

TriMedia Manager and Embedded Linux

NDK 6.1

Application Note v0.81, 10 July 2009

Introduction

The TriMedia Manager (tmman) driver allows system designers to pair a (Linux) host processor (such as an x86 or an ARM) with one or more TriMedia “slaves” being used as a media processing accelerators.

The tmman driver was developed for use on Windows hosts, and was first ported to Linux years ago. The development of tmman for embedded Linux has advanced recently. tmman is best seen as an open source driver that has been ported to numerous OS over the years. This app note will help you understand whether it has already been ported to your OS, or else what work you will have to undertake to port it yourself. This application note describes the tmman driver and its usage to enable a Linux host to control a PNX1xxx TriMedia chip.

The TCS tools documentation page includes an API reference for the user library that is exported by the tmman driver. This API reference [[link to book 9 of TCS docs](#)] provides quite a bit of information useful to system designers who will be using the tmman driver. Please do refer to it.

1 About tmman

The tmman driver enables communication between a host (for example, Linux ARM) and a target (TriMedia). The main functions of the tmman driver are:

1. As a kernel driver, it reserves host system resources, and enumerates the TriMedia SoCs present on the PCI bus at load time.
2. It provides an API to download and run programs on the TriMedia.
3. It provides a relatively rich API to support interrupt moderated communication between the host and the target programs.

The tmman driver assumes a PCI connection between the host and the target, and free access to the entire memory space by both the host and the target. In the past, tmman was ported to systems that disregard these assumptions. When these assumptions are violated, more parts of the driver system must be modified to achieve the functionality required. This is possible, but it is generally beyond the scope of this document.

The tmman driver consists of a kernel mode driver and a user mode library (tmman32). Most users access the TriMedia from the user mode. The “host_comm” library and tmCrt (C Run Time) library implement functions that rely on tmman and hence these are often considered with tmman.

The `host_comm` API is used to implement the `tmCrt`. `tmCrt` is a remote procedure call server which runs on the host processor. It accepts client requests to run Standard IO functions (`printf`, console IO, file IO) for the target on the host. TriMedia programs can thus use `printf`, `fread`, and `fwrite`. This function is distinct from `tmman`. Many helpful facilities of the TriMedia/NDK system are enabled by this access to `stdio` in a hosted system.

The sources for `tmman` have traditionally been delivered with the TriMedia Compilation System (TCS), and a version of the sources are found in the TCS/`tmComm` directory. This version is functional for basic DSP development, but more recent sources are available for use in embedded systems. The most recent source is always available from the TCSHelp support site. When each NDK release is made, the latest sources are packaged with NDK.

`tmman` assumes that each processor has relatively full access to the memory of the other processor. `tmman` makes the entire DRAM visible to host whereas it only makes the shared memory region of the host memory visible to TriMedia CPU. The communication API available on each side of the bus is approximately symmetric.

On the Linux host, the `tmman` kernel driver code is provided under a Lesser GPL so that it can be integrated with kernel. The user libraries and the TriMedia side of the code is not under a Lesser GPL license.

For more information on the `tmman` driver, refer to the `tmman` user manual in the TCS package.

1.1 Features supported by `tmman` driver

The basic features of `tmman` are detailed in this section.

Kernel driver

1. Enumeration/resource allocation

The `tmman` kernel driver identifies various memory apertures of the PCI devices already enumerated by the operating system and maps these into the virtual memory space(s) of the operating system.

2. Interprocessor synchronization

The `tmman` kernel driver allows interprocessor interrupt generation and handling. This is used to construct a messaging API.

3. Shared memory

Three types of “shared memory” are used.

- A very small block of memory is used by the core `tmman` driver to support the API. This uncached memory must be visible to both processors. The `tmman` code assumes this memory, used for mailboxes, does not require software cache management. This small block is meant to communicate pointers to larger areas of memory that may be cached. This block is actually only 1k, but it is allocated from the same block of memory as the next type.

- `tmman` supports access to a “shared memory” block. The kernel driver allocates this uncached contiguous memory and a `tmman` API allows users to access it. The amount allocated is typically 64K and it is typically contiguous with the 1k required by the driver. With Linux, the amount can be controlled by configuring the kernel. See other parts of this document that give details. The Windows driver also gives some control via a registry entry, but this contiguous shared memory is usually a limited resource.

- Memory allocated on the host by calls to `malloc` from user space can be shared using the `SGBuffer` feature of `tmman`. The `SGBuffer` feature looks behind the virtual addresses of the user space memory to find the underlying physical addresses. The page table is involved. The

physical memory that underlies an allocated block of virtual memory is not typically contiguous. Instead, a “scatter-gather” list describes the various blocks of physically contiguous memory. tmman provides an API to convert user space memory into such “SGBuffer” memory. The slave processor can then access the list of physical memory segments and copy data to or from it. DMA can be used in this process. Since this results in a series of DMA transfers each with size 4k, this transfer mechanism may be less efficient than shared RAM. But it can be available.

User driver

1. tmman32 enumeration and control API's

Programs on each side of the bus need to know the address of memory on either side of the bus. These addresses can be retrieved using function calls. On the host side, an API is also available to download, start and stop programs on the TriMedia.

2. Messaging

tmman provides a queued messaging API. The recipient of the messages registers an “on receipt” notification callback. The message uses the shared memory for temporary storage. Messages are typically short and can have specific structures. The recipient is notified by interrupt.

3. Shared memory management.

The kernel driver gives access to continuous shared memory and SG buffer memory. The user driver exposes an API by which user space applications can access this memory.

HostComm / tmCrt

The hostcomm and tmCrt (C Run Time) libraries are utilities that are commonly associated with tmman. They are technically distinct. This introduction is provided to help you understand where they fit in.

1. argc and argv handling

Programs started on TriMedia can pass arguments using the traditional argc and argv mechanisms.

2. Remote file IO (host file system)

Programs running on TriMedia can access the host file system.

3. Console IO

Programs running on TriMedia can access a console on the host.

4. Remote socket IO

Programs running on TriMedia can make socket calls remotely on the host.

For more information on tmman features, refer to the tmman user manual.

1.2 Reference Ports of tmman for Linux

NXP “fully supports” tmman running on a number of processors and operating systems. tmman is portable, and it can be ported to others. NXP has the hardware and software to test on the reference systems.

x86 Linux using RedHat Enterprise Linux 5 is used as a reference. RHEL 4 is also supported. Fedora Linux has traditionally been close to this. Ubuntu has traditionally been more different.

NXP uses a Marvell 88F5182 ARM system as a reference port of tmman with embedded Linux. Going away from x86 forced the engineers doing the port to solve a number of issues. A new “tmload” utility replaces tmrun when the TCS libload library is not available. tmload handles memory images, not .out files. tmload also serves as an example to download and start the DSP. The source code can be used for reference to create host side embedded applications as per specific application requirements. How shared memory is handled is also different from x86 Linux.

This port to ARM is considered the “flagship” embedded port. NXP has most experience with this port. If you port to other embedded processors running Linux, NXP will often refer to this port as your reference.

NXP is also working with a Motorola PowerPC reference port. The PowerPC runs in big endian mode while the TriMedia runs in little endian mode. This presents unique challenges in the port.

1.2.1 Features of the ARM port

The following examples

The features supported on ARM Linux reference port are:

- Messaging
- Shared memory API
- Interprocessor Synchronization
- C Runtime library
- tmload utility (along with the x86 timage utility)
- tmbring up tools
- tmbringup/dpdump: Utility for dumping debug buffers. Valid only for NDK applications.
- tmbringup/tddump: Utility for dumping timing information. Valid only for NDK applications.
- example (go,memory,message,tmapi)
- examples/dmatest (using NDK driver for UDMA)

The features not supported on ARM Linux reference port are:

- examples/tminterrupt
- tmgmon or tmmon
- tmbringup/pcisdram

1.2.2 Examples

The following examples are supported, tested and available in the directory `$TCS/tmComm/src/examples`.

go: demonstrates how to download the PNX executable on PNX (functionally the same as tmrun(x86)/tmload (ARM))

message: demonstrates use of tmman messaging API.

Memory: demonstrates use of tmman shared memory API.

dmaTest: You may need to contact TCSHelp for the latest, but we can demonstrate UDMA based data transfer.

ttmapi: demonstrates messaging, shared memory and other API usages. Useful for testing all essential APIs of tmman.

All examples directories contain a Readme.Linux file which provides the required information to build and execute the respective examples.

1.3 Using tmman on other processors

Use the latest version of tmman to port to other host processors. Unless you are using exactly one of the supported processors, you should see the bringup of tmman as a porting effort. The port may be more or less complicated depending on how similar your processor is compared to the references. We have seen other ARM ports that were quite complicated. Often the driver must be adjusted to live with the kernel port provide by the processor vendor. There is no substitute for following a checklist to ensure that each part of the port is actually working.

As always, contact the TCSHelp system to obtain the latest version. As we see the code run on other processors, we update the master copy with what we have learned. The support engineers at TCSHelp may have some very helpful advice.

The PowerPC is a rather special case as it usually runs in big endian mode. A TriMedia usually runs in little endian mode. Hence, the tmman code has provisions to run in a cross-endian mode. The host can be big endian while the TriMedia runs little endian. The necessary byte-swapping is handled by the drivers. This feature was not tested in recent tmman releases, but as of this some activity is now underway.

An Oxford OXE810 platform uses an ARM9TDMI core with a PCI interface. The limited nature of the PCI interface on this chip makes it unable to support a normal tmman port. The OXE810 host can map only 8M of memory on the PCI bus, and hence cannot address the entire PNX memory. It is possible to make the two processors talk to each other, but this violation of the assumptions of tmman results in non-tmman behaviour. However, it is possible to make tmman work on OXE810 by enabling limited RAM (only 4MB) using a workaround in PCI module of kernel and modifying the script for PNX device.

As noted in the tmman API documents, mapping multiple large blocks of memory (eg, 256M DDR) can be hard on the OS. As more and more people use the system in this way, we are using various strategies to address the problem.

The “Viper/Mips/VxWorks” port was a unique port as it ran on a multi-core SoC. The shared memory issue was handled differently as both processors worked out of the same DRAM. This port is no longer supported.

It is also worth mentioning a number of processors that have been the subject of support discussions.

- Micrel ARM, with a PCI bridge that does bus address translations.
- Intel Atom, much like x86

1.4 Dependencies of the tmman system

This section describes the hardware resources available on PNX system and how the tmman driver accesses the resources.

1. MMIO Memory Region and host access:

In hosted mode, the host system requires access to the memory register to read and write values to it. The host system normally accesses a memory map from the PCI sub system and configures registers by writing the addresses to registers like TM32_DRAM_LO and TM32_DRAM_HI. MMIO.

Access to the memory region is also required to start, stop and interrupt the DSP whenever required. The size of this region is 2M. The host Operating System must enumerate and enable access to this memory region.

Linux Note:

At boot time on x86 Linux (as well as on ARM), the kernel enumerates this PCI region and assigns a physical address range. The tmman driver then accesses this physical address and makes this region available in the kernel address space.

It is required for tmman driver to map MMIO address space in user address space (mmap). This memory region must be visible as non-cached memory in kernel mode as well as in user mode.

2. DRAM memory region and host access:

In hosted mode, a host can read and write to the entire DDR address space of the PNX device. Communication features in hosted mode depend on this support. The size of this region can vary from 64M to 256M depending on the PNX card. The communication is dependent on host access to the DRAM memory region, hence it is necessary for the host to enumerate and enable this memory region. Failing to enumerate or enable this region effectively disables most of communication features of tmman subsystem.

Linux Note:

At boot time on x86 Linux (as well as on ARM), the kernel enumerates this PCI resource and assigns a physical address and size. The tmman driver then makes this region visible to kernel, and maps this region in applications user address space. This memory region should be visible as non-cached memory in kernel mode as well as user mode.

3. PNX access to host memory:

You can configure the PNX card to access section of host's memory (System RAM). Typically the tmman driver allocates contiguous memory on the host side. This memory, referred to as shared memory, shares the data structure between host and target for communication. The tmman driver configures the registers of PNX card so that this memory is visible to TriMedia. Access to this memory must be uncached else communication features may not work.

Linux Note:

On x86 Linux driver allocates 65K of memory using `get_free_pages` calls and accesses the memory physical address. The driver configures access for PNX card to this region by writing to appropriate MMIO registers of the PNX.

It is important that the target and host processors do not cache this memory region. Typically on x86 architecture, hardware handles the cache coherency. However, this may differ for other platforms such as ARM and PPC.

2 Desktop (x86) Linux port

The desktop version of tmman is traditionally developed and supported only on RedHat Linux enterprise Linux RHEL 4 and 5. This code continues to be supported and is tested as a reference in the embedded tmman.

The TCS libload library is supported on x86. The library is provided in binary form only. tmman uses API exported by this library to download .out files on PNX. TCS 5.2 distributes a reference tool tmrun to download and execute PNX .out files.

On some x86 systems, PCI interrupts are incorrectly marked as edge triggered. To fix this issue, enable `pci=noacpi` to kernel boot parameters. See appendix A for more information about configuring the kernel.

The kernel mode driver source uses macros to distinguish between the kernel of RHEL4 and RHEL5. To build `tmman` for x86 other than RHEL, such as UBUNTU, you must identify the relevant macros in source code, and enable sections of code appropriate to the source version of the kernel installed on respective system.

3 Marvell ARM port

This section details the build, installation and features of the Marvell ARM port.

The TCS 5.2 Linux `tmman` is ported on ARM Linux for Marvell's ARM 88F5182 chip.

3.1 Prerequisites for porting `tmman` to Embedded Linux

To port `tmman` to embedded Linux, you must meet the following requirements:

- Flexible shared memory management: An x86 has hardware cache coherence and the shared memory is stored in x86 RAM. Other processors do not have this functionality, hence the shared memory management must be more flexible.
- The TCS loader library (LIBLOAD) is not usually provided as source. It is provided in binary form for x86, but when the embedded host processor is not x86, provisions must be made to load memory images, not relocatable `.out` files. You cannot use `tmrun` on embedded platform as it depends on LIBLOAD library. Instead, use `tmload` to download and execute the PNX executable. `tmload` requires memory images.
- GUI download tools like `dvpMon` and `tmgmon` are not available. Instead, we use command line utilities referred to as the “`tmbringup`” utilities. The utility sources are also provided with TCS.
- Reference embedded Linux `tmman` port does not use macros.
- On embedded platform, the host applications are responsible for downloading the PNX executable. Refer to [Embedded Linux porting issues](#) for more information. A package that addresses these issues is now available in NDK 6.x and to customers who request it on TCSHelp (www.tcshelp.com/support.html). This updated version of `tmman` for embedded Linux continues to support operation with an x86 host as well as an ARM host.

Modifications to the `tmman` driver ported on ARM Linux

The `tmman` driver code requires the following modifications to work on this platform:

- *Shared Memory allocation and user space mapping changes*

The `tmman` subsystems on host and target require access to the shared memory region. On x86 this shared memory is allocated using `get_free_pages` calls. Since this memory is accessed by the PCI bus when target accesses it, x86 architecture maintains cache coherency. This does not apply to ARM, and it requires a different method of memory allocation. For more information, refer to [Appendix B: Shared Memory: Importance, Location and Strategies](#). ARM Marvell `tmman` port uses `dma_alloc_coherent` API for this allocation.

By default, x86 uses `nopage` method for (mmap) mapping allocated shared memory into user address space. The `nopage` method is not suitable for memory allocated using `dma_alloc_coherent` API. The `trimedia_mmap` is modified accordingly for shared memory mapping in user space. For more information, refer to the `trimedia_mmap` method in `tmif.c`.

- *MMIO and SDRAM user space mapping changes*

The `tmman` driver uses `ioremap_nocache` calls to access MMIO and SDRAM. It also maps this memory into user space before the first use of the device, using `remap_pfn_range` call. On x86 it maintains cache coherency, but on ARM the memory region is cached. The `trimedia_mmap` code is modified and a `pgprot_noncached` call is made before a `remap_pfn_range` call. The `pgprot_noncached` call makes memory non-cached for the user `vma`. For more information, refer to `trimedia_mmap` method in `tmif.c`.

- *Retrieving a physical address*

The x86 `tmman` code uses the function `virt_to_phys` to retrieve the physical address of supplied kernel virtual address. This function does not work on the ARM port as this function is not authorized to call on the memory allocated using the function `dma_alloc_coherent`. Changes are made accordingly to retrieve the correct physical address. For more information, refer to `rtal.c`.

Embedded Linux porting issues

1. *Build issues:*

The current driver is compiled and tested on RHEL4/5 Linux hosts. The driver kernel module is compiled and linked with Linux kernel v2.6, and the user mode module is linked with `glibc` library available on the RHEL Linux platform with x86 `gcc` compilation tool. For specific processor ports, parts of the build system may not work such as flags to `gcc`.

2. *Hardware resources (PCI resources):*

PNX1500/1700/1005 use PCI interface to communicate with the host system. The operating system on the target platform (for example, ARM Linux) must enumerate the PNX PCI device in PCI device tree. The OS PCI enumeration must also enable two memory regions (one non-prefetchable 2M size MMIO region, and one prefetchable 64-128M DRAM region) of the PNX device and its IRQ line. If the OS fails to do so, the installation of ported `tmman` driver will fail. To check if PNX is enumerated by the OS, use the `lspci` (Linux) tool to display the PNX device and its associated resources.

On some system (x86), it is observed that the interrupts are marked as edge triggered by BIOS/PCI code. This is incorrect as PCI interrupts are level triggered. After loading the driver check `cat /proc/interrupts`. Host target communication will not work unless interrupts are marked correctly. For more information refer to [Appendix A: Configuring Linux machines](#).

3. *Location of shared memory:*

The `tmman` driver depends on shared memory for communication. In the traditional x86 architecture, the shared memory resides on the x86 side of the PCI bus, and cache coherency is maintained by the x86 hardware. This may change for particular platforms. For example, shared memory implementation of ARM reference port differs from x86 implementation. Host communication will not work if shared memory is not accessible to any processor or it is cached by one of the processor. For more information [Appendix B: Shared Memory: Importance, Location and Strategies](#).

4. *Memory images:*

The `tmman` driver is modified to work without the `LIBLOAD` library. Host applications which use the `tmman` driver to load, start and communicate with DSP require static linking with `LIBLOAD` library. The `LIBLOAD` library provides a set of APIs which allows the host application to download the target executable (`.out` files).

Applications link LIBLOAD through static linking with the tmman32 user library. When a host application wants to download a target application, it uses tmman32 API tmmanDSPload.. This API in turn uses the LIBLOAD library calls for relocating the .out image on the target with board-specific memory map.

The LIBLOAD library provides an easy way to load .out files on target with the help of tmman, but it is tightly implemented on the x86 Linux platform. Users may need to use tmman driver and user libraries on other embedded systems such as ARM Linux and PPC Linux. In such scenarios, LIBLOAD porting is difficult and time consuming.

To help users implement all other functionalities of the tmman drivers, a new utility tmload is available. tmload is a small host side application which gets the handle of DSP and queries for resources such as the address range of SDRAM and MMIO. It then copies the target image in .mi format on the target and starts the DSP.

To generate .mi files, an offline utility tmimage is needed, which is available as part of TCS tools (x86). tmimage creates a relocated image (.mi) file after supplying board-specific parameters. For more information on use of tmload and tmimage, refer to the readme file available at `$TCS/tmComm/tools/tmload`.

These changes are not officially supported by TCS tools. Contact support at TCSHelp (www.tcshelp.com/support.html) to obtain the changes.

Note: The tmrun utility is available on the x86 platform but cannot be used on embedded Linux as it depends on the LIBLOAD library.

3.2 Building and installation

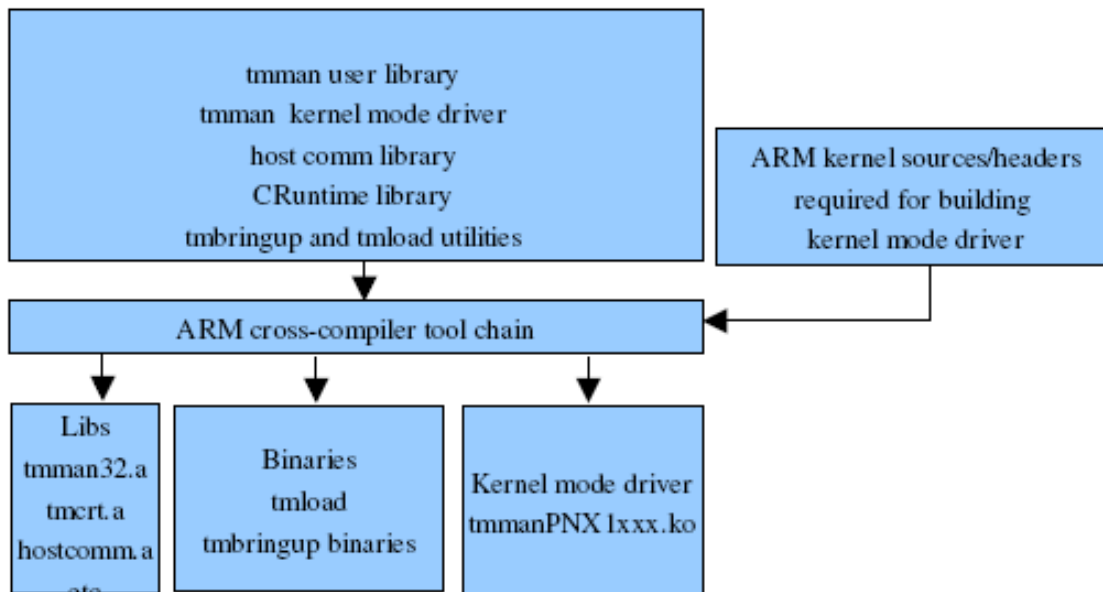
By default, the Marvell ARM port is not available with TCS 5.2 tools. To obtain a tmComm package, contact tcshelp (www.tcshelp.com/support.html).

To build the tmman driver and associated components using a tmComm package, follow the steps below:

1. Unzip the downloaded tmComm package to a new tmComm directory.
2. Replace the tcs5.2/tmComm directory with the new tmComm directory.
3. Refer to *uservars.sh* and *Linux.mk* files to build drivers and other libraries.
The driver and binaries build is located at BUILDTOP (*uservars.sh*).
4. Copy the kernel mode driver built for PNX card from BUILDTOP.
5. Install the script associated with the kernel mode driver on the board.
6. Load the driver using *insmod*.
7. Create an entry using *mknod*.

For example, `/dev/pnx1700`

Figure 1 shows how tmman and associated libraries gets built using cross compilation tool chain.



Building tmman libs,driver and binaries on build system.

Figure 1: tmman build using cross-compilation tool chain

3.3 Executing Application

This section details the procedure to create and load an executable memory image on any PNX series processor connected to an embedded Linux host, such as ARM and MIPS processors via PCI bus.

In a PC environment, the tools tmrun or dvpmom are used to download a relocatable .out file on PNX1xxx processors. These tools use the tmDownloader API to convert the relocatable .out file to a fixed address memory image, by reading the system parameters.

The tmDownloader API is a part of LIBLOAD, which is not available in source form with the TCS package built for the embedded processors. Users of the embedded systems can download the source code from the host using the pre-built memory images.

The tmload tool helps users obtain the system parameters of an embedded system, and download the .mi file (created using the system parameters) to the pnx1xxx processors in the system.

To create and load an executable memory image on any PNX series processor connected to an embedded Linux host, follow the steps below:

1. Copy the tmload executable for ARM on the target board.
2. Run `tmload -d0 -r` to obtain the system parameters of the platform for dsp 0.

To obtain the system parameters of more than one dsp, increment the value of `-d` option. The resultant output is displayed below:

```

# ./tmload -d0 -r
SDRAM LO = 0xE0000000
SDRAM size = 0x4000000
MMIO base = 0xFA200000
  
```

Shared Memory = 0x07048000

- Run tmimage on PC using the following parameters:

```
[root@sjolwstb4 bin]# $TCS/tmComm/bin/tmimage -R _TMMANShared=0xaabbccdd -mb
0xe0000000 -ib 0xfa200000 -ms 64 -o tstSdram.mi /home/tstSdram_ram.out
```

where:

-ms 64 is the size of SDRAM in MegaBytes.

/home/tstSdram_ram.out is the .out file to be converted

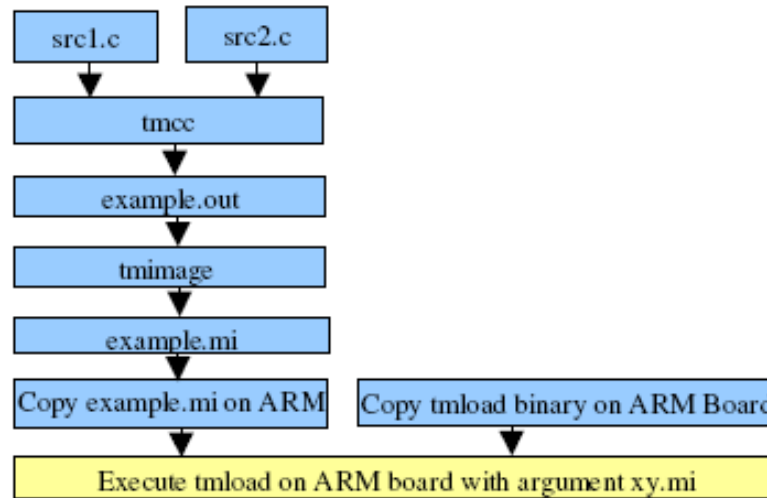
-o tstSdram.mi is the mi file which will be used to download

Note: Always use `_TMMANShared=0xaabbccdd` as it is referenced by `tmload`.

- Copy the .mi file on the target board and run `# ./tmload -d0 tstSdram.mi`.

This command executes the code on pnx1xxx.

[Figure 2](#) displays the PNX application build and load process.



Building and creating PNX .mi files and execution on ARM

Figure 2: PNX application build and load

4 tmman on other processors

This section details the procedure to port tman to other processors.

4.1 tmman bring up (tmbringup) tools

This section details the various utilities available to bring up the tmman driver on the system, and to identify and debug issues, if any, with the driver.

These utilities are available in the directory `$TCS/tmComm/src/tools/tmbringup`.

4.1.1 Building driver and tools

The source codes of the driver, libraries and tools are located in the directory `TCS5.2/tmComm/src` (`TCS5.2/tmComm/src/driver/tmman32/arm_linux`, `TCS5.2/tmComm/src/tool/tmbringup`).

To build the driver and tools, follow the instructions below and place the files in the directory `TCS5.2/tmComm/src`.

Install driver

1. Build scripts for all PNX1xxx devices at `TCS5.2/tmComm/src` using the default build driver. The build is located at BUILDTOP.
2. Locate the driver and installation scripts of the required PNX card at BUILDTOP.
3. Load the driver.

4.1.2 Using tmbringup utilities

You can build sources of tmbringup utilities using the default build. The binaries of these utility sources are available at BUILDTOP.

Check access to memory apertures

- Is MMIO space readable?

If the tmman driver is loaded correctly, then the MMIO registers are readable. Use the *tmreadmmio* MMIO register contents at offset 0x040040 (PCI Device/Vendor ID) to read the space.

The source code of *tmreadmmio* is located at `TCS5.2/tmComm/src/tools/readmmio`.

The specific PCI Device/Vendor ID values are:

- 0x54051131 for PNX15XX
- 0x54061131 for PNX17XX
- 0x540b1131 for PNX1005
- Is MMIO space writable?

Use *tmwritemmio* to check if you can write any value to MMIO register at offset 0x063600, and read back the same value. This is the SCRATCH MMIO register. It should act just like a 32-bit read/write register.

The source code of *tmwritemmio* is at `TCS5.2/tmComm/src/tools/writemmio`.

- Is DDR memory space visible?

Use *tmwritemem* and *tmreadmem* to check if you can write and read DRAM of the PNX card. Use *tmwritemem* to write and *tmreadmem* to read back the value.

The source code of these utilities is located at `TCS5.2/tmComm/src/tools/readmem` and `TCS5.2/tmComm/src/tools/writemem`.

- Verify information using *tmmandebug*

tmmandebug reports information about the memory map, and PCI information about the card. Verify this reported information with the information of the card.

The source code is at `TCS5.2/tmComm/src/tools/tmmandebug`.

Download and run a program

When you have access to the PNX, you can download programs to run on it. A TriMedia program compiled with `-host=Linux` must behave similar to a program compiled with `-host=WinNT`. You can also download a `.out` file and run on your target. Console and file IO work on the PNX.

Examples:

To compile a non-NDK hello world, use the following command line:

```
tmcc -target pnx1700 -host Linux hello.c -o hello.out
```

To compile a test that only writes to the scratch MMIO register, use the following command line. This program proves whether the processors are running or not.

```
tmcc -target pnx1700 -host nohost scratch.c -o scratch.out
```

When you download and run a program, check for the following issues:

- To run the `.out` file

On x86 you can use `tmsrun` to run the generated `.out` file. However, on embedded Linux, you must use `tmload`. For information on how to use `tmload`, refer to the readme file at `$TCS/tmComm/tools/tmload`

`tmload` downloads the `.mi` image and starts the DSP. `tmsrun/tmload` also sets up C Runtime server (`tmcrct`) such that `printf` and `getc` calls made on TriMedia are visible on the console. When you use `tmload`, the PNX should run and console IO should be directed to Linux command line.

- PNX does not run

The program does not start

Compile a simple TM program to write into the scratch MMIO register:

```
tmcc -target pnx1700 -host nohost scratch.c -o scratch.out
```

You can read the program counter of the TriMedia using `tmreadmmio` and check if it runs. Note that the TriMedia program compiled with **-host nohost** will not be able to run if `tmload` uses `CRuntime`. To properly load `scratch.mi`, disable all calls to `cruntime` in `tmload` source, rebuild `tmload` and then load `scratch.mi` on the board.

Run `tmreadmmio` in a separate console to read the content of MMIO register at offset `0x10004c`. This is `TM32_PC`. Its initial value is the start of RAM, but you should see a higher value within the DDR memory range. If the value umps out of RAM, it indicates the program was not properly relocated.

- To verify the program is downloaded correctly

Modify `readmem` to read back the downloaded image and verify the content.

- The program starts but `printf` fails

Check where the shared memory resides. On x86, the `.out` file has the symbol `_TMMANShared`. This is resolved when `tmman` downloads the `.out` file using DSP Download API. On an embedded system, `tmload` determines the signature (created by `tmimage` tool) and resolves it with correct value; it also displays the value of "Shared Memory". Ensure access to the Shared Memory address from host as well as from target side. Also check that the host and target agree on interrupt assignment, that is, they can interrupt each other.

4.2 Checklist for Bringup

Use this table as a checklist to guide your porting efforts. No test is likely to work unless all preceding tests also work. Find the lowest test on the list that fails and solve that problem before continuing.

Table 1: Checklist and guide for porting

Check	Title	Description	Proves
	Kernel and tools build	Using local Linux SDK	
	Kernel driver installs	Inspect the memory mapping reported by <code>./tmload -d0 -r</code> . Are these values sane?	
	tmmandebug	Check that tmmandebug reports good values	Driver is able to map memory
	Shared memory	Shared memory method is defined and handled. This can be quite tricky. Take the time to understand how it is solved in your port.	Required for communication
	Read MMIO	Prove you can read MMIO registers. Eg: <code>readmmio 0x040040</code> reads 0x540b1131 for PNX1005	Able to read MMIO registers
	Write MMIO, read back	Prove you can write to MMIO registers and read back. Eg: <code>writemmio 0x063600 47</code> <code>readmmio 0x063600</code> Should return what you wrote. Try a few things.	Able to write MMIO registers
	Read memory	Using <code>readmem</code> in the DDR space, see that you get different random values	Able to read DDRAM
	Write memory, read it back	Using <code>writemem</code> , write to some locations in the DDR space. Read them back. All correct? Create a more comprehensive memory test if you are not sure.	Able to write DDRAM
	Test console IO	Build simple TCS helloWorld program and see that <code>printf</code> works.	Shared memory is working. PCI interrupts are connected.
	Test trivial program	build scratch program, download it and run. Use <code>readmmio</code> to verify that scratch register is changing. Note that you build with <code>-host nohost</code> option and also <code>tmload</code> program should not use Cruntime library.	At Least TriMedia is able to run
	Test exNDK	Prove that a larger NDK program runs reliably. Retrieve DP buffer.	Full tool suite is working

4.3 How to report to TCSHELP

Table 2: Reporting to TCSHELP

Check	Title	Description	Required for
	Kernel version (uname -a)	This will display the complete version of kernel running on the platform.	Required for identification base kernel version.
	printenv at uboot prompt	This will show common environment variables of bootloader like bootargs, etc.	Required to understand how kernel is getting initialized with boot parameters
	uboot and kernel boot log	Boot log from hardware reset to kernel shell prompt.	Boot log shows most of the necessary information with which kernel is booted on the system such as total memory, any PCI resource allocation failures etc.
	cat /proc/meminfo	Displays allocate/used/free Memory resource on system.	Need to understand how memory like vmalloc (ioremap) is allocated/used and free on system.
	cat /proc/iomem	Displays memory map of the system.	This will display how System RAM, other devices has acquired memory resources.
	cat /proc/ioport	Display ioport usage of system.	
	output of lspci -vvv	Display PCI related info in human display format.	Useful for PCI related debugging.
	output of lspci -xxx	Display PCI configuration in hex format	Useful for PCI related debugging>
	Kernel's .config file	Configuration of the running kernel	
	Board details	Board details such as: which core, (PPC/ARM) core version ARM 11/ARM 9 etc, endianness of the host CPU, datasheet, schematics, etc.	
	Board details	Board details such as: which core, (PPC/ARM) core version ARM 11/ARM 9 etc, endianness of the host CPU, datasheet, schematics, etc.	

Appendix A: Configuring Linux machines

This section details the issues which can cause the tmman system to malfunction, especially in an x86 system. Some of the same issues may be noted in other cases, though the details will vary.

The driver may fail to load or the TriMedia program may appear not to run, due to the following issues:

1. Amount of memory

Several factors affect the amount of memory the TriMedia can access. These factors must be consistent for the TriMedia to run and communicate with the host. These factors include:

- The amount of memory installed on the board
- The amount of memory the bootscript declared
- The amount of memory mapped into Linux host space by the driver *tmconfig.c* (`tmComm\src\drivers\tmman32\x86_linux\PNX1005\tmconfig.c`)
- The relocation of the .out file to a memory image

2. Configuration of interrupt operation

Assumption: The (standard) bootscript is built to be compatible with the memory chips installed on the board.

The kernel determines the amount of memory available. The tmman driver specifies an amount of memory to be mapped in the file

`tmComm\src\drivers\tmman32\x86_linux\PNX1005\tmconfig.c`.

Edit the amount of memory to match your system and rebuild tmman. The Linux kernel may or may not be able to meet this request, as it is constrained by the parameters given to it at boot time. You can change these parameters in the grub.conf file, as described later in this section.

Choose to relocate a .out file into a memory image so that it uses less than the full amount of available memory.

3. Level or Edge Triggered Interrupts

Some BIOS may choose to make the TriMedia PCI interrupt edge-triggered. This will not work as it must be level-triggered. Use the grub.conf file to fix this. If the triggered interrupt is wrong, you will have no problem mapping memory and you will be able to run a program that does not use host communication. However, host communication will fail.

4. GRUB Configuration Commands

Assuming that your Linux installation uses the GRUB boot loader, the following information summarizes settings useful with TriMedia processors in Linux systems:

If you have Linux installed on a multi-core machine and `'cat /proc/cpuinfo'` shows only one CPU, ensure that the BIOS does not have "Software Single Processor Mode" enabled. You may also need to configure the kernel parameter `pci=nocpi` and `pci=nommconf`. For more information about configuring Linux kernel, refer to the following link (Appendix L):

<http://us.download.nvidia.com/XFree86/Linux-x86/1.0-9755/README/appendix-l.html>

Depending on your Linux machine (single core vs. dual or quad cores, installed memory of 1GB or more, different BIOS, mother boards), you may need to use the following kernel parameters in the grub configuration (`/boot/grub/grub.conf`) file:

- `vmalloc=256M` (parameter on kernel line)

Increases *vmalloc* size to 256M. This is especially needed if your system contains more than one TM board. If this is not set, the kernel driver may fail to map TM memory on the host. In such a scenario, typing 'dmesg' will display "*kernel: allocation failed: out of vmalloc space - use vmalloc= to increase size*".

- *uppermem 524288*

Some versions of the GRUB boot loader have problems calculating the memory layout and loading the initrd if the 'vmalloc' kernel parameter is used. Using the *uppermem* parameter is a workaround for a grub bug when using *vmalloc=256M*.

You can use the 'uppermem' GRUB command to force GRUB to load the initrd into a lower region of system memory to work around the problem. This will not adversely affect system performance once the kernel has been loaded.

Without the *uppermem* kernel parameter setting, you may see the following error message and boot will hang.

VFS: Cannot open root device "LABEL=/" or unknown-block(0,0)

Please append a correct "root=" boot option

Kernel Panic - not syncing: VFS Unable to mount root fs on unknown-block(0,0)

- *pci=noacpi* (parameter on kernel line)

This option specifies not to use ACPI to set up PCI interrupt routing, and is needed because some BIOSs will incorrectly cause PCI boards (including TriMedia boards) to use edge-triggered interrupts instead of level-triggered interrupts.

After installing TriMedia drivers successfully, check the */proc/interrupts* file to see if the interrupt associated with a TriMedia board is marked edge or level. It should not be edge. PCI standards says that PCI interrupts are level-triggered, and if this is configured wrong, host-target communication (all I/O) will not work.

- *pci=nommconf* (parameter on kernel line)

Additional support for Memory-Mapped PCI Configuration Space accesses is available for 2.6 kernels. However, this mechanism is not error-free, and the latest kernel updates are more careful about enabling this support. This option specifies that the kernel must not use memory mapped config space for PCI devices.

Depending on the user's mother board, BIOS and Linux kernel version, Dual core or Quad core machines may show up as single core machines. Setting *acpi=off* enables boot to proceed on multi-core Linux machines, but disables multi-core functionality. Hence, set *pci=nommconf* to allow the kernel to boot and enable multiple cores.

Summary

For dual core/quad core machines with memory of 1GB or more, the use of *uppermem*, *pci=noacpi*, *pci=nommconf* in */boot/grub/grub.conf* works accurately.

The following example displays a */boot/grub/grub.conf* file on tcslnx13, a quad core processor running RHEL 5.

Example grub.conf files:

GRUB configurations

```
tcslnx13 @ SJ0
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE: You have a /boot partition. This means that
```

```

# all kernel and initrd paths are relative to /boot/, eg.
# root (hd0,0)
# kernel /vmlinuz-version ro root=/dev/sda2
# initrd /initrd-version.img
#boot=/dev/sda
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title Red Hat Enterprise Linux Client (2.6.18-53.1.14.e15)
    root (hd0,0)
    uppermem 524288
    kernel /vmlinuz-2.6.18-53.1.14.e15 ro root=LABEL=/ pci=noacpi pci=nommconf rhgb
quiet vmalloc=256M
    initrd /initrd-2.6.18-53.1.14.e15.img
title Red Hat Enterprise Linux Client (2.6.18-53.e15)
    root (hd0,0)
    uppermem 524288
    kernel /vmlinuz-2.6.18-53.e15 ro root=LABEL=/ pci=noacpi pci=nommconf rhgb quiet
vmalloc=256M
    initrd /initrd-2.6.18-53.e15.img

pc67249004 @ EHV (core duo, RHEL5)
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE: You have a /boot partition. This means that
# all kernel and initrd paths are relative to /boot/, eg.
# root (hd0,1)
# kernel /vmlinuz-version ro root=/dev/sda5
# initrd /initrd-version.img
#boot=/dev/sda
default=0
timeout=5
splashimage=(hd0,1)/grub/splash.xpm.gz
#hiddenmenu
title Red Hat Enterprise Linux Client [ uppermem=524288 pci=noacpi pci=nommconf ]
(2.6.18-53.1.14.e15) -- JvdH20080410
    root (hd0,1)
    uppermem 524288
    kernel /vmlinuz-2.6.18-53.1.14.e15 ro root=LABEL=/ rhgb quiet noexec32=stack
pci=noacpi pci=nommconf
    initrd /initrd-2.6.18-53.1.14.e15.img
title Red Hat Enterprise Linux Client (2.6.18-53.1.14.e15) -- proven ok till 20080410
    root (hd0,1)
    kernel /vmlinuz-2.6.18-53.1.14.e15 ro root=LABEL=/ rhgb quiet noexec32=stack
initrd /initrd-2.6.18-53.1.14.e15.img
title Red Hat Enterprise Linux Client (2.6.18-53.e15)
    root (hd0,1)
    kernel /vmlinuz-2.6.18-53.e15 ro root=LABEL=/ rhgb quiet noexec32=stack
    initrd /initrd-2.6.18-53.e15.img
title Other
    rootnoverify (hd0,0)
    chainloader +1

```

Appendix B: Shared Memory: Importance, Location and Strategies

This appendix describes the relevance of shared memory for tmman. It also describes x86 Linux implementation and other strategies on embedded Linux platforms.

Shared memory is important for tmman to work. Both host and target access shared memory for communication such as messaging. Shared memory is also described in [Embedded Linux porting issues](#) and [Appendix B: Shared Memory: Importance, Location and Strategies](#).

At implementation level shared memory is divided into the following two sections:

- Shared Data: used for sharing the data structures such as object names
- Memory Block: used for general allocation like buffer for messages.

The important properties of shared memory are listed below:

- It is allocated per PNX card on the system.
- It must have physical contiguous memory.
- It must not be cached by any processor on the system.
- It must be able to map allocated memory in user space.

Check if x86 Linux implementation is suitable for that platform.

Note: The strategies are for reference and are not the only ways to allocate shared memory. You can always use platform-specific features and functions for such allocation, if allocated memory has properties listed above.

x86 Linux implementation

1. The tmman driver allocates two blocks using `get_free_pages`, one for “Shared Data” and the other for “Memory Block”. Default allocation is 1K+64K per PNX on system.
2. The .out file, the target executable built for host mode, contains the symbol `_TMMANShared`. The target side code refers to this symbol value for location of the host allocated shared memory. tmman32 with LIBLOAD (which creates relocated image as per PNX board memory map) determines the physical address of the “Shared Data” and assigns this value to `_TMMANShared`.
3. Cache coherency is ensured by x86 hardware on host side. On the target caching is not an issue as memory resides on the PCI bus.
4. tmman on Linux uses `nopage` (mmap) for it's user space mapping.

Other Shared Memory location strategies

The x86 implementation method is sufficient for x86 Linux system as cache coherency is ensured by x86 hardware. However for other hardware (such as ARM) the same implementation method may not work. Other Shared Memory location strategies available are:

1. Shared Memory location on host: Coherent memory allocation.
2. Shared Memory location on target: Using part of non-cache region of TriMedia DRAM.
3. Shared Memory on host: Re-using unmapped System RAM.

Shared Memory location on host: Coherent memory allocation

The tmman driver uses `pci_alloc_coherent/dma_alloc_coherent` calls to allocate coherent memory. Linux kernel exports these calls for the driver to use on almost all platforms.

Disadvantages: These calls are limited by the amount of total memory they can allocate. Check for platform-specific limitations. Allocated memory may not be directly mapped in user space using *nopage* method on x86. In some instances, the platform provides functions which can map this memory in user space (for example, on ARM, *dma_mmap_coherent*). On other platforms this function may not available and need alternate methods to implement.

Note: The reference Marvell port uses *dma_alloc_coherent* for shared memory allocation.

Changes required to default (TCS 5.2) x86 implementation:

- Modify *MmAllocateContiguousMemory* calls to use *dma_alloc_coherent* call.
- Modify *MmFreeContiguousMemory* accordingly.
- Modify *MmGetPhysicalAddress* so that it will return correct physical addresses.
- *nopage* method is not required to map memory in user space. Modify *trimedia_mmap* accordingly. (On ARM, *dma_mmap_coherent* provides functionality which maps allocated memory into user space.)

Shared Memory location on target: Using part of non-cache region of TriMedia DRAM

The TriMedia DRAM is divided between two sections, one cached and other non-cached, separated by offset *CLIMIT_OFFSET*. The region from *CLIMIT_OFFSET* and DRAM end address is not cached by TriMedia DSP. It is possible to use part of such memory for shared memory.

Advantages: This strategy does not depend on any support from the host operating system. Memory is guaranteed to be non-cached for host and target.

Disadvantages: Part of the *CLIMIT_OFFSET* and DRAM memory region is also used by some components of NDK. It is important to remember this detail while implementing this scheme to avoid any memory overlapping/corruption issues.

Changes required to default x86 implementation:

- Modify *MmAllocateContiguousMemory* calls so that it can allocate memory from *CLIMIT_OFFSET - DRAM END*. Note that this memory should be visible for host. A call to *ioremap_nocache* will make this memory visible to kernel.
- Modify *MmFreeContiguousMemory* accordingly.
- Modify *MmGetPhysicalAddress* so that it will return correct physical addresses.
- *nopage* method is not required to map memory in user space. Modify *trimedia_mmap* accordingly.

Shared Memory on host Side: Re-using unmapped System RAM

Use this strategy if other methods do not work for the platform. By default, the Linux kernel maps all the System RAM available (*/prop/iomem*). It is possible to send boot arguments to the kernel to map system memory to a specific size.

For example, if a system has 64M RAM, the kernel can map only 60 M by sending *mem=60M* as boot argument. The remaining memory (in this example 4 M) of known physical address can be used for shared memory location.

Advantages: There are no memory size constraints.

Disadvantages: Less system memory is available to the kernel even if the *tmman* driver is not loaded. This may not cause any errors on a system which has sufficient RAM.

Changes required to default x86 implementation:

- Modify *MmAllocateContiguousMemory* calls so that it can allocate memory from unmapped system RAM address. Note that this memory should be visible for host. A call to *ioremap_nocache* will make this memory visible to kernel.
- Modify *MmFreeContiguousMemory* accordingly.
- Modify *MmGetPhysicalAddress* so that it will return correct physical addresses.
- *nopage* method is not needed for user space mapping. Modify *trimedia_mmap* accordingly.

4.3.1 How to Get more Shared Memory

On a Linux system, there are multiple ways to allocate contiguous shared memory. Which method works best depends on the details of your needs and on the methods available on your processor and kernel port. You are forced to consider these methods when you need to transfer large amounts of data between the host and target. In this case, it is sensible to use the UDMA feature of the TriMedia processor to move data across the PCI bus. In order to do this, you need to have access to enough host buffer memory.

4.3.1.1 Methods of Allocating

Method A: *dma_alloc_coherent()* - this is the default if shared memory physical address is not specified while installing the module.

You can use *dma_alloc_coherent()* to allocate uncached, contiguous memory. This is supported on our reference platforms, but may not be supported in some systems:

Map this memory to user space using *dma_mmap_coherent()*. This is supported on our ARM platform, but not on x86 nor on PowerPC.

Free the memory using *dma_free_coherent()*. This is supported on our reference platforms, but may not be supported in some systems:

Pro:

- User space mapping in case of ARM platform is directly available.
For PPC and x86, the driver can implement *dma_mmap()* code to map pages to user space.

Cons:

- There is limitation on MAX memory that *dma_alloc_coherent* can allocate, on x86 this limit is 2MB.
- This memory pool is also used by other drivers, which may cause *tmman* driver to fail while allocating larger SHM.

Method B: *Restrict memory at boot time and ioremap()*

The kernel can be booted with "mem=MEMORY_SIZE" where MEMORY_SIZE is less than actual available memory. With this approach, physically contiguous memory is available but not mapped in address space. *ioremap()* and *ioremap_uncache()* kernel methods can be used by driver to map this memory and get virtual addresses.

Pro:

- You can obtain a lot of memory

Con:

- You may restrict the operation of the kernel

- Not all kernel ports support the mapping methods required.

Method C: `alloc_pages()`

The driver can request the kernel to provide free physical pages and these pages can be allocated from the DMA zone in the kernel so that they are uncached. Once these pages are allocated, they can be mapped into kernel and inturn mapped to user space.

o Not all kernel ports support this.

Method D: SG Buffers

The SGBuffer (Scatter Gather Buffer) approach is sometimes seen as an alternative. In this approach, the memory to be shared is allocated by user programs using a normal `malloc()` call. This range of virtual addresses must then be translated to a range of physical addresses to be used for DMA transfer. To be specific:

- Allocate buffers on host side, from user space using `malloc()`.
- Pass these buffer to driver using APIs() on host side.
- Driver shall find out physical addresses for these buffers.
- Buffers may not be physically contiguous.
- Driver shall lock pages(if needed) and then pass list of physical address (pointer) and size to application on TriMedia.
- Trimedia applications will initiate DMA transfer on these pointers.

Pro:

It is easy for the application to work with many large buffers. As we have seen, it can be complicated to allocate large amounts of coherent DMA memory.

Cons:

- SGBuffer memory may not be contiguous, hence the TriMedia's aperture1 must be opened over the entire host DRAM range. This can disable the RSE mechanism of TriMedia. Dynamic reconfiguration of aperture1 is possible, but complicated.
- Many small (4k) buffers leads to inefficient use of the DMA hardware. The transfer rate may be adversely affected.
- Even though some of the buffers in the SG llist may be contiguous, merging them is not supported in the driver. Adding this increases the complexity of the driver
- Even when the host kernel has some notion of an SG buffer framework, this may be adapted to the use of host side DMA. It may not be easy to reuse it for TriMedia DMA.
- Driver need to find out Kernel space Virtual addresses for the user space addresses, and then need to find out physical addresses, which later need to be passed to TriMedia's DMA. All these pages need to "locked" (protected from paging or swapping) before use for DMA.

4.3.1.2 Methods of Mapping to User Space

To map memory, you must obtain the physical address of that memory and then add page table entries to user address space.

Method A: Mapping memory obtained using `dma_alloc_coherent()`

The kernel for the Marvell ARM port provides the `dma_mmap_coherent()` method to accomplish mapping to userspace. The device driver's `mmap()` implementation must call the `dma_mmap_coherent()` method.

For x86 and PPC, the `dma_mmap_coherent()` method is not available. The kernel could certainly be extended and the ARM platform's `dma_mmap_coherent()` implementation can be used as a reference.

Method B: Mapping "Restrict memory at boot time and ioremap()"

Mapping this memory to user space involves two steps:

1. Mapping physical memory to "kernel Virtual address".
2. Mapping physical address to user space.

Mapping physical memory involves the usage of a "high memory" area for which kernel logical addresses are not defined, i.e there is no 1:1 mapping. Virtual addresses allocated falls under `VMALLOC_START` to `VMALLOC_END`.

The `ioremap()` method allocates VMA from this high memory and adds pages to page table. Physical addresses that were not previously part of the kernel memory map are mapped into kernel address space. The driver can use this address. Then user space applications can call `mmap()` to map SHM memory into userspace. The driver is responsible for implementing the `mmap()` call.

As a part of this implementation, the user space VMA that caused a fault is available to the driver. `ioremap()` based SHM mapping relies on `remap_pfn_range()`, where physical address is the `shm_addr` that was passed at the module installation time. (see section on `shm_phys_start`). `remap_pfn_range()` must associate a physical address with the userspace. Then an application in userspace can access the SHM using this virtual address.

4.3.1.3 Recommended approach:

Method A, `dma1_alloc_coherent()`, is the default if no action is taken. It is likely to be workable for smaller sizes of shared memory.

Method B (ioremap of memory reserved at kernel boot) is preferred when more memory is required and this approach is feasible

- boot the kernel with `mem=SIZE`
- pass `shm_phys_start` and `shm_size` while installing `tmman`.

SHM with `ioremap()` + DMA Transfer appears to be the solution that best meets the performance criteria. Because large buffers are available, DMA transfers are efficient. The kernel driver issues seem less complicated. And the PCI aperture¹ of the TriMedia is more easily controlled hence preserving the memory protection provided by the TriMedia's RSE mechanism.

So that user space host applications can retrieve virtual addresses, memory mapping for this region must be included in the `tmman` driver.

Approach Details :

- Assume total physical memory is N MB, and shm requirement is M MB
 - Boot Linux Kernel with "`mem=(N-M) MB`", This is to be passed from boot parameter to kernel from `uboot` environment.
 - Modify `tmman` driver to have module parameters `shm_phys_start` and `shm_size`
 - Driver can get the physical address and size from command line and map this memory using `ioremap()`
 - Map M i.e `sizeof(messaging area + SHM)` and pass the virtual address to user space. This shall be a part of driver's `mmap()` implementation.

- Once this this memory is available at user space, application can fill in data and ask TriMedia to initiate DMA to transfer.
- If `shm_phys_start` and `shm_size` are not specified, then driver can use `dma_alloc_coherent()` and allocate default 64K memory. (default mode of driver)

Appendix C: Kernel calls reference

This section briefly documents kernel calls referenced in this document.

1. *void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *dma_handle, int flag)*
*void *pci_alloc_consistent(struct pci_dev *dev, size_t size, dma_addr_t *dma_handle)*

Consistent memory is memory for which a write by either the device or the processor can immediately be read by the processor or device, without having to worry about caching effects.

This routine allocates a region of <size> bytes of consistent memory. It also returns a <dma_handle> which may be cast to an unsigned integer of the same width as the bus, and used as the physical address base of the region.

This routine returns a pointer to the allocated region (in the processor's virtual address space) or NULL, if the allocation failed.

Note: Consistent memory can be expensive on some platforms, and the minimum allocation length may be as big as a page, so you should consolidate your requests for consistent memory as much as possible.

2. *void dma_free_coherent(struct device *dev, size_t size, void *cpu_addr dma_addr_t dma_handle)*
*void pci_free_consistent(struct pci_dev *dev, size_t size, void *cpu_addr dma_addr_t dma_handle)*

These calls free a region of consistent memory previously allocated. dev, size and dma_handle must all be the same as those passed into the consistent memory allocated. cpu_addr must be the virtual address returned by the consistent memory allocated.

3. *unsigned long __get_free_pages(int flags, unsigned long order)*

This call is for the page-oriented allocation functions. It allocates free memory pages in kernel mode in the order of 2^{order} . It returns the kernel logical address of the first byte of the allocated area.

4. *void *ioremap_nocache(unsigned long offset, unsigned long size)*

This function allocates or frees a contiguous virtual address space. The parameter ioremap_nocache accesses physical memory through kernel virtual addresses. Memory access in kernel mode is uncached.

Appendix D: tmman implementation of mmap

This appendix describes various address types, and tmman implementation of mmap.

Address types and access

Linux is a virtual memory system - the address seen by any software component is always a virtual address and does not correspond to the physical address used by hardware.

The different address types used in Linux are listed below:

- User virtual address: The addresses seen by a user program
- Physical address: Address seen by hardware
- Kernel logical address: Addresses seen by drivers, kernel and kernel modules

Virtual Memory Area (VMA)

In Linux, the address space of each process is divided between memory regions called virtual memory areas (VMA). Each VMA is range of pages with identical permissions. Examples of vma are code segment, data segment, file-mapping.

Memory mapping

Devices are files on Linux, hence when the device memory needs access in user space, it requires a corresponding VMA and page table entries translating these addresses. The user application calls mmap system call to make this physical memory available in user mode, assuming drivers responsible for devices provide implementation of mmap.

Implementation of mmap in Linux

On Linux, mmap in the device driver can be implemented using the following methods:

- `remap_pfn_range` method
- `nopage` method

`remap_pfn_range` method maps the entire memory area by building page tables at once.

`nopage` method returns memory page for a particular address after its first access.

Implementation of mmap in tmman

When the tmman kernel mode driver initializes, it makes the following physical memory visible in kernel mode driver.

- DRAM
- MMIO

It also allocates memory for the shared memory region using one of the kernel memory allocation routines.

Shared memory

After initialization of driver, the tmman user library maps shared memory in user space. x86 uses `remap_pfn_range` user mapping method for physical memory, and implements `nopage` method for shared memory mapping.

Reference ARM ports use the `remap_pfn_range` method for all three memory mappings.

Revision history

Version 0.1: Created by Girish Pathak

Version 0.2: Updated introduction by C Peplinski

Version 0.3: Added Section on tmbringup tools

Version 0.4: Updated Section 6 with diagrams.

Version 0.5: Added Appendix 1,2 and 3. Added section about examples.

Version 0.6: Added Section about Desktop vs Embedded tmman.

Version 0.7: Edited by Girish Pathak and Sabita Rao

Version 0.8: Restructured document

Version 0.81: Chuck edited, added checklist, and added shared memory info from Nandkishor