Windows CE 6.0 Stream Driver

Harshitha.K.Raj, S.R.Sujatha, Dr.M.Siddappa, P. Satish Kumar

Abstract: Developing device drivers is one of the most difficult tasks to develop or port operating systems. A device driver needs to be described according to the target device and OS. A major design goal in operating system developments is stability and one of the challenges of this stability is device drivers. Device drivers are more likely to crash the operating system for two reasons: on one side they typically run in kernel mode and not as easily be confined as the normal application programs and on the other hand side many drivers are built by independent equipment manufactures. This paper proposes Windows CE 6.0 support system for stream driver development.

Keywords: Win CE6.0, device driver, stream driver.

I. Introduction

Writing device drivers is one of the most difficult tasks to develop or port operating systems (OSs). Some of the reasons are as follows:

- Programmers of device driver must know information about hardware such as specifications of devices and carefully describe complex parts such as timing control.
- When two devices have different chips (controllers) even if they offer the same services, the programmers must write two different device drivers for each of them.
- If we change an OS but use the same devices, we need to have the device drivers for new one.

As internet is grown and multi-media are progressed, various devices would be developed. It is a more serious problem to spend much time and make efforts to write the device drivers. Device driver developers require in-depth understanding of innumerable peripherals that exist in a typical embedded system, programming tools, operating systems, bus protocols, network programming, and system management. In this paper, we aim at lightening the burden. We propose a support system for device driver generation. The support system proposed in this paper is the Windows CE6.0 operating system and the device driver used for the proposed system is stream driver.

- Harshitha.K.Raj, M.Tech(4thsem),SSIT, Tumkur. harshitha.k.raj249@gmail.com
- S.R.Sujatha, Asst.Professor,SSIT, Tumkur. sujathassit@gmail.com
- Dr.M.Siddappa, HOD(Comp.Sci),SSIT, Tumkur. Siddappa.p@gmail.com
- P. Satish Kumar., Member (Research Staff), CRL-BEL, B'lore psatishkumar@bel.co.in

II. Support system for Device Driver Generation

It is considered that inputs for the system are various elements such as functions, values, timing control. We attempt to simplify inputs for the system. The inputs for the system are determined as follows:

- **device driver specification**

It shows operations of the device. It is described that functions, data structure, and code in the functions which the generated device driver uses.

- **OS dependent specification**

It shows dependent parts on the OS. It is described that names, arguments, return values of device driver interfaces which the OS provides.

- **device dependent specification**

It shows dependent parts on the device. It is described that dependent parts on the hardware in the functions of the device described in the device driver specification.

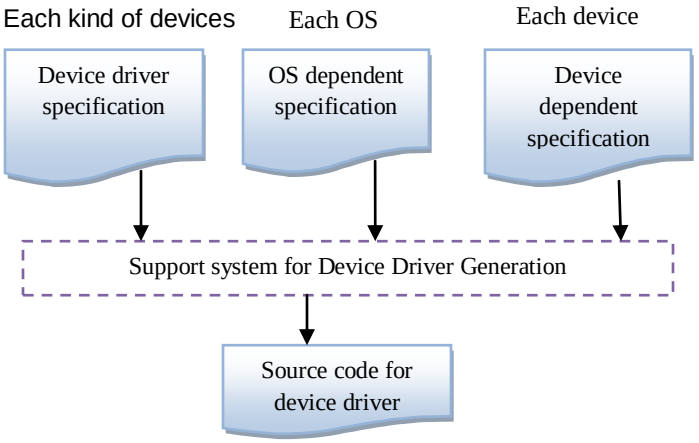

Fig 1: Outline of support system for device driver generation

III. Support system architecture: Windows CE6.0

Windows Embedded CE 6.0 is an embedded OS designed to turn vision and ingenuity into superior business results by offering a great combination of the creative tools and technology. Windows Embedded CE 6.0 provides a 32-bit native hard real time, small footprint operating system, a re-designed kernel, and powerful embedded development tools. CE 6.0 interoperates with industry standards and existing Microsoft desktop and server technologies to help you create differentiated devices for a broad range of

device categories, from commercial devices to consumer electronics products. It is mainly developed to address the needs of handheld, mobile devices. The Windows Embedded CE 6.0 is chosen as the OS for the next generation HHC based on portability issues of the existing application and new features, advantages and rich set of application development tools offered when compared to the other embedded operating systems. Windows Embedded CE can be adapted to a variety of devices like Smartphone, PocketPCs, set - top boxes, thin- client terminals, digital cameras, DVRs, VoIP, network routers, wireless projectors, industrial automation, home and building automation, robotics, data acquisition, and human – machine interfaces. In Windows CE6.0 layout critical driver, File System, GWEs are moved into the kernel. This greatly reduces the overhead of system calls between these components, reduces overhead of all calls from user space to kernel space and increases code sharing between base OS services. The Win CE6.0 architecture is shown in Fig2.

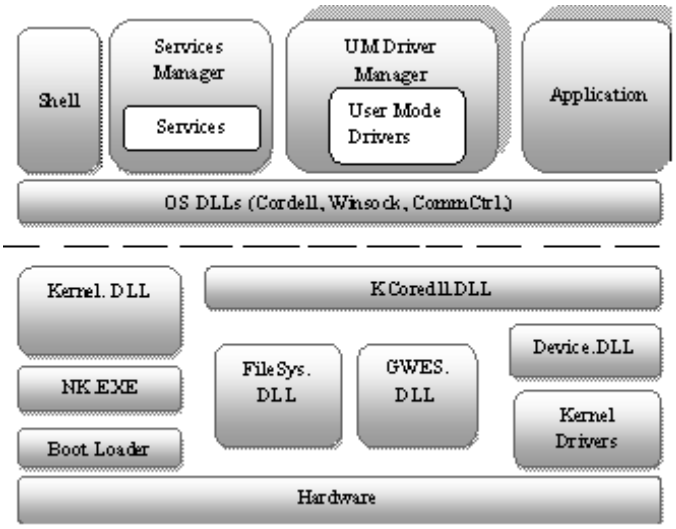

Fig 2: Windows CE6.0 Architecture

IV. Device Driver Overview

The term device, as used in this paper, does not refer to the primary central processing unit (CPU), or main memory, but a specific hardware resource for a dedicated task. The device is either attached to or embedded in computer system architecture and can interact with the CPU and other hardware resources in the system via a single system bus or through a bus hierarchy. A device driver is a low-level software component in the OS, which allows upper-level software to interact with a device. A device driver implements an interface to the kernel and application developers for an underlying device, and provides a lower-level communication channel to the device. It acts as a translator from the kernel interface to the hardware interface. As a form of communication, it requires kernel services, and often also offers services to other kernel components. The Device Manager is loaded by the kernel, it runs continuously, and it manages loaded device drivers and their interfaces. When the Device Manager loads, it also loads the I/O Resource Manager to read a list of available resources from the registry. The Device Manager tracks interfaces advertised by drivers and supports searches for drivers based on a globally unique identifier (GUID). Device Manager runs under the Windows Embedded CE operating system tracking loaded drivers and their interfaces. It runs continuously and launches from the kernel. Device Manager can notify the user when device interfaces become available and unavailable. A user or the system itself can make device interfaces available or unavailable. Additionally, Device Manager notifies the kernel that devices interface supports file operations, such as Create File, to access devices that expose the stream interface. Device Manager sends power notification callbacks to device drivers and provides power management services. Drivers can programmatically advertise interfaces by calling DMAdvertiseInterface. DMAdvertiseInterface enables drivers to add more searchable GUIDs to their associated lists. DMAdvertiseInterface is exposed by Devmgr.dll, which also implements most Device Manager Functionality. Because only the Device Manager can load Devmgr.dll, only device drivers can call DMAdvertiseInterface. If a device driver does not advertise the unavailability of its interfaces when the driver is unloaded, Device Manager automatically cleans up the interface advertisement notification. Device Manager consists of three components: nk.exe, device.dll, and devmgr.dll. Nk.exe loads device.dll, which is a thin shell that loads devmgr.dll. Devmgr.dll implements the core device manager functionality. Mainly in this paper Windows CE6.0 device driver is used. Windows Embedded CE, a device driver is a dynamic-link library (DLL). A device driver provides the operating system with an interface to the platform's peripheral devices. The driver exposes a set of known functions and provides the logic to initialize and communicate with the hardware. Software developers can then call the driver's functions in their applications to interact with the hardware. Windows CE supports two different types of drivers: native drivers and stream drivers. Native CE drivers typically support input and output peripherals, such as display drivers, keyboard drivers, and touch screen drivers. The Graphics, Windowing, and Events Subsystem (GWES) loads and manages these drivers directly. Native drivers implement specific functions according to their purpose, which GWES can determine by calling the GetProcAddress API. Stream drivers, on the other hand, expose a well-known set of functions that enable Device Manager to load and manage these drivers. For Device Manager to interact with a stream driver, the driver must implement the Init, Deinit, Open, Close, Read, Write, Seek, and IOControl functions. In this paper mainly discussion is done on the stream driver. Stream driver is a type of windows CE driver used in development of the keypad, audio device driver for custom Ti boards.

V. Stream Driver

The stream interface facilitates interaction with peripherals that primarily produce or consume data. The stream interface functions allow other software modules to interact with peripherals as if they were files. A stream interface driver is any driver that exposes the stream interface functions, regardless of the type of device controlled by the driver. The stream interface is appropriate for any I/O device that can be thought of logically as a data source or a data sink. That is, any peripheral that produces or consumes streams of data as its primary function is a good candidate to expose the stream interface. An example is a serial port device. An example of a device that does not produce or consume data in the

traditional sense would be a display device, and indeed, the stream interface is not exposed for controlling display hardware. The stream interface can use another underlying device driver to access the physical peripheral devices that the driver manages, or they can access the device directly if the device is mapped into memory. Audio device drivers for built-in audio hardware and keypad are an example of direct access. The stream interface functions themselves are designed to closely match the semantics of the usual file system application programming interfaces (APIs) such as Read File, IOControl. Despite the generic characteristics of the stream interface, it can be implemented in different ways. For example, even though the stream interface is typically implemented by independent hardware vendors (IHVs) for peripheral devices, original equipment manufacturers (OEMs) might choose to expose the stream interface for certain built-in devices.

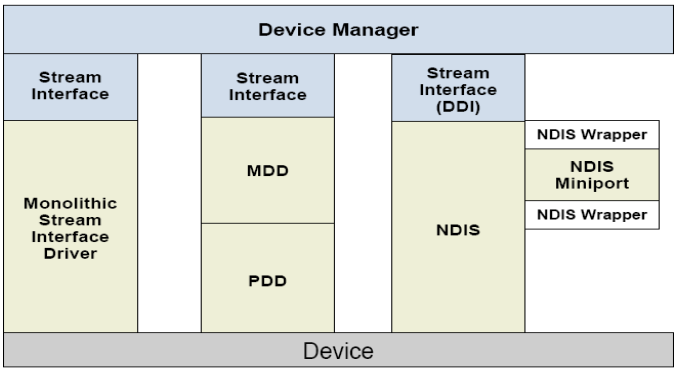

Fig 3: Stream Driver Model

VI. Stream Driver Architecture

A stream interface driver receives commands from the Device Manager and from applications by means of file system calls. The driver encapsulates all of the information that is necessary to translate those commands into appropriate actions on the devices that it controls. All stream interface drivers, whether they manage built-in devices or installable devices, or whether they are loaded at boot time or loaded dynamically, have similar interactions with other system components. The following illustration shows the interactions between system components for a generic stream interface driver that manages a built-in device.

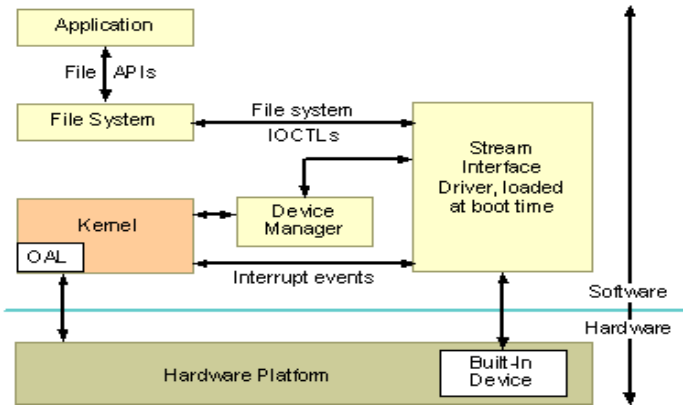

Fig 3: Stream Driver Architecture

Various issues can affect your design decisions as you implement a stream interface driver dynamic-link library (DLL). To implement the stream interface driver, create a DLL containing the required entry points for the drivers, and then decide whether you want to enforce either single or multiple accesses to the driver. Here and elsewhere, ***XXX*** refers to a three character prefix you choose for your device driver. When implementing a stream interface driver replace ***XXX*** with a prefix appropriate for your specific implementation.

VII. Stream Driver Function

Device Manager uses the ***XXX*** prefix. When implementing a stream interface driver, replace ***XXX*** with a prefix appropriate for your specific implementation.

XXX_Close (Device Manager) : This function closes the device context identified by hOpenContext. This function is required to access the device with Create File. If you implement XXX_Close, you must implement XXX_Open.

XXX_Deinit (Device Manager): This function de-initializes a device. It is called by Device Manager. This function is required by drivers loaded by ActivateDeviceEx, ActivateDevice, or RegisterDevice.

XXX_Init (Device Manager): This function initializes a device. It is called by Device Manager. This function is required by drivers loaded by ActivateDeviceEx, ActivateDevice, or RegisterDevice.

XXX_IOControl (Device Manager): This function sends a command to a device. This function might or might not be required, depending on the device capabilities that the driver exposes. This function requires an implementation of XXX_Open and XXX_Close.

XXX_Open (Device Manager): This function opens a device for reading, writing, or both. An application indirectly invokes this function when it calls Create File to obtain a handle to a device. This function is required to access the device with Create File.

XXX_PowerDown (Device Manager): Optional. This function ends power to the device. It is useful only with devices that can be shut off under software control.

XXX_PowerUp (Device Manager): Optional. This function restores power to a device.

XXX_PreClose (Device Manager): Optional. This function marks the closing handle as invalid and wakes any sleeping threads.

XXX_PreDeinit (Device Manager): This function marks the device instance as invalid and wakes sleeping threads. This function is required if the XXX_PreClose function is implemented.

XXX_Read (Device Manager): This function reads data from the device identified by the open context. This function might or might not be required, depending on the device capabilities that the driver exposes. This function requires an implementation of XXX_Open and XXX_Close.

XXX_Seek (Device Manager): This function moves the data pointer in the device. This function might or might not be required, depending on the device capabilities that the driver exposes. This function requires an implementation of XXX_Open and XXX_Close.

XXX_Write (Device Manager): This function writes data to the device. This function might or might not be required, depending on the device capabilities that the driver exposes. This function requires an implementation of XXX_Open and XXX_Close.

VIII. Conclusion

The main purpose of this paper is to aim at reducing the burdens in writing device drivers. So to avoid burden support system for device driver generation has proposed. Mainly the support system used is the Windows CE6.0 operating system and the driver for this system is the stream interface driver. The support system generates a source code of a device driver by giving three specifications such as device driver specification, OS dependent specification, and device dependent specification. The stream interface driver is appropriate for any I/O device that can be thought of logically as a data source or a data sink. So the proposed system which produces or consumes streams of data as its primary function is a good candidate to expose the stream interface.

References

[1] Tetsuro Katayama, Keizo Saisho, and Akira Fukuda: *"Proposal of a Support System for Device Driver Generation",* 1999 IEEE.

[2] Hui Chen, Guillaume Godet-Bar, Frederic Rousseau, and Frederic Petrot: "*Me3D: A Model-driven Methodology Expediting Embedded Device Driver Development"*, 2011 IEEE.

[3] A. Weggerle and C. Himpel and T. Schmitt: *"Transaction Based Device Driver Development",* MIPRO 2011.

[4] Mizell.A.M: *"Understanding device drivers in OS/2",* IBM Enter System Divisions, Bocu Raton, USA.

[5] Levin. V: *"Static driver verifier, a formal verification tool for Windows device drivers",* 2004 IEEE.

[6] Clerc. D, Garcé s-Erice L, Rooney S: *"OS streaming deployment",* 2010 IEEE.

[7] Stanislav Pavlov, Pavel Belevsky: *"Windows® Embedded CE 6.0 Fundamentals."*

[8] Samuel Phung: "*Microsoft® Windows Embedded CE 6.0."*