bSQUARE

The Mobile & Embedded Systems Experts

BSQUARE SchemaBSP

Three Steps to Windows® CE BSP Development

Executive Summary: *BSQUARE SchemaBSP minimizes time-to-market for Windows CE BSP development by reducing the scope of work, and diminishing the engineering effort required. SchemaBSP is used to develop a BSP systematically. When the BSP is ready, itís simply imported into Platform Builder to finish configuring the Windows CE ROM image.*

Introduction to BSQUARE SchemaBSP

You've just been assigned to write a Board Support Package (BSP) for your company's latest hardware platform. How do you start? The answers from most software engineers range from "use one of the reference BSPs," to "start from scratch," to "I'm not sure." BSQUARE has a better answer: the BSQUARE SchemaBSP

SchemaBSP contains three major parts:

- DeviceWare, a source code library;
- CoreLinx IDE, an Integrated Development Environment; and
- CompWare, a set of hardware designs.

DeviceWare is a library of production ready BSP source code that gets your product to market faster with fewer engineers. Microsoft® Platform Builder comes with a lot of source code, but the DeviceWare library is significantly better in several ways. First, it is a library of source code used by your project, not edited for your project. Second, the hardware abstraction is taken further than the sample code provided by Platform Builder. Finally, the source code is production ready.

The source code is in a library. This has several very important implications. It is source code that you can review and even modify when needed. Unique to SchemaBSP is that your project is separated from the DeviceWare library. This means that the source code can be updated quickly and easily with new versions of SchemaBSP. This is contrary to one of the most popular methods of BSP development, modifying a reference BSP. When using a modified reference BSP, it is very difficult and tedious to integrate updates when and if they are provided. By separating the source code from the project, BSQUARE can and does send updates which usually only require a re-build to integrate.

According to *Total Cost of Development*, a report from **Embedded Market Forecasters**, an average of 8.3 software engineers is required to develop a Windows CE product. Most SchemaBSP projects only require two to four engineers. The need for engineers is reduced because DeviceWare contains hardware abstraction that surpasses the native Microsoft model device driver (MDD)/platform-dependent driver (PDD) layering. The Windows CE device driver model divides source code into the MDD and PDD layers. The PDD layer is the hardware abstraction layer of the driver. To port the PDD layer requires engineers to reverse-engineer the PDD layer. This requires valuable project time to become familiar with a large amount of source code for the Bootloader, OAL, and device drivers.

The DeviceWare library is engineered to focus your development on the hardware. To accomplish this, the DeviceWare library pulls out all hardware interactions to a set of configuration files. It also provides a small set of APIs to use for configuring. We will look at this in more detail below.

CoreLinx is an Integrated Development Environment (IDE) designed to manage the development of BSPs. Microsoft Platform Builder is a great tool for managing a project once the BSP is created. CoreLinx, on the other hand, focuses on developing and building the BSP.

CompWare provides a set of hardware schematics and other documentation that can be used to accelerate hardware development. Its use is not required, but when used it can provide solutions to common problems with battery-operated embedded devices.

Three Step Process for Building a Windows CE BSP

Software development using BSQUARE SchemaBSP is accomplished in three basic steps: create the BSP, configure the BSP and configure Windows CE.

Step 1 – Create the BSP

Start software development by creating a BSP using the New Platform Wizard found in CoreLinx. The New Platform Wizard allows you to use a system block diagram as a guide in selecting OAL, Bootloader, and Device Driver components. The block diagram typically contains most of the information needed for this wizard.

When the wizard starts, you are prompted to enter basic information about the project. This includes project name, CPU, and Windows CE version. Once this is established, the wizard will look at what is available in the DeviceWare library to determine the components compatible with your initial settings.

The next step is to select the chips on the platform. These might be companion chips or external peripherals like CODECS, FLASH ROM, battery gauge, display controller, and others. When the hardware and OS are established, the wizard displays a list of software components that are available for your platform (please see Figure 1).

After the software components are selected, the wizard will create a small set of configuration files. These files define the hardware characteristics of your platform.

Step 2 – Configure the BSP

The next step in the development process is to configure the BSP based on Product Schematics and Specifications. This is the main step in creating the BSP. The product schematics and specifications provide the requirements for configuring the BSP. A USB Function driver provides a good example of how a BSP is configured.

For this example, the USB Function driver will use an Intel PXA255 internal USB Function controller. The DeviceWare driver supports the PXA255 USB controller, but the controller does not have a way to determine that the cable is connected

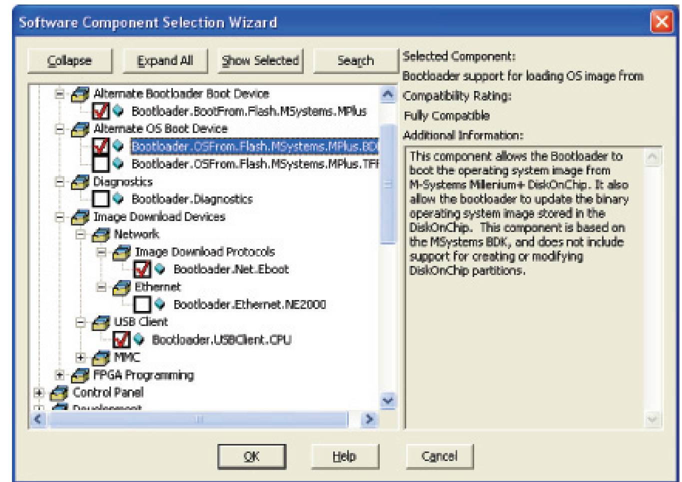

Figure 1 - New Platform Wizard: Software Components

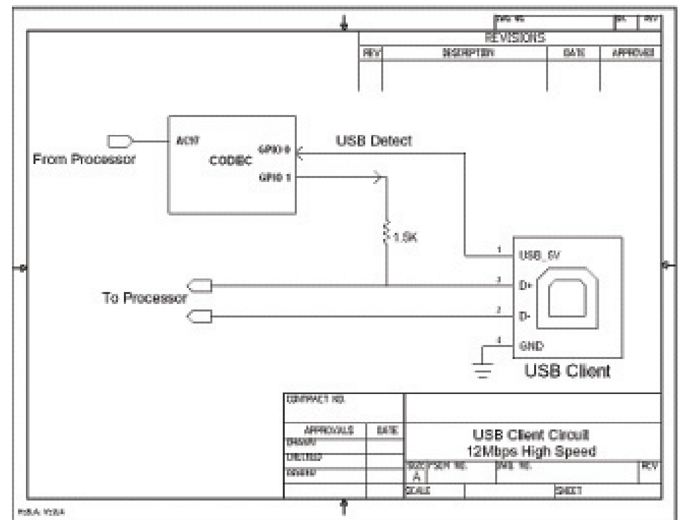

Figure 2 - Simplified USB Client Circuit

to a host controller. So an electrical engineer needs to provide a signal to allow the software to determine that the cable is connected. There are many possible ways to accomplish this, including connecting a signal to an internal GPIO or an external GPIO on some peripheral chip like a CODEC or companion chip. If electrical engineers have already used all of the internal GPIOs, they can connect the signal GPIO 0 to a CODEC.

The DeviceWare library already knows how to control the internal USB Function controller, so for this part of the driver select the PXA25x USB controller.

```
// EP0 Control is always used
#define USB_CLIENT_DRIVER_USES_EP1  TRUE // Bulk IN Required for ActiveSync
#define USB_CLIENT_DRIVER_USES_EP2  TRUE // Bulk OUT Required for ActiveSync
#define USB_CLIENT_DRIVER_USES_EP3  FALSE // Isochronous IN
…
#define USB_CLIENT_EP4_BUFFER_SIZE 0x1000
```

The PXA255 has some options for DMA control. The defaults are typically used in most Windows CE devices. These options are available if the device requires different settings.

```
// USB Client Controller Select Constants
…
#define USB_CLIENT_CONTROLLER_PXA25x                5

// Select from on of the above interfaces
#define USB_CLIENT_CONTROLLER_SELECT USB_CLIENT_CONTROLLER_ PXA25x
```

The next step is a relief for those of you who have trouble mapping virtual addresses to real addresses. This section of the configuration tells the driver how the hardware communicates with the software. It is very simple from the configuration side, but there is substantial code in the DeviceWare library to support it. For this section, set the correct macros to TRUE to identify the hardware interaction that the driver needs. When the code is compiled, all of the mapping will be done automatically.

Now we need to read from the CODEC. Set USB_CLIENT_DRIVER_USES_CO-DEC_GPIO to TRUE. This adds the code necessary to instantiate and initialize the variables needed for the DeviceWare API to read from the CODEC.

```
// Does the driver need any ranges? If so, set to TRUE
// Do not delete unused ranges!
#define USB_CLIENT_DRIVER_USES_ONCHIP_GPIO          FALSE
#define USB_CLIENT_DRIVER_USES_CODEC_GPIO           TRUE
#define USB_CLIENT_DRIVER_USES_COMPANION_GPIO       FALSE
…
```

The CODEC GPIO pin 0 needs to be initialized to be an input. The driver needs to do this both when it starts on boot and when it resumes from suspend. The DeviceWare API provides a simple mechanism to do this, a macro named AC97_GPIO_DIREC-TION_xxxx. xxxx is either KERNEL or USER, but there is no need to worry about this, since the driver passes it in as the variable “mode.”

AC97_GPIO_DIRECTION_xxxx takes three arguments: value, mask and inpowerhandler. Value tells the macro which pins to set; in this case, only pin 0 is set. Mask is used to indicate which pins to modify so that, in this case, only pin 0 is modified. The Boolean argument inpowerhandler indicates whether or not the driver is in a power handler function, which in Windows CE has some API restrictions. For inpowerhandler, pass on the variable inpowerhandler that is passed in from the driver.

```
// What special initialization should be done when the driver loads
#define USB_CLIENT_INITIALIZATION(mode, inpowerhandler) \
AC97_GPIO_DIRECTION ## mode ( BIT0, BIT0, inpowerhandler )

// What special steps should be performed at power up (resume)
#define USB_CLIENT_POWER_UP(mode, inpowerhandler)
AC97_GPIO_DIRECTION ## mode ( BIT0, BIT0, inpowerhandler )
```

This hardware design does not require any components to be powered down when the system is suspended. Nothing needs to be done with the USB_CLIENT_POWER_DOWN macro.

```
// What special steps should be performed at power down (suspend)
#define USB_CLIENT_POWER_DOWN(mode, inpowerhandler)
```

The next section configures the reading of the cable detect signal. Again the DeviceWare API provides a macro to assist with this. AC97_GPIO_READ_xxxx takes the same three arguments as the pin direction macro. However, this time value is the return value from the read and mask indicates the pins to read.

```
// Does the system have cable detection hardware support?
// If this is FALSE a 12 second timeout is used. If there is no
// traffic to any USB endpoint in this amount of time then the driver
// will indicate a disconnect (WaitCommEvent(EV_RLSD))
#define USB_CLIENT_CABLE_DETECT_SUPPORT_ENABLE             TRUE

#if USB_CLIENT_CABLE_DETECT_SUPPORT_ENABLE
// Sets the timeout period in milliseconds. If there is no traffic
// to any USB endpoint in this amount of time then the driver will
// call USB_CLIENT_CABLE_DETECT which returns a status of connected
// or not. This is the longest period of time USB_CLIENT_CABLE_DETECT
// can go without being called.
// ActiveSync generates keep alive traffic every three to four seconds.
#define USB_CLIENT_CABLE_DETECT_POLLING_RATE        500

// This is called for every USB event and if the USB_CLIENT_
// CABLE_DETECT_POLLING_RATE time expires before an event occurs.
// Set connected to TRUE if cable connection to host is detected.
#define USB_CLIENT_CABLE_DETECT(connected, mode, inpowerhandler) \
       AC97_GPIO_READ_ ## mode(connected, BIT0, inpowerhandler);
#endif
```

Start by enabling the cable detect support in the driver by setting USB_CLIENT_CABLE_DETECT_SUPPORT_ENABLE to TRUE. Then set the polling rate for checking the cable detect signal. In this case, it is 500 milliseconds. Then add a call to the DeviceWare API to read the signal.

Finally, configure the thread priority. This allows fine tuning of your system if needed. Typically the defaults are acceptable.

```
// Set the thread priority for the USBF Serial event handler thread
#define USB_CLIENT_EVENT_THREAD_PRIORITY 130
#define USBF_SERIAL_EVENT_THREAD_PRIORITY 130
```

The USB Function driver is now ready for use. It is significant to note that the driver configuration did not require virtual to physical address mapping, setting up an ISR, or reverse engineering the driver source code. The separation of the hardware from the software allows you to focus your development efforts on your hardware design rather than the software functions and data.

Step 3 – Configure Windows CE

Microsoft Platform Builder is used to configure Windows CE. When the platform is built, the CoreLinx tool automatically creates the files necessary to integrate the BSP into Platform Builder. These files include a platform CEC file and batch files. All that is needed is to import the CEC file to use your BSP in Platform Builder. Then you can create a Platform Builder project and select OS components and add applications.

With a BSP for the unique hardware design and the OS components, you are ready to ship your new product. However, completing the device software and shipping is not necessarily the last step in the process. Maintaining the device with the latest technology is an ongoing process.

Now is when the separation of hardware and software brings added benefits. For one thing, the separation makes the task of maintenance easier. BSQUARE is constantly adding support for new embedded processors and Windows CE versions to the DeviceWare library, and regularly delivers updates to SchemaBSP users. A simple push of a button in CoreLinx is all it takes to rebuild the BSP using your existing hardware configuration and the new software.

Conclusion

BSQUARE SchemaBSP makes it significantly easier and less expensive to develop a Windows CE BSP for a new hardware platform. This, however, is just one of the benefits that SchemaBSP users receive.

Original Design Manufacturers (ODMs) often find the library of source code enables them to get many products to market quicker. This is accomplished through easier BSP development since the library makes it possible to share a common code base between projects. By sharing a common source code base, each project learns from the other projects. New features and drivers can be enabled quickly in all projects.

Original Equipment Manufacturers (OEMs) with long-term embedded projects can leverage BSQUAREís engineering expertise for maintenance. This is delivered in the form of updates to the DeviceWare library, including support for new versions of Windows CE and popular processors.

OEMs that need to get new products to marker quickly, or risk missing out on important opportunities, are discovering that SchemaBSP provides them with a competitive advantage. So when youíre ready to start your next project, take a look at the advantages that SchemaBSP Development Suite provides.

Take the Next Step: Contact BSQUARE Today

To find out more about SchemaBSP, as well as BSQUARE professional engineering services, Windows Embedded licensing and development tools that can accelerate your time to market, contact BSQUARE at 1-888-820-4500 or email sales@bsquare.com. To view an online demo of the SchemaBSP product, visit www.bsquare.com/products/schemabsp.asp

For more information, please visit **www.bsquare.com**

About BSQUARE:

BSQUARE is a solution provider to the global embedded device community. Our teams collaborate with OEMs at any stage in their device development to bring quality products to market faster. Since 1994, BSQUARE has been a trusted partner to smart device makers worldwide.

©2006 BSQUARE Corporation. BSQUARE is a registered trademark of BSQUARE Corporation. All other names, product names and tradenames are registered trademarks of their respective holders. CS-SCHEMABSP-2003, Rev2006-10

By email at sales@bsquare.com

BSQUARE Headquarters
110 110th Ave NE Suite 200
Bellevue, WA 98004, USA
Phone: 888-820-4500 or
direct at 425-519-5900

BSQUARE San Diego
6450 Lusk Blvd. Suite E210
San Diego, CA 92121, USA
Phone: 858-535-9265

BSQUARE Akron
3480 West Market Street
Suite 105
Fairlawn, OH 44333, USA
Phone: 330-864-2300

International:

BSQUARE Vancouver
3751 Shell Road Suite 100
Richmond, BC V6X 2W2 , Canada
Phone: 425 519 5900

BSQUARE Taiwan
18F.-B, No. 89
Songren Road, Xinyi District
Taipei City 110, Taiwan
Phone: +886-2-8780-9100