return value indicates that the mutex object is abandoned. Typically, an abandoned mutex object indicates that an error has occurred and that any shared resource being protected by the mutex object is in an undefined state. If the thread proceeds as though the mutex object had not been abandoned, the object's abandoned flag is cleared when the thread releases ownership. With the flag cleared, typical behavior is restored if a handle to the mutex object is subsequently specified in a wait function.

The following code example shows how a process uses **CreateMutex** to create a named mutex object.

```
HANDLE hMutex;

if (!(hMutex = CreateMutex (NULL,         // No security attributes
                            FALSE,        // Mutex not owned
                            TEXT("MutexToProtectDatabase"))))
                                          // Mutex-object name
{
  // CreateMutex failed. Insert code here for error handling.
  // ...
}
```

Before a thread of this process can write to the database, it must have ownership of the mutex. It first requests ownership of the mutex. If it gets ownership, the thread writes to the database, and then releases ownership.

The following code example shows how to open a handle to an existing mutex object. It also uses the **try-finally** structured exception handling syntax to ensure that the thread properly releases the mutex object. To prevent the mutex object from being abandoned inadvertently, if the **try** block includes a call to the **TerminateThread** function, the **finally** block of code will not run.

```
BOOL WriteToDatabase (HANDLE hMutex)
{
  DWORD dwResult;
```

```
    dwResult = WaitForSingleObject (
                            hMutex,   // Handle of mutex
                            5000L);   // Five-second time-out interval
    switch (dwResult)
    {
      case WAIT_OBJECT_0:
        __try
        {
          // Insert code here to write to the database.
          // ...
        }
        __finally
        {
          if (!ReleaseMutex (hMutex))
          {
            // Insert code here for error handling.
            // ...
          }
        }
        break;

      case WAIT_TIMEOUT:
        // Cannot get mutex ownership due to time-out
        return FALSE;

      case WAIT_ABANDONED:
        // Got ownership of the abandoned mutex object
        return FALSE;
    }
    return TRUE;
  }
```

USING THREAD OBJECTS

A thread synchronization object is created when a new thread is created by calling either the **CreateProcess** or CreateThread function. Its state is set to non-signaled while the thread is running, and set to signaled when the thread terminates.

MANAGING MEMORY

Windows CE optimizes memory management for game development by using virtual memory and memory-mapped files.

The Hitachi SH-4 CPU for the Dreamcast system uses 32-bit memory addressing, which allows the SH-4 to address up to 4 GB of memory. The Windows CE kernel partitions this space into 2 GB of physical addresses and 2 GB of virtual addresses. The following illustration depicts the Windows CE memory architecture.

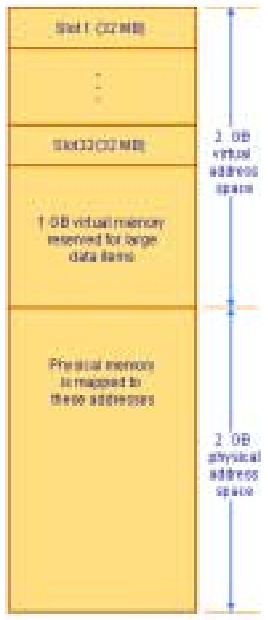
Slot 1 (32 MB)
Slot32 (32 MB)
1 GB virtual memory reserved for large data items
Physical memory is mapped to these addresses
2 GB virtual address space
2 GB physical address space

Figure 5: Windows CE for Dreamcast Memory Architecture

The 2 GB of physical address space corresponds directly to the underlying hardware memory and is available only to the kernel. The 2 GB of virtual memory is accessible by the processes running on Windows CE. A process perceives the virtual address space as one contiguous, unshared block of memory. The kernel maps all virtual memory addresses into physical addresses in hardware memory.

Of the 2 GB of virtual address space, 1 GB is made available for individual use by the processes. The 1 GB is subdivided into 32 slots, of 32 MB each. The slot memory of a process contains its code, data, stack, and heap. While the slot architecture limits the number of processes to 32, the total number of threads within a process is limited only by available memory.

The kernel isolates each process by assigning it to a unique slot and protecting that slot against access by other processes. The kernel prevents a process from accessing memory outside its own slot by generating an exception. Applications can check for and handle such exceptions with **try-except** statements.

The remaining 1 GB of virtual memory is shared among all processes and is used for large data items, such as reserved physical memory blocks and memory-mapped files. These files are useful for interprocess communication because more than one process can map the same file and share its contents. Memory mapping allows for fast data transfer between cooperating processes and between a driver and an application. Page protections and demand paging are supported.

VIRTUAL MEMORY

The addresses that a Dreamcast game uses to refer to its code and data are *virtual addresses*, not physical addresses. A virtual address is an artificial address created at startup by Windows CE that exists independently from the physical memory in the game console. The kernel maintains the relationship between the virtual address space of an application and the physical address space on the hardware.

The virtual addresses referenced by an application's code are translated into physical addresses by the SH-4 CPU. This translation is performed by the SH-4's *translation look-aside buffer (TLB)*, an internal table that maps addresses transparently. Each entry in the TLB maps one page of virtual address space onto a page of physical memory.

No correspondence is needed between an address location in virtual and physical memory. Memory pages that are contiguous in virtual space can be mapped to non-contiguous pages of physical memory. The mapping is invisible to the application, since applications use virtual addresses exclusively.

The following illustration shows how the kernel maps virtual memory to physical memory.

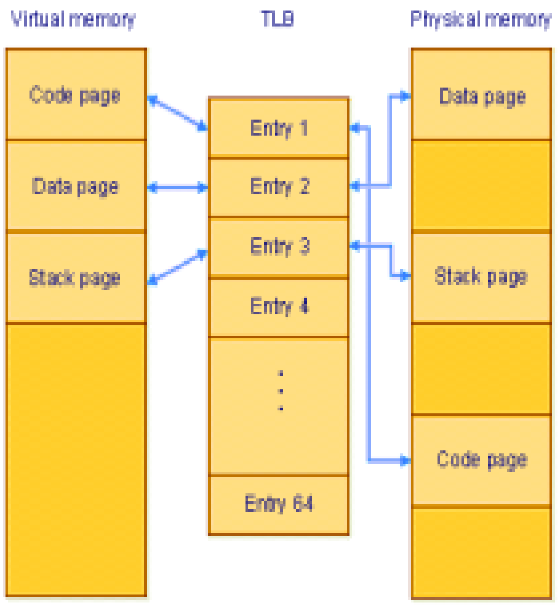

Figure 6: Windows CE for Dreamcast Memory Mapping

In Windows CE for Dreamcast, the page size is 4 KB. For certain kinds of data, a page size of either 64 KB or 1 MB may be used. Because each TLB entry contains information about the size of the page to which it refers, references to pages of different sizes may coexist in the same TLB. An application has no control over the page size used and therefore cannot change the page size.

Each section of an executable file or dynamic-link library (DLL) is aligned on a page boundary when it is loaded. This results in an average of half a page of unused memory. For example, to allocate 1 KB of memory, an entire 4 KB page must be dedicated. The 3 KB of unused memory is not available to any other process. However, if the same application requests an additional 1-3 KB of memory, that additional memory is allocated from the unused portion of that same 4 KB page.

With virtual memory, the virtual address space can be larger than the physical address space. Windows CE keeps track of which pages in virtual memory are present in physical memory and which pages are not.

When an application references a page that is not present in physical memory, a *page fault* occurs. A page fault is an interrupt that is intercepted by Windows CE, which then loads the desired page automatically and transparently. This process allows an application to be written as if the hardware on which it runs contained more physical memory than it does.

An additional benefit of virtual memory is protection from system crashes. Windows CE is aware of which pages belong to which processes, and uses the built-in SH-4 CPU protection features. These features prevent a process from accessing pages that do not belong to it.

As the SH-4 processor's TLB contains 64 entries, a maximum of 64 pages of memory can be mapped at one time. When a game references virtual memory that is not mapped to physical memory, a page fault interrupt occurs. This interrupt is serviced by the kernel, which then loads a new entry into the TLB. Before writing the new entry, the kernel first determines which of the old entries to discard. The game then resumes. If a discarded entry is referenced at a later time, the process repeats, and the page is loaded again.

While Windows CE for Dreamcast does support virtual memory, little use is made of demand paging. Under Windows CE for Dreamcast, executable files are always loaded completely and locked in memory. Their pages are never discarded. Furthermore, Dreamcast includes no write-able storage media, making it impossible to swap out data pages, so these pages are never discarded either. That is the reason why game playing will not be subject to unpredictable pauses while the operating system loads some previously memory page from the disc.

VIRTUAL MEMORY ISSUES

Only data is subject to paging. When a process is started or **LoadLibrary** is called, all of the appropriate code is loaded into memory immediately—it is never paged out until the process is terminated or the library is removed using **FreeLibrary**.

To minimize the number of page fault interrupts, pay attention to the distribution of the game's memory references. Whenever possible, the data references within a section of code should all be made to the same set of memory pages.

The scarcity of TLB entries can be offset through the use of pre-allocated blocks of physical memory. These blocks can use page sizes of either 64 KB or 1 MB, instead of the normal 4 KB. Using a physical memory block increases the amount of memory that can be addressed without causing a page fault.

Since the Dreamcast CPU's translation look-aside buffer (TLB) can map only virtual memory pages to physical memory at any one time, when a game references a virtual memory address which is outside of these 68 pages, a page fault occurs, forcing the operating system to suspend the game's execution and map the virtual page containing the offending address to physical memory. In order to do this; the operating system must first make room in the TLB by unmapping a previously mapped page, creating the potential for future page faults.

These 64 pages may be thought of as a form of cache, and a page fault may be thought of as a cache miss. As with more conventional caches, excessive cache misses can degrade overall performance. Programmers must strive to localize both code and data references as much as possible.

MEMORY-MAPPED FILE

A *memory-mapped file* is a file that is mapped to a region of virtual address space. A space is not necessarily present in physical memory. Using a memory-mapped file is a convenient way to read from files that are too large for physical memory.

The physical memory associated with a memory-mapped file is mapped into the 1 GB of virtual address space that is shared by all processes, not into the 32 MB slot of a specific process.

The pages of a memory-mapped file are loaded as an application references them. If sufficient physical memory is not available, previously loaded pages are discarded to make room. A later reference to a discarded page causes that page to be loaded again.

To use a file as a memory-mapped file, open it with **CreateFileForMapping** and then call **CreateFileMapping** to get a handle to a file-mapping object. Pass this handle to **MapViewOfFile** to map all or part of the contents of the file to virtual memory. **MapViewOfFile** returns a pointer to the virtual memory region, from which the contents of the file can be read.

HANDLING SYSTEM OBJECTS

A system object is a data structure that represents a system resource, such as a file, graphic image, or thread. The system object categories include:

- User, which supports window management.
- Graphics Device Interface (GDI), which supports graphics.
- Kernel, which supports memory management, process execution, and interprocess communications.

An application cannot directly access object data or the system resource that the object represents. Instead, an application must obtain the object's handle to examine or to modify that system resource. Each object handle has an entry in an internally maintained table. These entries contain the private addresses of the resources and the means to identify the resource type.

OBJECT MANAGER

A system object consists of a header and object-specific attributes. All system objects have the same structure, which allows a single object manager to maintain all objects. The object header includes items such as the object name so that other processes can reference that object by name. The object header also includes a security descriptor so that the object manager can control which processes access the system resource.

The object manager performs the following tasks:

- Create objects
- Verify that a process has the right to use the object
- Create object handles and return them to the caller
- Maintain resource quotas
- Close handles to objects
- Object Interface

Windows CE provides functions that perform the following operations:

- Create an object
- Get an object handle
- Get information about the object
- Set information about the object
- Close the object handle
- Destroy the object

When a process terminates, the system automatically closes handles and destroys objects created by the process. However, when a thread terminates, the system usually does not close handles or destroy objects. The only exceptions are window objects, which are destroyed when the creating thread terminates.

OBJECT HANDLE LIMITATIONS

Some objects support only one handle at a time. The system provides the handle when an application creates the object and invalidates the handle when the application destroys the object. Other objects support multiple handles to a single object. The operating system automatically removes the object from memory after the last handle to the object is closed.

The total number of open handles in the system is limited by the amount of memory available. Some object types support a limited number of handles for each process, while others support a limited number of handles in the system.

MANAGING USER OBJECTS

User objects support window management. The ratio of objects to handles is 1:1, that is, there can be only one handle per object. However, the ratio of processes to handles is M:1, and there is no per-process limit on user object handles.

A user object handle is available to every process, provided that the process has security access to the user object.

After the window object has been created, the application uses the window handle to display or change the window. The handle remains valid until the window object is destroyed.

The **CreateWindow** and **CreateWindowEx** creator functions either create the window object and an object handle or return the existing window object handle. The **DestroyWindow** destroyer function removes the object from memory, which invalidates the object handle.

The following illustration shows how an application creates a window object. (Currently, the only user object type is Window.) The **CreateWindow** function creates the window object and returns an object handle.

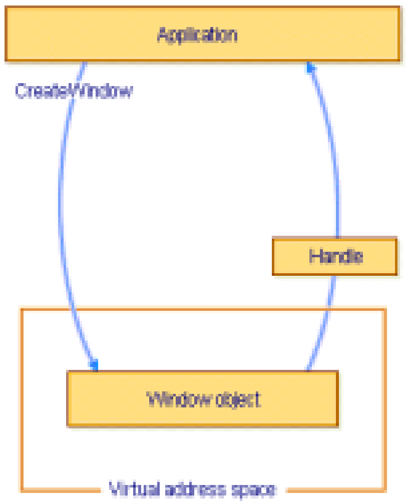

Figure 7: Window Creation in Windows CE for Dreamcast

The following illustration shows how the application destroys the window object. The **DestroyWindow** function removes the window object from memory, which invalidates the window handle and makes the handle inaccessible.

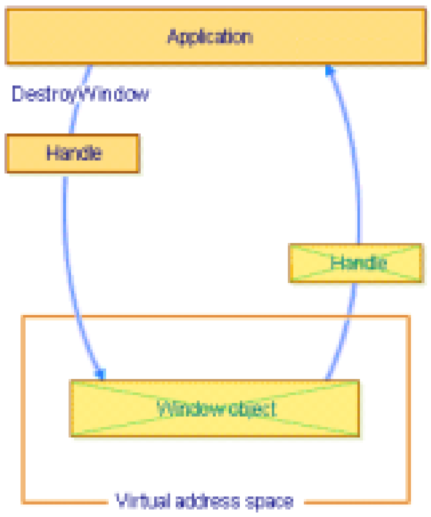

Figure 8: Window Destruction in Windows CE for Dreamcast

MANAGING GDI OBJECTS

GDI objects support graphics. The ratio of objects to handles is 1:1, only one handle per object. GDI object handles are global and are accessible to every process, provided that the process has security access to the GDI object.

The following table lists the GDI object types, along with the creator and destroyer functions of each object. The creator functions create the object and return a new object handle. The destroyer functions remove the object from memory, which invalidates the object handle.

GDI object	**Creator function**	**Destroyer function**
Bitmap	**LoadBitmap**, **CreateBitmap**, **CreateCompatibleBitmap, CreateDIBSection**	**DeleteObject**
Brush	**CreatePatternBrush**, **CreateDIBPatternBrushPt**, **CreateSolidBrush**	**DeleteObject**
Device Context	**CreatCompatibleDC, CreateDC**	**DeleteDC**
Font	**CreateFontIndirect**	**DeleteObject**
Memory DC	**CreateCompatibleDC**	**DeleteDC**
Pallette	**CreatePallette**	**DeleteObject**
Pen	**CreatePen**,	**DeleteObject**

	CreatePenIndirect	
Region	**CreateRectRgnIndirect**, **CreateRectRgn**	**DeleteObject**

MANAGING KERNEL OBJECTS

Kernel objects support memory management, process execution, and interprocess communications. They are process-specific. A process must either create the kernel object or open an existing kernel object to obtain a kernel object handle. The per-process limit on kernel handles is 2^30. Most kernel objects, those that are used for process and thread communications, support multiple handles to a single object.

Any process can create a new handle to an existing kernel object, for example, one created by another process, provided that the process knows the name of the object and has security access to it. Kernel object handles include access rights that indicate the actions that can be allowed or denied to a process. An application specifies access rights when it creates an object or obtains an existing object handle. Each type of kernel object supports its own set of access rights. For example, event handles and file handles can have set access, or wait access, or both.

The following illustration shows how an application creates an event object. The **CreateEvent** function creates the event object and returns an object handle.

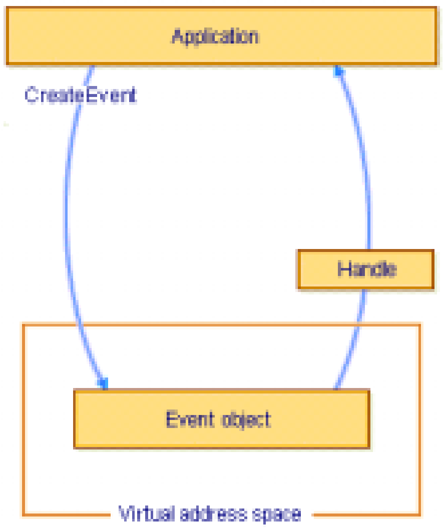

Figure 9: Event Object Creation in Windows CE for Dreamcast

After the event object has been created, the application can use the event handle to set or wait for the event. The handle remains valid until the application closes the handle or terminates.

An object remains in memory as long as at least one object handle exists. The following illustration shows how an application uses **CloseHandle** to close event object handles. When there are no event handles, the system removes the object from memory.

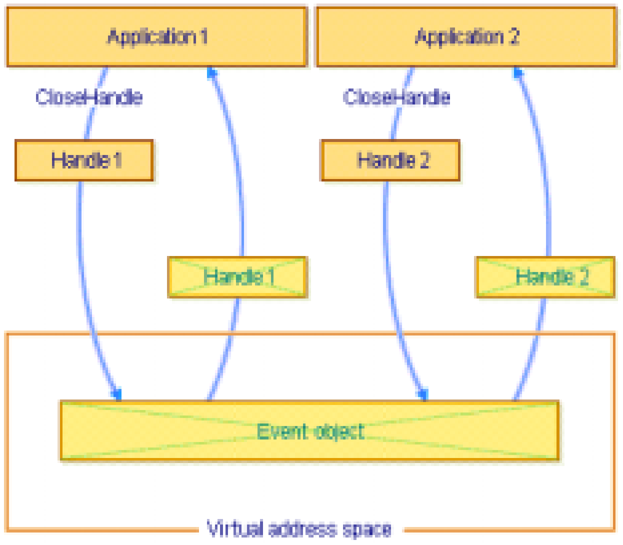

Figure 10: Event Object Destruction in Windows CE for Dreamcast

The system manages file objects somewhat differently from other kernel objects. File objects contain a file pointer to the next byte to be read. Whenever an application creates a new file handle, the system creates a new file object. The following illustration shows how more than one file object can refer to a single file.

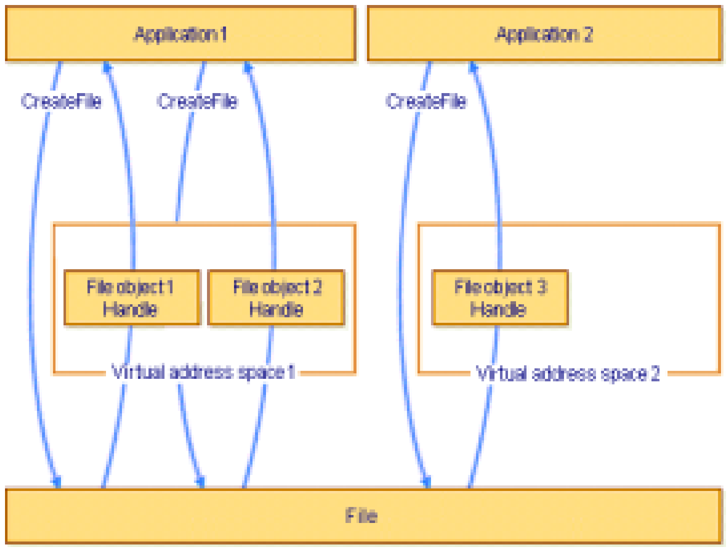

Figure 11: Multiple File Objects Referring to a Single File in Windows CE for Dreamcast

The following table lists the kernel object types, along with the creator and destroyer functions of each object. The creator functions either create the object and an object handle or create a new handle to an existing object. The destroyer functions close the object handle. When an application closes the last handle to a kernel object, the system removes the object from memory.

Note: CreateFile is used when a mapped file is not needed. **CreateFileForMapping** is used for mapped files, along with **CreateFileMapping**.

Kernel object	**Creator function**	**Destroyer function**
Communications device	**CreateFile**	**CloseHandle**
Event	**CreateEvent**	**CloseHandle**
File	**CreateFile**	**CloseHandle,**
File mapping	**CreateFileMapping**	**CloseHandle**
Find file	**FindFirstFile**	**FindClose**
Heap	**HeapCreate**	**HeapDestroy**
Module	**LoadLibrary**, **GetModuleHandle**	**FreeLibrary**
Mutex	**CreateMutex**	**CloseHandle**
Process	**CreateProcess**, **OpenProcess**, **GetCurrentProcess**	**CloseHandle** **TerminateProcess**
Socket	**socket**, **accept**	**closesocket**
Thread	**CreateThread**, **GetCurrentThread**	**CloseHandle**, **TerminateThread**

TIMERS

A *timer* is a system resource one can set to notify an application at regular intervals. Windows CE for Dreamcast supports two broad categories of timers:

Low-resolution timers are used for events that do not require a high degree of start-time accuracy, such as accessing hardware resources or performing application maintenance.

High-resolution, or multimedia, timers are used to coordinate events that require a high degree of synchronization, such as matching sound to graphics.

Timer Terminology

Resolution: The accuracy, or tolerance factor, of the timer. For example, a timer resolution of 10 milliseconds results in the timer being accurate to within plus or minus 10 milliseconds.

Interval: For a low-resolution timer, it is the time-out value. For a one-shot high-resolution timer, it is the amount of time that passes before the timer starts. In the case of a periodic high-resolution timer, it is the amount of time before the timer starts, plus the amount of time between each notification to the application, until a call to **timeKillEvent** is made and the timer is destroyed.

<u>Start lag</u>: The lag time between when the timer is set to start and when it actually starts. This number cannot be determined in advance and depends on a number of factors, including system resources usage.

<u>Greatest Common Factor (GCF)</u>: The lowest common denominator of all timer intervals currently running on the timer thread. The system automatically determines and adjusts the GCF, which directly affects the amount of system resources dedicated to monitoring the starting times of all timers currently running on the thread.

<u>High-resolution Timer</u>: a timer that requires a high degree of precision (has a very low tolerance level), but consumes a higher amount of system resources. For example, setting the value to zero would cause the timer to start at the most precise time requested, within the resolution's capabilities. High-resolution timers are recommended for tasks that require a high degree of timing precision, such as when coordinating multimedia events. Use **timeSetEvent** to create and set a high-resolution timer.

<u>Low-resolution Timer</u>: a timer that requires a low or moderate degree of precision (has a greater tolerance level), but consumes a lesser amount of system resources. For example, setting the value to 50 causes the timer to start within 50 milliseconds of the time requested. Low-resolution timers are Windows-based and are recommended for tasks that do not require a high degree of timing precision, such as requesting hardware resources. Use **SetTimer** to create and set a low-resolution timer.

<u>Example of Creating Timers Using SetTimer</u>

The following code example shows how to create two timers using **SetTimer**. The first timer is set for every 10 seconds, the second for every 5 minutes.

```
// Set two timers.
 SetTimer (
          hwnd,               // Handle of main window
          IDT_TIMER1,         // Timer identifier
          10000,              // 10-second interval
          (TIMERPROC) NULL);  // No timer callback

 SetTimer (
          hwnd,               // Handle of main window
          IDT_TIMER2,         // Timer identifier
          300000,             // 5-minute interval
          (TIMERPROC) NULL);  // No timer callback
```

<u>Example of Determining a Timer's Best Resolution</u>

The following code example shows how to call **timeGetDevCaps** to determine the minimum and maximum timer resolutions of high-resolution timers.

```
TIMECAPS timecaps;

if (timeGetDevCaps (&timecaps, sizeof (TIMECAPS)) != TIMERR_NOERROR)
{
  // Error occurred; application cannot continue. Insert code here for
  // error handling
  // ...
}
```

Example of Processing WM_TIMER Messages

The following code example shows how to process the WM_TIMER messages generated by the two example timers. A WM_TIMER case statement can be added to the window procedure specified by the *hWnd* parameter of **SetTimer**.

```
case WM_TIMER:

  switch (wParam)
  {
    case IDT_TIMER1:
      // Process the 10-second timer.
      return 0;

    case IDT_TIMER2:
      // Process the 5-minute timer.
      return 0;
  }
```

Example of Using a Callback Function to Process WM_TIMER Messages

An application can also create a timer whose WM_TIMER messages are processed by an application-defined callback function, not by the main window procedure. The following code example creates a third timer and uses the callback function **MyTimerProc** to process the timer's WM_TIMER messages.

```
// Set the timer.
 SetTimer (hwnd,                       // Handle of main window
          IDT_TIMER3,                 // Timer identifier
          5000,                       // 5-second interval
          (TIMERPROC) MyTimerProc);  // Timer callback
```

The calling convention for **MyTimerProc** must be based on the **TimerProc** callback function.

Example of Monitoring the Message Queue

The following code example shows a message loop for an application that has created a timer without specifying a window handle. The loop monitors the message queue for WM_TIMER messages and dispatches them to a window procedure to process messages.

```
MSG msg;                  // Message structure
HWND hwndTimer;           // Handle of the window for timer
                          // messages
while (GetMessage (&msg,  // Message structure
                   NULL,  // Handle of window receiving the message
                   0,     // Lowest message to examine
                   0))    // Highest message to examine
{

  // Post WM_TIMER messages to the window (hwndTimer) procedure.
  if (msg.message == WM_TIMER)
    msg.hwnd = hwndTimer;

  // Translates virtual-key messages into character messages
  TranslateMessage (&msg);
```

```
    // Dispatches message to the window (hwndTimer) procedure
    DispatchMessage (&msg);
  }
```

Example of Destroying Timers

The following code example shows how to destroy the timers identified by the constants IDT_TIMER1, IDT_TIMER2, and IDT_TIMER3.

```
// Destroy the timers.

KillTimer (hwnd, IDT_TIMER1);

KillTimer (hwnd, IDT_TIMER2);

KillTimer (hwnd, IDT_TIMER3);
```

FILE SYSTEMS AND FILES

As with other Win32 operating systems, Windows CE for Dreamcast employs handle-based file access. **CreateFile** returns a handle that references a created or opened file. The read, write, and information functions all use that handle to determine which file to act on. The read and write functions use a *file pointer* to determine where in the file they read and write. Windows CE for Dreamcast does not use the concept of the *current directory*. Instead, all references to an object are given in the full path name.

Windows CE for Dreamcast supports the following file systems:

- CD-ROM
- Windows
- PC
- CD-ROM File System

The CD-ROM file system is a 1 GB double-density partition and is stored on the Dreamcast CD (the game application CD). The application views it as the \CD-ROM\ directory.

The Windows CE file system does not use drive letters, while the Windows 9x or Windows NT file systems do. When using the CD-ROM file system, the full path name must always be included.

The CD-ROM file system has higher transfer rates than a standard CD-ROM. Because Dreamcast CDs rotate at a constant angular velocity, the transfer rate varies with the location on the disc. Toward the center of the disc, drive performance is equivalent to a 6X CD-ROM drive. Toward the outer edge of the disc, performance is equivalent to a 12X CD-ROM drive.

To access a directory, a directory handle must be obtained, and it is a unique identifier for each directory. Pass the directory handle to **GetFileInformationByHandle** and **GetFileSize**.

The following code example shows how to obtain a handle to a directory by using **CreateFile**.

```
hDir = CreateFile (szDirName,
                   GENERIC_READ,
                   FILE_SHARE_READ,
                   NULL,
                   OPEN_EXISTING,
                   FILE_FLAG_BACKUP_SEMANTICS,
                   NULL);
```

Windows File System

The Windows file system is stored in the OS image on the Dreamcast CD and is loaded into memory when a game starts. The application views it as the \Windows\ directory. The file system contains the operating system binary and driver files, such as the GD-ROM and Maple drivers. One can optionally store additional files, such as DirectX DLLs, game executable file, and any data files that need to be resident in memory.

PC File System

The PC file system is stored only on the development computer and is viewed as the \PC\ directory. The PC file system is a convenient location to store files needed during development and debugging, but not need to be placed on the GD-ROM emulator.

File System Functions

The following table lists the file system functions supported under Windows CE for Dreamcast.

Function	**Description**
CloseHandle	Closes an open object handle.
CreateFile	Creates, opens, or truncates a file, directory, communications resource, disk device, or console.
DeviceIoControl	Sends a control code, such as IOCTL_SEGACD_CD_PLAYTRACK, directly to a specified device driver.
FindClose	Closes the specified search handle.
FindFirstFile	Searches a directory for a specified file or subdirectory.
FindNextFile	Continues a file search from a previous call to **FindFirstFile** or **FindNextFile**.
GetFileAttributes	Retrieves attributes of a file or directory.
GetFileInformationByHandle	Retrieves information about a file.
GetFileSize	Retrieves the size, in bytes, of the specified file.

ReadFile	Reads data from a file.
SetFilePointer	Moves the file pointer of an open file.
WriteFile	Writes data to a file.

File Read/Write Example

The following code example shows how to append one file to the end of another. The example uses **CreateFile** to open two files: One.txt for reading and Two.txt for writing. **ReadFile** and **WriteFile** then append the contents of One.txt to the end of Two.txt by reading and writing the 4 KB blocks.

```
void AppendExample (void)
{
  HANDLE hFile, hAppend;
  DWORD dwBytesRead, dwBytesWritten, dwPos;
  char buff[4096];
  TCHAR szMsg[1000];

  // Open the existing file.

  hFile = CreateFile (TEXT("\\CD-ROM\\ONE.TXT"),       // Open One.txt
                      GENERIC_READ,           // Open for reading
                      0,                      // Do not share
                      NULL,                   // No security
                      OPEN_EXISTING,          // Existing file only
                      FILE_ATTRIBUTE_NORMAL,  // Normal file
                      NULL);                  // No attribute template

  if (hFile == INVALID_HANDLE_VALUE)
  {
    // Error-handling code goes HERE.
    wsprintf (szMsg, TEXT("Could not open ONE.TXT"));
    return;
  }

  // Open the existing file, or if the file does not exist,
  // create a new file.

  hAppend = CreateFile (TEXT("\\PC\\TWO.TXT"),       // Open Two.txt
                        GENERIC_WRITE,          // Open for writing
                        0,                      // Do not share
                        NULL,                   // No security
                        OPEN_ALWAYS,            // Open or create
                        FILE_ATTRIBUTE_NORMAL,  // Normal file
                        NULL);                  // No attribute template

  if (hAppend == INVALID_HANDLE_VALUE)
  {
    wsprintf (szMsg, TEXT("Could not open TWO.TXT"));
    CloseHandle (hFile);            // Close the first file.
    return;
  }

  // Append the first file to the end of the second file.

  do
  {
    if (ReadFile (hFile, buff, 4096, &dwBytesRead, NULL))
    {
      dwPos = SetFilePointer (hAppend, 0, NULL, FILE_END);
      WriteFile (hAppend, buff, dwBytesRead,
                 &dwBytesWritten, NULL);
    }
  }
  while (dwBytesRead == 4096);

  // Close both files.

  CloseHandle (hFile);
  CloseHandle (hAppend);
  return;
} // end of AppendExample code
```