

Driver Synthesis: Automating the Creation of BSPs

Everyone knows a BSP is needed for all high-end embedded design projects, but what exactly is a BSP? What is involved in developing one? What means are available to accelerate the development of a BSP? This article identifies the different types of BSPs, the issues a BSP has to handle, and describes existing means to expedite the development of a BSP. This article also presents an emerging approach to automate the creation of a BSP through driver synthesis.

THE STRUCTURE OF A BOARD SUPPORT PACKAGE

The term board support package (BSP) has different meanings to different people. To the RTOS vendor, a BSP is the boot code and set of drivers needed to handle the operating system. For someone who needs to control the complete set of peripherals on a target board, the BSP means the full set of drivers to control this board. Hardware engineers who need to verify that a board is working (but do not care about the operating system), would need a BSP that does not include any OS specific code. Figure 1 shows the structure of an embedded system, and

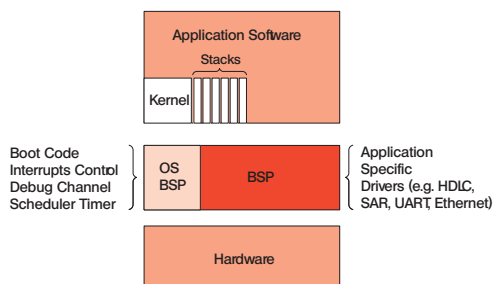


Figure 1. The BSP in the structure of an embedded system

shows the OS BSP as part of the BSP.

For purposes of this article, I am defining the BSP as follows: A BSP is the full set of drivers and boot code for a board. This includes an OS BSP, which is the partial set of boot code and drivers required for the operation of the RTOS, and the test BSP, which is needed by hardware engineers to test a board without the operating system.

The general structure of the BSP includes the following:

1. **Setting up the CPU:** When the CPU comes out of a reset, some of its registers should be set to allow it to function.
2. **Setting up memory:** Identifying where the different parts of memory are located (program, data, stack, etc.)
3. **Setting up the interrupts:** The interrupt controller should be set to handle the basic OS interrupts, and the I/O interrupts.

4. **Setting up the operating system scheduler timer:** The timer needs to be set to provide time ticks for the scheduler to handle task switching.
5. **Setting up the communication drivers for the operating system debug channel:** The debug channel is used for communication between the host and the target. Usually the debug channel is a serial port or an Ethernet channel.
6. **Setting up the application specific drivers for the board:** These drivers are not required for the operating system to operate, but are required by the application. These drivers could be fairly simple drivers, such as additional timers or serial ports. However, they could be extremely complex, such as ATM drivers.
7. **Downloading the OS components from an outside source (optional):** In some cases the boot sequence is divided into two parts. Initially, after a reset occurs, a ROM based boot code brings the board up. Once the board is up, the operating system modules and the application code are loaded from a remote location (e.g. the host) The system then reaches its final configuration and the application starts running.
8. **Running the OS:** At some point, the operating system has to be fired up, and the first task is launched.
9. **Running the application:** After the OS is ready to run, the application tasks are launched.

Each operating system has a different way of running through these activities. Some are stricter on the order and contents, while others are more relaxed and let the user decide how the system should be brought up. For example, VxWorks has a very specific set of actions in bringing up the system, due to its extensive support of I/O and networking functionality. In comparison Nucleus, allows the user to decide how the I/O would operate.

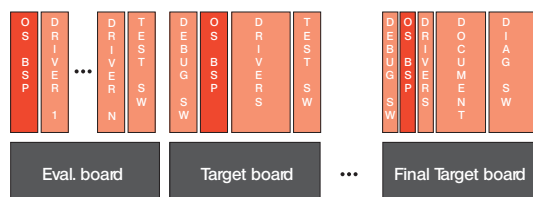


Figure 2. Stages in BSP development (the process flows from left to right)

DEVELOPING A BOARD SUPPORT PACKAGE

Creating a BSP requires interdisciplinary knowledge of:

1. **CPU hardware architecture:** The BSP developer has to know how the CPU handles reset, memory management and interrupts. In addition, most of the initial code has to be written in assembly language.
2. **On-board or on-chip peripherals:** Coding the drivers for the application requires understanding of the peripherals that perform the hardware functionality. That includes understanding the registers that control the peripheral, the timing of setting them up, and the interaction with the CPU.
3. **Operating system behavior:** The operating system is hardware independent, and the BSP serves as the connection between the OS and the hardware. There is a set of APIs that the OS expects the BSP to provide in order for it to function. These APIs have semantic meaning that needs to be mastered to provide a fully functional OS connection to the hardware.
4. **Application requirements:** In order to provide the right services for the application, the BSP developer must understand what the application is expecting the hardware to deliver.

Going through the learning process has always been complex. But these days, with the increased complexity of the hardware, the learning task has become as long as the whole project schedule. It is highly recommended that developers take the OS vendors' BSP training courses to better understand the BSP structure for the specific OS, as well as the chip vendors' training courses on their architectures, to understand the



Figure 3. Selecting an OS and compiler using DriveWay

hardware side of the system.

Because of the complexity of the task and the fact that, in many cases, the target hardware is not available initially, the BSP is usually developed in stages. The first stage is done on an evaluation hardware that has been verified in the past and can be used until the target hardware becomes available. The operating system vendors or the board manufacturers usually provide a sample BSP that works in popular evaluation boards. Using the evaluation board and the sample BSP, the first task is to make them work. This requires a lot of learning and (usually) extensive debugging of the design environment. Once the vendor supplied BSP works on the evaluation board, the application drivers can be developed.

When the hardware prototype arrives, the OS vendor supplied sample BSP is no longer applicable, and two BSPs have to be written: the hardware testing BSP and the target BSP. The hardware testing BSP includes test code to exercise the hardware and help debug the board. In order to reduce complexity, this BSP is better

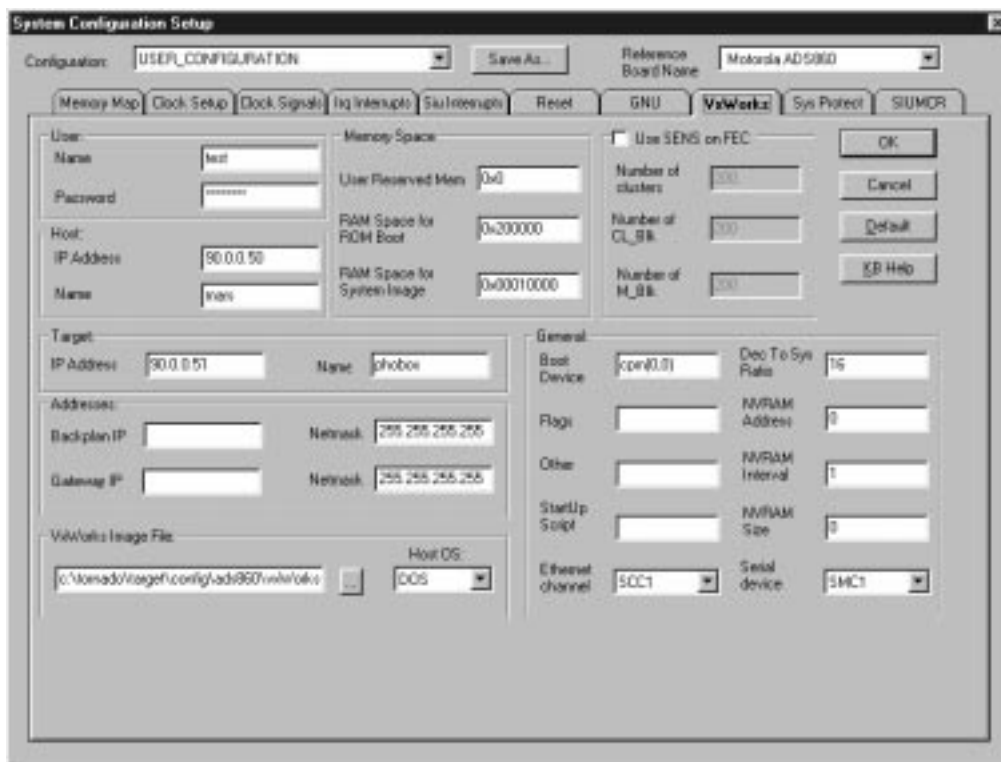


Figure 4. VxWorks system configuration screen

off without an operating system. Once the board is verified, the OS BSP has to be created. The vendor provided BSP does not run on the target board, so the developer has to write and debug a new BSP. This is a lengthy process that gets trickier because of the differences between the evaluation board and the target board. Also, based on the application requirements (e.g. if the vendor supplied BSP uses a resource -- such as a specific UART -- that has to be used for the application), the BSP must be tailored to use a different resource. Once the OS BSP is working, the application BSP (drivers) must be written and debugged to complete the full target BSP.



Figure 5. Evaluation board selection

As the design progresses, hardware bugs are found on the prototype board, requiring a re-spin of the board. There are generally a few cycles of target board manufacturing until the final board is working. At this point, prior to final product release, the board diagnostic has to be developed (this includes power up diagnostics and manufacturing diagnostics).

The code developed during this process must be documented. This is often the most neglected piece of the process, but one of the more important ones for the product life cycle. Once the code is integrated into the product, the project team moves on to the next project.

Insufficient documentation makes it impossible to maintain the code and, due to the amount of specific knowledge required to create the BSPs, this part of the code can become un-maintainable very quickly after the project is finished.

Figure 2 presents the BSP creation process, from the BSP used on the evaluation board to the final version. (The part of the BSP that is available from the OS vendor is the white block on the left.)

BSP CREATION TECHNOLOGY

Until not long ago, developers had no choice - they had to go through the BSP development process. Instead of focusing on the application coding, which is where the value of the product resides, a small group of people were busy creating and debugging BSPs. Finally, a new way of creating BSPs has begun to emerge: driver synthesis, which automatically creates BSPs for both the evaluation and target boards. A point and click interface is used to specify the design requirements and parameters. Based on these definitions, the system creates the BSP configured according to the definitions provided by the user for the specific design.

Let's take the example of the MPC-860 processor. This processor is one of the most highly integrated processors. It features a PowerPC core and about 25 different peripherals, which include complex peripherals such as Ethernet and HDLC. Creating a BSP for this chip takes nine to 18 person months, a significant part of which is in the critical path of the design. Using DriveWay, one of the first driver synthesizers, this BSP will take one to three weeks to develop. Let's follow the BSP creation process for a VxWorks based design.

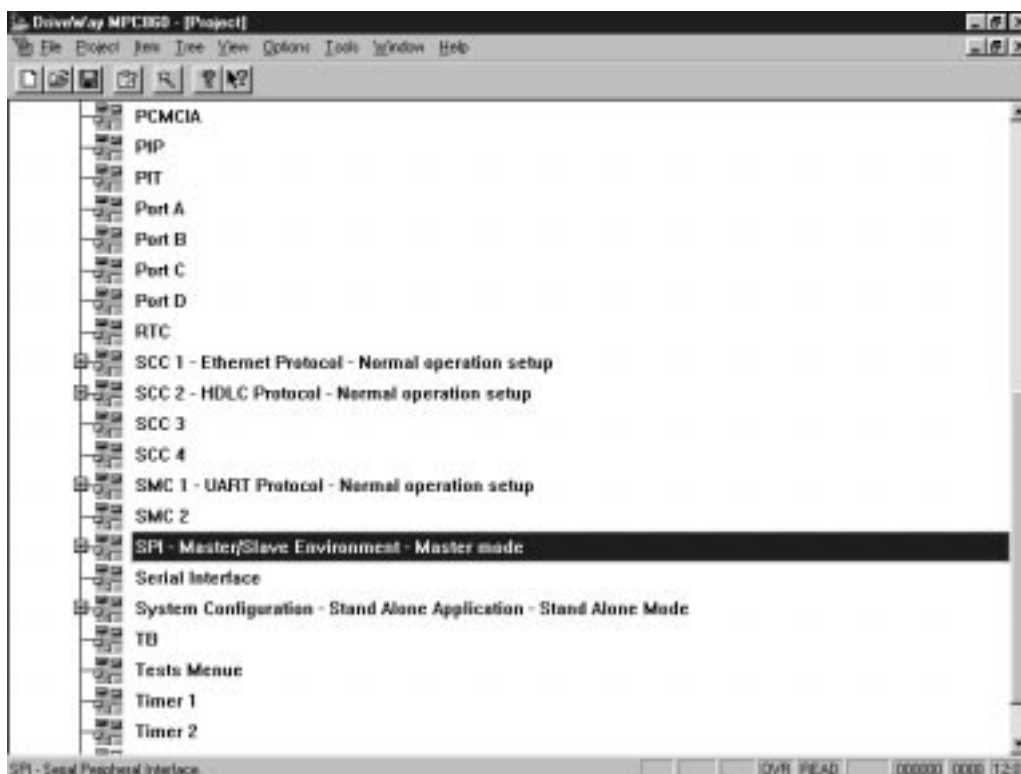


Figure 6. Peripheral Navigator (partial view) with Ethernet, HDLC and UART selected

dw_860.h - MPC860 Memory Map
 pq_hand.h - MPC860 Interrupt Handlers Header File
 cpic.c - Driver code
 cpic.h - Driver header
 system.c - Driver code
 system.h - Driver header
 project.h - Project include
 projectc - The Project Main File
 ads860.h - Motorola MPC860ADS Board Header
 booth - Boot Definitions
 brgs.c - DriveWay Baud Rate Generators (BRGs) Software Interface
 brgs.h - DriveWay Baud Rate Generators (BRGs) Software Interface
 config.h - Motorola 860ADS Board Configuration Header File
 configNeth - configNet for use in VxWorks with SENS
 ctdt.c - ctors and dtors arrays for VxWorks BSP
 dw_860.c - Main Project File
 dw_brg.h - DriveWay BRGs Interface
 dw_cache.h- DriveWay Data Cache Interface
 dw_clock.h- DriveWay Clocks Interface
 dw_cpic.h - DriveWay CPM Interrupt Controller Interface
 dw_gpt.h - DriveWay General-Purpose Timers Interface
 dw_i2c.h - DriveWay I2C Header File
 dw_idma.h - DriveWay IDMA Channels Interface
 dw_mem.c - DriveWay Memory Management Software Interface
 dw_mem.h - DriveWay Memory Management Software Interface Header File
 dw_memc.h - DriveWay Memory Controller and Banks Interface
 dw_mmu.h - DriveWay MMU Interface
 dw_pip.h - DriveWay PIP Interface
 dw_pith - DriveWay PIT Interface
 dw_port.h - DriveWay Ports Interface
 dw_porta.h- DriveWay Port A Interface
 dw_portb.h- DriveWay Port B Interface
 dw_portc.h- DriveWay Port C Interface
 dw_portd.h- DriveWay Port D Interface
 dw_reset.h- DriveWay Reset Interface
 dw_risc.h - DriveWay RISC Controller Interface
 dw_ritt.h - DriveWay RISC Timers Table Interface
 dw_rtc.h - DriveWay RTC (Real Time Clock)
 dw_scc.h - DriveWay SCCs Interface
 dw_scca.h - DriveWay SCCs AppleTalk (LocalTalk) Protocol Interface
 dw_sccb.h - DriveWay SCCs Bisync Protocol Interface
 dw_scce.h - DriveWay SCCs Ethernet Protocol Interface
 dw_scch.h - DriveWay SCCs HDLC Protocol Interface
 dw_scct.h - DriveWay SCCs Transparent Mode Interface
 dw_sccu.h - DriveWay SCCs UART Protocol Interface
 dw_sdma.h - DriveWay SDMA Channels Interface
 dw_si.h- DriveWay Serial Interface/TSA Interface
 dw_smc.h - DriveWay SMCs Interface
 dw_smct.h - DriveWay SMC Transparent Mode Interface
 dw_smcu.h - DriveWay SMC UART Protocol Interface
 dw_spi.h - DriveWay SPI Interface
 dw_wind.h - DriveWay WindRiver/ VxWorks header file
 end860.c - Tornado for sens END file on FEC
 end860.h - Tornado for sens END file on FEC
 gnu.bat- Set the GNU environment in a stand alone mode(without VxWorks)
 gnu.h - GNU Compiler Header File
 gnu.txt- DriveWay GNU Interface
 io.c- IO system services
 io.h- IO system services header file
 link.cmd - Link Command File for the GNU Compiler
 make.def - Makefile definitions for GNU in a standalone environment
 makefile - VxWorks Makefile
 makefile.gnu - GNU makefile for a standalone environment
 memtest.txt - memory test text file
 pq_brg.h - MPC860 BRGs
 pq_cache.h- MPC860 Cache
 pq_clock.h- MPC860 Clocks and Power control
 pq_cpic.h - MPC860 CPM Interrupt Controller
 pq_defs.h - Project General Definitions and Constants.
 pq_dpram.h- MPC860 Dual Port RAM
 pq_gpt.h - MPC860 General Purpose Timers
 pq_hand.c - MPC860 Interrupt Handlers
 pq_i2c.h - MPC860 I2C Header File
 pq_idma.h - MPC860 IDMA Channels
 pq_memc.h - MPC860 Memory Controller
 pq_mmu.h - MPC860 MMU
 pq_pcmcia- MPC860 PCMCIA
 pq_pip.h - MPC860 PIP
 pq_pith - MPC860 PIT (Periodic Interrupt Timer)
 pq_port.h - MPC860 Ports
 pq_ppc.h - MPC860 PowerPC Core
 pq_reset- MPC860 Reset
 pq_risc.h - MPC860 RISC Controller
 pq_ritt.h - MPC860 RISC Timers Table
 pq_rtc.h - MPC860 RTC (Real Time Clock)
 pq_scc.h - MPC860 SCC
 pq_scca.h - MPC860 SCC AppleTalk Protocol
 pq_sccb.h - MPC860 SCC BISYNC Protocol
 pq_scce.h - MPC860 SCC Ethernet Protocol
 pq_scch.h - MPC860 SCC HDLC Protocol
 pq_scct.h - MPC860 SCC Transparent Mode
 pq_sccu.h - MPC860 SCC UART Protocol
 pq_sdma.h - MPC860 SDMA Channels.
 pq_si.h- MPC860 Serial Interface/TSA
 pq_siu.h - MPC860 SIU
 pq_smc.h - MPC860 SMC
 pq_smct.h - MPC860 SMC Transparent Mode
 pq_smcu.h - MPC860 SMC UART Protocol
 pq_spi.h - MPC860 SPI
 pq_tb.h- MPC860 Timebase (TB)
 project.txt - Project documentation
 risc.c - DriveWay CPM RISC Controller Software Interface
 risc.h - DriveWay CPM RISC Controller Software Interface
 romlnits - ROM Initialization Module for VxWorks Operating System
 services.s- General System Services in ASM
 sysALib.s - System-Dependent Assembly for VxWorks Operating System
 sysLib.c - Board-Specific Routines for VxWorks Operating System
 sysSerial.c - MPC860 SMC UART BSP Serial Device Initialization
 sys_info.txt - System Information text file
 target.h - VxWorks Target board configuration file
 vxworks.txt - DriveWay WindRiver/VxWorks Documentation File

Figure 7. DriveWay Created BSP File List



Figure 8. DriveWay generated BSP components

The first stage is to define the operating system. Figure 3 shows the screen where the OS and compiler for the specific design are selected.

Once VxWorks and GNU are selected, the next step is to set up the board. This is done via a system configuration screen where all the board parameters and VxWorks parameters are set. Figure 4 presents the VxWorks tab of the system configuration screen.

If an evaluation board is used in this stage of the design, DriveWay has a set of supported boards that can be selected and set up automatically. Figure 5 presents the standard boards currently supported for the MPC860 board.

Once the board parameters are provided, the OS BSP is ready to be generated. If some application drivers are desired, the user can get into the chip navigator screen and set up the peripherals. Figure 6 shows the map of peripherals available on the 860, and the set-ups for them.

Once the peripherals are set, The BSP is ready for generation. A click on the generate button will create about 20,000 lines of source code (about 70% of which is documentation). The code is spread across approximately 100 files that include the OS BSP files, the drivers files and project related files (makefile, link.cmd file, etc.) Figure 7 provides a list of the files created for the VxWorks project. The generated BSP can be compiled using the GNU compiler, and loaded into the evaluation board, or the target board. The components of the created BSP can be seen in Figure 8.

In order to build a project to run with a Nucleus operating system, the following steps are necessary:

1. In the development environment dialog, select "OS: Accelerated/Nucleus."
2. In the same dialog, select "Compiler: DIAB Data C Compiler."
3. Select and configure the peripherals according to your application's requirements.
4. Generate the project files. One of these files is a makefile that includes all the drivers that your project contains.
5. After the code is generated, open the 'MAKE.DEF' file and set the NUCLEUS_PATH to point to the directory where Nucleus is installed, the default directory is "c:\nucleus."
6. DriveWay uses the NU_Create_Memory_Pool(), NU_Allocate_Memory() and

NU_Deallocate_Memory() system calls from Nucleus, in order to create, allocate and free memory buffers and not the malloc() and free() system calls.

The Dual Port Ram Set Parameters dialog under System Configuration includes parameters that set up the memory allocation type. The "Nucleus memory alloc" option should be selected when working with Nucleus. The initialization of the memory pool is done in the Application_Initialize() service located at the demo.c file. Note that these services cannot be called from a low level ISR. The "Simple linked list of buffers" option can be called from anywhere in the program.

DriveWay-MPC860 comes with a demo project called 'Nucdemo.dwp.' This project can be used as a starting point for developing a Nucleus based application. Use the Nmake utility to build the application according to the makefile.

SUMMARY

Creating a BSP becomes more difficult as the hardware and the systems become more complex. The process requires interdisciplinary knowledge of hardware and software, and entails many stages that reduce the degrees of freedom of the design. The time it takes to create a BSP delays the project schedule, and increases the R&D costs significantly. A new methodology - driver synthesis - enables automatic creation of the BSP based on user defined parameters. This new methodology saves months in the design process, saves in design costs, and enables the design engineers to focus on adding value to the developed system instead of spending time on debugging the drivers ■

Shaul Gal-Oz is founder, president and CEO of Aisys Inc. He has 18 years experience in the embedded systems field. Mr. Gal-Oz received his bachelors degree in Electronic Engineering from Ben-Gurion University in 1981. Email: uli@aisysinc.com