

Booting Nexperia Media Processors

NDK 5.7/MPTK 2.4



Application Note vX.X, 12 December 2007

Introduction

This document provides a software developer's overview of how the TriMedia processor boots in a Nexperia system. It focuses on what software components, why they exist, and how it is appropriate to modify them. Important aspects covered here are:

- Overview of the boot process and the phases involved.
- How Flash Boot is handled.
- The role of the Board Support Library (BSL)

1 Booting Overview

There are plenty of opportunities for diversity in the TriMedia boot process. As an introduction, it is helpful to list the steps along the way. Each of these steps is described in more detail in the sections that follow.

1. L0 – The boot script is executed. The boot script is read from an I²C EEPROM. It initializes a few critical registers, then takes the action necessary to jump to the appropriate code. This phase is called, when the CPU is not yet running, "L0" boot.
2. Boot scripts come in a number of flavors. Practically, the boot script may be unique to the board built in production. In most cases, it matches one of several templates, in the following list:
 - Hosted Boot – When the TriMedia is a slave on the PCI bus, as in the PC based development systems, the boot script simply initializes the clocks and then waits for the host to provide a program and start the processor.
 - JTAG boot – Developers often rely on JTAG to download programs. In this case, the boot script contains a small program that is able to receive a program downloaded from the host over JTAG. The size of this program is limited by the size of the EEPROM.
 - Flash boot – In embedded systems, the boot script uses DMA to load a program from flash into DDRAM. The TriMedia typically executes its programs out of DDR, so the boot script for NAND or NOR flash typically copies the program in before jumping to it.
 - It is certainly possible to boot from other locations (such as an IDE drive). The NDK package does not contain this code as of this writing.
3. Optional L1 boot – This phase is not present in a hosted boot configuration. The L0 phase has loaded some program into memory. Both in the JTAG case and in the flash cases, an L1 boot program is loaded. This program's job is simply to load a larger program. The L1 code runs on the CPU. It is typically built to exclude most sophisticated software features and hence be very small. In the JTAG case, it is close to 1.5kB. In the NOR flash case, it is less than 8k.

4. The chapter on flash boot describes a further L2 boot phase that is not present in most other boot situations. Also, dvpMon over JTAG uses an L2 boot. After the normal JTAG L1 boot program runs, dvpMon uses it to download its own loader. This program is called `jtagMonitor.out`. Its sources are in the tools/host target package the binary resides in the `bin` directory. Note that this loader takes some memory and that is why the starting address for dvpMon JTAG boot is typically close to 1M.
5. The TriMedia Compilation System (TCS) typically provides some boot code customized for each SoC, as described in the "SoC Orientation" document found in the TCS portion of the document tree. You can refer there for more information. Unless you instruct the TriMedia linker to skip this feature, programs constructed using the TriMedia compiler (TCS) include a set of boot code that can be found in a location like the following:

```
Nexperia\TCS5.11\libdev\src\targets\pnx1500\lib\startup
```

6. The program then jumps to `main()` which is typically initializes pSOS before calling the `root()` function as the entry to the user's application. There is a document in the pSOS section of the API reference document that gives details about the TriMedia implementation of pSOS. Rather than implementing the `root()` function directly, Nexperia programs use the `tmMain()` macro to perform some important initializations before entering user code.

You might be interested to know that the implementation of the `main()` function is defined in the file corresponding to the following:

```
Nexperia\TCS4.61\OS\pSOS\pSOSystem\configs\sysinit.c.
```

Application programmers do not typically edit this file. Experienced pSOS programmers might be surprised to learn that the pSOS driver and BSP models are not used at all. pSOS is treated as a simple embedded OS.

7. One of the initializations performed by `tmMain()` is to invoke the Board Support Library (BSL) to install a set of function pointers that are appropriate for this hardware. Whenever a new board is created, someone has to provide the BSL for that board. How to do this is discussed in [Section 4, tmMain and the Nexperia BSL](#) and in other parts of this document.

2 The Boot Script

The boot block on the Nexperia chip executes the boot script, or L0 (el-zero) boot. The CPU is not involved. The design of the pnx1500 suggested that an on chip boot script could be used, but hardware problems have precluded this. Today's Nexperia chips require an off-chip boot script. While the hardware problem may be fixed in the future, every development board should have provision for an off-chip boot script.

The boot script can be generated in any number of ways. The most supported method uses a program provided as `bootscript.c`. This program is easily compiled with any C compiler and it generates a binary image of the boot script.

2.1 Hosted Boot – The minimum boot

The hosted boot can also be seen as the minimal boot script. Reference to the following:

```
NDK5.4\prod\mptk\apps\btscriptgen\src\btscriptgen.c
```

shows you that the minimum boot script sets the clock rates. It might also write the Ethernet MAC address into some registers (hence the boot script is unique to each board and contains the MAC address). In hosted boot, the boot script will not initialize all of the PCI apertures because the PCI host does that.

2.2 JTAG Boot

The JTAG boot is the next most complicated boot script. After initializing the clock rates, etc., it does write the PCI apertures to reflect the size and location of memory. We do not discuss some of the small details that may also be initialized by any boot script. You can read those in the code. The JTAG boot script copies a program from the EEPROM into DDRAM, points the CPU's program counter there, and then starts the CPU.

2.2.1 The JTAG L1 program

The source for the small JTAG loader program is found at a location like the following:

```
NDK5.4\prod\mptk\apps\btscriptgen\l1jtagMon\src
```

This is an effective example of how to build a very small TriMedia program. This JTAG loader is used by all of the tools that download over JTAG. You could also use it as a prototype of any other L1 (el-one) program that you want to make very small and load directly from the EEPROM. Notice that it does not use any standard IO or any other library functions. Note that EEPROMS larger than 2k require an extra byte at the start of the boot script.

2.3 Flash boot

Compared to hosted or JTAG boot, booting from flash is a relatively involved process. The next chapter is devoted to this. The following can serve as a brief introduction.

The L0 boot contains code to move the L1 boot loader from flash into memory. The flash boot sequence uses the Boot Flash File System (BFFS) to organize its data files on the flash memory. Chapter 5 has more information.

The flash boot procedure supported with NDK 5 is conceptually similar to what has been used previously. But it is simplified and extended to NAND flash as well as NOR flash.

2.4 Changing the Boot Script

Most system designers who make their own board should expect to provide a boot script customized to their own hardware. Some of the things that are configured in the bootscript include:

- Board ID (see BSL chapter)
- Type and configuration of DDRAM
- size of PCI aperture
- Size of SDRAM (SDRAM used interchangeably with DDRAM here)
- Clock speed of the CPU and DDRAM
- Memory layout (addresses) of memory and peripherals
- MAC address for PHY or other unique board identifier

Some of these things (like SDRAM_BASE and SDRAM_LIMIT) can be changed at the command line of the example boot scripts. Others (like the DDRAM controller configuration) just need to be edited in the C file.

2.4.1 Changing the memory size:

Most of the examples refer to systems with 64M. In case you are building a system with a different size, here is a summary of what you have to change:

1. You have to change the SDRAM_LIMIT in the bootscript program.

2. You also have to change the PCI setup, e.g.:

```
scriptWriteSingle(0x1be40010, 0x01d60e83) ; // PCI_SETUP - 64MB
#else
scriptWriteSingle(0x1be40010, 0x01d60e03) ; // PCI_SETUP - 32 MB
```

3. You have to change the parameters you use to download your program, for instance with `dvpMon`. Change the `memLimit` and the `cacheLimit`. Set the cache limit to memory size -64k or so.
4. If you are lowering the memory size, you also need to set the memory size in the makefile for the `I1jtag` program. Look for `SDRAM_LIMIT` in the makefile in the `I1jtag` (or `I1jtagMon`) directory below the `bootscript` directory and rebuild the `I1jtag` program. If you do not do this, the JTAG boot program will fail to run as its stack will be up at 64M while your memory ends at 32M.
5. Should you want to lower the memory size below 32M, there is another change you need to make. This change goes in your BSL. Each BSL has a define something like this:

```
#define TMBSL_LCP_TOTALRAM_SIZE 0x2000000 // Total system RAM size
```

Even though the comment says “Total system RAM size,” it really means “minimum” size, as explained in the comment for the `tmbslBoardGetBootInfo` function. The default minimum is thus 32 MB. For instance, with 16M or RAM, you need to change this to `0x1000000`.

2.4.2 A note about the MAC address

Systems that use the on chip MAC (or for that matter, any MAC) need to have a unique identifier somewhere. By convention, the boot script writes the MAC address into the on chip MAC’s registers. This allows the MAC address (a unique board identifier) to be stored in the boot script. By convention, the MAC address is stored in the first few bytes of the EEPROM, and one of the tools commonly used for EEPROM programming (`jpgmLcp`) allows you to specify this as a command line parameter. This is designed to be a “production friendly” convention.

3 Flash Boot

A Boot Flash File System (BFFS) is provided with the Nexperia Developers Kit to support booting from flash. A common question is why not just write the program in flash? The BFFS is not really much more complicated than this. It is described in the OS API reference section of the docs. See the [Booting and Board Support APIs](#) volume, [Chapter 1](#) for more information. This booting procedure and the BFFS meets the needs of systems in production, including compression, data storage, and in field upgrade.

The program “`exProductionBurn`” is your gateway to this entire process. Its makefile builds everything necessary for flash boot. Running `exProductionBurn` formats and initializes the flash. The makefile for `exProductionBurn` contains a lot of comments that are critical usage information. After reading this document, please look carefully at the comments in the `exProductionBurn` makefile.

3.1 The boot sequence from flash

In brief, the boot sequence is as follows:

1. After reset the external flash boot script in the eeprom loads exL1Boot from flash into RAM and launches it. The L1 boot is restricted to be less than 8KB. The exL1Boot resides at location 0 of the flash memory (in the boot record), and it is loaded at offset 0x00100000 (1 MB) in RAM with the end address at 0x00200000 (2 MB). The start address is required for internal bootscripts; it is not required for external bootscripts, e.g., from EEPROM's.

Note: Neither the PNX1500 nor the PNX1700 can reliably boot from their internal boot scripts. Since you must use an external bootscript, some of these restrictions could be relaxed, if there were a good reason. One of these restrictions is the size of the L1 boot code. It should be as small as possible but 8K is not a hard requirement. For NAND it could be as large as 15.5K and for NOR it could be as large as 127.5K based on the NAND and NOR parts we have used to date. At the end of the first block we store the 512-byte boot record. Since NAND blocks are 16K and NOR blocks are 128K this explains the (15.5K and 127.5K) limitations.
2. The exL1Boot loads the exL2Boot into RAM and launches it. The exL2Boot is typically loaded in the middle of RAM. If your board has 64M, the offset 0x02000000 (32 MB) in RAM with the end address at 0x03fa0000 is appropriate. If you have a different RAM size, it is correct to change this. The exL1Boot finds the L2 boot using the last block of the boot record where exProductionBurn places it.
3. The exL2Boot reads the BIF (Boot Image File) of the main application, decompresses it, checks the CRC, loads it into RAM, and executes it. The simplest example you can put into flash is the HelloFlash application. It reports to `stdout` (usually via `jcon`) the type (NAND or NOR) and size of the flash ROM it finds.

A Boot Image File is created by building your application program as a .mi (memory image) file. The BIF may be compressed with a header that describes the contents of the file. The exL2Boot loads the application at offset 0x00001000 with the end address at 0x03fa0000.

Note: With the exception of the load address for the exL1Boot, the other addresses are arbitrary. The internal bootscript, which doesn't work, is designed to always load the exL1Boot at offset 0x00100000.

3.2 Flash Boot EEPROMs

The flash boot process must be done using an external flash boot eeprom. The code for the flash boot eeprom is contained in the "unified" boot script generator program:

```
Nexperia\NDK5.4\prod\mptk\apps\btscriptgen
```

This program was discussed earlier in this document.

3.3 exL1Boot: Tiny initial boot

The code for the exL1Boot is hardware-specific. The version for the LCP1500, LCP1700 and similar boards is in this file:

```
.../NDK5.4/prod/mptk/apps/exL1Boot/src/exL1Boot.c
```

The main purpose of exL1Boot is to prepare for and launch the exL2Boot loader application. The exL1Boot performs the following functions in the order specified:

1. Perform initialization to access the appropriate flash (NOR or NAND) media.
2. Locate exL2Boot and determine its size.
3. Read exL2Boot code into memory.
4. Flush data cache.
5. Clear any interrupts.

6. Clear instruction cache.
7. Return start address of exL2Boot to the caller which will cause exL2Boot to start execution.

The exL1Boot code (and the rest of the boot code) has an important debugging facility. A series of values are written to the JTAG DATA_OUT register to show the progress of the boot. These are easily monitored using a program like MDS' pcijtag.exe or usbjtag.exe. This is a most effective debugging method.

Because the internal boot scripts do not work, the limits on the size of the exL1Boot binary are set by the size of a sector in the flash.

The main reasons to modify exL1Boot would be to accommodate changes in the hardware configuration such as the presence of LED's or other required peripheral initialization. The L1boot is otherwise relatively constant in function.

3.4 exL2Boot: Full featured boot loader

An example implementation of exL2Boot is found in the NDK:

```
.../NDK5.4/prod/mptk/apps/exL2Boot/src/exL2Boot.c
```

It may be modified to match the application's needs. For instance, you might modify it to display a "splash screen" as early as possible at bootup. The example programs (exQvcp) for the QVCP device library might be a good place to start on this project. Another example is in field upgrade. The products that have implemented this using the BFFS did it by extending the L2Boot program.

The main purpose of exL2Boot is to prepare for and launch the final user application. It performs the following functions in the order specified:

1. Initialize the tmBffs for read access.
2. Using the tmBffs, open the BIF (Boot Image File), typically named "sys.bif", and located in BFFS partition 1 in the flash memory.
3. Read the BIF header. For a description of the BIF header (tmBffs_ImageFileHeader) see:

```
.../nexperia/NDK5.4/prod/mptk/comps/tmBffs/inc/tmBffs.h
```

4. Process the BIF header.

Although the exL2Boot is architected to support several executable types (static or dynamic, relocatable or nonrelocatable) the exL2Boot only supports non-relocatable static images (*.mi files).

Several BIF options are possible (compressed or not, encrypted or not, with or without digital signature), but the exL2Boot only supports compressed or uncompressed.

- a. If the executable is compressed, allocate memory from the heap and read the compressed code into memory. Then decompress the compressed code from the heap into the starting address for the application (specified by header.sdramBase). The compression/decompression library is zlib. The sources for it can be found in


```
.../nexperia/tcs4.61/examples/compression/zlib
```
- b. If the executable is uncompressed, read the uncompressed code directly to the starting address for the application specified by header.sdramBase.
- c. If memory was allocated for decompression free it.
5. Close the BIF file.
6. Do a CRC check on the executable, comparing it against what was in the BIF header when the BIF was created.

7. Launch the application. This consists of the following steps:
 - a. Clear the BIS (Boot Information Structure).
 - b. Flush data cache.
 - c. Clear any interrupts.
 - d. Clear instruction cache.
 - e. Jump to the application's start address in `header.sdramBase`.

The `exL2Boot` never exits because the new application has taken over the processor.

3.5 exProductionBurn

The build process for the `exProductionBurn` program packages constructs an image of the flash in the release directory and packages it into `exProductionBurn.out`. When you run the `.out` file, it burns that image into the flash. There are many options; they are described as comments in the makefile for `exProductionBurn`:

```
.../NDK5.4/prod/mptk/apps/exProductionBurn/makefile
```

The makefile for `exProductionBurn` takes care of building everything in order.

3.6 Related resources

Various related documents and applications exist in the NDK that are helpful and even necessary to develop standalone booting applications.

1. The tool to test the BFFS for both NAND and NOR flash is in the following:

```
prod\mptk\comps\tmBffsIoDriver\tst\tmBffsIoDriverTest
```

It operates in two modes: batch and interactive. In batch mode all BFFS API functions are tested on the flash media and speed tests are performed. The interactive mode allows you to browse the contents of the BFFS partitions, the data file system (`tmFlashFS`, `TargetFFS`), and the host file system with Unix-like commands.

2. Documentation for the BFFS (Boot Flash File System) is in the OS API section [Booting and Board Support APIs](#) volume, [Chapter 1](#).
3. Please do read the comments in the makefile for `exProductionBurn`.

4 tmMain and the Nexperia BSL

As mentioned above, standard C programs start with `main()`. pSOS programs start with `root()`. This is one detail of application construction that is hidden in `tmMain()`. Nexperia programs start with `tmMain() { }`. `tmMain()` is a macro. It expands to the right program entry point, depending on the OS and processor in use. Further, `tmMain` initializes the Nexperia kernel. This includes the debugging trace library (`tmDbg`, not to be confused with `tmdbg.exe` which is the debugger program), and the Board Support Library (BSL).

`tmMain` is interesting in its own right. It can easily be customized. Definitions, such as `tmMain_DBG_BUFF_SIZE` which controls the size of the `DebugPrint (DP)` buffer is often overridden in an application. The approach taken with `tmMainStreaming` is another valid use of `tmMain()`. `tmMainStreaming()` is a clone of `tmMain` in which the initialization has been extended to cover initialization of the TSSA system. The Mbox project has its own version of `tmMain()` where even more project specific initialization is hidden. This approach is appropriate when you have a

collection of applications, all of which need the same initialization to function correctly. Placing this initialization in a customized `tmMain` means that the applications do not have to be changed when the `init` sequence changes.

It can be important to note the order of execution in `tmMain()`. Look into `tmMain.c`. More on this in the sections on audio and video BSL's.

Although you will see code in `tmMain()` to initialize the BSL, in the NDK 5.3 release this actually happens before `main()` is called! The call in `tmMain()` returns immediately because it has already been initialized.

4.1 Nexperia BSL Overview

Nexperia uses a Board Support Library to abstract details of the board-based hardware away from user applications. When the BSL is properly deployed, any of the standard Nexperia demo applications should be able to run on a user's board without modification to the source code. Similarly, a properly constructed user application should be able to run on the reference hardware. Of course, there are "issues" to be addressed when reaching toward this goal. This chapter attempts to make you aware of the capabilities of the underlying mechanisms so that you can use them effectively.

Why BSL?

The traditional OS driver mechanism provided by pSOS seemed unsuited to use with audio and video drivers. The designers also wanted to be able to include drivers for more than one board in an image. The BSL mechanism serves these needs.

Board Support Theory

At its root, the BSL is just a mechanism to retrieve a set of data based on the identification of one board or another. It uses a simple registry to do this. More than one BSL can be linked into a program. At system initialization, one of these is identified as matching the hardware. The "registry" is then used to retrieve the data appropriate for this board. That data commonly takes the form of function tables that are used to implement a simple form of run-time linking. Some of the common Nexperia libraries (e.g. audio and video) define standard structures that are used in the BSL. But nothing stops any user from creating his own entries in the BSL.

Further, the BSL provides an access point to initialize all of the things that vary from board to board. This can include details of memory and interrupt usage.

Identifying the board

Each BSL includes an "activate" function. When the BSL is initialized, the activate function for each board in the image is called in turn until one of them returns with no error. The activate function must be able to uniquely identify its own board. The subsystem ID in the boot EEPROM is typically used to make this identification. When the activate function has been completed, the board is "registered", meaning that the BSL's registry has been populated with keys and values that are appropriate for this board.

Example – Using the BSL for video

The activate function will typically include a call to a function like `myBoard_VideoInit()`. Reference to one of the examples provided will show that functions like `tmbslVencAnaRegisterComponent()` are called. This particular function records a pointer in the BSL's registry with a key known to the `VencAna` component and a value corresponding to a function table as required by `VencAna`.

4.2 Creating your own BSL

You create your own BSL by copying and customizing an existing BSL. An approach to doing this is given here, step by step. Note that the BSL concept has been deployed beyond the scope of TriMedia and pSOS. You may see reference to this in some places, but this discussion does not cover that scope.

We recommend that search all the files in the entire copied BSL directory tree and replace the example's name with your own name. For example, search every file for "mdsbslLcp1500" using a non-case-sensitive search. If you don't do this, you will miss changing files you might not think need to be changed, for example the file "diversity.mk".

Each of the following steps will be covered in more detail in the next sections.

1. Copy an existing BSL component. Start from a working example.
2. Name and identify your board (hardware ID and global names).
3. Initialize on-chip resources.
4. Initialize off-chip resources.
5. Test.

For this document, the new board being created is a PNX1500 (Nexus) based development board. The example board's name will be abbreviated from "Nexus Development" board to "Ndev" or "NDEV" board.

4.2.1 Copy an Existing BSL

NDK contains a number of BSLs that can serve as an example. The two canonical references have traditionally been the LCP1500 and the NREF. In NDK 5.3, the LCP BSL was updated to support both the pnx1500 and the pnx1700. At the time of writing, this is the best available example BSL (...**NDK5.4\sd\os\osinfra\comps\mdsbslLcp**). By the time you read this, other boards may be common as references. Each of these BSL's contains similar code for activating (detecting, initializing and registering system information) the board. Check the different boards available and pick the one that is closest to the hardware configuration of your board. Or alternatively, point out to the hardware designers how much simpler the software will be if they exactly copy an existing board. To the extent that the same off-chip components and memory configurations are used, the same BSL can be used. Any of these values can be changed, but it's faster and simpler to start with a source board that is intended for your target chip and similar to your target board.

1. Copy the entire source `tmbssl<board>` (`tmbsslLcp`) directory to a new directory in your project tree. Do not put it in the NDK tree. For this example, the new reference board will be called `tmbsslNdev` so you create `Nexperia/myProject/comps/tmbsslNdev`.
2. Rename all of the files in the subdirectories accordingly (`tmbsslLcp.c` -> `tmbsslNdev.c`, etc.).
3. Update the makefile `Nexperia/myProject/comps//tmbsslNdev/makefile` to reflect the filename changes.

You'll need to update the makefile again later based on any components added or removed from the board files.

Notice that the reference board has a subdirectory for `tm_pSOS`. It may also have subdirectories for other target CPUs or OSs. These are typically named "`<cpu>_<os>`" such as `tm_pSOS` for the TriMedia CPU and pSOS OS, respectively. This application note does not cover any case other than `tm_pSOS`.

Note – A possible shortcut

This document discusses the "right" way to create a new BSL. When you follow this method, your BSL can co-exist with that of other boards and you can use reference boards to validate your software and isolate the cause of problems. But there is a shortcut that has to be mentioned. You can skip the renaming activity and just use the BSL of the reference board. This should work fine if your board is indeed identical to the reference. You start to enter a gray area when you modify the reference BSL without changing its name or ID. This can be confusing.

4.2.2 Board Identification and Naming

Nexperia allows multiple boards to be included in a single OS binary image. This is done to help developers whose clients may be running more than one board. When you create a board, you must identify it. On the surface, this ID has two aspects. One is the name of the component in the SDE2 directory tree. The other is the "board ID" typically found in the EEPROM. As you dig deeper, you find other aspects that are covered here.

Naming the component

Put the component in your own project directory and name it something reasonable. For the purposes of discussion, we'll call it `tmbs1Ndev`.

Naming the files and functions

Do a search and replace of the function names and change the file names. Change the makefile. Changing all of these things allows your BSL to be linked into the same image as the original reference without linker errors. You might note the use of static variables in the BSL code to limit the scope of names. Variables declared static do not have to be renamed.

The Board ID

A 32-bit number stored in the EEPROM typically identifies a board. This number has two 16-bit parts. There is a "vendor ID" and a "subsystem ID". The PCI SIG technically grants the vendor ID. The PCI vendor ID of Philips and MDS are visible in the code base. The subsystem ID is unique to a board. To choose the ID for your board, use the following Q&A.

Q – Will this board be plugged into a PCI slot, or is it "standalone"?

If it is standalone, you can pick any number that is different from the reference boards with no problems resulting. The only possible conflict you could ever see is if you build a development image having two BSL's that share an ID. This might happen if you copy the reference BSL and do not change the ID.

If the board will be plugged into a PCI slot, answer the next question.

Q – Is the PCI system "closed" or "open"? That is, can customers plug in their own arbitrary boards and more important, could another TriMedia based board be plugged in?

Closed – If the system is basically closed, then the situation is as above. As long as you do not conflict with a reference board, you are free to do as you please. Since the board ID process looks both at the chip ID and the vendor ID, this discussion only applies to chips like the PNX1500 and PNX1700, as supported by `tmman`.

Open – It is only if the system is open that you must really be careful when choosing a board ID. Then you must be sure that your ID is different from that of any other device that might possibly be plugged into your system. Of course the most likely sources of problems are the TriMedia reference boards. So just like above, be sure you do not copy their ID!

Exported Macros

A number of macros are defined in each BSL and must be set reasonably. These are summarized in the following table:

Table 1:

TMBSL_BOARD_ID_VARIABLE	The activation function of your BSL will be entered into a table (by the linker) which is checked by the <code>tmbslCore</code> module. The position in the table is determined by this macro. When more than one BSL is linked into an image, you must ensure that no two BSL's use the same entry. The table appears to have nine entries and the valid keys are listed in <code>tmbslCore.h</code> . They are of the form: <code>tmbslMgrBoardActivateOfBoardId0x10001131</code> .
TMBSL_BOARD_ID_NUMBER	This <code>#define</code> must match the board ID in the EEPROM. Note that if you use an on-chip boot script, the board ID must be written into the Boot Info Structure (BIS).
TMBSL_BOARD_ID_STRING	This is the board name as an ASCII string with a maximum length of 16 bytes including the Null terminator. Let's call our example "NexusDev."

```
#define TMBSL_BOARD_ID_VARIABLE tmbslMgrBoardActivateOfBoardId0x10001131
#define TMBSL_BOARD_ID_NUMBER 0x12345678 // Ndev board ID
#define TMBSL_BOARD_ID_STRING "NexusDev"
```

System RAM Layout

The BSL has facilities to identify RAM size, but these are not usually used with stand-alone TriMedia systems. Instead, the RAM size is determined when the `.out` file is relocated to become a memory image, either by `dvpMon`, `tmdbg.exe`, or directly by linking. This structure is found in the `CPU/os` specific subdirectory, in a file like

```
NDK5.4\sd\os\osinfra\comps\mdsbslLcp\src\tm_posos\mdsbslLcp_0sSpecific.c.
```

4.2.3 CPU and On-chip Hardware Initialization

The BSL always includes a call to the BSL of the SoC that is the CPU. In this document, that is the `pnx1500`, or now the `pnx1700`. You'll notice that a `tmbslCoreUtilities_t` structure is initialized with functions that pertain to the CPU.

Since the `pnx1500` and `pnx1700` are pin-compatible, it is common for a board to be able to be used with either of these. This section describes how this is implemented.

A component called `tmbslTmOnly` (...**NDK5.4\sd\os\osinfra\comps\tmbslTmOnly**) exists to serve as the placeholder for the 1500 or 1700 BSL. This component can be used in different ways.

- In NDK 4 – only the `pnx1500` was supported and you typically included `tmbslPnx1500.h` and called it directly. The `tmbslPnx1500` component included sources.
- In NDK 5 – the `tmbslPnx1500` does not contain sources. Instead, it refers to `tmbslTmOnly` for the sources. If you include `tmbslPnx1500`, you have a backwards compatible solution for the 1500. But in fact, the `tmbslPnx1700` component refers to the same sources. The difference is that they are compiled for the different core based on macros defined by the build in `tmFlags.h`. With this method, you typically have two BSL's, such as `mdsbslLcp1500` and `mdsbslLcp1700` that must be maintained in parallel.
- You should not write your BSL to refer directly to `tmbslTmOnly`.

4.2.4 Off-chip Hardware Initialization

For off-chip hardware that needs to be detected and/or initialized, code should be added to either the `tmbslBoardHwInit` or `tmbslBoardOsSpecificInit` functions. The choice of which function to place your initialization code in depends on whether your hardware initialization is standard for all target OS's or it is different for each target OS:

<code>tmbslBoardHwInit</code>	Is shared for all target CPU/OS combinations – Initialization can be performed within it to ensure that the hardware is in a known state independent of the target OS. Most hardware initialization would fall into this category.
<code>tmbslBoardOsSpecificInit</code>	Is unique for each target OS – This allows the function to be customized to add or remove some initialization that would not be appropriate for all target OS's.

Off-chip hardware that might require initialization before use could include:

- ROMs connected through the XIO aperture,
- GPIO pins dedicated for specific uses (interrupts, control lines, etc.), or
- Devices used for debugging purposes (external UARTs).

Prior to adding your hardware initialization code, it's a good idea to go through the source files and remove any code from the example board (`tmbslNref` in this case) that is unused for your board. This will result in cleaner source files and prevents possible errors or hangs due to invalid hardware accesses by code that no longer applies to the new board.

4.2.4.1 BSL Structure

BSL's are typically organized in a few files. This is not technically necessary, but it makes it easy to find things. This table lists the major files and tries to give you an idea of what to expect in them.

File name	Purpose or contents
<code>tmbslNdev.c</code>	Contains the main entry points to the BSL. Typically covers all codes that are not audio or video. IDE, PHY and Flash tables (which might be different on your board) are here.
<code>tmbslNdev.h</code>	Defines the board ID (which might be different on your board).
<code>tmbslNdev_aud.c</code>	BSL code for audio stuff (which might be different on your board).
<code>tmbslNdev_vid.c</code>	BSL for video stuff (which might be different on your board).
<code>tm_pSOS/tmbslNdev_IntDrv.c</code>	Holds table of interrupt related functions that are not usually changed between the prototype and your customer board.
<code>tm_pSOS/tmbslNdev_OsSpecific.c</code>	Defines memory layout and a function table with pointers to basic utilities that do not typically change when you create a new board.

4.2.4.2 Modifying the Basic BSL

Refer to following for modifying the basic BSL:

- `midsbslLcp1500.c`
- or –
- `tmbslNref.c`

You will notice these defines do exclude things. We have them. We do not argue whether they are useful or necessary.

IDE

The IDE code on TriMedia requires a BSL. The BSL enables you to use the on-chip IDE interface, or a BSL can be written to use an external IDE (or e.g. SATA) chip. The on-chip IDE is accessed through the `tmbslIdeXio` peripheral.

The IDE BSL table would have two entries if you wanted to use a master/slave IDE arrangement. This has been tested, but you might want to be aware of some issues if you think you want to go in this direction. ATAPI (used in DVD and CD drives) is more complex than ATA used in hard discs. A master or slave transaction must be atomic. Mixing ATA and ATAPI commands requires some code that is not present in the IDE device library in NDK 5.3 or earlier. If you want to use these features, ask tcs-help.com for the latest information.

Notice that the BSL specifies which interrupt pin and which address is used on the board. If you change these values from those demonstrated, expect to carefully test and debug the affect of those changes.

PHY

The Ethernet MAC on the 1500 requires an off-chip PHY. The example boards use National Semiconductors' DP83847. Other PHYs can easily be used. The interface to the PHY is apparently rather standard now and the PHY BSL provided for the DP83847 will also work on other PHYs (e.g., Micrel KS8721BL) if you remove the DP83847 ID check.

NOR Flash

The early BSLs for the 1500 contained a BSL for NOR flash. This is not used by the BFFS (boot flash file system). Also, it is not used by the most recent implementation of Blunk's flash file system on the 1500. That goes directly to the hardware. It is not strictly necessary to register the NOR flash interface in the BSL, but the BSL must at minimum set the XIO profile (with `tmhwXioSetProfile()`) for NOR flash if present.

NAND Flash

The current code includes a `tmbslNand` component that is supported and used by the BFFS.

```
tmbslBoardActivate(), tmbslBoardHwInit() and tmbslBoardRegister()
```

Implementations of these three functions are present in each BSL. As mentioned above, the activate function is used to identify the board. You can see that the typical implementation relies on common helper functions rather than reading the ID directly.

Note: The BSL is called before main using the TCS chaining mechanism. This is invoked in `... \NDK5.4\sd\os\oshome\comps\tmbslCore\src\tm_pSOS\tmbslOsSpecific.c`

with the line

```
Int __custom_boot[] = {0, (Int) tmbslMgrInit};
```

Because the `BoardHwInit` function is called before `main()`, you cannot use OS functions. Since the `BoardHwInit` function is called before the `tmDbg` library is enabled, you can use `tmbslCorePrintf()` to write to the DP buffer. You cannot use `DBG_PRINT`, as the `tmDbg` library is not yet initialized. This also prevents you from using other high level libraries like `dll2c` and `tmOsal`. As much of the init of peripherals as possible should be postponed to the point in the application where that peripheral is addressed. At that point, all OS and debug features are available. In case you need to make some I2C accesses here, `tmhwI2c` is available. It is a low level method of accessing the I2C hardware appropriate for these circumstances. Remember that `hwI2c` has no OS dependency and hence is not protected for multi-threaded operation. Use it only when necessary, and when you are sure that no other task will be accessing the I2C hardware.

4.2.4.3 Audio BSL

The audio portion of the BSL is typically found in a separate file (`xxbsl_yyy_aud.c`). This is customary, but not necessary. As with any BSL, the audio BSL consists of data structures and function tables that are used by a device library. The BSL is not called directly by applications. Applications call the device library which calls the BSL. There are four audio device libraries that use BSLs:

- `tmdlAo` – I2S style audio output
- `tmdlAi` – I2S style audio input
- `tmdlSpdo` – S/P DIF output
- `tmdlSpdi` – S/P DIF input

The I2S format audio inputs and outputs have more useful settings than the S/P DIF BSLs. This is because the SPDIF IO's are usually completely on chip, while the I2S connected AI and AO ports use codecs that often are configurable.

In each case, the code that calls these BSLs is readily accessible. It is in the device libraries. In case of any question, refer immediately to the device library. There are certainly things in the BSL that there for legacy reasons or no longer used.

Supported Types

For each of the BSLs there is a group of macro definitions that are appropriate for the board. The SPDIF variants are not typically used. For audio input and output, these macros describe which formats of data are supported. For example, you see:

```
// Audio Unit 0
#define SUPPORTED_AO0_TYPES          (atfLinearPCM)
#define SUPPORTED_AO0_SUBTYPES      (apfMono16 | apfMono32 | apfStereo16 | apfStereo32)
#define OUTPUT_AO0_ADAPTERS        (aaaGeneric)
```

This is where you state that your board can support only stereo and mono operation, in either 16- or 32-bit format. If your board can also support six-channel output, this is the place to say it. On the PNX1500 or PNX1700, any board that supports stereo can also support mono. The device library and audio renderers take care of the details. Still, you must declare this support in your BSL.

tmbslAoConfig_t and its ilk

The AO BSL includes a structure (representative of other parts of the audio BSL) of `tmbslAoConfig_t` type. Notice there are `init`, `term` and other functions that are each called in specific places by the device library. If your audio hardware comes right up in the right state, you can make these `null`. If your AO hardware must be initialized by I²C writes, this is the place to do it. The audio BSL functions are invoked from the application when it starts up the audio services. This is typically well after the OS services were initialized, so you can use OS services. This has not always been true in the past, so some legacy BSLs rely on the `hwApi` interface to avoid reliance on the OS. In fact, this is dangerous for the audio inits because `hwApi` calls are not thread safe.

There are entries for things that are not supported on the 1500 and 1700. The TSU pins meet that description. There are entries that depend on your hardware. The oversampling clock factor is one of these. So is the "serial master" entry. To be sure of how any of these entries is used, please refer to the corresponding device library's source code.

The audio renderer/digitizer refers to the BSL via the device library to know its maximum and minimum sample rates.

When you create the BSL for your board, you will want to (need to) fill in these tables with code and data that is appropriate to your board.

About using I2C to initialize audio components

Earlier we noted that device libraries cannot be used in the early BSL init because the BSL is called so early in the boot process. This in turn means that you cannot use `tmdI2c` to initialize I2C peripherals from the activate calls. This is interesting to note, but generally not a problem. The code to initialize I2C audio peripherals should be placed in the appropriate audio init function in the audio BSL. You can look into the AO device library (`tmdIAo`) source code to see that the BSL's init function is called from the `InstanceSetup` function. This is always called after basic system initialization.

Examples and test programs

You can use example programs such as `exoIArendAo` and `exoIAio` to verify your board. There are also device library level examples such as `sd\avs\aproc\comps\tmdIAo\ex\exdISineAo` that can be used to test. It is generally good practice to first test AO, then test AI because the AI tests depend on AO.

- If you have trouble getting the audio tests to come up on your board, you can run a program like `exdISineAo` under `dvpMon` (or the debugger) and then open up a memory window on the AO (or AI) MMIO register space. Then poke in values until you find a set that works. Knowing the desired values, you can trace back how to force the libraries to write these values for you.
- Allowing the TriMedia to be the clock master of the AO and/or AI devices allows you to rely on the DDS's to implement sophisticated sync algorithms. For instance, when the clock source is SPDIF input, you have a choice. You can let the DDS be the AO clock source vary the clock frequency using the DDS controlled by the `Ap11` component to match the clock of SPDIF input. Or you can let the DAC be the clock master (fixed output clock) and spend a couple dozen MIPS to run the asynchronous sample rate converter. Using the DDS saves a few MIPS.

Configuring the audio DAC

In some previous releases of the Nexperia software you might find descriptions of controlling the audio DAC for mute or volume control functions. This functionality is not currently tested or supported in the device library. If you want to use it, you should choose whatever method seems most reasonable.

4.2.4.4 Video BSL

The video portion of the BSL is typically found in a separate file (`xxbslyyy_vid.c`). This is not necessary. As with any BSL, the video BSL consists of data structures and function tables that are used by a device library. The BSL is not called directly by applications. Applications call the device library. There are two video device libraries that use BSLs:

- `tmVencAna` – video encoder, used on output
- `tmVdecAna` – video decoder, used on input

As noted in the section about I2C usage in audio, the I2C initialization should be done using `tmdI2c` and it should be done in the init function for `VencAna` or `VdecAna`.

Video Output – VencAna

If your board has video out, you should expect to fill in the `tmbslVencAnaConfig_t` and `tmbslVencAnaExtConfig_t` structures. These describe the video output hardware. (See [Chapter 9 Analog Video Output BSL: tmVencAna](#) in the *Video IO Components* volume for more information.) With NDK 5, some slight updates are made to the capabilities of this system. These are made to

simplify applications and other control code that needs to support different kinds of hardware. The deployment of DVI and HDMI in modern video systems drove this change. A requirement of the change is that an existing BSL must still work. See the release notes for more information.

The VencAna interface is designed to be used by the application to control application level aspects of the off-chip video output hardware. This is traditionally an "encoder", but the VencAna interface is applied to control whatever is out there, even if it is an LCD or HDMI interface chip.

Referring to the BSL for the LCP, you can see a case where three output devices are supported. Each of these has its own functions. Alternatively, the BSL for the Nref board has only one such entry. Either way works, as long as it is consistent. In recent work, a more sophisticated method is being deployed on the MDS DMA board using exoIVideoSimple.

Unlike VencAna, the VencAnaExt interface is designed to be used by the video driver, such as VrendGfxVo.

What belongs in the VencAna BSL?

The BSL should report what "adapters" are available on the hardware. In previous releases, the "encoding type" attempted to do this job. The adapter type describes which output jack is being addressed.

The BSL should report which video modes are supported by each adapter.

The BSL should set the VDI/VDO routers. The application can also do this, and in legacy applications this was the case. But as boards deploy more sophisticated video out systems (including dual output using FGPO), the BSL becomes the one place that knows all of the correct settings.

The BSL should report the settings necessary to make the QVCP talk to this device.

And of course, the BSL must be able to configure the external hardware to work in the selected configuration.

Video Input – VdecAna

If your board has video input, you should expect to fill in the `tmbs1VdecAnaConfig_t` and `tmbs1VdecAnaExtConfig_t` structures. These describe the video input hardware. The VdecAna interface also supports a VBI section. (See [Chapter 8 Analog Video Input BSL: tmVdecAna](#) in the *Video IO Components* volume for more information.)

If your board has no video decoder, but you want to use the on chip test signal generator as a video source, you may still need to fill in the VdecAna function tables. Unlike the audio function tables, VdecAna does not accept null function pointers.

Testing the video BSL

The canonical video test program `exoIVideoSimple`.

4.2.4.5 Extending the BSL definition for new components

Support for components in the BSL is built up from a simple basis. You may decide that your new component needs to be configured in a way that is really board specific. In that case, the information should be found in the BSL. The assignment of GPIO pins might be a good example of this.

The `bslCore` component contains the interface used to support this situation. You 'register' the component in the BSL and 'look up' the registered information in the client component. The information to be provided by the BSL is defined by agreement between the BSL writer and the client component. The component owner defines the necessary data structure in an `.h` file. The

data structure desired by the client component is identified with a registry key. This is typically the component's ID (CompID). The BSL calls the register function, `tmbslCoreRegisterComponent`, and the client calls the get interface function, `tmbslCoreGetInterface`.

It is appropriate to use this feature when you find yourself maintaining a component that is going to be used on multiple boards. Just remember to code proper behavior in case the client library finds no entry by this name. There will certainly be some legacy BSL's around that exercise this path through the code. One appropriate example is to raise an assertion forcing the user of the code to understand the new requirement on the BSL. If you do this, be sure to document what is required in a comment near the assertion! If some default value is appropriate, that might also be okay. But then the user of this library may never learn that the BSL is supposed to be updated.