Book 5—System Utilities

Part A:

Support Libraries

TriMedia

Version 2.0 beta

Table of Contents

Chapter 1 TriMedia Utility Functions

 ©1999 Philips Semiconductors 10/08/99

Chapter 3 TriMedia Component Manager API

Chapter 4 Clock Support API

Chapter 7 Programmable Interrupt Controller (PIC) API

Chapter 8 File I/O Drivers API

 © 1999 Philips Semiconductors 10/08/99

Chapter 10 TriMedia Flash File System API

 © 1999 Philips Semiconductors 10/08/99

Chapter 11 General Purpose Compression API

 © 1999 Philips Semiconductors 10/08/99

Chapter 12 Downloader API

Chapter 13 Dynamic Linking API

Chapter 14 TriMedia Manager API for Windows

 © 1999 Philips Semiconductors 10/08/99

 © 1999 Philips Semiconductors 10/08/99

Chapter 1

TriMedia Utility Functions

TriMedia C Library API Function Descriptions

The header file tmlibc.h contains function prototypes from the TCS standard C library libc.a that are not declared by the usual standard headers. This section describes those functions.

The header file which contains information on the debugging printf function, **DP** is called dprintf.h, and it resides in the same include directory as the tmlibc.h header file. The syntax for **DP** is similar to **printf**. **DP** maps to a minimally intrusive function which writes its string to a buffer in SDRAM. The contents of this buffer are retrieved from the host. Using this mechanism, **DP** can be called from time critical code like interrupt service routines. For more information, see Chapter 18, *Debugging TriMedia Applications Using JTAG,* in Book 4, *Software Tools*, Part C.

IMPORTANT
Always use the **DP_*** macros to do your "debug printf." Do NOT call the underlying functions. The **DP_*** macros accept the same arguments as their corresponding functions. Once you have finished debugging, simply recompile your src with the **–DNO_DP** option. All of the **DP_*** macros will be compiled out, that is, they become comments and do not impact on the final code size or execution speed at all.

 © 1999 Philips Semiconductors 10/08/99

tmAssert

```
tmAssert(
   Bool    condition,
   Int     ErrorCode
)
```

Parameters

`condition`	Assert when this condition is false.
`ErrorCode`	Error code to be printed, along with the file name and line number where the assertion was generated.

Description

`tmAssert` is a macro, defined in tm1/tmAssert.h. It is very similar to the ANSI standard **assert**. **tmAssert** is used extensively while bringing up programs that use the TriMedia Software Architecture (TSA) libraries. When the debug version of a library is used, an assertion can be generated on numerous error conditions. These conditions include invalid input and null pointers. An assert stops the execution of the current thread. Asserts are designed to quickly point out programming errors, rather than attempting to provide handlers for diverse error conditions.

Note
When **tmAssert** is triggered in a pSOS system, the **exit** function causes the current task to exit. The rest of the system may still be able to run.

_dcball

```
extern void _dcball(
   void
);
```

Parameters

None.

Return Codes

None.

Description

Copyback of the data cache.

Side effect: Copyback of entire data cache.

Note
This routine is non-interruptible.

 © 1999 Philips Semiconductors 10/08/99

_dclr

```
extern void _dclr(
   void
);
```

Parameters

None.

Return Codes

None.

Description

Clears the data cache.

Side effect: Clears entire data cache, discarding its contents.

Note
This routine is non-interruptible.

_dlock

```
extern  void _dlock(
   UInt32    address,
   UInt32    size
);
```

Parameters

`address`	Base address of block to clear.
`size`	Size of block to clear.

Return Codes

None.

Description

Locks the data cache.

Side effect: Locks **size** bytes starting at address in the data cache.

Note
Implements software workaround for hardware data cache locking bug.

 © 1999 Philips Semiconductors 10/08/99

_cache_copyback

```
#include <tmlib/tmlibc.h>
void _cache_copyback(
   void   *start_address,
   int     size
);
```

Parameters

`start_address`	Address of block to copy back.
`size`	Size of block to copy back.

Return

None.

Description

Copyback a range of memory (to SDRAM).

Side effect: copyback **size** bytes starting at **start_address** in the data cache.

_cache_invalidate

```
#include <tmlib/tmlibc.h>
void _cache_invalidate(
   void   *start_address,
   int     size
);
```

Parameters

`start_address`	Base address of block to invalidate.
`size`	Size of block to invalidate.

Return

None.

Description

Invalidates a range of memory.

Side effect: invalidate **size** bytes starting at **start_address** in the data cache.

© 1999 Philips Semiconductors 10/08/99

_cache_malloc

```
#include <tmlib/tmlibc.h>
void *_cache_malloc(
   size_t   size,
   int      set
);
```

Parameters

`size`	Request size of block to be allocated.
`set`	Desired cache set of the result (refer to warning below).

WARNING
Unlike the normal **malloc**, this function requires both the size and the set parameters. (If you do not know about cache sets, just pass –1 as second argument.) It is very easy to forget the second parameter, especially when tmlibc.h (containing the prototype) has not been included. Passing only one parameter will cause problems.

Return

A pointer to the allocated memory block.

Description

Allocates D-cache aligned memory block.

Note
If **set** is not **ANYSET (–1)**, **set % NSETS (32)** gives the desired cache set of the result. The requested size is rounded up to NBLOCK (64) multiple. The result can be freed with **_cache_free**, but not with the standard free.

_cache_free

```
#include <tmlib/tmlibc.h>
void _cache_free(
   void   *ptr
);
```

Parameters

`ptr`	Pointer the memory block to be freed.

Return Codes

None.

Description

Frees D-cache aligned memory block.

Note
The memory block must have been allocated with **_cache_malloc**, and must not be blocks allocated with the standard **malloc**.

 © 1999 Philips Semiconductors 10/08/99

_add_free

```
void _add_free(
   void     *ptr,
   size_t    size
);
```

Parameters

`ptr`	Pointer to block to be freed.
`size`	Size of block to be freed.

Return Codes

None.

Description

Add non-malloc'd memory block to memory free list.

Note
The memory block must not have been allocated with **malloc**.

_long_udiv

```
void _long_udiv(
   UInt    n[2],
   UInt    d
);
```

Parameters

n	64-bit unsigned dividend (first argument).
d	32-bit unsigned divisor.

Return Codes

None. The result is placed back into **n**.

Description

In-place 64/32-bit unsigned integer division as follows:

$$n = \frac{n[1] \times 2^{32} + n[0]}{d}$$

Implementation Notes

The only possible error is **d==0** (division by zero). However, similar to normal integer division in C there is no possibility of detecting this other than checking **d** before or after a call to this function. In case of division by zero, this function completes with n undefined. This function completes in a constant time of about 84 instruction cycles, or 490 cycles with cache effects taken into account. The corresponding numbers for 32-bit integer division in the TriMedia SDE are 65/250.

The parameter **n** has the least-significant 32-bits in **n[0]**, and the most significant 32-bits in **n[1]**. The 2-element **UInt** array is arranged in little-endian fashion.

Note
This is not to be confused with the individual element's endianness, which could be either big endian or little endian.

The (32*32) 64-bit integer multiplication can be performed as follows:

```
#include <custom_defs.h>
UInt a,b,result[2];
result[0]= a * b;
result[1]= UMULM(a,b);
```

It divides the first argument (array of (2) 32-bit unsigned integer) by the second argument (a 32-bit unsigned integer) and places the result back into the first argument.

 © 1999 Philips Semiconductors 10/08/99

TriMedia Types API Overview

The header file tmtypes.h defines types for use in shared and public header files. Such header files should use definitions made here, or standard C types. This header file is platform-specific.

Integer and **Float** types fall into those categories selected for specific precision (for example, use in files), and those categories in which control of precision is sacrificed for machine efficiency.

String is a pointer to a string of characters that is guaranteed to be null-terminated. (Use **char***, otherwise.) Bools normally occupy machine-efficient storage. Use **:1** or **:8** for more precise control of packing within structures. **Pointer** represents a reference to an unspecified type, whereas **Address** is ready for use in address-arithmetic. **Char** and **Int** are defined for completeness and consistency.

TriMedia Types

The table below describes the general TriMedia type definitions. Following the table is a structure that defines the specific version.

Typedef	Type Name	Purpose
char*	Address	Ready for address-arithmetic.
unsigned int	Bool	Null is 0, False is 0, True is 1.
char	Char	Machine-natural character.
float	Float	Fast float.
float	Float32	Single-precision float.
double	Float64	Double-precision float.
int	Int	Machine-natural integer.
signed char	Int8	8-bit signed integer.
signed short	Int16	16-bit signed integer.
signed long	Int32	32-bit signed integer.
void*	Pointer	Pointer to anonymous object.
char*	String	Guaranteed null-terminated.
unsigned int	UInt	Machine-natural unsigned.
unsigned char	UInt8	8-bit unsigned integer.
unsigned short	UInt16	16-bit unsigned integer.
unsigned long	UInt32	32-bit unsigned integer.
Int	Endian	Big Endian is 0, Little Endian is 1.

tmVersion_t

```
typedef struct tmVersion_t{
   UInt8    majorVersion;
   UInt8    minorVersion;
   UInt16   buildVersion;
} tmVersion_t, *ptmVersion_t;
```

Fields

majorVersion	Major version.
minorVersion	Minor version.
buildVersion	Build version.

Description

Specifies the version: major, minor or build.

 © 1999 Philips Semiconductors 10/08/99

tmprof Profiler

This module contains the API of profiler library exported to the user. For more information, refer to tmlib/tmprof.h file.

tmprof API Functions

The following section describes the tmprof API function descriptions, which are contained in tmlib/tmprof.h header file.

profileInit

```
void profileInit (
   struct profileCaps    *pcaps,
   int                  (*writefunc)(),
   int                    handle
);
```

Parameters

pcaps	Pointer to initialization parameters
writefunc	Function to write data. Second and third arguments are address of buffer and data.
handle	The value of handle is passed as the first argument when writing data.

Return Codes

Returns 0 if OK, otherwise error code.

Description

Initialize hardware and buffer pointers for profiling.

 ©1999 Philips Semiconductors 10/08/99

profileStart

```
void profileStart();
```

Parameters

None.

Description

Starts profiling.

profileStop

```
void profileStop();
```

Parameters

None.

Description

Stops profiling (TFE exception).

profileFlush

```
void profileFlush();
```

Description

Convert buffer from internal to external format for use by **tmprof**.

profileDtrees

```
int profileDtrees();
```

Parameters

None.

Returns

Returns the number of bytes necessary for the profile buffer (estimate).

Note
This does not include locked text.

Description

Calculate the size of the trace buffer.

 © 1999 Philips Semiconductors 10/08/99

profileArgs

```
int profileArgs(
   struct profileCaps   *pcaps,
   char                **argv,
   int                   argc
);
```

Parameters

pcap	Pointer to initialization parameters.
argv	Vector of command line arguments.
argc	Argument count.

Returns

Returns the new argument count.

Description

Initialize profiling options. Profiling specific options are removed from the arguments.

AppModel Functions

The "appModel" functions provide a number of primitive services independent of an operating system. The appModel functions are available whether an OS is installed or not. The decision is made at link time. If an OS is available, the appModel functions are implemented in terms of that OS. If no OS is installed, simple implementations are used.

The following section describes the AppModel API functions.

 © 1999 Philips Semiconductors 10/08/99

AppModel_suspend_scheduling

```
extern void  AppModel_suspend_scheduling(void);
```

Parameters

None.

Return

None.

Description

Using whatever operating system is installed, scheduling is suspended. This is a portion of a larger operating system abstraction layer that is not publicly documented. This function is generally called at the start of an interrupt service routine, or at the beginning of a section that should be atomic against task switching. Under pSOS, it maps to **ienter**. This function is appropriate for use in libraries that may or may not be used in the presence of an operating system like pSOS. See also **AppModel_resume_scheduling**.

AppModel_resume_scheduling

```
extern void AppModel_resume_scheduling (void);
```

Parameters

None.

Return Codes

None.

Description

Using whatever operating system is installed, scheduling is resumed. This is a portion of a larger operating system abstraction layer that is not publicly documented. This function is generally called at the end of an interrupt service routine, or at the close of a section that should be atomic against task switching. Under pSOS, it maps to **ireturn**.

This function is appropriate for use in libraries that may or may not be used in the presence of an operating system like pSOS. See also **AppModel_suspend_scheduling**.

 ©1999 Philips Semiconductors 10/08/99

Mutual Exclusion Semaphores

A "mutex" type of semaphore is available for your use. A mutex semaphore provides mutually exclusive access to a resource. The include file "AppMutex.h" contains these definitions. The type **AppMut_Mutex**, presented next, forms the basis for mutual exclusion.

AppMut_Mutex

```
typedef struct {
   volatile Int      count;
   Bool              casted;
   volatile Pointer  first;
   volatile Pointer  last;
   volatile Pointer  owner;
} *AppMut_Mutex;
```

Description

Users do not generally have to know about the contents of this structure because it is always accessed through functions.

 © 1999 Philips Semiconductors 10/08/99

AppMut_create

```
extern AppMut_Mutex AppMut_create();
```

Parameters

None.

Description

Create a mutual exclusion semaphore.

Return

The function returns a pointer to a new mutex semaphore, or **NULL** when creation failed.

AppMut_delete

```
extern void AppMut_delete(
   AppMut_Mutex  mut
);
```

Parameters

`mut`	Pointer to the mutex semaphore to be deleted.

Description

Deletes a mutex semaphore. This routine can handle mutex semaphores which were created by casting. CAVEAT: Applications which are blocked on the mutex will never be released.

Return

Void.

AppMut_cast

```
extern void AppMut_cast(
   AppMut_Mutex  mut
);
```

Parameters

`mut`	Pointer, returned, to a memory block to be casted into a mutex.

Description

Casts a specified memory block into a mutex with specified capacity. This operation always succeeds.

Return

Void.

 © 1999 Philips Semiconductors 10/08/99

AppMut_lock

```
extern void AppMut_lock(
   AppMut_Mutex  mut
);
```

Parameters

mut	Pointer to the mutex semaphore to be locked.

Description

Put the mutex semaphore in a locked state.

Return

Void.

AppMut_unlock

```
extern void AppMut_unlock(
   AppMut_Mutex  mut
);
```

Parameters

mut	Pointer to the mutex semaphore to be unlocked.

Description

Puts the mutex semaphore in its unlocked state.

Return

Void.

AppMut_attempt_lock

```
extern Bool AppMut_attempt_lock(
   AppMut_Mutex mut
);
```

Parameters

`mut`	Pointer to the mutex semaphore to be locked.

Description

Attempts a lock operation on the mutex, but returns immediately and leaves the mutex untouched when this would cause a block.

Return

`True`	A lock operation has been applied to **mut**.
`False`	The lock operation could not be done without causing the current process to be blocked. The semaphore remains untouched.

 © 1999 Philips Semiconductors 10/08/99

OS-Independent Semaphores

The file AppSem.h defines an OS-independent semaphore. This device is appropriate for use when an OS is not available or when it is desired to be independent of an OS. The OS-independent semaphores use the **AppSem_Semaphore** type, presented next.

AppSem_Semaphore

```
typedef struct {
   volatile signed long count;
   int                  casted;   /* boolean */
   volatile void       *first;
   volatile void       *last;
} *AppSem_Semaphore;
```

Description

Users do not generally have to know about the contents of this structure because it is always accessed through functions.

 © 1999 Philips Semiconductors 10/08/99

AppSem_create

```
extern AppSem_Semaphore AppSem_create(
   Int32 count
);
```

Parameters

`count`	Required capacity of the semaphore.

Description

Create a semaphore with specified capacity.

Return

The function returns a pointer to a new semaphore, or **NULL** when creation failed.

AppSem_delete

```
extern void AppSem_delete(
   AppSem_Semaphore  sem
);
```

Parameters

`sem`	Pointer to the semaphore to be deleted.

Description

Deletes a semaphore. This function can handle semaphores which were created by casting. CAVEAT: Applications which are blocked on the semaphore will never be released.

Return

Void.

AppSem_cast

```
extern void AppSem_cast(
   AppSem_Semaphore  sem,
   Int32             count
);
```

Parameters

`sem`	Pointer, returned, to a memory block to be cast into a semaphore.
`count`	Required capacity of the semaphore.

Description

Casts a specified memory block into a semaphore with specified capacity. This operation always succeeds.

Return

Void.

 ©1999 Philips Semiconductors 10/08/99

AppSem_p

```
extern void AppSem_p(
   AppSem_Semaphore  sem
);
```

Parameters

sem	Pointer to the semaphore to be acquired.

Description

The calling process attempts to acquire the semaphore 'token'. If the semaphore token count is positive, then this call returns the semaphore token immediately. If the semaphore token count is zero, the task will be blocked until a semaphore token is released.

Return

Void.

AppSem_v

```
extern void AppSem_v(
   AppSem_Semaphore  sem
);
```

Parameters

sem	Pointer to the semaphore to be released.

Description

The calling process intends to release a semaphore token. If a task is already waiting at the semaphore, it is unblocked and made ready to run. If there is no task waiting, then the semaphore token count is simply incremented by 1.

Return

Void.

AppSem_attempt_p

```
extern Bool AppSem_attempt_p(
   AppSem_Semaphore  sem
);
```

Parameters

`sem`	Pointer to the semaphore to be acquired.

Description

Attempts to acquire the semaphore, but returns immediately and leaves the semaphore untouched when this would cause a block.

Return

`True`	The semaphore was acquired.
`False`	The semaphore could not be acquire without causing the current process to be blocked. The semaphore remains untouched.

 © 1999 Philips Semiconductors 10/08/99

Chapter 2

TriMedia Registry Manager API

©1999 Philips Semiconductors 10/08/99

Introduction

The functionality of the TriMedia Registry is roughly equivalent to that of the registry found in Microsoft Windows®. The registry is a hierarchically-structured tree consisting of directories and data containers, which are referred to as *entries*. Entries can be strings, integer values, or any customized data.

The registry behaves much like a file system that resides in memory. Like a file system, you can write to or read from the registry, and add, remove, or scan through the entries. Unlike a file system, the directory (which is a container of subentries and is itself an entry) *can* have data associated to it. Therefore, a directory is an entry, but an entry does not have to be a directory, since it may not have subentries.

Apart from this dual personality of "directory entries," there are no major differences between a file system directory and the TriMedia registry.

Why Use the Registry

Normally when you want to share data between two modules, you must declare an external variable. This method creates name pollution, wherein the variable becomes visible to the whole world, thus preventing any other module from using its name. When you use the registry, however, you store the data in a hidden structure so as not to adversely affect any other modules.

One could argue that name pollution is shifted to the registry, but that is not completely true, since you can create separate directories. Declaring external variables is similar to a file system that allows you to store files only in the root directory. Using the registry is similar to a file system that lets you use all the directories to store your files.

Package-Oriented Data

This allows you to have a package-oriented approach to data. A package is then a directory, and variables belonging to that package are entries in this directory. For example, you could store in a directory called "network" the IP address, DNS, and so forth, so that the data is accessible to all applications without creating an **ip** variable.

Exploring a Registry

Since the registry has features similar to a file system, you can explore the registry. For example, an application might try to open the network directory, and if it fails, will conclude that there are no networking facilities. Conversely, if you had stored data in variables and have no networking facilities on a specific platform, the linker would complain about an unresolved reference.

Through the registry, your application can adapt itself to the other modules without having to be recompiled. One typical use is to register device drivers so that a program developed for one platform can adapt itself to other platforms just by reading the regis-

 ©1999 Philips Semiconductors 10/08/99

try. For example, on some boards, there might be user-writable flash memory, but on others there may not be. In this case, an application can read the registry to see if flash memory is present, and act accordingly if it isn't.

Of course, you don't have to use the registry within your applications, but if you write device drivers, it is probably a good idea. Refer to Chapter 5, *Device Libraries*, of Book 3, *Software Architecture*, Part A, for information on how to write a device driver using the registry.

Using the Registry API

All entries are given a name that is easy to understand from the human point of view. This name is case-sensitive and consists of a string with a maximum length of 32. At the present time, names can use any character except * (which is used for pattern matching), the **'\0'** (which is used as a string terminator) or the **'/'** (which is used as a path separator).

Note
Though at present almost any character can be used as an entry name, you should use only alphanumeric characters. We cannot guarantee that non-alphanumeric characters will be supported in future implementations.

When you first access the registry, you will see that three directories already exist. These are created during the boot process by the component manager.

Directory	Description
bsp	This directory contains all data that is specific to the board. In this directory, you will find the **boardName**, the **boardID**, and a description of the different peripherals connected to this board (the number of Audio Out units, for example). An application designer should never write to this directory, as the bsp (board support package) and therefore the entire architecture are relying on it. However, it is possible to read in this directory. There are many helper functions available that allow you to read directly in the registry. These are documented in Chapter 19, *TMBoard API*, of Book 5, *System Utilities*, Part C.
apps	This directory may contain data specific to applications. If you want to use this directory, you should add a subdirectory for every application you write.
misc	This one is free for your use.

Note
Although you can add other directories at the "root" level, the preferred method is to use the apps and misc directories.

To use the registry, just include <tm1/tsaReg.h> and link against libdev.a (this is automatically added for you by **tmcc**). However, if you are still debugging your application, you should use the debug version of this library, since numerous error conditions are trapped using asserts. The debug version is in libdev_g.a.

Limitations

The following are some limitations to the registry.

1. Be aware that the registry will be slower than direct access to a variable because of the time required to find the data.
2. Avoid storing large amounts of data in the registry (structures larger than 1K, for example). If you want to reference a large amount of data, it is more efficient to store in the registry only the pointer to the data. Of course, this implies that this data will have to be **static** or **malloc**'d.
3. To avoid possible memory fragmentation, you should also be careful about the accessing of entries by applications that are running "forever."
4. The registry is not the place for persistent storage. It is not copied back to any kind of device, so it must be reconstructed at every boot.

Demonstration Program

The demonstration program resides in the examples/misc/tsaRegExample directory and shows how to use the TriMedia Register API. It can be decomposed into four sub-programs. These four sub-programs correspond to the functions described below.

AddEntryTest

This test can be split into three different entities corresponding to **regArray**, **regArray2**, and **regBuggyEntries** arrays. These three arrays contain entries that have to be added to the registry with the **tsaRegAddEntry** function. The first array (**regArray**) contains entries that are correctly formatted to access the registry.

The second array contains some deliberate formatting errors (there are some misplaced forward-slash characters in the paths, for example). This involves a little more work from the library to reformat internally to the correct names, but this flexibility is convenient. This array also shows how to create an entry without needing to create all the sub-directories. This is probably the easiest way to add entries in the registry.

The third array contains a list of entries that are not valid for various reasons, and the library will return various error codes (see the example for an explanation).

QueryEntryTest

This test tries to retrieve the data previously stored in the registry.

RemoveTest

This test tries to remove two directories (and all their contents). One of them does not exist and trying to remove this directory produces an error.

 © 1999 Philips Semiconductors 10/08/99

FindTest

This test demonstrates how to use the find functions to explore the registry. This allows you to scan for entries that match special criteria. Only at this point can you try to find entries whose name begins with the pattern given as an argument. Therefore, a pattern can only follow the "pattern*" format. This can be a very convenient way to walk through the registry.

Note
In future implementations, the find capability might be improved to support pattern matching inside the entry itself, making searches with targets such as "bsp/*/1" possible.

Registry API Data Structures

This section describes the Registry API device library data structures. These data structures are defined in the tsaReg.h header file.

 © 1999 Philips Semiconductors 10/08/99

tsaRegEntryClass_t

```
typedef enum{
   recNull       = 0x00000000,
   recData       = 0x00000001,
   recFunction   = 0x00000002,
   recCustom     = 0x000000ff,
} tsaRegEntryClass_t;
```

Fields

`recNull`	The entry does not contain data. Mostly used for sub-directory entries.
`recData`	The entry contains an array of data elements.
`recFunction`	The entry contains a pointer to a function.
`recCustom`	The entry contains some data whose type is private.

Description

This enumerated type is used to describe the contents of an entry.

tsaRegEntryDataType_t

```
typedef enum{
   redtInt     = 0x00000001,
   redtUInt    = 0x00000002,
   redtFloat   = 0x00000003,
   redtChar    = 0x00000004,
} tsaRegEntryDataType_t;
```

Fields

`redtInt`	The entry contains an array of integers (C type is **Int**).
`redtUInt`	The entry contains an array of unsigned integers (C type is **UInt**).
`redtFloat`	The entry contains an array of floating point values (C type is **Float**).
`redtChar`	The entry contains an array of characters (C Type is **Char**). This array does not have to be a null-terminated string.

Description

When the entry contains data (that is, it has the **recData** type as described in **tsaRegEntryClass_t**), this enumerated type describes the type of the values in the array. If you want to store elements whose type is not in this enumerated type (such as your own structure), you should use the **recCustom** type (see **tsaRegEntryClass_t**).

© 1999 Philips Semiconductors 10/08/99

tsaRegDataEntry_t

```
typedef struct{
   tsaRegEntryDataType_t   dataType;
   UInt32                  dataLength;
   Pointer                 data;
} tsaRegDataEntry_t, *ptsaRegDataEntry_t;
```

Fields

`dataType`	Type of data stored in this entry.
`dataLength`	The number of elements in this data array.
`data`	Pointer to the first element of the array.

Description

When the entry you want to register has the **recData** type (see **tsaRegEntryClass_t**), this describes the data you want to store in the registry.

tsaRegEntryAdd_t

```
typedef struct{
   Char                *path;
   Char                 keyString[TSA_REG_MAX_KEY_SIZE];
   tsaRegEntryClass_t   entryType;
   UInt32               entrySize;
   Pointer              entry;
   UInt32               flags;
}tsaRegEntryAdd_t, *ptsaRegEntryAdd_t;
```

Fields

`path`	Path to entry. This can consist up to **TSA_REG_MAX_PATH_SIZE** characters (256 in this implementation) including the **\0** terminator.
`keyString`	This string identifies the entry to be added in path. This consists in up to **TSA_REG_MAX_KEY_SIZE** (32) characters including the **\0** terminator.
`entryType`	Describes the type of the entry (see **tsaRegEntryClass_t**).
`entrySize`	Describes the length of the entry in bytes. This field is only used when the entry type is a customized entry type. Otherwise, it is ignored.
`entry`	Pointer to the entry to be added. This part will be copied into the registry. This field is not used if the entry type is **recNull**.
`flags`	Describes additional flags you want to add be used when creating the entry. At the present time, only one flag is supported: `TSA_REG_CREATE_ALWAYS` This tells the **tsaRegAddEntry** function to try to create the entry even if all the subdirectories leading to the entry are not yet created. If this flag is not set, the **tsaRegAddEntry** function would fail.

Description

This structure is passed as an argument to the **tsaRegAddEntry** function and completely describes the entry to be added, and how the entry should be added (with the flags field).

tsaRegFind_t

```
typedef struct {
   Char                  keyString[TSA_REG_MAX_KEY_SIZE];
   tsaRegEntryClass_t    entryType;
   Pointer               entry;
   UInt32                flags;
   Pointer               handle;
} tsaRegFind_t, *ptsaRegFind_t;
```

Fields

`keyString`	Name of the entry that was found by the **tsaReg-FindNext** or the **tsaRegFindFirst** functions.
`entryType`	Type of the found entry (see **tsaRegEntryClass_t**).
`entry`	Pointer to the entry. Its type depends on the field **entryType**.
`flags`	Reserved for future use.
`handle`	This is reserved. You should not set or modify this field.

Description

This structure is used when walking through the registry. The members are set by the **tsaRegFindFirst** and **tsaRegFindNext** functions.

Registry API Functions

This section presents the Registry API functions.

 © 1999 Philips Semiconductors 10/08/99

tsaRegAddEntry

```
tmLibdevErr_t tsaRegAddEntry(
   tsaRegEntryAdd_t   regEntry
);
```

Parameters

`regEntry`	Pointer to a structure that contains the description of the entry to be added.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NULL_PARAMETER`	In the debug version of the library, this assert is given if **regEntry** is null. This is also triggered in the following cases: **regEntry–>entry** is null and **regEntry–>entryType** is not **recNull**. **regEntry–>entry->data** is null and **regEntry–>entryType** is **recData**.
`TSA_REG_ERR_PATH_TOO_LONG`	Returned if the path given in **regEntry** exceeds **TSA_REG_MAX_PATH_SIZE** (256).
`TSA_REG_ERR_INVALID_PATH_NAME`	Returned if the path contains invalid characters such as '*'.
`TSA_REG_ERR_KEY_NAME_TOO_LONG`	Returned if the entry length exceeds **TSA_REG_MAX_KEY_SIZE** (32).
`TSA_REG_ERR_INVALID_KEY_NAME`	Returned if the entry contains invalid characters such as '/' or '*', or if the entry is an empty string.
`TSA_REG_MEMALLOC_FAILED`	Returned if the function was unable to allocate memory for the entry.
`TSA_REG_ERR_PATH_NOT_FOUND`	Returned if **TSA_CREATE_ALWAYS** is not set in the flags of **regEntry** and the path to the entry to be created is not found.
`TSA_REG_ERR_ENTRY_EXISTS`	Returned when attempting to create an entry that already exists.
`TSA_REG_ERR_UNKNOWN_ENTRY_TYPE`	Returned when the type of the entry is not among the predefined types (see **tsaRegEntryClass_t**).
`TSA_REG_ERR_UNKNOWN_DATA_TYPE`	Returned when the type of the entry is **recData** and the data does not have one of the predefined types (see **tsaRegEntryDataType_t**).

Description

This functions creates a new entry as defined in the **regEntry** structure.

 © 1999 Philips Semiconductors 10/08/99

tsaRegAddDirectory

```
tmLibdevErr_t tsaRegAddDirectory(
   Char   *path
);
```

Parameters

path	The path to be added in the registry

Return Codes

TMLIBDEV_OK	SuccessSuccess.
TMLIBDEV_ERR_NULL_PARAMETER	In the debug version of the library, this assert is given if **regEntry** is null. This is also triggered in the following cases: **regEntry->entry** is null and **regEntry->entryType** is not **recNull**. **regEntry->entry->data** is null and **regEntry->entryType** is **recData.**
TSA_REG_ERR_PATH_TOO_LONG	Returned if the path given in **regEntry** exceeds **TSA_REG_MAX_PATH_SIZE** (256).
TSA_REG_ERR_INVALID_PATH_NAME	Returned if the path contains invalid characters such as '*'.
TSA_REG_ERR_KEY_NAME_TOO_LONG	Returned if the entry length exceeds **TSA_REG_MAX_KEY_SIZE** (32).
TSA_REG_ERR_INVALID_KEY_NAME	Returned if one of the tokens between two '/' contains invalid characters such as '/' or '*'.
TSA_REG_MEMALLOC_FAILED	Returned if the function was unable to allocate memory for the entry.
TSA_REG_ERR_ENTRY_EXISTS	Returned when attempting to create an entry that already exists.

Description

This function creates the directory path and all the sub-directories if necessary. For example, you can create the directory /foo/bar/baz even if /foo does not exist. All the created entries are created with the type **recNull**. If you would prefer to use another type, you should **tsaRegAddEntry** instead.

tsaRegRemoveEntry

```
tmLibdevErr_t tsaRegRemoveEntry(
   Char    *keyString,
   UInt32   flags
);
```

Parameters

keyString	The complete path and the entry name to be removed.
flags	Specifies how the entry should be removed. The following flag is currently supported: **TSA_REG_DELETE_SUBTREE** All subentries are removed recursively.

Return Codes

TMLIBDEV_OK	Success.
TMLIBDEV_ERR_NULL_PARAMETER	In the debug version of the library, this assert is given if **keyString** is null.
TSA_REG_ERR_PATH_TOO_LONG	Returned if the path given in **regEntry** exceeds **TSA_REG_MAX_PATH_SIZE** (256).
TSA_REG_ERR_INVALID_PATH_NAME	Returned if the path contains invalid characters such as '*'.
TSA_REG_ERR_PATH_NOT_FOUND	Returned if the entry to be destroyed was not found.
TSA_REG_ERR_CANT_REMOVE_ROOT_ENTRY	Returned if trying to remove the root entry.
TSA_REG_ERR_ENTRY_HAS_SUB_TREE	Returned when the entry has sub-directories and the **TSA_REG_DELETE_SUBTREE** flag was not set.

Description

Removes the entry described by **keyString**. If the **TSA_REG_DELETE_SUBTREE** is set in flags, then the entry and all its subentries will be removed.

tsaRegQuery

```
tmLibdevErr_t tsaRegQuery(
   Char                  *keyString,
   tsaRegEntryClass_t    *entryType,
   Pointer               *regEntry
);
```

Parameters

keyString	Name of the entry to be opened.
entryType	Pointer to a buffer that will contain the type of the entry.
regEntry	Pointer to a pointer that contains the entry.

Return Codes

TMLIBDEV_OK	Success.
TMLIBDEV_ERR_NULL_PARAMETER	In the debug version of the library, this assert is given if **keyString**, **entryType**, or **regEntry** is null.
TSA_REG_ERR_PATH_TOO_LONG	Returned if the path given in **regEntry** exceeds **TSA_REG_MAX_PATH_SIZE** (256).
TSA_REG_ERR_INVALID_PATH_NAME	Returned if the path contains invalid characters such as '*'.
TSA_REG_ERR_PATH_NOT_FOUND	Returned if the path to the entry is not found.

Description

This function gives you the properties of an entry. In accordance with **entryType**, **regEntry** can be a pointer to a pointer to a function, to a data descriptor (see **tsaRegDataEntry_t**), or to some custom data.

Note that with this function you get a direct access to the registry. This means that you may modify the contents of the registry through this pointer, though you are advised not to do so, as it may lead to unpredictable results if not done properly.

tsaRegFindFirstEntry

```
tmLibdevErr_t tsaRegFindFirstEntry(
   Char           *keyString,
   ptsaRegFind_t   findInfo
);
```

Parameters

keyString	Name of the entry to be opened.
FindInfo	Buffer to a structure that contains the description of the first object matching the criteria

Return Codes

TMLIBDEV_OK	Success.
TMLIBDEV_ERR_NULL_PARAMETER	In the debug version of the library, this assert is given if **keyString**, or **FindInfo** is null.
TSA_REG_ERR_PATH_TOO_LONG	Returned if the path length exceeds **TSA_REG_MAX_PATH_SIZE** (256).
TSA_REG_ERR_KEY_NAME_TOO_LONG	Returned if the ending part of **keyString** exceeds **TSA_REG_MAX_KEY_SIZE** (32) characters.
TSA_REG_ERR_INVALID_SEARCH_PATTERN	Returned if the search pattern is not terminated by an '*' (the only search pattern presently supported), or if an '*' is found at an inappropriate place (not at the end). This error code's behavior may change in future implementations as new search methods are added.
TSA_REG_ERR_PATH_NOT_FOUND	Returned if the path to the entry is not found.

Description

This function returns in **findInfo** the first entry that matches **keyString**. At the present time, '*' is the only recognized meta character. Moreover, it is only accepted if placed at the end of **keyString**. If successful, the **findInfo** structure contains a description of the found entry. This structure should be used as a parameter of the **tsaRegFindNextEntry** function to find the other entries that match the search criteria.

 © 1999 Philips Semiconductors 10/08/99

tsaRegFindNextEntry

```
tmLibdevErr_t tsaRegFindNextEntry(
   Char           *keyString,
   ptsaRegFind_t   FindInfo
);
```

Parameters

keyString	Name of the entry to be opened.
FindInfo	Buffer to a structure that contains the description of the next object matching the criteria.

Return Codes

TMLIBDEV_OK	Success.
TMLIBDEV_ERR_NULL_PARAMETER	In the debug version of the library, this assert is given if **keyString**, **FindInfo, or FindInfo->handle** is null.
TSA_REG_ERR_PATH_TOO_LONG	Returned if the path length exceeds **TSA_REG_MAX_PATH_SIZE** (256).
TSA_REG_ERR_KEY_NAME_TOO_LONG	Returned if the ending part of **keyString** exceeds **TSA_REG_MAX_KEY_SIZE** (32) characters.
TSA_REG_ERR_INVALID_SEARCH_PATTERN	Returned if the search pattern is not terminated by an '*' (the only search pattern presently supported), or if an '*' is found at an inappropriate place (not at the end). This error code's behavior may change in future implementations as new search methods are added.
TSA_REG_ERR_PATH_NOT_FOUND	Returned if the path to the entry is not found.

Description

In **findinfo**, the function returns the next entry that matches **keyString**. At the present time, '*' is the only recognized meta character. Moreover, this character is accepted only at the end of the **keyString**. If successful, the **findinfo** structure contains a description of the found entry. The **findInfo** parameter should be initialized by a call to **tsaRegFindFirst-Entry** before any use of the **tsaRegFindNextEntry**.

 © 1999 Philips Semiconductors 10/08/99

Chapter 3

TriMedia Component Manager API

Overview

The component manager provides a mechanism to control the order of system initialization. Software "components" that are managed by the component manager are initialized before the start of user code. In this way, the component manager provides a way to install drivers for all sorts of functions, be they hardware or software based.

The initialization that takes place before main typically consists of the following phases.

- Taking the chip out of reset.
- Initializing the malloc/free functions.
- Executing all modules that run at **custom_boot**.
- Initializing the host communication (if appropriate).
- Executing all modules that run at custom_driver.
- Opening **stdin**, **stdout**, and **stderr**.
- Parsing the command line (**argc**, **argv**).
- Launching the dynamic loader.
- Initializing the debugger monitor.
- Launching all modules that run at **custom_start**.

The "chaining" mechanism supported by **tmld**, the TriMedia linker/loader, allows the adding of other software modules with the three symbols **custom_boot**, **custom_driver**, and **custom_start**. (For a full description of **tmld**, see Chapter 11, *Linking TriMedia Object Modules*, in Book 4, *Software Tools*, Part B.) In short, the linker allows a user to create a linked list of functions to be called before **main**. Experience with the software system shows that this mechanism is not flexible enough. This mechanism does not provide a way to control the order in which the different modules are launched. Two modules that are declared to be launched at the same level (for example at **custom_boot**) will get run in a random order, preventing any dependencies between the two modules.

Moreover, this kind of initialization implies that the initialization of board support components is performed only when needed, after program start. This can be very inconvenient if you want to develop a module that needs to be launched before main but that is also dependent on the launching of the board support modules.

The component manager offers a generic way to launch (in an appropriate order) all the software components that are needed before main. The component manager uses the "chaining" mechanism internally, but it is transparent to a developer.

We will refer to a piece of software that needs to be launched before main as a *component*. This includes, but is not limited to, chip, board, host communication, and flash memory initialization. A component is a black box that may require the prior launching of other components. These other required components are referred to as *inputs*. A component may also export some properties, and these are referred as *outputs*. The inputs and out-

 ©1999 Philips Semiconductors 10/08/99

puts will be referred to as *symbols* in the rest of this chapter. You may notice some family resemblance between the operation of the component manager and that of the linker.

The component manager builds a tree of dependencies out of this list of components, their inputs, and their outputs. Then it launches all the components one after another according to the tree. The component manager makes sure that each component has been properly initialized before launching another component that relies on it. If the component manager cannot resolve a dependency (either because some of the components that could provide a required functionality failed to initialize, or because none of the components could satisfy the dependencies), then the component manager stops exploring that leaf (unless told to do so). The component manager will launch all the components that can be launched according to the dependency tree.

An example of a component is the board support package. A board support package does not have any input, because it is the first component to be launched. But many components rely on it. Therefore it must indicate that it was launched. Therefore, the board support package is declared to have an output called 'bsp/boardID'. These inputs and outputs are symbols that are stored in the registry. See Chapter 2, *TriMedia Registry Manager API,* for details on the TriMedia registry mechanism.

The component manager tries to launch all of the available components in an appropriate order. Then the component manager will check whether all of the symbols that were declared (inputs or outputs) are defined. By default, the component manager does not complain if there is a missing symbol at the end of the exploration of the component tree. This way, it is possible to link components to an executable even if they are not used. This can be convenient, for example, when building a component that takes care of a TCP/IP stack that relies on a modem connection that may be present on some boards, but may not on some other boards. This mechanism allows you to run a program that does not need this TCP/IP stack to run anyway.

When the exploration of the tree is done, then it will launch **main** (and eventually **root** of this is a pSOS application).

This method ensures that a program that is launched will run as expected. This does not mean that all components have been launched successfully, but that all components have been given a chance to initialize. It is especially important to see the difference between these two concepts when developing two components that provide similar functionality (two board support packages, for example) but cannot run at the same time. With this model, these two components will have the same output, therefore preventing one component to run if the other was successfully activated. This method also ensures that one and only one of these components will run.

Advanced Features

The following features are not used in most cases, and should be used only if necessary.

Components With "Required" Flag

By default, the component manager does not complain if a symbol was not created by any of the components. This behavior can be overridden by attaching a "data property" to one of the symbols. By default, all symbols are given basic properties. One of them is a flag that describes whether the symbol is required or not. This specific data property is called "required." If this flag is set, the symbol will be considered by the component manager as a necessary symbol to launch **main**. This means that the component manager will stop execution (without launching **main**) if it is unable to locate this symbol at the end of the initialization of the components. This can be convenient, if a program cannot run if a symbol is not present. For example, a program that would use the board support package might want to make the symbol bsp/boardID a required symbol. Note that this flag is general: this puts a condition on the execution of the main program itself.

Disabling Components

This mechanism can be extended further when there are one or more components in the list of components that are not wanted. Another component called a "disabler" is then created. It will take care of disabling the former. This is done very simply by adding a new component that outputs the symbol "disabled/*name_of_component*," where *name_of_component* is the name of the component which is to be disabled. Great care should be taken when disabling a component, as you also disable implicitly all components that relied on the disabled component. This may cause the component manager to complain about unresolved dependencies at the end (if one of these symbols was declared as "required").

Symbol Qualifiers

The default behavior of the component manager is not to launch a component if one of the inputs is missing. This can be overridden by adding a flag to any of the inputs of a component. This tells the component manager to try by any mean to resolve all the inputs of a component, but if unable to do so, to launch the component anyway. For example, this method is used for the serial console driver component ($TCS/examples/boards/serialConsoleDriver). This component opens the UART and creates a new driver (/dev/console) for it. Many boards, though have multiple UARTs which can be used for different purposes. In this case this component has as an input the entry libio/SerialConsolePort. This entry contains the unit it should associate with /dev/console. But, if the entry is not present, it will use **unit0**, that is **COM1**. Then, if you want to build a board with 2 **COM**s, and you would rather use **COM2** for communications, you just add a new component that exports libio/SerialConsolePort. This method allows you to put default values in some of the settings of your components, but the settings can be overridden just by adding a new component.

Example: Audio In on a Daughter Board

To see how the component manager is used, we can examine some applications. One of these is the support for an audio input module that resides on an installable daughter board. This is, in fact, the way that the later DTV reference boards support audio input using a "NIM" (network interface module) board.

When a new component is created, the first thing to do is to figure out the dependencies on other components. These other components can either be components one may implement in the future or components that already exist. The easiest part is to find the needed inputs, since one already knows the components that exist. These include the board support package, flash file system, and so on. Most components will rely on the board support package, at least to get the characteristics of the board, the board ID, or the address of the Flash memory, etc.

When designing a dependency tree, cyclic dependencies should be avoided, since the component manager cannot solve this kind of dependencies. The component manager stops execution as soon as it detects a cyclic dependency.

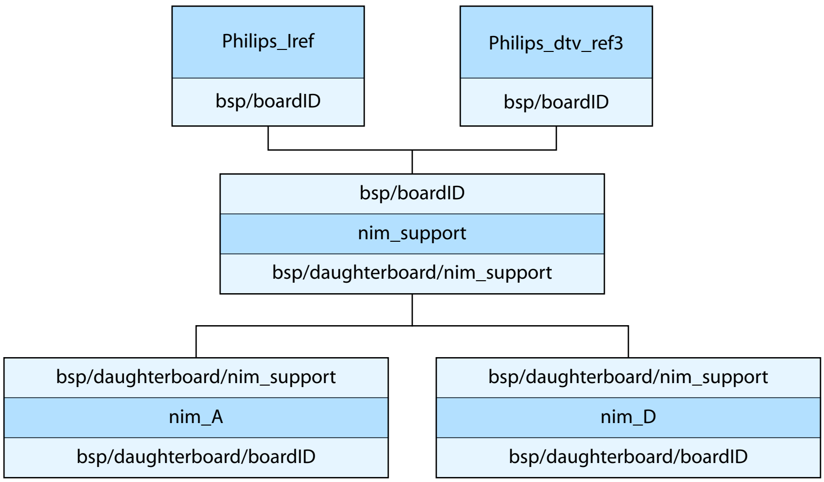


Figure 1 Tree of Dependencies

In our example, we build three separate components. The first one handles the communication between the daughter board and the main board. We suppose that there is only one-way communication. We call it "*nim_support.*" The second one is a component that should be installed when we are using a board called "*nim_A*" (network interface module, analog) to support analog audio input. The third component is another board called "*nim_D*" that can provide digital audio input. We can build the tree of dependencies as shown in Figure 1.

The tree has five components. The two board support packages are called "*Philips_dtv_ref3*" and "*Philips_Ire.f*" "*Philips_Iref*" is installed by default in the device library. The *dtv_ref3* package would have to be installed by the programmer. These two components have the same output called "bsp/boardID." Declaring two boards that have the same output ensures that at least one of them will run. The "*nim_support*" component needs to know on which board it is running to start its initialization. Hence it takes as an input "bsp/boardID." From this value, it will decide either to initialize or to disable itself. It will only run if "bsp/boardID" corresponds to an *Philips_dtv_ref3* board. Then the "*nim_support*" component proposes a special interface to access the daughter board. To do this, it registers the symbol "bsp/daughterboard/comm" that contains a list of the functions that can be used to access the daughter board. The two daughter boards cannot be plugged in at the same time, and some special code needs to be run in case we use the daughter board for analog or digital input. Both of these components have to make sure that there is a standard way to communicate with the daughter boards, so they rely on the fact that the "*nim_support*" component ran successfully, thus creating the dependency. These two daughter boards have another output called "bsp/daughterboard/comm". Having the same output ensures that only one of them will run. Due to the nature of the component manager, the failure to initialize a daughter board component (if the board is not plugged in or if we do not have any software for it), will not prevent the program from running.

A diagram like Figure 1 can be very helpful in an analysis of the dependencies between the different components.

General Rules About Creating a Dependency Tree

- Avoid components that have no outputs, because it prevents any other component from relying on it. Besides, by not having an output, it is impossible to make sure that the component was properly loaded from the component manager point of view.
- Use components without any inputs with great care, since they might be launched before any other component, especially before any board support package. In most cases adding a dependency on the board support package is a good idea (by adding "bsp/boardID" as an input).
- Components should define only the necessary symbols. For example, board support packages should only output "bsp/boardID" even if they actually register many capabilities such as an audio out or video out unit. It is the role of other components to use the **boardID** to query the board interface and find out the capabilities of an audio out or video out unit.

 ©1999 Philips Semiconductors 10/08/99

- Like any public name space, component and symbol names should be given with care. Avoid names like **A** or **B**. Explicit names should be used instead.
- Component names are limited to 24 characters and should be treated as a C language variable.
- Symbol names, in contrast to component names, must be completely unique. They exist during the complete execution of the program (see further to create temporary symbols), and therefore are visible to any program. Symbol names like **apps/mycompany/myproject/mycomponent/mysymbol** are recommended to avoid conflicts with any other existing components. Symbol names should comply with the registry rules. Therefore, alphanumeric names are recommended. See Chapter 2, *TriMedia Registry Manager API*, for details on the TriMedia registry mechanism.

  In the bsp directory, information about the board can be found. The following two entries are present:

  boardName. This entry contains the board Name. Its class type is **recData**, **redtChar** (see Chapter 2).

  boardID. This entry contains the board ID. Its class type is **recData**, **redtUInt**.

  In this same directory, one can find the following subdirectories:

  AO, AI, VO, VI, SSI, TP, GPIO (on TM2). These directories describe the different capabilities of the board in terms of I/O.

  In all of these directories, all the different interfaces for all the units are registered. For example, the entry **bsp/AO/00/Default** contains a description of the first Audio Out unit (units are numbered starting at 0). **Default** is actually a pointer to a **boarddevConfig_t** structure. See Chapter 19, *TMBoard API*, of Book 5, *System Utilities*, Part C, for details.

  At the root directory, the misc and apps directories can be used. Two extra directories, “temp” and “disabled,” also exist, as described below.

  The “temp” directory should contain symbols that are only needed during the booting process (until **main** begins). This directory is automatically destroyed when the component manager finishes.

  The “disabled” directory contains the list of the component names that should be disabled. This directory is automatically destroyed when the component manager finished.

The Activation Function

The declaration of a component consists not only of its name, inputs and outputs, but also of an entry point that the component manager uses to access the component. The entry point is represented by a function called the activation function. The activation function should be of the type **compActivateFunc_t** (see <tm1/tsaComponent.h>), and should always be declared **static** to avoid name space pollution. Hence it should also reside in the same file as the component declaration. The activation function takes as a parameter the definition of the component, describing its outputs, inputs, and name. It does not have any return value. This function should return a **tmLibdevErr_t** that can be used for debugging purposes.

The first phase could be called "detection" or "probing" phase, since it is in this part that the hardware and software (in terms of other installed components) are probed. In our example, we test to determine that we are running on the **Philips_dtv_ref3** board before doing anything else. If the component is a board, this phase should read the boot EEPROM using the IIC bus to identify the board. Great care should be taken when writing the detection phase, since it is very important that any component does not mistakenly try to launch itself when it is not safe to do so.

Also in the detection phase, the hardware is interrogated. No assumption should be made about the underlying hardware. Components can try to get the board ID and board name (if this component is itself not a Board Support Package), and the processor version. This is done using the functions described in Chapter 19, *TMBoard API*, of Book 5, *System Utilities*, Part C.

If this function succeeds in detecting all the adequate hardware and software components, then the second phase can be called. This phase takes care of the initialization. This can either be software initialization (allocation of buffers, etc.), or hardware initialization (taking the hardware out of reset, etc.). Failures should be forced into the detect phase and the initialization phase should not fail.

Registration is the only way that the component manager can know about the success of the activation function. If the symbols that the component is supposed to output are not there, the component manager will conclude that the component has failed. Other software will assume that the component is not installed.

There are two criteria that the component manager uses to declare that a component initialization has succeeded. The first verifies that all the outputs of these components are there after exploring the components tree. Since multiple components could have identical outputs, this criterion is not sufficient to declare a component initialization successful. Therefore, the component manager inspects the return value given by the activation function. A **TMLIBDEV_OK (zero)** should signal a successful completion, and any nonzero value a failure to complete. It is not advisable to use assert conditions (**assert**, **tmAssert**, or **exit**) to trap a problem. Since the component manager is running at custom boot, these functions will cause an apparent **tmcons** hang. Since host communication happens after custom boot, **tmcons** is unable to establish a connection with the TriMedia program since it already finished, hence causing the apparent hang.

This return value can be very convenient when debugging a component. This error code is used by the component manager to print a small report on the execution of the com-

 ©1999 Philips Semiconductors 10/08/99

ponents. This report can be inspected in the DP buffer. The report has the following format.

```
These components failed during initialization :
  Philips_Iref : f010017
  Philips_dtv_ref2 : f010017
  Philips_dtv_nim : f010017
End of logs
```

Or it can look like the following.

```
These components failed during initialization :
  comp2 : 3f01000a
These components have not been started during initialization
  comp5
End of logs
```

The first log was taken from a run with the components **dtv_ref2**, **dtv_ref3**, **iref**, and **dtv_nim** on a dtv_ref3 board without any NIM board attached to it. Out of the four components, three fail and all return **0xf010017**, which is actually **BOARD_ERR_UNKNOWN_BOARD**.

The second log was taken from a run of the component manager example in $TCS/examples/misc/compmanager/. In this case, there are components that were not even launched because the dependencies were not satisfied. Please read the readme.txt in the example directory for further details on this example.

How to Implement a New Component

The declaration of the properties (name, inputs, outputs and activation function) is performed through a set of macros that are defined in <tm1/tsaComponent.h>. These macros are called **TSA_COMP_DEF_XX_COMPONENT** where **XX** can be **I**, **O**, or **IO** depending on the fact that the component to be declared has inputs only, outputs only, or a mix of inputs and outputs. One of these macros should be placed at the end of one of the files that describe the new component. Adding this macro is the only change that has to be made to register a component.

Linking a Component Into an Application

Every component should be contained in a separate .o file. Components cannot be linked from an archive (.a). Since a component does not export any external functions, the linker would not link this component if it was included in an archive. Besides, there should be only one .o per component. In many cases, it is convenient to split the implementation of a component in many files, therefore this creates many .o. You should use **tmld** to merge multiple .o into one.

This linking mechanism ensures that the component manager is not linked (and therefore the registry), if the linker does not detect any component to be linked. This reduces the memory footprint for applications that do not need the component manager, or that are running on systems with little memory.

If the component you want to install is a board support package, it can be convenient to have it automatically linked every time you develop a program for this board. You can then modify your **tmconfig** file:

```
# Default boards to be linked
BOARD_LIST_EB= $TCS/lib/eb/libBSPiref.o $TCS/lib/eb/libBSPdtv_ref2.o
BOARD_LIST_EL= $TCS/lib/el/libBSPiref.o $TCS/lib/el/libBSPdtv_ref2.o
```

These two lines define the components (actually the board support package) that are automatically linked. For example, if you are developing on a Philips DTV ref3 board (also known as the GOMAD board), you might want to specify:

```
BOARD_LIST_EB= $TCS/lib/eb/libBSPdtv_ref3.o
BOARD_LIST_EL= $TCS/lib/el/libBSPdtv_ref3.o
```

This will tell the linker to link the board support package of the DTV Ref 3 board. This will also reduce the memory footprint of the executable, since the IREF and the DTV Ref 2 are no longer linked. If you need to develop a program that would run on any kind of board (IREF, DTV Ref2, or DTV Ref3), just add the DTV Ref 3 BSP to the variables instead of replacing their content.

The variables **BOARD_LIST_EL** and **BOARD_LIST_EB** are ignored when linking a **tmsim** executable: **tmsim** is unable to simulate boards, therefore board support packages cannot run reliably under this environment.

The TriMedia component manager is contained in the archived device library libdev.a. The libdev.a device library is linked automatically.

If you are developing a dynboot executable (see Chapter 11, *Linking TriMedia Object Modules,* of Book 4, *Software Tools*, Part B), then the components are embedded in the dynboot executable. This can advisable in this case to tune the list of the required components to avoid having an unnecessary large executable. If you are debugging a component, a specific version of the component manager should be linked, as detailed below.

Debugging a New Component (Example Program)

The example program can be found in $TCS/examples/misc/compmanager/. Please refer to the readme.txt file for build instructions and for description of functionalities of the program.

When using the component manager, many things are absent that could be useful for debugging. These include host communication, and therefore **printf** and other I/O functions because these are initialized as components. Therefore, development of a new component should be split into three phases. The first phase consists of making the new

 © 1999 Philips Semiconductors 10/08/99

driver work after **main** by explicitly calling the component **activate** function. This enables the use of **printf** and any other host communication.

```
main(){
   comp1_activate();
   comp2_activate();
   comp3_activate();
   ...
}
```

Note that in this phase, the calls to the **activate** functions and the initialization order are explicit.

The second phase consists of integrating the component into the list of the components by using one of the **TSA_COMP_DEF_XX_COMPONENT** macros. This macro should be added only when the functionality of the new component has been proven. In this second phase, the component manager will be forced to be launched after **main**. This is possible by explicitly calling the function **tsaCompInitComponentManager** and linking the **_dbg** version of the component manager. This version can be linked by adding on the **tmld** command line **$TCS/lib/$ENDIAN/tsaComponent_dgb.o** (you must replace **$TCS** and **$ENDIAN** by your specific paths and endianness). Calling the component manager explicitly allows you to use **tmdbg** and all of the usual debugging techniques except of course STDIOs. You can make your component dependent on STDIO, making **printf**s work. Though in many cases, your component (for example, a BSP) must be started before, because STDIO initialization relies on it. In this case, making your BSP rely on STDIO could create a circular dependency that the component manager would be unable to resolve. Note that running the component manager after **main** implicitly disables software modules that need component manager to run before **main**.

```
/* main_dbg.c */
...
main(){
   tsaCompInitComponentManager();
   ...
}
```

```
# Makefile
...

$(OUT_DBG): $(OBJ_DBG)
   $(CC) $(LDFLAGS) -o $@ $(OBJ_DBG) \
      $(TCS)/lib/$(ENDIAN)/tsaComponent_dbg.o
...
```

The last phase consists of restoring the normal behavior of the component manager (that is, to make it run at custom boot). **$TCS/lib/$ENDIAN/libtsaComponent_dgb.o** should be removed from the **tmld** command line, so that the normal version of the component manager is used. In this component, the only method you will have to debug is the use of DPs (debug prints). That is why it is best to tune your component, with the _dbg version of the component manager and with the help of tmdbg, before using the normal version of the component manager.

```
main(){
   printf("The component manager has already been launched\n");
   ...
}
```

```
# Makefile
...

$(OUT) : $(OBJ)
   $(CC) $(LDFLAGS) $(OBJS) -o $@
...
```

It is possible to skip one of the two first phases, though it is not recommended.

In phases II and III, the component manager's execution report can be used to debug a component. This report (printed in the **DP** buffer) contains the description of the different component initialization failures that were encountered. The use of **DP**s is highly recommended during the last development phase—only **DP**s can perform reliable I/O at this time. The component manager enables a **DP** buffer for you, so that there is no need to call **DP_START**. On the contrary, calling **DP_START** will erase the component manager logs.

When your component is stable enough, you might want to leave a few **DP**s in your code, so that developers of other components using your component will understand why your component failed. Ideally, you should provide a table that describes all the error codes that can be returned by your component, so that it should be easy to understand a component's failure to initialize.

 ©1999 Philips Semiconductors 10/08/99

Macros

This section presents the set of macros that should be used to define new components.

One type definition applies:

compInputQualifier_t

```
typedef enum {
   compInputRequired     = 0,
   compInputNotRequired  = 1
} compInputQualifier_t;
```

Fields

`compInputRequired`	Specifies that this input is required to launch this component. This value is the default value for a component.
`compInputNotRequired`	Specifies that this input is not needed by the component manager to launch this component, although the component manager will try to resolve this symbol if possible before launching this component.

Description

Describes the way the component manager should try to resolve the symbols before launching a specific component. This type is used with the macros:

```
TSA_COMP_BUILD_ARG_LIST_1_M
TSA_COMP_BUILD_ARG_LIST_2_M
TSA_COMP_BUILD_ARG_LIST_3_M
```

The macros

```
TSA_COMP_BUILD_ARG_LIST_1
TSA_COMP_BUILD_ARG_LIST_2
TSA_COMP_BUILD_ARG_LIST_3
```

have **compInputRequired** built in.

 © 1999 Philips Semiconductors 10/08/99

TSA_COMP_DEF_IO_COMPONENT

```
TSA_COMP_DEF_IO_COMPONENT(
   name,
   Pointer             *inputs,
   Pointer             *outputs,
   compActivateFunc_t   activate
);
```

TSA_COMP_DEF_I_COMPONENT

```
TSA_COMP_DEF_I_COMPONENT(
   name,
   Pointer             *inputs,
   compActivateFunc_t   activate
);
```

TSA_COMP_DEF_O_COMPONENT

```
TSA_COMP_DEF_O_COMPONENT(
   name,
   Pointer             *outputs,
   compActivateFunc_t   activate
);
```

Parameters

`name`	Name of the component to be defined. This should *not* be enclosed between quotes. This should also be a valid C variable—only numbers and letters may be allowed. The number of characters is limited to 24.

`inputs`	An array of names defining the different symbols the component manager is to find before launching this component. The last element of this array should be Null. The different elements in this array are strings that comply with the registry naming convention described on page 53 in Chapter 2, *TriMedia Registry Manager API*. You can use the component manager macros
	`TSA_COMP_BUILD_ARG_LIST_1`
	`TSA_COMP_BUILD_ARG_LIST_2`
	`TSA_COMP_BUILD_ARG_LIST_3`
	to construct the array.

`outputs`	An array of names defining the different symbols the component manager expects this component to output after it is launched. The last element of this array should be Null. The different elements in this array are strings that comply with the registry naming convention. You can use the component manager macros
	`TSA_COMP_BUILD_ARG_LIST_1`
	`TSA_COMP_BUILD_ARG_LIST_2`
	`TSA_COMP_BUILD_ARG_LIST_3`
	to construct the array.

activate	This is the entry point of the component. This function is called by the component manager when it has resolved all the dependencies imposed by the inputs array. This function should initialize the component and register its outputs.

Description

These macros define the component's name, inputs (for **TSA_COMP_DEF_I_COMPONENT** and **TSA_COMP_DEF_IO_COMPONENT**), outputs (for **TSA_COMP_DEF_O_COMPONENT** and **TSA_COMP_DEF_IO_COMPONENT**), and the component's activation function.

Use **TSA_COMP_DEF_O_COMPONENT** if a component doesn't require input. Use **TSA_COMP_DEF_I_COMPONENT** if a component doesn't output anything, and use **TSA_COMP_DEF_IO_COMPONENT** for components with both inputs and outputs.

Each component should only use one of those macros once. The macro should be called at the end of the file where the activate function is defined. These macros should never be placed inside a function body.

These macros define one of two static variables called **inputs**_*xxxx* and **outputs**_*xxxx* (or both, in the case of **TSA_COMP_DEF_IO_COMPONENT**) where "*xxxx*" should be replaced by the field name. These two arrays contain the description of the inputs and the outputs of the component called **name**. These macros also define a dummy variable that is needed by the linker to decide when to link the component manager.

These macros also chain this component in the external variable **__component_list**. For more information, see *List Construction by tmld* on page 130 of Book 4, *Software Tools*, Part B.

 ©1999 Philips Semiconductors 10/08/99

TSA_COMP_DEF_DATA_PROP

```
TSA_COMP_DEF_DATA_PROP(
   String   name,
   Bool     required
);
```

Parameters

name	Name of the symbol (data property).
required	Boolean value that describes whether the component manager is supposed to expect this symbol to be created before launching main.

Description

This macro changes the data properties of a symbol. By setting **required** to True, the symbol name is expected to be present before launching main. If it is not, then the component manager stops execution after printing a message explaining the reason of the failure.

TSA_COMP_BUILD_ARG_LIST_1

```
TSA_COMP_BUILD_ARG_LIST_1(
   String   name1
);
```

Parameters

`name1`	Name of the first input or output that is attached to a component. This name must comply with the registry naming conventions (see page 53 of Chapter 2).

Description

This macro creates the array

```
{ { name1,comInputRequired }, { Null,0 } }
```

that is needed in **TSA_COMP_DEF_IO_COMPONENT**, **TSA_COMP_DEF_I_COMPONENT**, or **TSA_COMP_DEF_O_COMPONENT** as parameter “inputs” or “outputs.”

 ©1999 Philips Semiconductors 10/08/99

TSA_COMP_BUILD_ARG_LIST_2

```
TSA_COMP_BUILD_ARG_LIST_2(
   String    name1,
   String    name2
);
```

Parameters

name1	Name of the first input or output that is attached to a component. This name must comply with the registry naming conventions (see page 53 of Chapter 2).
name2	Name of the second input or output that is attached to a component. This name must comply with the registry naming conventions.

Description

This macro creates the array

```
{ { name1, compInputRequired },
  { name2, compInputRequired },
  { Null,  0 } }
```

which is needed in **TSA_COMP_DEF_IO_COMPONENT**, **TSA_COMP_DEF_I_COMPONENT**, or **TSA_COMP_DEF_O_COMPONENT** as parameter “inputs” or “outputs.”

TSA_COMP_BUILD_ARG_LIST_3

```
TSA_COMP_BUILD_ARG_LIST_3(
   String   name1,
   String   name2,
   String   name3
);
```

Parameters

name1	Name of the first input or output that is attached to a component. This name must comply with the registry naming conventions (see page 53 of Chapter 2).
name2	Name of the second input or output that is attached to a component. This name must comply with the registry naming conventions.
name3	Name of the third input or output that is attached to a component. This name must comply with the registry naming conventions.

Description

This macro creates the array

```
{ { name1, compInputRequired },
  { name2, compInputRequired },
  { name3, compInputRequired },
  { Null,  Ø } }
```

which is needed in **TSA_COMP_DEF_IO_COMPONENT**, **TSA_COMP_DEF_I_COMPONENT**, or **TSA_COMP_DEF_O_COMPONENT** as parameter “inputs” or “outputs.”

 © 1999 Philips Semiconductors 10/08/99

TSA_COMP_BUILD_ARG_LIST_1_M

```
TSA_COMP_BUILD_ARG_LIST_1_M(
   String                 name1,
   compInputQualifier_t  qualifier1
);
```

Parameters

`name1`	Name of the first input or output that is attached to a component. This name must comply with the registry naming conventions (see page 53 of Chapter 2).
`qualifier1`	Describes the attributes of the symbol **name1**.

Description

This macro creates the array

```
{ {name1, qualifier1}, { Null, 0} }
```

that is needed in **TSA_COMP_DEF_IO_COMPONENT**, **TSA_COMP_DEF_I_COMPONENT**, or **TSA_COMP_DEF_O_COMPONENT**. The qualifier is ignored for outputs.

TSA_COMP_BUILD_ARG_LIST_2_M

```
TSA_COMP_BUILD_ARG_LIST_2_M(
   String                 name1,
   compInputQualifier_t   qualifier1,
   String                 name2,
   compInputQualifier_t   qualifier2
);
```

Parameters

`name1`	Name of the first input or output that is attached to a component. This name must comply with the registry naming conventions (see page 53 of Chapter 2).
`qualifier1`	Describes the attributes of the symbol **name1**.
`name2`	Name of the second input or output that is attached to a component. This name must comply with the registry naming conventions (see Chapter 2, page 53).
`qualifier2`	Describes the attributes of the symbol **name2**.

Description

This macro creates the array

```
{ { name1, compInputRequired },
  { name2, compInputRequired },
  { Null,  0 } }
```

which is needed in **TSA_COMP_DEF_IO_COMPONENT**, **TSA_COMP_DEF_I_COMPONENT**, or **TSA_COMP_DEF_O_COMPONENT**. The qualifiers are ignored for outputs.

 ©1999 Philips Semiconductors 10/08/99

TSA_COMP_BUILD_ARG_LIST_3_M

```
TSA_COMP_BUILD_ARG_LIST_3_M(
   String                 name1,
   compInputQualifier_t   qualifier1,
   String                 name2,
   compInputQualifier_t   qualifier2,
   String                 name3,
   compInputQualifier_t   qualifier3
);
```

Parameters

`name1`	Name of the first input or output that is attached to a component. This name must comply with the registry naming conventions (see page 53 of Chapter 2).
`qualifier1`	Describes the attributes of the symbol **name1**.
`name2`	Name of the second input or output that is attached to a component. This name must comply with the registry naming conventions (see page 53 of Chapter 2).
`qualifier2`	Describes the attributes of the symbol **name2**.
`name3`	Name of the third input or output that is attached to a component. This name must comply with the registry naming conventions (see page 53 of Chapter 2).
`qualifier3`	Describes the attributes of the symbol **name3**.

Description

This macro creates the array

```
{ { name1, compInputRequired },
  { name2, compInputRequired },
  { name3, compInputRequired },
  { Null,  0 } }
```

which is needed in **TSA_COMP_DEF_IO_COMPONENT**, **TSA_COMP_DEF_I_COMPONENT**, or **TSA_COMP_DEF_O_COMPONENT**. The qualifiers are ignored for outputs.

 ©1999 Philips Semiconductors 10/08/99

Chapter 4

Clock Support API

Clock Support Overview

The clock support module provides some generic clock functions to components or applications. You can have an unlimited number of clock instances, each with their own frequency.

Only if the alarms must be set on a clock will the clock support module use a timer. All clock module instances share the one timer.

Clock Support API Data Structures

This section describes the clock support API data structures found in the file tsaClock.h.

 ©1999 Philips Semiconductors 10/08/99

tsaClockFunc_t

```
typedef void (*tsaClockFunc_t)(
   Int instance,
   void* args
);
```

Parameters

`instance`	A clock instance, as generated by a call to **tsaClockOpen**.
`args`	Arguments, determined by the user.

Description

A callback function to be called when a clock instance's alarm goes off.

tsaClockCapabilities_t

```
typedef struct {
   ptsaDefaultCapabilities_t   defaultCapabilities;
} tsaClockCapabilities_t, *ptsaClockCapabilities_t;
```

Fields

defaultCapabilities	Pointer to default capabilities struct. (See tsa.h.)

tsaClockInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t  defaultSetup;
   UInt32                      frequency;
   Int                         numAlarms;
} tsaClockInstanceSetup_t, *ptsaClockInstanceSetup_t;
```

Fields

defaultSetup	Pointer to default instance setup struct. (See tsa.h.)
frequency	Frequency at which the clock should run.
numAlarms	Maximum number outstanding alarms the clock instance can have.

 © 1999 Philips Semiconductors 10/08/99

Clock Support API Functions

This section presents the clock support API functions present in the file tsaClock.h.

Name	Page
tsaClockGetCapabilities	100
tsaClockOpen	101
tsaClockClose	102
tsaClockGetInstanceSetup	103
tsaClockInstanceSetup	104
tsaClockStart	105
tsaClockStop	106
tsaClockGetTime	107
tsaClockSetTime	108
tsaClockSetAlarm	109
tsaClockTimeDiff	110
tsaClockTimeAdd	111
tsaClockTimeSub	112
tsaClockTimeDiv	113
tsaClockTimeMul	114

tsaClockGetCapabilities

```
tmLibappErr_t tsaClockGetCapabilities(
   ptsaClockCapabilities_t    *cap
);
```

Parameters

cap	Pointer, returned, to a capabilities structure.

Return Codes

TMLIBAPP_OK	Success.

Description

Return a pointer to the capabilities of the clock. There is no precondition for this function.

 © 1999 Philips Semiconductors 10/08/99

tsaClockOpen

```
tmLibappErr_t tsaClockOpen (
   Int   *instance
);
```

Parameters

instance	Pointer, returned, to the clock instance.

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_MEMALLOC_FAILED	Memory allocation for the instance variables failed.
CL_ERR_PROC_CAP	Problem obtaining the processor frequency (uses **procGetCapabilities**).

Description

Assigns an instance of the clock for use.

tsaClockClose

```
tmLibappErr_t tsaClockClose (
   Int   instance
);
```

Parameters

`instance`	The clock instance to close.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Not a valid instance

Description

Unassigns the clock instance for usage. The clock instance must have been opened with **tsalClockOpen**.

 © 1999 Philips Semiconductors 10/08/99

tsaClockGetInstanceSetup

```
tmLibappErr_t tsaClockGetInstanceSetup(
   Int                      instance,
   tsaClockInstanceSetup_t  &setup
);
```

Parameters

instance	A clock instance, generated by a call to tsaClock-Open.
setup	Address at which to return the setup structure.

Return Codes

TMLIBAPP_OK	Success.

Description

Returns the **tsaClockInstanceSetup_t** with the present setup of the current instance.

tsaClockInstanceSetup

```
tmLibappErr_t tsaClockInstanceSetup (
   Int                       instance,
   tsaClockInstanceSetup_t  *setup
);
```

Parameters

`instance`	A clock instance, generated by **tsaClockOpen**.
`setup`	Pointer to the FR setup structure **tmalClockInstanceSetup_t.**

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Not a valid instance.
`CL_ERR_INV_FREQ`	Frequency setup parameter has an invalid value (it is larger than the CPU clock frequency).
`CL_ERR_ALARMS_OUTSTANDING`	The clock has outstanding alarms that have not been serviced.
`CL_ERR_NO_TIMER`	The clock instance could not obtain a timer (used for alarm functionality only).
`CL_ERR_TIMER_SETUP`	Setup of the timer failed.
`TMLIBAPP_ERR_MEMALLOC_FAILED`	Memory allocation for the alarms failed.

Description

Sets up the instance of the clock. Setup includes the clock frequency and number of possible alarms.

 © 1999 Philips Semiconductors 10/08/99

tsaClockStart

```
tmLibappErr_t tsaClockStart (
   Int   instance
);
```

Parameters

`instance`	A clock instance, generated by **tsaClockOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Not a valid instance.
`TMLIBAPP_ERR_NOT_SETUP`	Instance has not been set up previously.
`TMLIBAPP_ERR_ALREADY_STARTED`	Instance has already been started.

Description

Starts data streaming for the clock instance.

tsaClockStop

```
tmLibappErr_t tsaClockStop (
   Int   instance
);
```

Parameters

`instance`	A clock instance, generated by **tsaClockOpen**.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Not a valid instance.
`TMLIBAPP_ERR_NOT_SETUP`	Instance has not been setup previously.
`TMLIBAPP_ERR_ALREADY_STOPPED`	Instance has already been stopped.

Description

Stops data streaming for the clock instance.

 © 1999 Philips Semiconductors 10/08/99

tsaClockGetTime

```
tmLibappErr_t tsaClockGetTime (
   Int              instance,
   tmTimeStamp_t   *time
);
```

Parameters

`instance`	A clock instance, generated by **tsaClockOpen**.
`time`	Pointer to the timestamp.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Not a valid instance.
`TMLIBAPP_ERR_NOT_SETUP`	Instance has not been set up previously.

Description

Returns the current clock time.

tsaClockSetTime

```
tmLibappErr_t tsaClockSetTime (
   Int              instance,
   ptmTimeStamp_t   time
);
```

Parameters

`instance`	A clock instance, generated by **tsaClockOpen**.
`time`	Pointer to timestamp (a return value).

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Not a valid instance.
`TMLIBAPP_ERR_NOT_SETUP`	Instance has not been setup previously.

Description

Sets the current clock time.

 © 1999 Philips Semiconductors 10/08/99

tsaClockSetAlarm

```
tmLibappErr_t tsaClockSetAlarm (
   Int              instance,
   ptmTimeStamp_t  time,
   Bool             periodic
   tsaClockFunc_t  func,
   void           *args
);
```

Parameters

`instance`	A clock instance, generated by **tsaClockOpen**.
`time`	Pointer to timestamp containing the alarm time.
`periodic`	If True, the alarm is enabled periodically. Otherwise, the alarm occur once only.
`func`	Pointer to function to be called when alarm expires.
`args`	Pointer to argument provided to callback function at the time the alarm expires.

Return Codes

`TMLIBAPP_OK`	Success.
`TMLIBAPP_ERR_INVALID_INSTANCE`	Not a valid instance.
`TMLIBAPP_ERR_NOT_SETUP`	Instance has not been set up previously.
`CL_ERR_LATE`	The time has already passed.
`CL_ERR_NO_UNUSED_ALRM`	All alarm slots are currently in use. (The number of alarm slots is one of the setup parameters.)

Description

Sets an alarm at the specified time. If **periodic** is True, the alarm is set periodically with time cycle given by argument time. That is, if the time argument sets a periodic alarm for 30 ms, then the alarm will be set at 30, 60, 90, 120 ms, and so on.

Note
The callback functions are called by the **TCS_handler** function. Interrupts are disabled during the call. Therefore, these should be short and cannot perform tasks which used interrupts (such as printing or communicating over IIC).

tsaClockTimeDiff

```
tmLibappErr_t tsaClockTimeDiff (
   Int              instance,
   ptmTimeStamp_t   time,
   Int             *timediff
);
```

Parameters

instance	A clock instance, generated by **tsaClockOpen**.
time	Pointer to timestamp containing the time.
timediff	Pointer to time difference (a return value).

Return Codes

TMLIBAPP_OK	Success.
TMLIBAPP_ERR_INVALID_INSTANCE	Not a valid instance.
TMLIBAPP_ERR_NOT_SETUP	Instance has not been set up previously.

Description

Calculates the integer time difference between a timestamp and the current clock value.

 © 1999 Philips Semiconductors 10/08/99

tsaClockTimeAdd

```
tmLibappErr_t tsaClockTimeAdd (
   ptmTimeStamp_t   time1,
   ptmTimeStamp_t   time2
);
```

Parameters

`time1`	Pointer to one timestamp.
`time2`	Pointer to a second timestamp.

Return Codes

`TMLIBAPP_OK`	Success.

Description

Adds two timestamp values and returns the result in the first timestamp.

tsaClockTimeSub

```
tmLibappErr_t tsaClockTimeSub (
   ptmTimeStamp_t   time1,
   ptmTimeStamp_t   time2
);
```

Parameters

time1	Pointer to one timestamp.
time2	Pointer to a second timestamp.

Return Codes

TMLIBAPP_OK	Success.

Description

Subtracts the second timestamp value from the first and returns the result in the first timestamp.

 © 1999 Philips Semiconductors 10/08/99

tsaClockTimeDiv

```
tmLibappErr_t tsaClockTimeDiv (
   ptmTimeStamp_t   time,
   float            value
);
```

Parameters

`time`	Pointer to timestamp (also used to store the result).
`value`	Floating point divisor.

Return Codes

`TMLIBAPP_OK`	Success.

Description

Divides the timestamp value by the floating point value and returns the result in time-stamp.

tsaClockTimeMul

```
tmLibappErr_t tsaClockTimeMul (
   ptmTimeStamp_t   time,
   float            value
);
```

Parameters

`time`	Pointer to timestamp (also used to store result).
`value`	Floating point multiplier.

Return Codes

`TMLIBAPP_OK`	Success.

Description

Multiplies timestamp by floating point value and returns the result in timestamp.

 © 1999 Philips Semiconductors 10/08/99

Chapter 5

TSA Timer (Stimer) API

TSA Timer API Overview

The TSA Timer library is a generalized timer libarary. The timer can be programed to generate alarms after specified delays from current time. It also provides generation of periodic alarms with a given time period. Unlike the Trimedia Clock library, this Timer is a software timer, based on periodic events generated by the OS. This timer overcomes the limitations of Trimedia Clock library in that the callback function is not called as a handler function, hence can be used for wider variety of applications.

TSA Timer Errors

No error callback functions and completion functions are provided in ths library.

TSA Timer Data Structures

This section presents the Timer library data structures.

 ©1999 Philips Semiconductors 10/08/99

tsaTimerCapabilites_t

```
typedef struct {
   tsaTimerDefaultCapabilites_t   defaultCapabilities;
} tpCapabilities_t, *ptpCapabilites_t;
```

Fields

`defaultCapabilities`	Default Capabilites structure.

Description

Provided for conformance wih TSA.

tsaTimerFunc_t

```
typedef void (*tsaTimerFunc_t)(
   Int instance,
   void*args
);
```

Description

This is the typedef for the callback function that is called when an alarm is triggered. You should provide this function. The instance is the pointer to the alarm that was triggered.

tsaTimerInstanceSetup_t

```
typedef struct {
   ptsaDefaultInstanceSetup_t   defaultSetup;
   UInt32                       resolution;
   Int                          numAlarms;
   Int                          priority;
   Int                          standbyPriority;
   Int                          timerEvent;
} tsaTimerInstanceSetup_t; *ptsaTimerInstanceSetup_t;
```

Fields

`defaultSetup`	For Conformance with TSA architecture
`resolution`	The resolution of Timer, alarms are triggered at time intervals integral multiples of resolution. Provided in milliseconds.
`numAlarms`	Maximum number of alarms allowed, currently this is unused.
`priority`	The priority at with the timer task runs.
`standbyPriority`	Priority of timer task when no alarms are pending.
`timerEvent`	The event to be used by timer for processing the alarms.

Description

This structure passes setup values to the **tsaTimerInstanceSetup** function. You can indicate the resolution, event and priority to use. The alarms are triggered at integral multiple of resolution. If an alarm delay is between n×resolution and (n+1)×resolution, the alarm will be triggered at (n+1)×resolution.

 ©1999 Philips Semiconductors 10/08/99

tsaTimerAlarmSetup_t

```
typedef struct {
   Int              delay;
   Bool             periodic;
   tsaTimerFunc_t   func;
   void*            args;
} tsaTimerAlarmSetup_t; *ptsaTimerAlarmSetup_t;
```

Fields

delay	Delay from the current time before the alarm is triggered.
periodic	Whether the alarm is periodic.
func	The callback function to be called when alarm is triggered.
args	Pointer to args to be passed to callback funciton.

Description

This structure sets up the alarms with function **tsaTimerSetupAlarm**.

TSA Timer Functions

This section presents the Timer device library functions.

 © 1999 Philips Semiconductors 10/08/99

tsaTimerGetCapabilities

```
tmLibdevErr_t tsaTimerGetCapabilites(
   ptpCapabilities_t   *pCap
);
```

Parameters

pCap	Pooointer, returned, to the capabilities structure.

Return Codes

TMLIBDEV_OK	Success.

Description

This function returns a pointer to the capabilites strucuture for the timer library. This function is provided for conformance with the other TSA architecture libraries.

tsaTimerOpen

```
tmLibdevErr_t tsaTimerOpen(
   Int*   instance;
);
```

Parameters

instance	Used to return handle to the timer instance.

Return Codes

TMLIBDEV_OK	Success.
TMLIBDEV_ERR_MEMALLOC_FAILED	Failed to allocate memory required for this instance.

Description

This function opens a new timer and returns the pointer to the timer Instance.

 © 1999 Philips Semiconductors 10/08/99

tsTimerClose

```
tmLibdevErr_t tsaTimerClose(
   Int   instance;
);
```

Parameters

`instance`	Instance value as returned by **tsaTimerOpen**.

Return Codes

`TMLIBDEV_OK`	Success.

Description

Close the timer Instance and release system resources acquired by **tsaTimerOpen**.

tsaTimerGetInstanceSetup

```
tmLibdevErr_t tsaTimerGetInstanceSetup(
   Int                         instance,
   ptsaTimerInstanceSetup_t   *setup
);
```

Parameters

`instance`	Instance handle previously created by **tsaTimer-Open**.
`setup`	pointer to the timer Instance setup structure.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This function returns pointer to the preallocated setup structure for the timer. This structure can be used to set up the timer using function **tsaTimerInstanceSetup**.

 ©1999 Philips Semiconductors 10/08/99

tsaTimerInstanceSetup

```
tmLibdevErr_t tsaTimerInstanceSetup(
   Int                        instance,
   ptsaTimerInstaceSetup_t    setup
);
```

Parameters

`instance`	Instance handle previously created by **tpOpenM**.
`setup`	Pointer to data structure containing the setup information.

Return Codes

`TMLIBDEV_OK`	Success.
TP_ERR_ALREADY_SETUP	This instance has been already setup.

Description

Initializes the timer task and allocates the system resources required for the timer library. This also sets up the proper priorities of tasks and resolution to be used for the timer

tsaTimerStart

```
extern tmLibdevErr_t tsaTimerStart(
   Int    instance
);
```

Parameters

`instance`	Instance handle previously created by **tsaTimer-Open**.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This function starts the timer, the alarms added become active only after starting the timer. The alarms can be themselves added and deleted with the timer is still active.

 © 1999 Philips Semiconductors 10/08/99

tsaTimerStop

```
extern tmLibdevErr_t tsaTimerStop(
   Int   instance
);
```

Parameters

instance	Instance handle previously created by **tsaTimer-Open**.

Return Codes

TMLIBDEV_OK	Success.

Description

Stop the timer. This disables generation of alarms.

tsaTimerCreateAlarm

```
extern tmLibdevErr_t tsaTimerCreateAlarm(
   Int    instance,
   Int*   alarmInst
);
```

Parameters

`instance`	Instance handle previously created by **tsaTimer-Open**.
`alarmInst`	Pointer to alarm returned by function.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_MEMALLOC_FAILED`	Failed to allocate memory required for this instance.

Description

This function is called to create an alarm for a specified timer. The alarm is disabled until both the timer and the alarm are started.

 ©1999 Philips Semiconductors 10/08/99

tsaTimerDestroyAlarm

```
extern tmLibdevErr_t tsaTimerDestroyAlarm(
   Int    instance,
   Int   alarmInst
);
```

Parameters

`instance`	Instance handle previously created by **tsaTimer-Open**.
alarmInst	Alarm handle previously created by tsaTimerCre-ateAlarm.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This function is called to destroy the alarm instance, the user memory is not freed by the API and is kept to be reused later.

tsaTimerSetupAlarm

```
extern tmLibdevErr_t tsaTimerSetupAlarm(
   Int    instance
);
```

Parameters

`instance`	Instance handle previously created by **tsaTimer-Open**.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This function is used to set up the parameters for the alarm, like callback function, delay and arguments to callback function. Also if the alarm is periodic. This function updates the alarm data structures appropriately.

 © 1999 Philips Semiconductors 10/08/99

tsaTimerStartAlarm

```
extern tmLibdevErr_t tsaTimerStartAlarm(
   Int    instance
);
```

Parameters

`instance`	Instance handle previously created by **tsaTimer-Open**.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This function starts the alarm, the alarm is insert into the timer to be scheduled at proper time. The timer also should be started by calling **tsaTimerStart** before alarms can be triggered by the timer.

tsaTimerStopAlarm

```
extern tmLibdevErr_t tsaTimerStopAlarm(
   Int    instance
);
```

Parameters

`instance`	instance handle previously created by **tsaTimer-Open**.

Return Codes

`TMLIBDEV_OK`	Success.

Description

This function is to stop the triggering of alarms. In particular the periodic alarms will continue to trigger untill the alarm is stopped by calling this function or the Timer itself is stopped.

 © 1999 Philips Semiconductors 10/08/99

Chapter 6

TriMedia Memory Manager API

Introduction

Many application programmers are not overly concerned with memory management. When they need only to allocate modest amounts of memory to assure that their applications function correctly, they can rely on the ANSI function **malloc** to do a satisfactory job.

Concerns change when applications run against the boundaries of the available memory or when other factors such as memory recycle schemes or memory leak debugging play a larger role. Memory allocation efficiency is especially critical for certain kinds of applications. To satisfy this range of needs, the TriMedia SDE provides several different memory managers.

Memory Management Trade-Offs

By design, each SDE memory manager has its own set of strengths and weaknesses. Each might coexist with, replace, or extend another, depending on the situation, but overall its performance is determined by its handling of the issues below.

- Controlling internal fragmentation.
- Controlling external fragmentation.
- Real-time, deterministic allocation performance.
- Ease of deallocation.
- Additional functionality.
- Implementation size.

This chapter provides an overview of the SDE memory managers and describes the trade-offs involved in choosing one over another.

Overview

Figure 2 shows all TriMedia system software components that can be used for allocating and deallocating blocks of memory.

These components can roughly be divided in two groups. The group on the left contains a number of memory managers that are implemented on top of one another. In this group, all allocated memory is eventually obtained via the simplest and lowest level allocator **sbrk**. The group on the right contains unrelated memory managers, some of which are available only in pSOS-based applications.

 ©1999 Philips Semiconductors 10/08/99

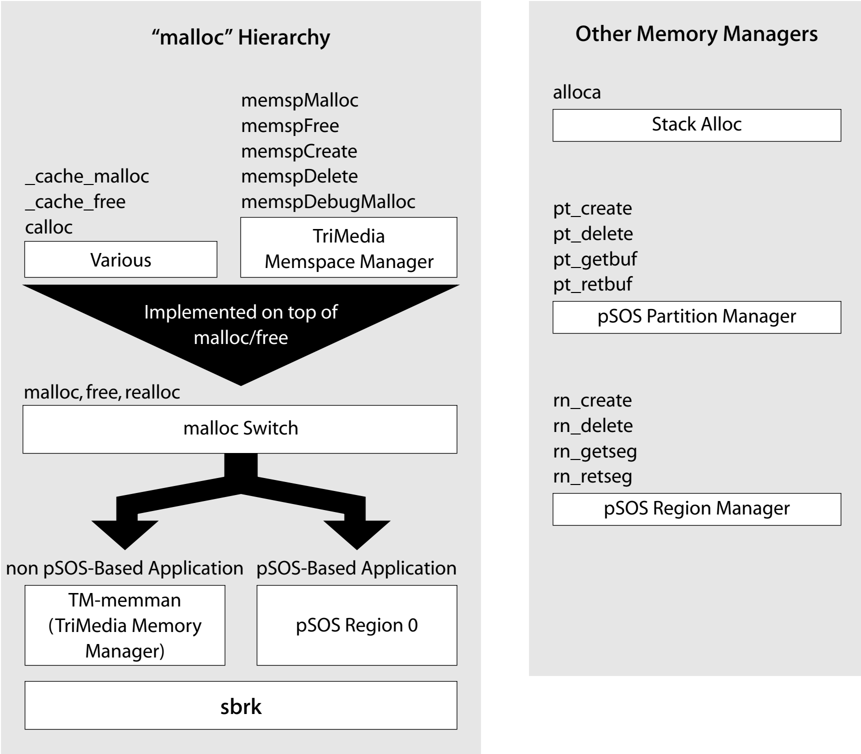


Figure 2 Memory Management Hierarchies

Memory Fragmentation

Memory fragmentation is the general problem of memory being inaccessible to the application because of the decisions made by the memory manager itself. *Internal* fragmentation refers to additional memory that is allocated per block (for whatever reason), but that is unused by the application and is hence wasted. *External* fragmentation refers to memory outside of allocated blocks that, although nominally available, cannot be used by the application because the memory is not contiguous.

A modest example of *internal* fragmentation is the situation where a memory manager returns word-aligned memory blocks. Any allocation of sizes that are not exact multiples of the word size result in some bytes being lost to padding. For example, a request for 7 bytes generally results in the allocation of 8 bytes or more. At least one byte is then lost due to internal fragmentation.

An example of *external* fragmentation is the scenario in which the allocation of a 4 megabyte block from an 8 megabyte memory area is not possible because of previously allocated blocks that are awkwardly positioned, as shown in Figure 3.

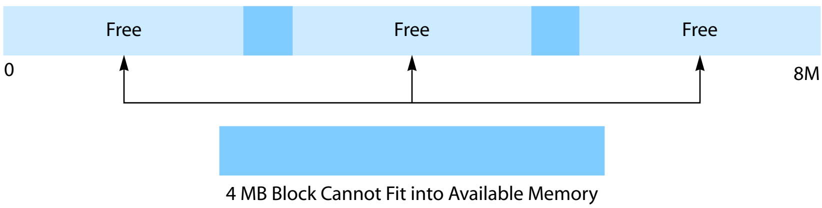


Figure 3 External Fragmentation

Unlike internal fragmentation, the issue of external fragmentation is directly influenced by the memory needs of the application. The smaller the requested block size, the greater the amount of free memory will be accessible. Conversely, if an application demands larger memory blocks, the more often it will suffer from the effects of memory fragmentation.

More precisely, external fragmentation tends to rise according to the number of (i.e. variety of) block sizes allocated from the same memory range. One extreme situation is formed by allocations of only one single block size x. In this case, inaccessible free memory would theoretically never be more than x, and any fragmentation would be small and predictable. The other extreme is what is illustrated in Figure 3.

There are several approaches to the problem of memory fragmentation. However, external and internal fragmentation are invariably on opposite sides of the negotiation, so fragmentation, though reducible, will never be eliminated.

Heap Partitioning

One approach, shown in Figure 4, consists of partitioning the available free memory into different heaps that are dedicated to different block sizes.

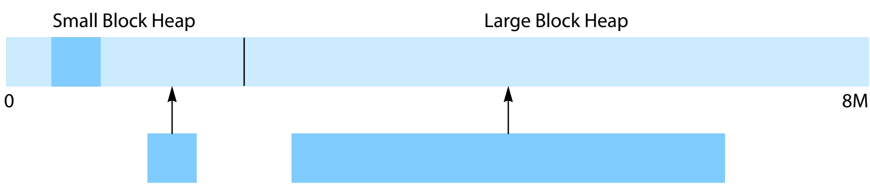


Figure 4 Heap Partitioning

This partitions (and reduces) memory fragmentation in two ways:

1. For each heap, the variety of block sizes is less, thereby reducing fragmentation.
2. Where the memory manager for the heap allows heap extension, each extension is just another large block to be allocated from the large block heap.

A disadvantage of this approach is that an individual heap is usually never filled. Because free space in one heap is generally unavailable for allocation into another heap, use of different heaps introduces a new, more internal, form of memory fragmentation.

Heap partitioning is supported by the pSOS Partition Manager and the pSOS Region Manager, both of which allow arbitrary ranges of memory to be cast into a heap. This memory range can be some large global variable, for instance, or a memory block that has been explicitly obtained from another heap. Following pSOS tradition, this other heap is usually “region 0,” which is created during pSOS initialization to hold all available free memory.

Both heap partitioning and heap extension are supported by the TriMedia Memspace Manager, which uses the underlying system memory manager (**malloc**) to obtain memory for its memspaces and memspace extensions. Extensions are automatically attempted when allocation requests to a memory space cannot be fulfilled with the memory currently allocated to that memory space.

Memory Units

Another approach to fragmentation is the use of memory units, where each allocated memory block is rounded up to a multiple of the unit size, as shown in Figure 5.

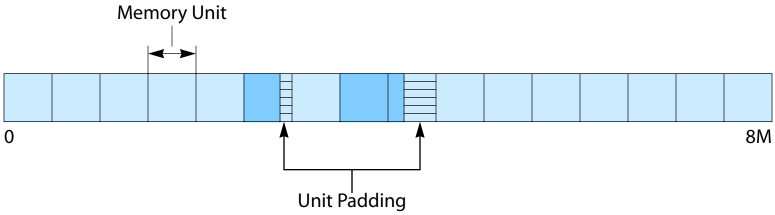


Figure 5 Memory Units

This prevents occurrence of blocks smaller than the unit size, thereby reducing the variety of block sizes. However, rounding adds padding in a memory block whenever the requested size is not an exact multiple of the unit size. Therefore, the use of memory units generally reduces external fragmentation at the cost of internal fragmentation.

Memory units are supported by the pSOS Partition Manager and the pSOS Region Manager, both of which allow specification of arbitrary unit size at partition/region creation. The difference between pSOS partitions and pSOS regions is that the partitions allow only allocation of single units, whereas the regions allow allocation of arbitrary block sizes, which are internally rounded up to unit multiples.

The TriMedia Memspace Manager is a mixture of these. For each memspace, it automatically maintains internal partitions for small sizes (when these are used), and uses one internal region for large block sizes. The division between “large” and “small” is fixed at 60 bytes. In addition, since an internal partition is created for each small block size that is used, no unit padding need be added, and the internal fragmentation is bounded.

Allocation Performance

A memory manager also has to balance the effect of fragmentation on the one hand against allocation time overhead on the other.

In soft real-time systems, the average allocation performance is most important, while time-critical systems generally like predictability, which suggests a hard upperbounds on the allocation time. However, multimedia applications (the target of the TriMedia SDE), generally do not impose hard real-time requirements onto their memory managers, especially applications that do all or most memory allocation up front, during startup.

Of the four memory managers shown in Figure 2, only one, the pSOS Partition Manager, has hard real-time properties, though at the cost of reduced functionality. The pSOS Region Manager is a close second. It does everything to remain predictable, but at the cost of increased heap fragmentation.

Both of the TriMedia memory managers attempt to minimize heap fragmentation while maintaining a good average performance, but do not claim to be hard real-time. Both of them might have occasional performance glitches, such as when they must escape to lower level memory managers for heap extension.

Additional Functionality

In addition to allocation/deallocation facilities, the SDE memory managers provide the following additional functionalities.

Functionality	**Memory Manager**
Collective deallocation of related memory blocks. Deallocation of entire heap.	pSOS Partition Manager pSOS Region Manager TriMedia Memspace Manager
Internal consistency checking. Automatic invalidation of deallocated memory blocks. Guard areas around allocated memory blocks. Inspection of heap statistics and of allocated blocks. Tracking where particular blocks have been allocated.	TriMedia Memspace Manager in debug mode
Allocation of TM data cache-aligned memory blocks.	**_cache_malloc** TriMedia Memspace Manager
Allocation of zero-initialized memory.	**calloc**, a variant of **malloc**
Allocation in current stack frame.	**alloca**

 © 1999 Philips Semiconductors 10/08/99

The "malloc" Hierarchy

This section describes what happens to available SDRAM after a TriMedia application starts executing. This discussion is closely related to the group of memory managers at the left side of Figure 2 on page 135. As shown in Figure 6, behavior is dependent on whether the application is pSOS-based.

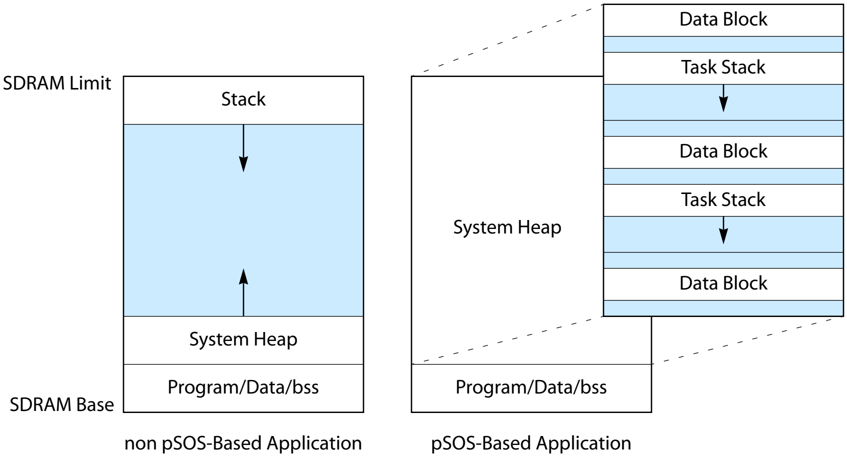


Figure 6 Memory Management Maps

All pSOS applications start executing identically to non-pSOS based applications, with pSOS gradually taking over after the core libraries of the SDE are initialized. This is an important juncture for memory management. Such ANSI functions as **malloc** and **free** are still to be mapped to pSOS memory management, while several core SDE features (I/O, dynamic loader, or user-defined components like flash-file system drivers) might already have needed allocation of memory during initialization, in a stage at which pSOS is not yet up and running.

In cooperation with the downloader, which is documented in Chapter 12, *Downloader API*, the first instructions of each application set up a memory map as shown in the left side of Figure 6. The stack pointer is initialized to the top of SDRAM and grows downward, and a system heap pointer is initialized to just after the loaded program. This heap pointer is managed by the low level system function **sbrk**, which implements a very rudimentary memory management facility. Using **sbrk**, only block allocation is possible, by moving the system heap pointer upwards. Memory obtained in this way can never be given back to the system heap.

Note
The system function **sbrk** should not be used by applications directly. It is intended as basis for higher level memory managers.

One of these managers is the default TriMedia Memory Manager (TM-memman). During startup, and by default in non-pSOS based applications, TM-memman handles all memory allocation. It is a small, general purpose memory manager that gradually extends the system heap (using **sbrk**) whenever it needs to extend its own heap. During application initialization for instance, TM-memman is used for allocation of memory that is needed for IO driver installation, and (in dynamic loader-based systems) for allocating memory needed for holding dynamic libraries that are loaded during application startup. These are mostly the dynamic libraries that have been linked in **immediate** mode to the initializing application. See Chapter 13, *Dynamic Linking API*, for information.

After core library initialization, the memory organization undergoes a drastic change for pSOS-based applications. While pSOS takes over, the entire system heap is allocated and given to “region 0,” to be managed by the pSOS Region Manager. Similarly to other pSOS tasks later on during execution, the root task is created with a stack allocated from this memory region, and the startup stack is no longer needed. From this point forward, the stack pointer always points to a task stack that has been allocated somewhere in region 0.

More importantly, pSOS convention requires that all **malloc/free** calls be further managed by the pSOS Region Manager. This is achieved by performing a dynamic switch that replaces the group of basic memory management functions (**malloc**, **free**, and **realloc)** by a group that uses region 0. This switch is an important milestone, in that memory allocated using functions **malloc** or **realloc** before this switch should never be passed to **realloc** or **free** after this switch.

It is important to note that all memory management services implemented on top of the **malloc** interface (as shown in Figure 2, page 135) will have their underlying memory manager silently replaced. This includes **calloc**, **_cache_malloc**, **_cache_free**, and the services of the TriMedia Memspace Manager.

Leaving TM-memman in Place

In certain cases, you might want to prevent this switch and let the **malloc** interface be mapped to TM-memman. One such case has to do with an implementation restriction of the pSOS Region Manager. Every region, including region 0, has a 32K upperbound to the number of units that it can manage. The unit size for region 0 is specified using **KC_RN0USIZE** in the pSOS application configuration file sys_conf.h and already defaults to 256 bytes, which places an upperbound of exactly 8 megabytes on the size of region 0. Larger SDRAM sizes can be handled by either increasing the unit size even more, which would substantially increase internal memory fragmentation, or by leaving TM-memman in place. The latter is achieved by means of **TCS_MALLOC_USE** in sys_conf.h.

```
/* TCS_MALLOC_USE:
 *    When YES, do *not* map malloc/free on rn_getseg/rn_free from region 0,
 *    as is standard in pSOS. Instead, use the TCS memory manager. The pSOS
 *    region manager might be more predictable in its real-time behavior,
 *    but this at the cost of larger unit sizes (see KC_RN0USIZE). Also,
 *    the pSOS region manager cannot hold more than 32K units, which is 8M
```

```
 *    with the current KC_RNOUSIZE, but proportionally less when the unit
 *    size is decreased. If this option is enabled, then define
 *    TCS_REGION0_SIZE such that region 0 does not occupy all free memory.
 */
#define TCS_MALLOC_USE     YES

/* TCS_REGION0_SIZE:
 *    When *not* defined, then all free memory (limited to 32K units) is
 *    given to region 0. Otherwise, region 0 is created with the specified
 *    size, but limited to 32K units; all other memory is available via the
 *    TCS memory manager. Use this option in combination with TCS_MALLOC_USE
 *    when the desired KC_RNOUSIZE results in a region 0 which is not able to
 *    contain all available SDRAM.
 */
#define TCS_REGION0_SIZE   512        /* empty region */
```

The TriMedia Memspace Manager

The table below summarizes how the TriMedia Memspace Manager is used.

Include File	$TCS/include/tmlib/Memspace.h
Libraries	$TCS/lib/<endian>/libmemspace.a $TCS/lib/<endian>/libmemspace_g.a
Sample Usage	tmcc main.c –lmemspace tmcc main.c –lmemspace_g

The TriMedia Memspace Manager is a high-level memory manager that serves three general purposes:

1. It supports creation of multiple, independent, and extendable heaps ("memory spaces") that can each be deleted at any time with all memory blocks currently allocated in them. Such memory spaces can be used for convenient collective deallocation of related memory blocks without having to keep track of each individual block. As described earlier, memory spaces may also help in reducing memory fragmentation.
2. In a debugging version, the Memspace Manager provides a number of tools that help in detecting heap-related memory errors:
   — Automatic internal consistency checking, mostly at the calls to the block deallocation function **memspFree**. Various checks are made to guarantee that the freed memory block is indeed a valid allocated block that has not been deallocated earlier. Invalid blocks trigger assertion failures.
   — Freed block corruption. The user contents of each freed block is overwritten by some magic pattern, increasing the likelihood of (early) problems when the application tries to use the contents of stale memory blocks.

— Guarded block allocation. The alternate memory allocation function **memspDebugMalloc** allows the allocated block to be surrounded by guard areas for detecting memory writes beyond the bounds of the allocated block. These guard areas are filled with magic patterns that are implicitly check-and-corrupted when the block is freed. All currently existing guarded blocks can be explicitly checked for block bound overwrites by using the function **memspCheck**.

3. In a debugging version, all currently existing memory blocks in a specific memory space that have been allocated by **memspDebugMalloc** can be listed using the function **memspPrintGuarded**. This shows the location in the program where the memory blocks have been allocated. This facility helps in analyzing memory leaks, and in observing the general allocation behavior of an application (for example, where the application spends its memory).

The following sections present an overview of memspace concepts and the functions to deal with them. The complete Memspace Manager API is included at the end of this chapter, starting on page 153.

Memspaces

Conceptually, memspaces are heaps that own a specific amount of memory in which they implement their own memory management scheme. A memspace has a name and an extension size, both assigned at creation. The name is for identification purposes (while debugging, for example). The extension size specifies the following:

1. The initial amount of memory that is assigned to the memspace at time of its creation.
2. The minimum amount of memory by which the memspace is extended when the currently owned amount of memory cannot satisfy a block allocation request. All memory extensions are allocated from the current system memory manager via calls to **malloc**.

The extension size reserves a certain amount of memory when the memory space is created, so all memory requests from the memspace up to the extension size are guaranteed to succeed. This reserved amount is no hard upperbound, in that the Memspace Manager will attempt a heap extension instead of immediately failing when a memspace's memory pool is depleted. However, this is no longer guaranteed to succeed.

The creation of a memspace results in a handle that is to be used in all further calls that operate on memspaces, as shown in the program below.

```
#include “tmlib/Memspace.h”

static void print_memspace(memspSpace space, Pointer data){
   memspSpaceInfo info;
   memspGetInfo(space, &info);
   printf(“\t----> memspace: ‘%s’\n”, info.name );
}

void main(){
   memspSpace s1,s2;
```

 ©1999 Philips Semiconductors 10/08/99

```
    printf("Before creation:\n");
    memspTraverseSpaces( print_memspace, Null );

    s1 = memspCreate( "large_blocks", 200000 );
    s2 = memspCreate( "small_blocks", 100000 );

    printf("After creation:\n");
    memspTraverseSpaces( print_memspace, Null );

    memspDelete( s1 );
    memspDelete( s2 );

    printf("After deletion:\n");
    memspTraverseSpaces( print_memspace, Null );
}
```

The output of this program is shown below.

```
tmcc main.c -lmemspace
tmsim a.out

Before creation:
   ----> memspace: 'System'
After creation:
   ----> memspace: 'System'
   ----> memspace: 'large_blocks'
   ----> memspace: 'small_blocks'
After deletion:
   ----> memspace: 'System'
```

The handles **s1** and **s2** are used in calls to **memspDelete**. A **Null** handle can be used as abbreviation of one special memspace: the *system* memspace. This special memspace is used for storing part of the administration of all user-created memspaces, and cannot be deleted. It can further be used as any other memspace, but it has an increment size of 0, which means that most memory requests (those of the "non-small" blocks) are directly passed to **malloc**.

API Summary

The following functions of the memspace API deal with entire memspaces:

memspCreate	Creates a memspace with specified name and extension size, and returns a handle, or **Null** when no memory could be allocated for it. Memspaces are created with their first memory extension.
memspDelete	Deletes a specified memspace, plus all memory ever allocated from it, and returns all its memory extensions to the system heap (using calls to **free**).
memspTraverseSpaces	Applies a specified function to all currently existing memory spaces.

memspGetInfo	Extracts information from the specified memspace. Information includes name, total amount of memory owned by the memspace, total amount of owned memory that is available for allocation, and largest free block. See example usage in the code example on page 142.
memspPrintGuarded	Prints a list onto the standard output stream (**stdout**) of all memory blocks that have been allocated (and not yet freed) from the specified memspace using **memspDebugMalloc** (described in *Allocation and Deallocation* starting on page 144). The list includes file name and line number as passed to this function.
memspCheck	Performs a consistency check on the internal state of the Memspace Manager. A call to this function might abort the program due to an assertion failure. Errors are reported onto the standard error stream (**stderr**).

Allocation and Deallocation

The basic function for allocating blocks of memory from memspaces is **memspMalloc**. This function can be used for allocating both "normal" and cache-aligned memory blocks. Cache-aligned blocks start at the boundary of a TM1 data cache page, and are silently padded at the end to completely fill the last data cache page. This padding prevents a cache-aligned memory block from sharing cache pages with other program data, which could be harmful if cache invalidate operations are made on the allocated block.

Any block, cache-aligned or not, and even memory blocks that have been allocated using **memspDebugMalloc**, can be freed using `memspFree` or resized using **memspRealloc**. For **memspFree**, this is illustrated in the code below, where it is used for deallocating both cache-aligned and normal blocks. Note that these calls to **memspFree** are actually redundant here—their memspace is deleted shortly afterwards.

```
static void print_memspace( memspSpace space, Pointer data ){
   Int i;
   memspSpaceInfo info;
   memspGetInfo(space, &info);

   printf( "\t\t-----------> memspace: '%s'\n", info.name );
   printf( "\t\t\t   total_size     : %d bytes\n", info.total_size     );
   printf( "\t\t\t   segment_size   : %d bytes\n", info.segment_size   );
   printf( "\t\t\t   increment_size : %d bytes\n", info.increment_size );

   printf( "\t\t\t\n" );
   printf( "\t\t\t   variable size block pool:\n" );

   printf( "\t\t\t     - total free space      : %d bytes\n",
           info.variable_block_info.total_free_space );

   printf( "\t\t\t     - largest free block    : %d bytes\n",
           info.variable_block_info.max_free_blocksize );

   printf( "\t\t\t     - amount of free blocks : %d\n",
           info.variable_block_info.nrof_free_blocks );
```

 ©1999 Philips Semiconductors 10/08/99

```
   printf( “\t\t\t\n” );
   printf( “\t\t\t   fixed size block pools:\n” );

   for( i=0; i<memspFastSizeBound; i++ ){
      if( info.small_block_info[i].amount_segments > 0 ){
          printf( “\t\t\t  block size= %d:\n”, i );
          printf( “\t\t\t     - amount of block segments : %d\n”,
                  info.small_block_info[i].amount_segments );
          printf( “\t\t\t     - amount of free blocks    : %d\n”,
                  info.small_block_info[i].nrof_free_blocks );
      }
   }
}
void main(){
   memspSpace s1;
   Pointer small, large, aligned;

   s1= memspCreate( “sample”, 20000 );

   print_memspace( s1, Null );

   small   = memspMalloc (s1,  4, 0                 );
   large   = memspMalloc (s1,100, 0                 );
   aligned = memspMalloc (s1, 34, memspCACHE_ALIGNED );

   print_memspace( s1, Null );
   memspFree( small   );
   memspFree( large   );
   memspFree( aligned );
   memspDelete(s1);
}
```

The output of this program is shown below.

```
tmcc main.c -lmemspace
tmsim a.out
      -----------> memspace: ‘sample’
         total_size     : 20032 bytes
         segment_size   : 4096 bytes
         increment_size : 20000 bytes

         variable size block pool:
           - total free space      : 20000 bytes
           - largest free block    : 20000 bytes
           - amount of free blocks : 1

         fixed size block pools:
      -----------> memspace: ‘sample’
         total_size     : 20032 bytes
         segment_size   : 4096 bytes
         increment_size : 20000 bytes

         variable size block pool:
           - total free space      : 15660 bytes
           - largest free block    : 11692 bytes
           - amount of free blocks : 3

         fixed size block pools:
         block size= 4:
           - amount of block segments : 1
           - amount of free blocks    : 1016
```

Memspace Organization

Internally, memspaces are organized as shown in Figure 7.

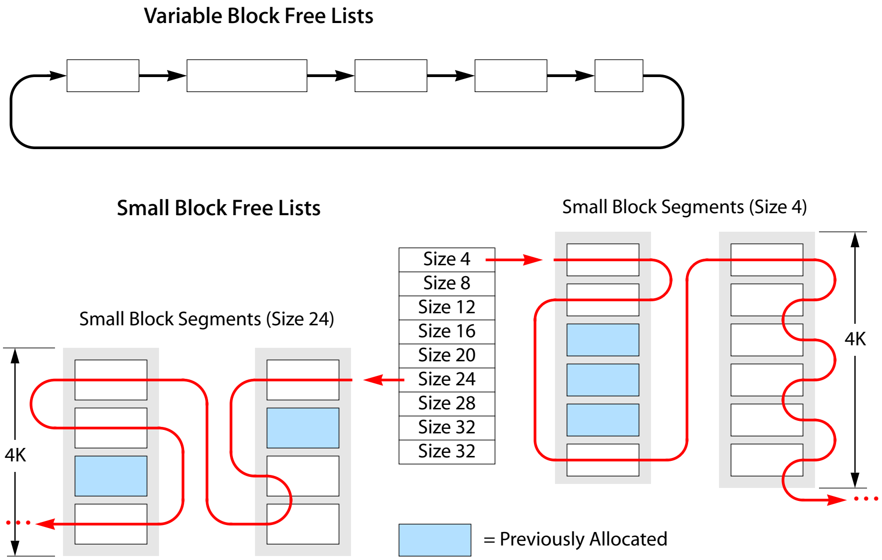


Figure 7 Internal Organization of a Memspace

Memspaces are organized as a combination of two things: a conventional memory manager that maintains a circular free list using a roving first-fit strategy, plus a page-based allocator for blocks smaller than 60 bytes. Such "small" blocks are allocated in 4K pages of identically-sized blocks. Separate free lists are maintained for each "small" size, so that allocation and deallocation for such small sizes can be performed very rapidly, and with very little memory manager administration overhead. The sample program output on page 145 shows that small block lists are indeed created only for sizes that are actually used. Furthermore, it shows that the overhead is minimal, one 4K segment yielding 1017 memory blocks of 4 bytes.

Tip

If you have a memspace allocating buffers of two different sizes, your application will eventually have many small packets in large buffer slots because of the first-fit algorithm. In this case, use two memspaces to decrease memory consumption.

Summary of Memspace API (Allocation/Deallocation)

The following functions of the memspace API deal with allocation and deallocation of memory blocks:

 ©1999 Philips Semiconductors 10/08/99

memspMalloc	Allocates a block from a specified memspace, of specified size. The block can be optionally cache-aligned.
memspDebugMalloc	The debugging version of **memspMalloc**. Allocates a block with guard areas and file/line number values attached. See "Overview of Debugging Features" starting on page 147 for more information. This function works as described only in the debugging version of the memspace library (lib_memspace_g.a). Otherwise, guard and file position information is ignored, and the function is identical to **memspMalloc**.
memspFree	Deallocates memory that has been previously allocated from **memspMalloc** or **memspDebugMalloc**, and that has not since been passed to either **memspFree** or **memspRealloc**.
memspRealloc	Adjusts the specified memory block to the specified size, and returns the block, which has possibly been moved. In any case, the returned block has the same properties as the input block. For instance, if the input block was cache-aligned, then the result of **memspRealloc** will also be cache-aligned. Similarly, when the input block was created (using **memspDebugMalloc**) with guard areas and file/line number information, the result will have the same guard areas and the same file/line number values.

Overview of Debugging Features

The debugging version of the Memspace Manager (lib_memspace_g.a) provides for two different debugging features that can aid in the detection of memory leaks or memory errors in an application. These features are disabled in the regular version of the Memspace Manager. They are independent of the concept of memory spaces, and hence can also be used for debugging conventional **malloc**-based applications.

The debugging features are described in the subsections below.

Consistency Checking of Internal Administration.

The feature provides two services. First, it validates memory blocks during various memspace operations (for example, checking whether blocks were indeed created, but not yet deleted by the Memspace Manager). Second, it attempts to detect internal corruption.

Using the alternate allocation function **memspDebugMalloc**, consistency checking can be explicitly enhanced by surrounding allocated memory blocks with guard areas, as shown in Figure 8.

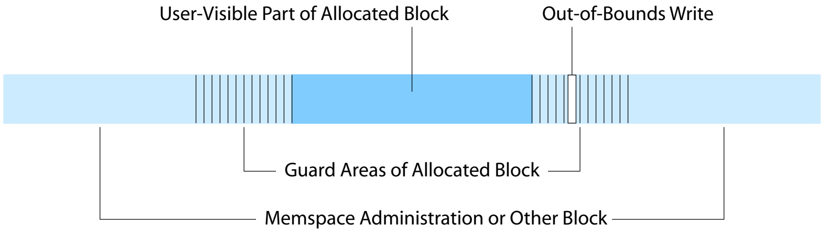


Figure 8 Guarded Memory Block

These guard areas have a dual purpose. First, by filling them with a specific pattern, the Memspace Manager is able to detect memory writes within these areas, which usually indicates out-of-bound block access. Second, these guard areas put some spacing between the memory block and the memory manager administration, thereby decreasing the likelihood of a fatal, unknown corruption due to out-of-bound writes. Instead, these are detected and reported. Sizes of the guard areas can be chosen on a per-block basis by means of parameters to **memspDebugMalloc**.

Consistency checking is mostly automatic, although it has some additional support in the form of function **memspCheck**, which explicitly triggers it at user-determined execution points. This means that the mere linking of an application to the Memspace Manager and the routing of all memory management calls to the Memspace Manager (preferably using **memspDebugMalloc** for allocation), will give a valuable level of consistency checking. Such a setup requires only minimal effort, and although the application's source code should be recompiled to include source location information, such a setup is also possible when these sources are not available or when recompilation is otherwise impracticable. This is described in *Redirecting Calls to malloc* starting on page 151.

Provoking Errors on Use of Stale Memory Blocks

This debugging feature provided by the Memspace Manager automatically invalidates the contents of freed memory blocks. Consequently, this tends to force application errors at an early stage in cases where the contents of already deallocated memory blocks are still used.

The use of such invalidated data as memory addresses (i.e. as pointers) will very likely show up as memory errors in some form or another shortly afterwards.

Tracking Allocated Memory

Using **memspDebugMalloc**, you can record the location in the application source code where memory blocks have been allocated. Two of the parameters to this function, an

 ©1999 Philips Semiconductors 10/08/99

integer and a string, will be kept with the resulting memory block and will appear whenever this block is listed, and as necessary in error messages and in the list produced by function **memspPrintGuarded**. Typically, the C file name and line number indicating the particular call to **memspDebugMalloc** are passed via these parameters by using the standard macros **__FILE__** and **__LINE__** provided by the C preprocessor.

Allocation tracking in this way has two important uses. First, a single call to **memspPrintGuarded** produces an allocation snapshot, showing exactly where all allocated memory is used at that particular moment. Second, a sequence of calls to **memspPrintGuarded** reveals all memory blocks that persist over time, and which are, consequently, potential memory leaks. Of course, the precondition to proper memory tracking is that all relevant memory will have been allocated using **memspDebugMalloc**.

Examples

The debugging features described are briefly illustrated in the code below.

```
#include “tmlib/Memspace.h”

/* Macro-redefine malloc, cache_malloc, and free to the corresponding
 * functions of the memspace library; this demonstrates how an existing,
 * non-memspace based application can be debugged for memory errors by
 * merely recompiling its source and running it. Note that all allocation is
 * done on the system memspace (referred to by ‘Null’): */

#define malloc(size) \
   memspDebugMalloc(Null,size, 0, 32,20,__FILE__,__LINE__ )

#define cache_malloc(size) \
   memspDebugMalloc(Null,size, memspCACHE_ALIGNED,
   32,20,__FILE__,__LINE__)

#define free(b) \
   memspFree(b)

void main(){

/* Do some allocations: */
   Address buggy         = malloc(34);
   Address block         = malloc(34);
   Address aligned_block = cache_malloc(34);

   printf(“- buggy              = 0x%08x\n”, buggy);
   printf(“- block              = 0x%08x\n”, block);
   printf(“- aligned_block      = 0x%08x\n”, aligned_block);

/* Write past the boundaries of block ‘buggy’, triggering a guarded block
 * error in the subsequent call to the guarded block list function, or in a
 * call to the consistency checker (memspCheck), or during the block’s
 * deallocation: */

   buggy[   -3 ]= 0;
   buggy[ 34+3 ]= 0;

   memspPrintGuarded(Null);

   printf(“\n----------------------\n”);
```

```
   memspCheck();

   printf("\n----------------------\n");

   free(buggy);
   free(block);
   free(aligned_block);
   free(buggy); /* spurious deallocation, will result in assertion failure */
}
```

The output of this program is shown below.

```
tmcc main.c -lmemspace_g
tmsim a.out
- buggy             = 0x0012aaa8
- block             = 0x0012c020
- aligned_block     = 0x0012e040
Guarded block 0x0012e060 ( 32/      64/  20) allocated at line   30 of
                             main.c, cache aligned
Guarded block 0x0012c040 ( 32/      34/  20) allocated at line   29 of
                             main.c
    Error: block 0x0012aaa8 allocated at line 28 of main.c in memspace
                             'System' corruption in guard space 3 bytes
                             before start
    Error: block 0x0012aaa8 allocated at line 28 of main.c in memspace
                             'System' corruption in guard space 3 bytes
                             after end
Guarded block 0x0012aac8 ( 32/      34/  20) allocated at line   28 of
                             main.c

-----------------------
    Error: block 0x0012aaa8 allocated at line 28 of main.c in memspace
                             'System' corruption in guard space 3 bytes
                             before start
    Error: block 0x0012aaa8 allocated at line 28 of main.c in memspace
                             'System' corruption in guard space 3 bytes
                             after end

-----------------------
    Error: block 0x0012aaa8 allocated at line 28 of main.c in memspace
                             'System' corruption in guard space 3 bytes
                             before start
    Error: block 0x0012aaa8 allocated at line 28 of main.c in memspace
                             'System' corruption in guard space 3 bytes
                             after end
assertion failed in memspace manager:   invalid block encountered
```

This example code first shows how **malloc** and **free** can be macro-redefined in terms of Memspace Manager calls. All allocation is redirected to memspace allocation from the system memspace, with guard spaces of 32 bytes before, and 20 bytes after the allocated blocks. Following that, three memory blocks are allocated, one of which ("**buggy**") is deliberately corrupted by writing beyond both its boundaries. This corruption is detected during the guarded block listing of the system memspace. Error messages reveal the corrupted block, along with all recorded information.

Additionally, the example shows that this block is also detected by an explicit call to **memspCheck**, and at deallocation of the block. Finally, it demonstrates that a spurious deallocation of a memory block causes an assertion failure.

 ©1999 Philips Semiconductors 10/08/99

None of the checks performed by the debugging Memspace Manager is complete. For a variety of reasons, error situations can be overlooked. For example, out-of-bounds access might write past the guard area, or it might write a pattern that is identical to the guard pattern. Similarly, block validation might let errors go unnoticed, for example if arguments to **memspFree** resemble valid blocks. This notwithstanding, the probability of overlooked errors is quite small, and in any case valid situations are never reported as errors.

Redirecting Calls to malloc

Even if applications do not make explicit use of the Memspace Manager, their memory allocation behavior can still be debugged or analyzed using the library. For this, all calls to **malloc** and **free** must be redirected. In the ideal situation, all sources are available and can be recompiled with macro redefinitions of **malloc** and **free**, as illustrated in the example code on page 149. Such redefinitions are then typically placed in a central include file. Macro redefinition is attractive because it allows memory allocation tracking by passing the standard macros **__FILE__** and **__LINE__** to the calls to **memspDebug-Malloc**.

In some cases, some of the source may not be available, and hence memory allocation tracking is not possible. However, it is still possible to redirect the calls from **malloc** and **free** to their memspace counterparts in order to take advantage of consistency checking services. This can easily be performed by using the linker **tmld** to rename the symbols **_malloc** and **_free** in the object files, to wrapper functions that call the Memspace Manager instead. This renaming is illustrated in the code sample below, where an equivalent

```
[18] tmcc -c main.c
[19] tmnm main.o
         U  _cache_malloc
         U  _free
00000000 T  _main
         U  _malloc
         U  _memspCheck
         U  _memspPrintGuarded
         U  _printf
[20] tmld main.o -o main.o \
         -symbolrename _malloc=_my_malloc,_free=_my_free,_cache_malloc=
         _my_cache_malloc
[21] tmnm main.o
00000000 T  _main
         U  _memspCheck
         U  _memspPrintGuarded
         U  _my_cache_malloc
         U  _my_free
         U  _my_malloc
         U  _printf
[22] tmcc main.o my_malloc.c -lmemspace_g
[23] tmsim a.out
...
```

program to that on page 149 is achieved by the renaming of the symbols **_malloc**, **_free** and **_cache_malloc** to wrapper functions that are provided in a separate C file. Note that these wrapper functions (shown below) should have the same prototype as their originals, because the compiler-generated calling sequences are left untouched. Apart from the fact that it is no longer known where the wrapper functions are called, this program should give results identical to the previous one.

```
#include “tmlib/Memspace.h”

Pointer my_malloc(Int size){
   return memspDebugMalloc( Null,size, 0, 32,20,”<my_malloc>”,0 );
}

Pointer my_cache_malloc( Int size ){
   return memspDebugMalloc( Null,size, memspCACHE_ALIGNED,
                            32,20,”<my_cache_malloc>”,0 );
}
void my_free(Pointer block){
   memspFree(block);
}
```

Summary of Memspace API (Debugging)

The following functions of the memspace API provide debugging support:

memspDebugMalloc Debugging version of **memspMalloc**. Allocates a block with guard areas and file/line number values attached. This function only works as described in the debugging version of the memspace library (lib_memspace_g.a). Otherwise, guard and file position information is ignored, and the function is similar to a “regular” **memspMalloc**.

memspGetInfo Extracts information from the specified memspace. Information includes name, total amount of memory owned by the memspace, total amount of owned memory that is available for allocation, and largest free block. See example usage in the code example on page 142.

memspPrintGuarded Prints a list onto the standard output stream (**stdout**) of all memory blocks that have been allocated (and not yet freed) from the specified memspace using **memspDebugMalloc** (see description on page 147). The list includes file name and line number as passed to this function.

memspCheck Performs a consistency check on the internal state of the Memspace Manager. A call to this function might abort the program due to an assertion failure. Errors are reported onto the standard error stream (**stderr**).

 ©1999 Philips Semiconductors 10/08/99

TriMedia Memory Manager API Data Structures

This section presents the data structures used in the Memory Manager API.

©1999 Philips Semiconductors 10/08/99

memspSpaceInfo

```
typedef struct memspSpaceInfo {
   String   name;
   Int      total_size;
   nt       segment_size;
   Int      Increment_size;
   struct {
      Int   Total_free_space;
      Int   max_free_blocksize;
      Int   nrof_free_blocks;
   } variable_block_info;
   struct {
      Int   amount_segments;
      Int   nrof_free_blocks;
   } small_block_info [memspFastSizeBound];
} memspSpaceInfo;
```

Fields

`name`	Memspace name, given at creation.
`total_size`	Total size (bytes) **malloc**'d for this space.
`segment_size`	Size of small block segments.
`increment_size`	Memory space extension chunk size.
`total_free_space`	Total free space in **var** block heap.
`max_free_blocksize`	Largest free block in **var** block heap.
`nrof_free_blocks`	Number of free blocks in **var** block heap.
`amount_segments`	Number of small block segments allocated for this size.
`nrof_free_blocks`	Number of small block available in this size.

Description

Memspace information structure, to be filled by function **memspGetInfo**. This structure exposes somewhat the internal details of memory spaces:

Each memory space consists of one variable block heap, plus a number of heaps from which fixed-size block allocation is possible. Such fixed-size allocation is automatically performed for blocks smaller than **memspFastSizeBound** bytes. Such fixed-size blocks are allocated in 4K segments with as little as zero memory overhead per block. Kept in separate lists, they can be allocated and freed very quickly. The idea is that the relative overhead, both in allocation time and in memory use, is largest for the smallest blocks.

Note

The "small block segments" are allocated from the variable block heap.

© 1999 Philips Semiconductors 10/08/99

Memory spaces implement a separate layer of memory management on top of large chunks allocated from the underlying system memory manager (**malloc**). The size of these memory chunks is one of the parameters to the memory space creation function **increment_size**. At creation, and each time the memory space runs out of memory, a chunk of this size is requested from **malloc**. When the memory space is deleted, all such chunks it has allocated since its creation are given back to the system memory manager.

memspSystemSpace

```
extern memspSpace memspSystemSpace;
```

Description

Global memory space. This system space is special, in that some of the administration of all user-created memory spaces is allocated from the system space. It cannot be deleted, and might be abbreviated by Null in all functions of this API.

For example, the two **malloc** calls below are identical.

```
memsp_Malloc( Null,             100, 0 )
memsp_Malloc( memspSystemSpace, 100, 0 )
```

memspBlockProperty

```
typedef enum {
   memspCACHE_ALIGNED  = 0x1
} memspBlockProperty;
```

Fields

memspCACHE_ALIGNED	Ensures that the result is cache-aligned, and that none of the TM data cache pages overlapping the result contain data that is otherwise in use by the application.

Description

Memory allocation properties, specifying properties requested for the blocks returned by **memsp(Debug)Malloc**. See flags parameter.

 © 1999 Philips Semiconductors 10/08/99

TriMedia Memory Manager API Functions

This section presents the functions used in the Memory Manager API.

memspCreate

```
Pointer memspCreate(
   String  name,
   UInt    increment_size
);
```

Parameters

name	Name for memory space.
increment_size	Size of chunks by which the memspace is to be extended.

Return

Returned new space handle. One chunk has already been allocated for the memspace.

Description

Creates new memory space.

memspDelete

```
void memspDelete(
   memspSpace   space
);
```

Parameters

space	Memory space to be deleted.

Description

Deletes a previously allocated memory space, and gives all its extension chunks back to the underlying memory manager.

 ©1999 Philips Semiconductors 10/08/99

memspMalloc

```
Pointer memspMalloc(
   memspSpace  space,
   Int         size,
   UInt32      flags
);
```

Parameters

`space`	Space from which to allocate.
`size`	Size in bytes of requested memory block.
`flags`	Required **memspBlockProperty** flags for the returned block.

Return Codes

Address of returned block. If no memory could be allocated, **Null** is returned.

Description

Attempts allocation of memory.

memspDebugMalloc

```
Pointer memspDebugMalloc(
   memspSpace  space,
   Int         size,
   UInt32      flags,
   UInt16      bsize,
   UInt16      asize,
   String      file,
   Int         line
);
```

Parameters

`space`	Space from which to allocate.
`size`	Size in bytes of requested memory block.
`flags`	Required **memspBlockProperty** flags for the returned block.
`bsize`, `asize`	Sizes of guard areas before and after the user part of the memory block.
`file`, `line`	For passing file block creator location information.

Return Codes

Address of requested block. If no memory could be allocated, **Null** is returned.

Description

Attempts allocation of guarded memory. Guarded memory blocks have guard regions immediately before and after their user space. Guard regions are filled with magic contents that can be checked for corruption at critical moments. Also, guarded memory blocks have file pos/line info attached, by which their creators can be located.

In the non-debugging form of this library, **bsize**, **asize**, **file**, and **line** are ignored.

A suggested use of this function is to redirect all calls to **malloc** by compiling sources with the following macro defined:

```
extern memspSpace malloc_space;
#define malloc(s) memspDebugMalloc(malloc_space,s,0,0,30,__FILE__,__LINE__)
```

 ©1999 Philips Semiconductors 10/08/99

memspFree

```
void memspFree(
   Pointer    address
);
```

Parameters

`addr`	Block to be freed.

Return Codes

In the debugging form of this library, various internal consistency checks are performed and the old contents of the block are corrupted.in incorrect applications. These checks may fail and result in messages on **stdout** plus calls to **exit**.

Description

Frees previously allocated memory.

memspRealloc

```
Pointer memspRealloc(
   Pointer    address,
   Int        size
);
```

Parameters

addr	Block to resize.

Description

Changes the size of a memory block, return pointer to the new, possibly resized block. This function correctly handles guarded, cache-aligned, and "normal" memory blocks. If a guarded block is passed, the result will also be guarded with the same guard parameters (i.e. **bsize**, **asize**, **file** and **line**). The same is true for cache-aligned input blocks.

memspFastFree

```
void memspFastFree(
   Pointer   mem,
   Int       size
);
```

Parameters

mem	Memory block to be freed (may be **Null**).
size	Size by which **mem** was obtained from **memspMalloc**, or **memspSpace_ANY_SIZE**, when not known. Note that the memory space manager will blindly trust your value, so it can be faster. (Because of this, you had better be correct, or conservative, and use **memspSpace_ANY_SIZE**).

Description

Fast memory block deallocation primitive. Can be used when size by which block was allocated is known. This function correctly handles guarded, cache-aligned, and 'normal' memory blocks.

 © 1999 Philips Semiconductors 10/08/99

memspGetInfo

```
void memspGetInfo(
   memspSpace        space,
   memspSpaceInfo   *info
);
```

Parameters

`space`	Memory space from which to get information.
`info`	Information block to fill.

Description

Gets current information of specified memory space.

memspPrintGuarded

```
void memspPrintGuarded(
   memspSpace    space
);
```

Parameters

`space`	Memory space to print.

Description

Prints list of all guarded blocks in specified memspace to **stdout**. Also, checks guard areas for each of the blocks, and prints diagnostics on **stderr**.

memspCheck

```
void memspCheck();
```

Description

Does consistency check on internal state on this memory manager, and print diagnostics on **stderr**.

memspTraverseSpaces

```
void memspTraverseSpaces(
   memspSpaceFun    fun,
   Pointer          data
);
```

Parameters

`fun`	Function to apply to all spaces.
`data`	User-specified data item to be additionally passed to each call to **fun**.

Description

Applies specified function to all memory spaces.

 ©1999 Philips Semiconductors 10/08/99

Chapter 7

Programmable Interrupt Controller (PIC) API

PIC API Overview

The PIC device library provides a standard way for various software modules to install and use interrupts, regardless of the details of the hardware. The PIC (Programmable Interrupt Controller) does this by providing a board support API to the hardware and a standard API to the application above it.

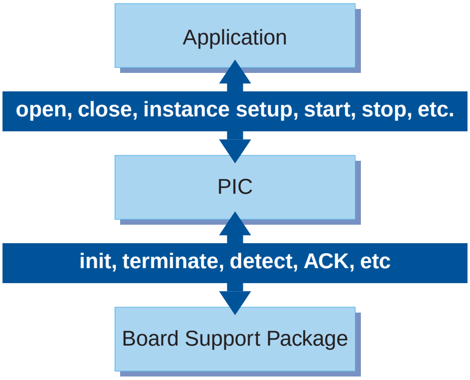


Figure 9 PIC Software Architecture

Interrupts managed by the PIC can come from any sort of external hardware, or even from software. The external hardware could be a dedicated interrupt pin (TriUserIrq), or a general purpose I/O pin (GPIO, as implemented on the TM-2700). Or it could use a sophisticated programmable interrupt controller chip, external to TriMedia. In each case, the PIC provides a way to write applications without specific dependencies on the hardware. These hardware dependencies are isolated into the board support package.

The PIC library currently supports up to 64 interrrupt sources, with names enumerated in the **tsaPICsource_t**. These names are used by applications to identify sources that might be handled in the board support package.

The BSP portion of the PIC expects that handlers are provided for the standard interrupt (e.g., detecting the interrupt source, acknowledging an interrupt source, enabling/disabling interrupt sources). The interface to the application is similar to that used in the TriMedia interrupt device library. An application opens an instance, and sets up that instance by installing a handler. The details of these operations are described in the following pages. You are also invited to examine the examples of PIC usage that are provided in the DTV reference boards. The PIC supports the UART and digital audio input.

To install an interrupt handler for a supported interrupt source, the application must first open an instance of the PIC library for this source:

```
err = tsaPICOpen( &instance, picSourceComm1 );
CHECK(err); /* This macro or its equivalent implemented by the
               application to handle returned error codes. */
```

 ©1999 Philips Semiconductors 10/08/99

The returned instance must be used in subsequent API calls. Now the application can install the handler by calling **tsaPICInstanceSetup**:

```
setup.handler = myHandler;
setup.handle  = (Pointer)localScope;
setup.enabled = True;
err = tsaPICInstanceSetup(instance, &setup);
CHECK(err);
```

If the **enabled** field in the setup structure is set to True, the interrupt for the installed source is immediately enabled. If the application sets this field to False, it must call **tsa-PICStart** to enable the interrupt source.

To disable an interrupt source, the application must call **tsaPICStop**.

Board Support Interface

For the PIC to support an interrupt source, code must be installed in the BSP to handle that source. The BSP includes a table that maps the **tsaPICsource_t** types to an index into that table, which is used in the BSP. This index is passed to the BSP functions as the interrupt "source." For example, a DTV reference board supports 5 or the 64 possible interrupt sources. The application will use the **tsaPICsource_t** (0–63) to identify the source. The PIC library converts this to a number between 0 and 4 (in this case) to identify the source.

An init function is called when **tsaPICInstanceSetup** is called. The init function can set up the hardware for the given interrupt. It probably does not actually enable the interrupt. That happens in the BSP's start function. Similarly, a stop function disables the interrupt. When an interrupt is detected by the hardware, a detect function is called to allow multiple handlers to be chained together. The detect function returns True when it sees that its interrupt has been triggered. This points out the need for each interrupt source to be able to respond to a query. The fact that the interrupt is shared also hints at the requirement that PIC interrupts be level triggered, and acknowledgable. The BSP also provides a place for an acknowledge function unique to each interrupt source.

Debugging PIC ISRs

The TriMedia debugger can behave in unexpected ways when stopped in an interrupt service routine. The DP (debug print) buffer is an invaluable tool when debugging interrupts because it does not use the debugger, nor does it make complex function calls. Refer to Chapter 18, *Debugging TriMedia Applications Using JTAG*, of Book 4, *Software Tools*, Part C, for more information.

The source code for the PIC is available. You might find it useful to compile this with some more DPs enabled.

PIC API Data Structures

This section describes the tsaPIC data structures.

 © 1999 Philips Semiconductors 10/08/99

tsaPICSource_t

```
typedef enum {
   picSourceNone  = 0,
   picSourceComm1 = 1,
   picSourceComm2,
   picSourceComm3,
   picSourceComm4,
   picSourceComm5,
   picSourceComm6,
   picSourceComm7,
   picSourceComm8,
   picSourceModem0,
   picSourceModem1,
   picSourceLpt1,
   picSourceLpt2,
   picSourceKeyboard,
   picSourceIrIn0,
   picSourceIrIn1,
   picSourceIrIn2,
   picSourceIrIn3,
   picSourceIrOut0,
   picSourceIrOut1,
   picSourceIrOut2,
   picSourceIrOut3,
   picSourceIrIO0,
   picSourceIrIO1,
   picSourceIrIO2,
   picSourceIrIO3,
   picSourceAudio1,
   picSourceAudio2,
   picSourceAudio3,
   picSourceAudio4,
   picSource1937,
   picSourceUSB,
   picSourceUsr0,
   picSourceUsr1,
   picSourceUsr2,
   picSourceUsr3,
   picSourceUsr4,
   picSourceUsr5,
   picSourceUsr6,
   picSourceUsr7,
   picSourceUsr8
}tsaPICSource_t;
```

Fields

`picSourceNone`	No valid source.
`picSourceComm`*X*	Source for serial ports. Here, *X* stands for the number of the serial port (ports 1–8).
`picSourceModem`*X*	Modem interrupt. Here, *X* is 0 or 1.
`picSourceLpt`*X*	Interrupt from parallel port. Here, *X* is 1 or 2.
`picSourceKeyboard`	Interrupt source is a keyboard.
`picSourceIrIn`*X*	Interrupt source is an infrared input device. Here, *X* is 0–3.
`picSourceIrOut`*X*	Interrupt source is an infrared outpout device. Here, *X* is 0–3.
`picSourceIrIO`*X*	Interrupt source is an infrared input/output device. Here, *X* is 0–3.
`picSourceAudio`*X*	Interrupt source is an audio device. Here, *X* is 0–3.
`picSource1937`	Interrupt source is an IEEE 1937 device.
`picSourceUSB`	Interrupt source is a USB port.
`picSourceUsr`*X*	These interrupt sources are for sources that are not specified in **tsaPICSource_t**. Here, *X* is 0–8.

Description

Specifies the interrupt source in **tsaPICOpen**. This type is also used in **tsaPICCapabilities_t** to specify the supported sources.

 ©1999 Philips Semiconductors 10/08/99

tsaPICCapabilities_t

```
typedef struct tsaPICCapabilities {
   tmVersion_t      version;
   UInt32           numSupportedInstances;
   UInt32           numCurrentInstances;
   Char             picName[DEVICE_NAME_LENGTH];
   tsaPICSource_t   supportedSources[PIC_MAX_NUM_OF_SOURCES];
} tsaPICCapabilities_t, *ptsaPICCapabilities_t;
```

Fields

`version`	Version of the PIC library.
`numSupportedInstances`	Number of supported instances (supported interrupt sources).
`numCurrentInstances`	Number of instances currently in use.
`picName`	Name of the PIC.
`supportedSources`	Array with the supported sources.

Description

A struct of this type is used to report the capabilities of the PIC library (see also **tsaPICGetCapabilities**).

tsaPICInstanceSetup_t

```
typedef struct tsaPICSetup {
   tsaPICHandler_t    handler;
   Pointer            handle;
   Bool               enabled;
} tsaPICInstanceSetup_t, *ptsaPICInstanceSetup_t;
```

Fields

`handler`	Handler that will be called in the PIC interrupt service routine if the related source asserted an interrupt.
`handle`	Passed to the handler as an argument.
`enabled`	If this flag is set to True the interrupt source will be enabled in **tsaPICInstanceSetup**. Otherwise **tsaPICStart** needs to be called to enable the interrupt source.

Description

A structure of this type is used in **tsaPICInstanceSetup** to initialize an interrupt source.

Note: the handler callback function gets called from an interrupt service routine. Therefore it should do only minimal processing (e.g., signaling an event to a task).

(Since the handler is called from an interrupt service routine, it should be written as a simple function, and not as an interrupt handler.)

 ©1999 Philips Semiconductors 10/08/99

PIC API Functions

This section presents the PIC device library functional interface.

tsaPICGetCapabilities

```
extern tmLibdevErr_t tsaPICGetCapabilities(
   ptsaPICCapabilities_t   *caps
);
```

Parameters

`caps`	Pointer (returned) to a static capabilities structure for the PIC library.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_MEMALLOC_FAILED`	Failed to allocate memory.

Description

This function gets the PIC capabilities.

It allocates memory for the capabilities structure and fills it with the capabilities that it gets from the board support package.

 © 1999 Philips Semiconductors 10/08/99

tsaPICOpen

```
extern tmLibdevErr_t tsaPICOpen(
   Int             *instance,
   tsaPICSource_t   src
);
```

Parameters

`instance`	Instance returned by **tsaPICOpen**. This instance must be used for subsequent PIC function calls.
`src`	Interrupt source that will be opened.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NO_MORE_INSTANCES`	No more instances available.
`TMLIBDEV_ERR_MEMALLOC_FAILED`	Failure to allocate memory.
`PIC_ERR_SOURCE_NOT_AVAILABLE`	Selected source not available.
`TMLIBDEV_ERR_NULL_PARAMETER`	Asserts this error if **instance** is a null pointer (but only in the debugging version of the library).

The function can also return error codes produced by the BSP or the interrupt device library.

Description

This function opens an instance of the PIC library. The application must specify which interrupt source it wants to use.

This function allocates all resources needed for this instance. It also opens, initializes and starts the interrupt if this is the first source opened for this interrupt.

Note that it is possible for the interrupt installed by the open function to be triggered immediately. If it is not possible to mask the source before opening (in the board_init function, for example), then the PIC BSP code should be prepared to acknowledge and clear the source immediately.

tsaPICInstanceSetup

```
extern tmLibdevErr_t tsaPICInstanceSetup(
   Int                      instance,
   ptsaPICInstanceSetup_t   setup
);
```

Parameters

instance	Instance previously opened by **tsaPICOpen**.
setup	Pointer to a setup struct that will be used to initialize the interrupt source.

Return Codes

TMLIBDEV_OK	Success.
TMLIBDEV_ERR_NOT_OWNER	Invalid instance passed to the function (can also be asserted in debug version).
PIC_ERR_ALLREADY_INITIALIZED	Function has been called before for this instance.
PIC_ERR_NO_HANDLER	Gets asserted if handler pointer in **setup** is Null (in debug version of the library).

The function can also return error codes produced by the board support package.

Description

Initializes a PIC source. The handler function is specified here. The BSP's init function is called as a result of this function. If the handler is specified to be enabled, then the BSP's start function is also called here. Otherwise, the start function will have to be called separately to enable the ISR. Call **tsaPICInstanceSetup** only once after opening.

 ©1999 Philips Semiconductors 10/08/99

tsaPICStart

```
extern tmLibdevErr_t tsaPICStart(
   Int   instance
);
```

Parameters

instance	Instance previously opened by **tsaPICOpen**.

Return Codes

TMLIBDEV_OK	Success.
TMLIBDEV_ERR_NOT_OWNER	Invalid instance passed to the function (can also be asserted in debug version).
PIC_ERR_NO_INSTANCE_SETUP	Source has not been initialized (gets asserted in debug version of the library).

The function can also return error codes produced by the board support package.

Description

Starts (enables) an interrupt source. This function calls the BSP's start function.

tsaPICStop

```
extern tmLibdevErr_t tsaPICStop(
   Int   instance
);
```

Parameters

`instance`	Instance previously opened by **tsaPICOpen**.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Invalid instance passed to the function (can also be asserted in debug version).
`PIC_ERR_NO_INSTANCE_SETUP`	Source has not been initialized (gets asserted in debug version of the library).

The function can also return error codes produced by the board support package.

Description

Stops (disables) an interrupt source. This function calls the BSP's stop function.

 © 1999 Philips Semiconductors 10/08/99

tsaPICClose

```
extern tmLibdevErr_t tsaPICClose(
   Int   instance
);
```

Parameters

`instance`	Instance previously opened by `tsaPICOpen`.

Return Codes

`TMLIBDEV_OK`	Success.
`TMLIBDEV_ERR_NOT_OWNER`	Invalid instance passed to the function (can also be asserted in debug version).

The function can also return error codes produced by the board support package.

Description

Closes a PCI instance. The function also frees all resources allocated for this instance in **tsaPICOpen**. This function calls the BSP's termination function.

To use the interrupt source again, you must reopen it using **tsaPICOpen**.

© 1999 Philips Semiconductors 10/08/99

Chapter 8

File I/O Drivers API

Introduction

This chapter describes the file i/o driver interface defined in <tmlib/IODrivers.h>. These routines allow a program to install *file I/O drivers*.

A file I/O driver provides access to file manipulation functions, either through the usual system call functions (**open**, **read**, **write**, **close**, and others), or through their standard C library counterparts (**fopen**, **fread**, **fwrite**, **fclose**, and others), which the standard library implements using the underlying system calls.

For additional information on File I/O Drivers, see *File I/O Drivers* in Chapter 2 of Book 3, *Software Architecture*, Part A.

 © 1999 Philips Semiconductors 10/08/99

File I/O Function Types

These definitions provide prototypes for file I/O functions. Most of these represent POSIX.1 system calls.

Name	Page
IOD_RecogFunc	21
IOD_InitFunc	22
IOD_TermFunc	23
IOD_OpenFunc	24
IOD_StatFunc	25
IOD_OpenDllFunc	26
IOD_CloseFunc	27
IOD_ReadFunc	28
IOD_WriteFunc	29
IOD_SeekFunc	30
IOD_IsattyFunc	31
IOD_FstatFunc	32
IOD_FcntlFunc	33
IOD_SyncFunc	34
IOD_FSyncFunc	35
IOD_UnlinkFunc	36
IOD_LinkFunc	37
IOD_MkdirFunc	38
IOD_RmdirFunc	39
IOD_AccessFunc	39
IOD_OpendirFunc	40
IOD_ClosedirFunc	41
IOD_RewinddirFunc	42
IOD_ReaddirFunc	43

IOD_RecogFunc

```
typedef Bool (*IOD_RecogFunc )(
   String  path
);
```

Description

Determines whether a given filename is recognized by this I/O driver.

IOD_InitFunc

```
typedef Bool (*IOD_InitFunc )( void );
```

Description

Initialization function.

IOD_TermFunc

```
typedef void (*IOD_TermFunc )( void );
```

Description

Termination function.

 © 1999 Philips Semiconductors 10/08/99

IOD_OpenFunc

```
typedef Int32 (*IOD_OpenFunc )(
   String   path,
   Int32    oflag,
   Int32    mode
);
```

Description

Opens a file, like POSIX.1 **open**.

IOD_StatFunc

```
typedef Int32 ( *IOD_StatFunc )(
   String       path,
   struct stat  *buf
);
```

Description

Stats a file (by name), like POSIX.1 **stat**.

IOD_OpenDllFunc

```
typedef Int32 ( *IOD_OpenDllFunc )(
   String  path
);
```

Description

Opens a DLL.

IOD_CloseFunc

```
typedef Int32 ( *IOD_CloseFunc )(
   Int32  file
);
```

Description

Closes a file, like POSIX.1 **close**.

IOD_ReadFunc

```
typedef Int32 (*IOD_ReadFunc  )(
   Int32    file,
   Pointer  buf,
   Int32    nbyte
);
```

Description

Reads from a file, like POSIX.1 read.

IOD_WriteFunc

```
typedef Int32 (*IOD_WriteFunc )(
   Int32    file,
   Pointer    buf,
   Int32    nbyte)
;
```

Description

Writes to a file, like POSIX.1 **write**.

 © 1999 Philips Semiconductors 10/08/99

IOD_SeekFunc

```
typedef Int32 ( *IOD_SeekFunc )(
   Int32  file,
   Int32  offset,
   Int32  whence
);
```

Description

Seeks on a file, like POSIX.1 **seek**.

IOD_IsattyFunc

```
typedef Int32 ( *IOD_IsattyFunc )(
   Int32  file
);
```

Description

Determines if a file is interactive, like POSIX.1 **isatty**.

IOD_FstatFunc

```
typedef Int32 ( *IOD_FstatFunc )(
   Int32         file,
   struct stat  *buf
);
```

Description

Stats a file (by file descriptor), like POSIX.1 **fstat**.

IOD_FcntlFunc

```
typedef Int32 ( *IOD_FcntlFunc )(
   Int32   file,
   Int32   cmd,
   Int32   flags
);
```

Description

Files control, like POSIX.1 **fcntl**.

IOD_SyncFunc

```
typedef Int32 ( *IOD_SyncFunc  )( void );
```

Description

Syncs a filesystem, like POSIX.1 **sync**.

IOD_FSyncFunc

```
typedef Int32 ( *IOD_FSyncFunc )(
   Int32  file
);
```

Description

Syncs a file, like POSIX.1 **fsync**.

 © 1999 Philips Semiconductors 10/08/99

IOD_UnlinkFunc

```
typedef Int32 ( *IOD_UnlinkFunc )(
   String  path
);
```

Description

Removes a file, like POSIX.1 **unlink**.

IOD_LinkFunc

```
typedef Int32 ( *IOD_LinkFunc )(
   String  src,
   String  dest
);
```

Description

Links a file, like POSIX.1 **link**.

IOD_MkdirFunc

```
typedef Int32 ( *IOD_MkdirFunc )(
   String  path,
   Int32   mode
);
```

Description

Creatse a directory, like POSIX.1 **mkdir**.

IOD_RmdirFunc

```
typedef Int32 ( *IOD_RmdirFunc )(
   String  path
);
```

Description

Removes a directory, like POSIX.1 **rmdir**.

IOD_AccessFunc

```
typedef Int32 ( *IOD_AccessFunc )(
   String  path,
   Int32   mode
);
```

Description

Checks file access, like POSIX.1 **access**.

IOD_OpendirFunc

```
typedef DIR* ( *IOD_OpendirFunc )(
   ConstString  path
);
```

Description

Opens a directory, like POSIX.1 **opendir**.

 © 1999 Philips Semiconductors 10/08/99

IOD_ClosedirFunc

```
typedef Int32 ( *IOD_ClosedirFunc )(
   DIR  *dir
);
```

Description

Closes a directory, like POSIX.1 **closedir**.

IOD_RewinddirFunc

```
typedef void ( *IOD_RewinddirFunc )(
   DIR  *dir
);
```

Description

Rewinds a directory, like POSIX.1 **rewinddir**.

IOD_ReaddirFunc

```
typedef struct dirent *( *IOD_ReaddirFunc )(
   DIR  *dir
);
```

Description

Reads a directory, like POSIX.1 **opendir**.

File I/O Driver Control Functions

These definitions provide prototypes for driver control functions.

 © 1999 Philips Semiconductors 10/08/99

IOD_install_fsdriver

```
UID_Driver IOD_install_fsdriver(
   IOD_RecogFunc     recog,
   IOD_InitFunc      init,
   IOD_TermFunc      term,
   IOD_OpenFunc      open,
   IOD_OpenDllFunc   open_dll,
   IOD_CloseFunc     close,
   IOD_ReadFunc      read,
   IOD_WriteFunc     write,
   IOD_SeekFunc      seek,
   IOD_IsattyFunc    isatty,
   IOD_FstatFunc     fstat,
   IOD_FcntlFunc     fcntl,
   IOD_StatFunc      stat,
   IOD_SyncFunc      sync,
   IOD_FSyncFunc     fsync,
   IOD_UnlinkFunc    unlink,
   IOD_LinkFunc      link,
   IOD_MkdirFunc     mkdir,
   IOD_RmdirFunc     rmdir,
   IOD_AccessFunc    access,
   IOD_OpendirFunc   opendir,
   IOD_ClosedirFunc  closedir,
   IOD_RewinddirFunc rewinddir,
   IOD_ReaddirFunc   readdir
);
```

Parameters

See *File I/O Function Types* beginning on page 19 for descriptions.

Return Value

Returns a new file driver id if successful, or NULL otherwise.

Description

Creates a new file I/O driver. The installed driver's init function is executed; if the init function fails, the **IOD_install_driver** call fails.

IOD_install_driver

```
UID_Driver IOD_install_driver(
   IOD_RecogFunc    recog,
   IOD_InitFunc     init,
   IOD_TermFunc     term,
   IOD_OpenFunc     open,
   IOD_OpenDllFunc  open_dll,
   IOD_CloseFunc    close,
   IOD_ReadFunc     read,
   IOD_WriteFunc    write,
   IOD_SeekFunc     seek,
   IOD_FstatFunc    fstat,
   IOD_FcntlFunc    fcntl,
   IOD_StatFunc     stat
);
```

Parameters

See *File I/O Function Types* beginning on page 19 for descriptions.

Return Value

Returns a new file driver id if successful, or NULL otherwise.

Description

Creates a new simple file I/O driver with the specified functions. This interface does not supply file system/directory manipulation functions. The installed driver's init function is executed; if the init function fails, the **IOD_install_driver** call fails.

 © 1999 Philips Semiconductors 10/08/99

IOD_uninstall_driver

```
void IOD_uninstall_driver(
   UID_Driver  driver
);
```

Parameters

`driver`	Driver to uninstall.

Description

Uninstalls the given driver. Call its term routine. The given driver subsequently is invalid for use as a driver id.

IOD_lookup_driver

```
UID_Driver IOD_lookup_driver(
   String  name
);
```

Parameters

`name`	File name to recognize

Return Value

The id of the first driver in the installed driver chain which recognizes name, or NULL if not recognized.

Description

Call the recognition functions of each installed driver, latest-installed first, to find the first driver which recognizes name.

IOD_lookup_dll

```
UID_Driver IOD_lookup_dll(
   String   name,
   Int32    *fd
);
```

Parameters

`name`	DLL name to recognize.
`fd`	Returned open file descriptor.

Result

The ID of the first driver in the installed driver chain which recognizes name, or NULL if not recognized.

Description

Calls the recognition functions of each installed driver, latest installed driver first, to find the first driver which recognizes name.

IOD_sync

```
void IOD_sync( void );
```

Description

Calls the sync function of each installed driver.

 ©1999 Philips Semiconductors 10/08/99

File I/O Data Structures

This section presents the File I/O data structure.

UID_Driver_t

```
typedef struct UID_Driver_t {
   UID_Driver         next;
   IOD_RecogFunc      recog;
   IOD_InitFunc       init;
   IOD_TermFunc       term;
   IOD_OpenFunc       open;
   IOD_CloseFunc      close;
   IOD_ReadFunc       read;
   IOD_WriteFunc      write;
   IOD_SeekFunc       seek;
   IOD_IsattyFunc     isatty;
   IOD_FstatFunc      fstat;
   IOD_FcntlFunc      fcntl;
   IOD_OpenDllFunc    open_dll;
   IOD_StatFunc       stat;
   IOD_SyncFunc       sync;
   IOD_FSyncFunc      fsync;
   IOD_UnlinkFunc     unlink;
   IOD_LinkFunc       link;
   IOD_MkdirFunc      mkdir;
   IOD_RmdirFunc      rmdir;
   IOD_AccessFunc     access;
   IOD_OpendirFunc    opendir;
   IOD_ClosedirFunc   closedir;
   IOD_RewinddirFunc  rewinddir;
   IOD_ReaddirFunc    readdir;
} *UID_Driver;
```

Fields

`next`	Link to next installed driver.
(others)	See File I/O Function Types beginning on page 19 for descriptions.

 ©1999 Philips Semiconductors 10/08/99

Chapter 9

The Operating System Wrapper (tmos.h)

Introduction

Programs that adhere to the TriMedia Software Architecture should not access pSOS directly. Instead, they use this wrapper API to ensure portability. The wrapper API is designed to clearly delineate the operating system functionality that is expected by the TriMedia software system. If ever it is necessary to change the OS, only a subset of pSOS might need to be emulated. That subset is clearly defined by this wrapper. Similarly, the error reporting behavior of the OS is clearly defined by this wrapper.

The source code for the pSOS wrapper is included with the SDE. Feel free to browse it. For more information about the specific functions, refer to the individual pSOS functions that are documented in *pSOS System Calls*. This is the reference OS implementation.

This table lists the functions available to applications that use the pSOS wrapper.

Functions	Page
tmosMain	39
tmosExit	39
tmosInit	40
tmosTaskChangePriority	41
tmosTaskCreate	42
tmosTaskDestroy	43
tmosTaskIdent	44
tmosTaskResume	46
tmosTaskStart	45
tmosTaskSuspend	46
tmosQueueCreate	47
tmosQueueDestroy	48
tmosQueueReceive	49
tmosQueueSend	50
tmosQueueSendUrgent	51
tmosSemaphoreCreate	52
tmosSemaphoreDestroy	53
tmosSemaphoreP	54
tmosSemaphoreV	55
tmosTimSleep	56

 © 1999 Philips Semiconductors 10/08/99

tmosMain

```
extern void tmosMain( void );
```

Parameters

None.

Description

This is a macro that maps to the pSOS root function. User code begins execution at this function. The standard C command line arguments are available as globals inside of this function. They must be declared in the user's program if used:

```
extern int    __argc;
extern char **__argv;
```

tmosExit

```
extern void tmosExit(
   Int32  val
);
```

Parameters

`val`	Exit code.

Description

This function causes the entire program to terminate, as opposed to the Exit function, which causes the current task to terminate.

tmosInit

```
extern void tmosInit( void )
```

Parameters

None.

Description

This function ensures that include files define the right things. The function calls dinette to initialize OS device drivers.

 © 1999 Philips Semiconductors 10/08/99

Tasks

tmosTaskChangePriority

```
extern UInt32 tmosTaskChangePriority(
   UInt32   tid,
   UInt32   newpriority,
   UInt32  *oldpriority
);
```

Parameters

`tid`	Task ID.
`newpriority`	New priority.
`oldpriority`	Pointer to old priority.

Description

Sets the priority of the specified task, using a call to the pSOS function:

```
t_setpri( tid, newprio, oldprio );
```

Errors are mapped through the “MapError” macro to return **TMLIBAPP_OK** in case of success. Possible errors are or’ed with **TMLIBAPP_ERR_OS_ERR**.

tmosTaskCreate

```
extern UInt32 tmosTaskCreate(
   char     name[4],
   UInt32   flags,
   UInt32   prio,
   UInt32   sstack,
   UInt32   ustack,
   UInt32  *tid
);
```

Parameters

`name`	Task name.
`flags`	Flags are listed below.
`prio`	Task priority.
`sstack`	Size of the system stack.
`ustack`	Size of the user stack.
`tid`	Pointer to the task ID (returned).

Description

Creates a task using a call to the pSOS function:

```
t_create( name, prio, sstack, ustack, flags, tid );
```

Errors are mapped through the “MapError” macro to return **TMLIBAPP_OK** in case of success. The system stack and the user stack combine into a single stack. Possible errors are or’ed with **TMLIBAPP_ERR_OS_ERR**. Legal values for the flags variable are listed here:

```
typedef enum tmosTaskCreateFlags{   /* for tmosTaskCreate */
   tmosTaskFlagsCreateStd = 0,
   tmosTaskFlagsLocal     = 0,
   tmosTaskFlagsGlobal    = 1,
   tmosTaskFlagsNoFPU     = 0,
   tmosTaskFlagsFPU       = 2
} tmosTaskCreateFlags, *ptmosTaskCreateFlags;
```

 © 1999 Philips Semiconductors 10/08/99

tmosTaskDestroy

```
extern UInt32 tmosTaskDestroy(
   UInt32  tid
);
```

Parameters

`tid`	Task ID.

Description

Deletes a task using a call to the pSOS function:

```
t_delete( tid );
```

Errors are mapped through the “MapError” macro to return **TMLIBAPP_OK** in case of success. Possible errors are or’ed with **TMLIBAPP_ERR_OS_ERR**.

tmosTaskIdent

```
extern UInt32 tmosTaskIdent(
   char     name[4],
   UInt32   node,
   UInt32   *tid
);
```

Parameters

`name`	Task name.
`node`	Task node.
`tid`	Pointer to the task ID.

Description

Given the name and node, the function looks up the TID (task identifier). The function is implemented using the pSOS function:

```
t_ident( name, node, tid );
```

Errors are mapped through the “MapError” macro to return **TMLIBAPP_OK** in case of success. Possible errors are or’ed with **TMLIBAPP_ERR_OS_ERR**.

 ©1999 Philips Semiconductors 10/08/99

tmosTaskStart

```
extern UInt32 tmosTaskStart(
   UInt32  tid,
   UInt32  flags,
   void    (*start_addr)(),
   UInt32  targs[]
);
```

Parameters

`tid`	Task ID.
`flags`	Flags are listed below.
`start_addr`	Pointer to a function that comprises the task.
`targs`	Array of (up to 4) arguments passed to the task.

Description

Start the specified task, using a call to the pSOS function:

```
t_start( tid, flags, start_addr, targs );
```

The flags determine some characteristics of the task. Errors are mapped through the "MapError" macro to return **TMLIBAPP_OK** in case of success. Possible errors are or'ed with **TMLIBAPP_ERR_OS_ERR**. Legal values for the flags are:

```
typedef enum tmosTaskStartFlags{   /* for tmosTaskStart */
   tmosTaskFlagsStandard  = 5,
   tmosTaskFlagsPreempt   = 0,
   tmosTaskFlagsNoPreempt = 1,
   tmosTaskFlagsNoSliced  = 0,
   tmosTaskFlagsSliced    = 2,
   tmosTaskFlagsNoAsyncSignalHandling = 4  /*should always be set*/ }
tmosTaskStartFlags, *ptmosTaskStartFlags;
```

tmosTaskSuspend

```
extern UInt32 tmosTaskSuspend(
   UInt32  tid
);
```

Parameters

`tid`	Task ID.

Description

Suspends the specified task. The function is implemented using the pSOS function:

```
t_suspend( tid );
```

Errors are mapped through the "MapError" macro to return **TMLIBAPP_OK** in case of success. Possible errors are or'ed with **TMLIBAPP_ERR_OS_ERR**.

tmosTaskResume

```
extern UInt32 tmosTaskResume(
   UInt32  tid
);
```

Parameters

`tid`	Task ID.

Description

Resume the specified task, if it was suspended. The function is implemented using the pSOS function:

```
t_resume( tid );
```

Errors are mapped through the "MapError" macro to return **TMLIBAPP_OK** in case of success. Possible errors are or'ed with **TMLIBAPP_ERR_OS_ERR**.

 © 1999 Philips Semiconductors 10/08/99

Queues

tmosQueueCreate

```
extern UInt32 tmosQueueCreate(
   char     name[4],
   UInt32   flags,
   UInt32   count,
   UInt32  *qid
);
```

Parameters

`name`	Queue name.
`flags`	Flags are listed below.
`count`	Queue size.
`qid`	Pointer to the queue ID (returned).

Description

Creates a queue using the pSOS function:

```
q_create( name, count, flags, qid );
```

Note that the order of the parameters in **q_create** is different. Errors are mapped through the "MapError" macro to return **TMLIBAPP_OK** in case of success. Possible errors are or'ed with **TMLIBAPP_ERR_OS_ERR**. Legal values for the flags are:

```
typedef enum tmosQueueCreateFlags{ /* for tmosQueueCreate */
   tmosQueueFlagsStandard = 0,
   tmosQueueFlagsLocal    = 0,
   tmosQueueFlagsGlobal   = 1,
   tmosQueueFlagsNoLimit  = 0,
   tmosQueueFlagsLimit    = 4
} tmosQueueCreateFlags, *ptmosQueueCreateFlags;
```

tmosQueueDestroy

```
extern UInt32 tmosQueueDestroy(
   UInt32 qid
);
```

Parameters

qid	Queue ID.

Description

Deletes a queue and frees all resources associated with it. The function is implemented with a call to the pSOS function:

```
q_delete( qid );
```

Errors are mapped through the "MapError" macro to return **TMLIBAPP_OK** in case of success. Possible errors are or'ed with **TMLIBAPP_ERR_OS_ERR**.

 ©1999 Philips Semiconductors 10/08/99

tmosQueueReceive

```
extern UInt32 tmosQueueReceive(
   UInt32  qid,
   UInt32  flags,
   UInt32  timeout,
   UInt32  msg_buf[4]
);
```

Parameters

`qid`	Queue ID.
`flags`	Flags are listed below.
`timeout`	Timeout period, in ticks.
`msg_buf`	Array of words in which to receive the message.

Description

Attempts to retrieve a packet from the specified queue. The function is implemented with a call to the pSOS function:

```
q_receive( qid, flags, timeout, msg_buf );
```

The timeout period is in ticks. The length of a tick is defined in the file sys_conf.h. Ticks are 10 ms by default. Errors are mapped through the "MapError" macro to return **TMLIBAPP_OK** in case of success. **ERR_NOMSG** is mapped to **TMLIBAPP_QUEUE_EMPTY**. **ERR_TIMEOUT** is mapped to **TMLIBAPP_TIMEOUT**. Possible errors are or'ed with **TMLIBAPP_ERR_OS_ERR**. Legal values for the flags are:

```
typedef enum tmosQueueReceiveFlags{   /* for tmosQueueReceive */
   tmosQueueFlagsWait   = 0,
   tmosQueueFlagsNoWait = 1
} tmosQueueReceiveFlags, *ptmosQueueReceiveFlags;
```

tmosQueueSend

```
extern UInt32 tmosQueueSend(
   UInt32   qid,
   UInt32   flags,
   UInt32   msg_buf[4]
);
```

Parameters

qid	Queue ID.
flags	Flags are listed below.
msg_buf	Array of words in which to send the message.

Description

Attempts to send a message through a queue using the pSOS function:

```
q_send( qid, msg_buf );
```

Errors are mapped through the “MapError” macro to return **TMLIBAPP_OK** in case of success. Possible errors are or’ed with **TMLIBAPP_ERR_OS_ERR**.

 ©1999 Philips Semiconductors 10/08/99

tmosQueueSendUrgent

```
extern UInt32 tmosQueueSendUrgent(
   UInt32   qid,
   UInt32   flags,
   UInt32   msg_buf[4]
);
```

Parameters

`qid`	Queue ID.
`flags`	Flags are listed below.
`msg_buf`	Array of words in which to send the message.

Description

Attempts to send a message through a queue using the pSOS function:

```
q_urgent( qid, .msg_buf );
```

The function sends the message to the head of the queue. Errors are mapped through the "MapError" macro to return **TMLIBAPP_OK** in case of success. Possible errors are OR'd with **TMLIBAPP_ERR_OS_ERR**.

Semaphores

tmosSemaphoreCreate

```
extern UInt32 tmosSemaphoreCreate(
   char     name[4],
   UInt32   flags,
   UInt32   count,
   UInt32  *smid
); .
```

Parameters

`name`	Semaphore name.
`flags`	Flags are listed below.
`count`	Initial value of the semaphore.
`smid`	Pointer to the semaphore ID (returned).

Description

Creates a semaphore using a call to the pSOS function:

```
sm_create( name, .count, .flags, .smid );
```

Errors are mapped through the "MapError" macro to return **TMLIBAPP_OK** in case of success. Possible errors are or'ed with **TMLIBAPP_ERR_OS_ERR**. Legal values for the flags are:

```
typedef enum tmosSemaphoreCreateFlags{
   tmosSemaphoreFlagsStandard = .0,
   tmosSemaphoreFlagsLocal    = .0,
   tmosSemaphoreFlagsGlobal   = .1,
   tmosSemaphoreFlagsPrior    = .2,
   tmosSemaphoreFlagsFifo     = .0
} .tmosSemaphoreCreateFlags, .*ptmosSemaphoreCreateFlags;
```

 ©1999 Philips Semiconductors 10/08/99

tmosSemaphoreDestroy

```
extern UInt32 tmosSemaphoreDestroy(
   UInt32  smid
);
```

Parameters

`smid`	Semaphore ID.

Description

Deletes a semaphore and free all associated resources using the pSOS function:

```
sm_delete(smid);
```

Errors are mapped through the "MapError" macro to return **TMLIBAPP_OK** in case of success. Possible errors are or'ed with **TMLIBAPP_ERR_OS_ERR**.

tmosSemaphoreP

```
extern UInt32 tmosSemaphoreP(
   UInt32  smid,
   UInt32  flags,
   UInt32  timeout
);
```

Parameters

smid	Semaphore ID.
flags	Flags are listed below.
timeout	Timeout period, in ticks.

Description

Attempts to acquire a semaphore using the pSOS function:

```
sm_p( smid, flags, timeout );
```

The timeout period is in ticks. The length of a tick is defined in the file sys_conf.h. Ticks are 10 ms by default. Errors are mapped through the "MapError" macro to return **TMLIBAPP_OK** in case of success. **ERR_TIMEOUT** is mapped to **TMLIBAPP_TIMEOUT**. Possible errors are or'ed with **TMLIBAPP_ERR_OS_ERR**. Legal values for the flags are:

```
typedef enum tmosSemaphorePFlags{
   tmosSemaphoreFlagsWait   = .0,
   tmosSemaphoreFlagsNoWait = .1
} .tmosSemaphorePFlags, .*ptmosSemaphorePFlags;
```

 ©1999 Philips Semiconductors 10/08/99

tmosSemaphoreV

```
extern UInt32 tmosSemaphoreV(
   UInt32  smid
);
```

Parameters

smid	Semaphore ID.

Description

Gives up a semaphore using the pSOS function:

```
sm_v(smid);
```

Errors are mapped through the “MapError” macro to return **TMLIBAPP_OK** in case of success. Possible errors are or’ed with **TMLIBAPP_ERR_OS_ERR**.

Timer

tmosTimSleep

```
extern UInt32 tmosTimSleep(
   UInt32  ticks
);
```

Parameters

ticks	Sleep period.

Description

Gives up control and sleeps for the specified number of ticks. The function is implemented using the pSOS function:

```
tm_wkafter(ticks);
```

The sleep period is in ticks. The length of a tick is defined in the file sys_conf.h. Ticks are 10 ms by default. Errors are mapped through the “MapError” macro to return **TMLIBAPP_OK** in case of success. Possible errors are or’ed with **TMLIBAPP_ERR_OS_ERR**.

Chapter 10

TriMedia Flash File System API

Note
This component library is not included with the basic TriMedia SDE, but is available as a part of other software packages, under a separate licensing agreement. Please visit our web site (www.trimedia.philips.com) or contact your TriMedia sales representative for more information.

Introduction

The Flash File System and flash-based standalone systems are two related topics. This chapter describes how TriMedia supports each of them.

First, it describes the generic Flash File System Manager provided by the SDE, which provides flash file access and storage and reliable updates of boot images.

Second, a number of scenarios are described, starting on 232, by which system software stored on flash can safely be upgraded by the embedded system itself. This process uses protocols that guarantee that the upgrade completes or, if write errors or power failures have occurred, that the original version remains installed.

Flash File System

Flash Basics

Flash memory functionality is a convenient mixture of ROM and RAM. Like ROM (but unlike RAM), its contents are persistent, in that they are not lost when the power is switched off. Like RAM (but unlike ROM), its contents can be modified in a running system[1]. Like both of them, contents of flash memory can be read by the processor using normal memory fetches.

In embedded systems, flash memory is typically used for storing boot images, system parameters, system logs (diagnostics), user data (address books, for example), and even for complete, general-purpose file systems. One especially attractive property of flash is that it can store system software that can be upgraded entirely by the system itself, with minimal user intervention. Such a system does not have to be disassembled for such an upgrade. Instead, it can merely be instructed to obtain the upgrade from a remote server via a network connection, to replace its predecessor in flash, and to reboot itself. The critical issue is that a careful update protocol must be followed. Otherwise, a power loss or write error during the upgrade process might corrupt the flash contents and render the system unusable.

Flash memory seems a convenient persistent storage device at a higher level, but there are some difficulties to be solved by lower level software. These difficulties have to do with writing to flash memory.

- Although flash can be read as RAM or ROM, writing a value into a flash memory location is considerably more involved, because it generally involves switching the entire chip to a programming mode, followed by some elaborate write protocol for transferring the value. This is illustrated by the sample **FLASH_write** function shown in the code sample on page 230.

1. This is unlike (EP)ROM, where the system using it must be stopped and the chip removed to update its contents.

 ©1999 Philips Semiconductors 10/08/99

- It is not always possible to write just any value into a given flash location. Rather, individual flash bits can only be toggled from a logical 1 to a logical 0 value. So although the contents of a particular location can be changed from 0xF0F0F0F0 to 0xF000F000, they cannot be changed to 0xF0FFF0FF. Consequently, flash locations generally must be erased to an all 0xFFFFFFFF pattern before they can be rewritten to an arbitrary value.
- Unfortunately, flash locations cannot be individually erased. Flash memory chips are organized in blocks[1], which are the units of flash erasure. In other words, before a flash location can be rewritten, its enclosing block has to be entirely erased. A typical flash block size is 32–256 kilobytes and, except for very specific applications, it is very likely that blocks will still be holding valid data when they have to be erased. Preserving this data through a block erasure, while guarding against power failures and flash write errors, requires prudent erasure schemes in flash system software.
- Block erasure occurs regularly in more intensively used flash file systems (for example, in preparation of particular file system sectors for updates). Unfortunately, there is an upper bound on the number of times that flash blocks can be erased or cycled. Depending on the type of chip, this amount is typically 10^5 to 10^6. When this upperbound has been reached, the probability of write and erasure errors rapidly increases. This means that attempted 1-to-0 writes and 0-to-1 erasures are more likely to fail, leaving the values of certain bits unmodified.

  Although this erasure upperbound is already quite high, and will tend to increase for newer flash devices, flash drivers that expect frequent flash updates generally implement some "wear leveling" scheme. Such a scheme periodically moves less frequently updated flash contents to more frequently erased flash blocks, with the intention to evenly spread erasure (and related wear) over all flash blocks.

Clearly, flash-based embedded systems must be very careful when updating flash contents. Power failures can always occur, and in any case, flash will eventually wear out. However, neither of these conditions should cause corruption of the logical flash contents. At worst, the system should roll back to a previous stable point. At best, it should continue functioning while avoiding errors, thus saving the flash contents until the chips can be replaced.

The TriMedia Flash File System software described in the next sections provides all these features.

- Wear leveling is supported.
- Power failures cause a rollback to the latest consistent state, which is the state before the last flash update (from a user point of view).
- Bad spots in flash are avoided as long as possible. When flash updates can no longer be made because the flash is full or because of unrecoverable write errors, the logical flash contents remain unaltered. A distributed file allocation table (FAT) maintained

1. Sometimes also called *sectors*. (The term *block* will be used here to avoid confusion with in file system sectors).

on flash ensures that this continuous consistency does not cause a noticeable performance degradation.

Generic Library

The table below lists the elements of the generic library.

Include File	$TCS/include/tmlib/tmFlash.h
Libraries	$TCS/lib/<endian>/libio.a
Sample Usage	tmcc main.c –lio –tmld –u _FlashFS ––
Mount Point	/flash
Examples	$TCS/examples/flash_file_system flash_demo, mkfs, autoboot, sample_drivers, write_boot, write_files, all_together

The flash file system manager provided by the TriMedia SDE is in the libio.a library. It can be used in both pSOS- and non pSOS-based applications and is fully reentrant.

It is also generic, in that it does not make any assumptions of what type of flash is used, how large it is, or at which memory range it is mapped. This flash-specific information may instead be provided to the application by the relevant board support package (BSP), in the form of a flash driver, as specified in *Flash File System Driver Specification* on page 229. General assumptions under which the flash manager works are specified in *Flash Assumptions* on page 225.

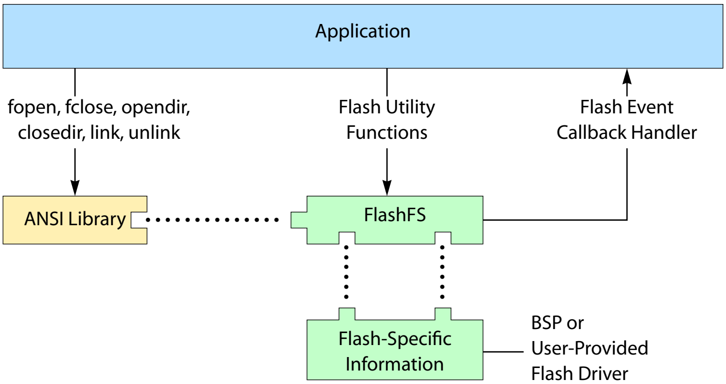


Figure 10 Flash File System as an Application Component

To give an application access to flash, merely link the flash file system manager and an appropriate BSP to that application. The file system manager automatically registers itself with the ANSI library (as shown in Figure 10), and when it detects a valid flash file system, this will be mounted at /flash. This file system can subsequently be accessed using

 © 1999 Philips Semiconductors 10/08/99

the usual ANSI and POSIX 1.1 I/O functions, by specifying file names that start with /flash/. For instance, a directory /flash/my_dir can be created by calling **mkdir("/flash/my_dir")**; a file /flash/my_dir/my_file can be opened for output by the function **fopen**; and directory structures can be traversed using the functions **opendir**, **readdir** and **closedir**. The flash_demo example provides a full demonstration of the functionality provided by the flash file system manager.

You are strongly encouraged to make use of the appropriate BSP to provide flash-specific information. You can include the BSP two ways. You can link the BSP on the command line or you can link the BSP automatically by modifying the **BOARD_LIST_EL** and **BOARD_LIST_EB** lines in **tmconfig**. We recommend modifying **tmconfig**. All examples in this chapter assume that the BSP is linked by modifying **tmconfig**.

The flash manager is usually enabled by presence of the "/flash" prefix in the file names that are given to I/O functions. Because of this, and since no explicit initialization is required by the application, it is possible that the flash file system can be used without any explicit calls to its functions. For this reason, library extraction of the flash file system manager from libio.a has to be forced during linking by placing an explicit reference to the symbol **_FlashFS** on the **tmld** command line, as in

```
tmcc main.c -lio -tmld -u _FlashFS --
```

Flash Event Handling

It is possible to act on the error codes returned by each individual flash access. Realistically, however, applications sometimes need some higher level of error handling. They can accomplish this by installing an event callback handler using the utility function **Flash_install_event_handler** that is described in *TriMedia Flash File System API Data Structures* on page 236. The flash manager defines a range of informational and warning events, such as:

- Flash getting full
- Flash definitely full
- Flash sector written
- Flash write error
- Failure of assumptions under which safe flash operation is guaranteed. See *Flash Assumptions* on page 225).

Long running applications should monitor the write error/sector write ratio. An increase in this ratio indicates that the flash is deteriorating, and a warning should be given that advises users to replace the flash.

Formatting Flash

The flash file manager is enabled only when it detects a valid file system. Flash memory can be formatted to an empty file system by means of the function

FlashUtil_init_filesystem. (See page 238). Formatting should be performed by a separate utility, since there is no way to reinitialize the flash file system manager after a reformat.

An example of such a utility can be found in $TCS/examples/flash_file_system/mkfs. Note that formatting erases all previous flash contents.

Forced library extraction of the flash file system should not take place as this will cause the initialization of the flash file system. The initialization of the flash file system by the BSP should also be disabled. This was done in the example $TCS/examples/flash.file.system/mkfs/mkfs.c. Initialization of the flash file system expects a valid file system to be in place, which may not be the case.

Copying Files Onto Flash

Directories and files can be created on flash using standard I/O functions. Therefore, directories and files can be easily copied to flash from any other storage medium that is also connected to the current TriMedia board. For example, simple applications can be devised for copying from a TCP/IP connection to flash or (in hosted systems) from the PC disk to flash.

An alternate solution is provided by the tool **tmSEA** (see *tmSEA: Self-Extracting Archives* in Chapter 11.). This tool can be used to embed an entire directory tree structure in compressed form in a TriMedia application. This application can subsequently be downloaded to a standalone, flash-based system (via JTAG, for example). When run, it unpacks its embedded directory tree and writes it to flash.

Boot Images

Apart from a file system, the flash manager maintains at most one boot image on flash. A boot image can be stored on flash using the function **FlashUtil_put_bootimage** (see page 238), thereby replacing any previously stored boot image. The boot image is not visible in the file system, but can be loaded into SDRAM and started via the function **Flash_boot**. This function is typically used in L1 boot programs, as is demonstrated in $TCS/examples/flash_file_system/auto_boot.

When building **auto_boot**, you must provide the flash-specific information in the form of the flash driver, as specified in *Flash Driver Boot Specification* on page 231. This is the only case where the BSP cannot be used. **auto_boot** must be less than 2K in size. Including the BSP will make auto_boot too large to fit in the EEPROM.

Updating the boot image stored on flash is facilitated by the tool **tmWRB** (see *tmWRB: Boot Image Writing* in Chapter 11), which converts a boot image into an executable that, when downloaded and started on a flash-based system, writes the boot image that it contains onto flash.

Boot images can also be compressed into a new boot image using tool **tmSEI** (see *tmSEI: Self-Extracting Load Images* in Chapter 11). This new boot image is still intended for system booting, but instead of immediately starting the actual system, it unpacks and

 ©1999 Philips Semiconductors 10/08/99

decompresses the original boot image, places it at the proper position in SDRAM, and starts it. Use of **tmSEI** causes boot images to occupy less flash space, with space savings of 50%, typically, for large images.

Figure 11 Cascading tmSEI and tmWRB

The **tmSEI** and **tmWRB** tools are typically used in a cascade, as shown in Figure 11. This first converts a boot image into a compressed, self-extracting image, and subsequently packs it into an application that updates the flash's boot image with this self-extracting image. This is fully demonstrated in $TCS/examples/flash_file_system/write_boot.

Although not visible to or accessible by regular I/O functions, the boot image is normally stored in the flash file system. This means that no special flash partition need be reserved, and that flash space that is not occupied by the boot image is automatically available for regular files. In particular, when no boot image has been stored, the entire flash is available for the file system. A boot image can always be written later without any need for reformatting, as long as there is space available.

As described in Flash Manager Properties on page 226, the writing of a new boot image is an atomic, safe operation. It either succeeds or leaves the logical flash state unaltered.

Flash Assumptions

The flash file system manager operates under the following assumptions.

1. Flash errors manifest themselves only as erase or write failures, by failing to write certain bits from a 1-to-0 value, or by failing to erase them from a 0-to-1 value. Flash memory contents remain stable, as long as they are not erased or overwritten.
2. Flash write operations currently have no more than seven write errors per byte. Such a heavy error condition is considered extremely unlikely, even for moderately deteriorated flash. However, if this threshold is exceeded at an awkward moment during certain internal commit operations, such a commit will fail and the flash file system might become inconsistent. The flash file system manager will detect such a situation, report a **FlashDangerousWriteError**, and try to recover from it, but inconsistencies might still result in the event of a subsequent power loss, or where recovery is not possible due to a flash full condition or unavailability of reliable flash sectors.

Flash Manager Properties

Update Safety Properties

As long as no **FlashDangerousWriteErrors** occur as discussed above, errors and power failures will generally let the operations succeed or, at the worst, cause them to fail with the logical state of the flash file system preserved.

However, although the *logical* state might be unaffected, the data at the *physical* level might have changed. For instance, prior to erasure of a certain flash block, data still in use might have been moved off this block. Consequently, logical flash sectors might have multiple (identical) physical instances at specific moments during flash operations. Usually this is corrected shortly afterwards by committing the data move and removing the old instance. However, a block might contain trash sectors, especially after power failures.

The following table specifies the safety properties of the individual flash file system operations. These safety properties can be used as basis for safe, higher level system upgrade protocols.

Category	Description	Operations
I	Not appropriate, because they do not alter the flash file system.	**opendll**, **isatty**, **seek**, **read**, **fstat**, **stat**, **access**, **open** (read mode only), **close** (read mode only), **opendir**, **closedir**, **readdir**, **rewinddir**
II	Either fully succeeds, or fails while leaving the flash file system unaltered.	**link**, **unlink**, **mkdir**, **write** boot image
III	File creation, using **open** in write mode, is committed when either the file is successfully closed, or successfully **sync**'d or **fsync**'d. Before that, it is unspecified whether the file is physically created in the flash file system.This situation is similar for file updates, using **write**.	**open** (for write), **write**, **close** (for write), **fsync**, **sync**

Flash Manager Space Overhead and Limitations

- Sector size: 2048 bytes.
- Maximum file name length: 300 characters.
- Maximum supported number of flash blocks: 249
- Maximum supported flash block size: 512 kilobytes
- Number of sectors allocated for each directory: 1
- Number of sectors allocated per file: 1 + (file size in bytes – 1716) / 2032.

 ©1999 Philips Semiconductors 10/08/99

This implies a file administration overhead of 1% of file size (asymptotically)

- Number of sectors allocated per boot image: (boot image size in bytes) / 2028.

  This implies a boot sector administration overhead of 1% of boot image (asymptotically)

- FAT overhead: 0.8% of entire flash
- Reserved amount of flash for wear leveling algorithm: 1 flash block

Sample Flash Performance Figures

Three key operations affect flash file system performance, namely writing, reading, and erasing. The following performance data were collected using the Philips NAB board containing Am29f016B flash memory on a 100 MHz TM-1000. The NAB board uses 8 megabytes of flash in 32 blocks. The data were taken with the compiler optimization set to **-O3**. The **tmcc** profiling and grafting options were not used.

The flash file system write performance test was carried out on a formatted flash file system. Over 7.5 megabytes of data was written. Various files were written to different directories. An average write speed of 0.25 Mb/s was achieved. The write speed varied between 0.37 Mb/s and 0.17 Mb/s. The flash write speed tended to decrease as the flash file system filled up.

The flash file system read performance test was carried out on an almost full flash file system. Various files were read from different directories. An average read speed of 4.37 Mb/s was achieved. The read speed varied between 6.05 Mb/s and 4.33 Mb/s.

There is no direct way of using the standard ANSI or POSIX 1.1 I/O functions to force the flash file system to erase a block. In normal operation, a block is erased only when it is required, i.e., when all blocks either contain valid data or invalid data which may be erased. Therefore, formatting the flash file system using the **mkfs** example program was chosen to give an estimate of the erasure time. There is overhead in setting up the flash file system. The overhead is estimated to be under 5% when formatting the entire 8 megabytes of flash. To run mkfs.out took between 23.2 and 20.8 seconds, giving an approximate erasure speed of between 0.34 Mb/s and 0.39 Mb/s. The speed difference depends on how much data was previously in flash. For wear-leveling reasons, before a block is erased, the NAB flash driver checks whether the block must be erased. Checking whether a block must be erased actually takes slightly more time than erasing the block.

Dynamic Libraries on Flash

A simple way of dealing with dynamic libraries on flash is provided by the flash file system manager. Dynamic libraries are searched for first in the directory /flash/old_dlls, and then in the directory /flash/dlls.

This scheme allows for safe updating of minimal system components that consist of one or more dynamic libraries (using the safety properties described in the previous section).

A minimal system component here is a subset of the collection of dynamic libraries that form an application, and can be further defined as follows:

1. The application works with the old version of the component.
2. The application works with the new version of the component.
3. The application does not work (or is not known to work) with part of the old, and part of the new version of the component.

In particular, the last property requires that component replacement be one atomic operation from the dynamic loader's point of view. An example procedure for safe updating of such a minimal system component is to keep all dynamic libraries in directory /flash/dlls. When updating, do the following:

1. Create a directory /flash/old_dlls, and move all dynamic libraries that form the old version of the component to this directory. This hides whatever happens to the component in /flash/dlls.
2. Move all dlls that form the new version of the component to /flash/dlls. All this is hidden from the dynamic loader.
3. Rename /flash/old_dlls to something like /flash/obsolete_dlls. This swaps the new version of the component in place in one atomic action.
4. Remove /flash/obsolete_dlls.

Because this procedure can be interrupted by errors and power failures, the /flash/old_dlls should be inspected at each application initialization. When files are detected in this directory, these should be placed back in /flash/old in order to roll back to the previous state.

Unimplemented Functionality

The flash file system manager currently does *not* provide the following functionality.

1. The (internal) flash file system format is endian-dependent. This means that a file system created by a big-endian executable cannot be used by little-endian executables, and vice versa.
2. No file protection modes are provided.
3. No file creation/modification times or dates are provided.
4. No garbage-reclaim utility (for garbage left after power failures) is currently provided.
5. The flash manager does not yet contain an option for transparently handling compressed files. This will be corrected in a future release, in which the flash manager will be able to decompress such files automatically upon reading them.

 © 1999 Philips Semiconductors 10/08/99

Flash File System Hardware Interface

This section describes the interface between the flash file system manager and the physical flash device. You have two options when choosing the hardware interface for the flash file system: link the relevant board support package with the application or use a flash-specific driver instead. We strongly encourage you to use an appropriate BSP to provide flash-specific information.

Using the Flash File System with the BSP

Linking the relevant board support package with the application will automatically take care of the flash file system hardware interface. The device library libdev.a is automatically linked with all non-dynamic applications.

When running dynamic applications, the flash file system DLL is required. Note that if your DLLs reside in a flash file system and you application is dynamic, the libtsaFlashFS.dll and libtsaFlash.dll must be embedded in your application using the **tmld** flag **-bembed**, in order to access the flash file system.

Flash File System Driver Specification

The sources of several sample flash drivers can be found in directory $TCS/examples/flash_file_system/sample_drivers. One of them, a driver for the Philips NAB board, is listed in Figure 12. Also included in this directory is a flash simulator by which flash-based applications can be tested without having actual hardware available. This simulator is capable of simulating flash write and erase errors with an adjustable error rate.

Flash Address Space

The flash driver completely hides the physical flash device from the flash file system manager. Although the physical flash may be partitioned in different segments or banks (that can only be accessed one at a time after having selecting it), and although the physical flash may be mapped at an arbitrary place in the current address space, the driver nonetheless provides a view of one logical, single flash bank starting at flash address 0. The size of this logical flash bank is defined by the flash driver by means of two global (constant) variables. It consists of NROF_FLASH_BLOCKS consecutive logical flash blocks, each of size SZOF_FLASH_BLOCK.

The flash is accessed by the flash file system manager using two read functions, two write functions, and an erase function provided by the driver. One read and one write function access flash on long word boundaries only, and read and write long words. The other read and write functions, and the erase function, access flash on logical flash block boundaries only, and read, write, and erase logical flash blocks. No byte or short access is needed by the flash file system manager. In addition to reading, writing and erasing, these functions are also responsible for logical-to-physical flash mapping. Usually, when the physical flash consists of a single bank, this mapping merely consists of adding an offset to the logical flash addresses, but when the physical flash consists of several banks, or when the physical flash only allows byte access, this mapping becomes more intricate.

Sample Driver

Shown below is a sample driver that uses a Philips NAB board containing Am29f016B flash memory:

```
#include "tmlib/tmtypes.h"
#define FLASH_BASE        ((Address)0xFF400000)
Int SZOF_FLASH_BLOCK    = (Int) 0x40000;
Int NROF_FLASH_BLOCKS   = (Int) 32;
#define RETRY_COUNT       20

Bool FLASH_block_erase( UInt ab, Bool check_if_necessary ){
   Int i;
   volatile UInt32 *blockbase = (Pointer)(FLASH_BASE+ab*SZOF_FLASH_BLOCK);
   volatile UInt32 *flashbase = (Pointer)FLASH_BASE;

   if( check_if_necessary ){
      Int i;
      UInt32 *pt = (Pointer)blockbase;
      Bool necessary = False;
      for( i = 0; i < (SZOF_FLASH_BLOCK/sizeof(Int)); i++ ){
         if( *(pt++) != 0xffffffff ){
            necessary = True;
            break;
         }
      }
      if( !necessary ) return True;
   }
   for( i = 0; i < RETRY_COUNT; i++ ){
      flashbase[0x555] = 0xAAAAAAAA;
      flashbase[0x2AA] = 0x55555555;
      flashbase[0x555] = 0x80808080;
      flashbase[0x555] = 0xAAAAAAAA;
      flashbase[0x2AA] = 0x55555555;
      blockbase[0x000] = 0x30303030;
      while( (*blockbase ^ *blockbase) & 0x40404040 ){}
      if( (*blockbase) == 0xffffffff ) return True;
   }
   return False;
}
Bool FLASH_write( Pointer address, UInt32 data ){
   Int i;
   volatile UInt32 *addr      = (Pointer)(FLASH_BASE+(UInt)address);
   volatile UInt32 *flashbase = (Pointer)FLASH_BASE;
   UInt32   old_data          = *addr;
   UInt32   new_data          = old_data & data;

   if (new_data == old_data) return data == new_data;

   for( i=0; i<RETRY_COUNT; i++ ){
      flashbase[0x555] = 0xAAAAAAAA;
      flashbase[0x2AA] = 0x55555555;
      flashbase[0x555] = 0xA0A0A0A0;
      *addr = new_data;
      while( (*addr ^ *addr) & 0x40404040){}
      if (*addr == new_data) { return True; }
   }
```

```
Bool FLASH_block_write( Pointer flash, Pointer image, Int nrof_words ){
   Int32 *f = ((Int32*)flash) + nrofwords;
   Int32 *i = ((Int32*)image) + nrofwords;
   Bool result;
   do{
      result = FLASH_write( --f, *(--) );
   } while( (Pointer)f != flash && result );
   return result;
}
UInt32 FLASH_read( Pointer address ){
   UInt32 *addr= (Pointer)(FLASH_BASE+(UInt)address);

   return *addr;
}
void FLASH_block_read( Pointer flash, Pointer image, Int nrof_words ){
   Int32 *f = ((Int32*)flash);
   Int32 *i = ((Int32*)image);
   while( nrof_words-- ){
      *(i++) = FLASH_read( f++ );
   }
}
Bool FLASH_init() { return True; }
```

Figure 12 Sample Driver for Philips NAB Board, Containing Am29f016B Flash Memory

Flash Driver Boot Specification

When booting from flash, the L1 boot program that resides in a 2K EEPROM must be capable of reading flash and copying the relevant L2 boot program from flash into SDRAM. A flash driver is required for this since BSPs are too big to include in an application that must fit in the 2K EEPROM.

The flash driver boot specification is a subset of the flash file system driver specification. The boot specification requires that the integers **SIZOF_FLASH_BLOCK** and **NROF_FLASH_BLOCKS** be defined, and that the functions **FLASH_read** and **FLASH_block_read** be defined.

In the example flash L1 boot program, as is demonstrated in $TCS/examples/flash_file_system/auto_boot, it is assumed that the flash hardware is initialized and can be used by default.

If you are using a BSP as the standard hardware interface for all other applications, you must ensure that the parameters **SZOF_FLASH_BLOCK** and **NROF_FLASH_BLOCKS** in the flash boot driver are equal to the corresponding parameter in the BSP. To do this, call the function **tsaFlashGetCapabilties** as defined in tsaFlash.h. Set the value of **SZOF_FLASH_BLOCK** to the returned **sectorSize** field multiplied by the size of a word. set **NROF_FLASH_BLOCKS** to the returned **numberOfSectors** field.

Standalone Flash-Based Systems

This following sections offer suggestions on how to set up a standalone system using the flash file system manager, and how to perform safe updates of system software stored on flash. The expression “safe updates” denotes the replacement of the system software, or parts thereof, in such a way that errors or power failures during an update do not result in an inconsistent, useless system.

Role of the Boot Image

A cold boot of a TM-1 processor in standalone mode causes an initial program to be loaded and started from the IIC-connected boot EEPROM. The TM-1 hardware requires this program to be smaller than 2 kilobytes, which is obviously too small for any realistic application. This situation has led to the L1/L2 standalone boot procedure in which the actual, unrestricted application is loaded by this initial program (from JTAG, flash, or EPROM, for example). In L1/L2 boot terminology, the initial program is referred to as L1, and the subsequently loaded application as L2. L1/L2 booting is fully described in Chapter 7, *Bootstrapping TriMedia in Autonomous Mode*, of Book 2, *Cookbook*, Part C.

In a flash-based setup using the TriMedia flash manager, the L2 application will be stored as the flash boot image, to be started by L1 using function **Flash_boot** of the flash manager API (see page 238). The size of this function is about 800 bytes, which is small enough to be used in an L1 image.

Although applications of any size can be easily stored as the flash boot image, it is simpler when this flash boot image instead holds the second stage of a new, three-stage boot procedure. In such an extended boot procedure, L2 is no longer the final application, but just a second, more powerful loader that subsequently loads the final application as an L3 from a regular flash file. This results in a situation in which all system software can simply be thought of as residing in one or more regular flash files, with an application-independent L2 loader stored as the boot image. By this, the issue of safely upgrading system software can now be entirely expressed in terms of manipulating regular flash files. Additionally, because this L2 loader is application-independent, it can be standardized as a generic boot component, to be used by different applications. It needs few if any updates.

Various application-independent functionality can be put into this L2 loader. For instance, it can decide between continuing to boot from flash or obtaining the final application L3 via an external port such as JTAG. Such flexibility is useful during development, but also attractive as a diagnostic or service option in production systems and hence should be standardized. Also, in a dynamic loader based setup (see 233), the L2 loader can have the form of an application shell as described in the linker documentation (see Chapter 11, *Linking TriMedia Object Modules*, in Book 4, *Software Tools*, Part B.). Such a shell contains most or all board-specific drivers and board support, and lets the subsequently loaded application remain more or less board-independent.

 © 1999 Philips Semiconductors 10/08/99

Use of the Dynamic Loader

Use of a single application image in a flash-based production system has a number of disadvantages:

- Most safe upgrade strategies are based on first copying the new version onto flash, toggling some form of a switch, and then deleting the old version. This means that for safely upgrading an image file of size N, flash space of at least 2N should be available. Since most multimedia applications consist of multi-megabyte images, the reserved amount of flash space needed for upgrading is considerable.
- It is hard to upgrade individual components, especially when an application consists of parts provided by multiple vendors. For instance, in a Java-based, monolithic application, an upgrade in the Java virtual machine can only be performed with the cooperation of the owner of the application, who should relink the application with the upgrade of Java, and redistribute it.
- Applications are necessarily system-dependent. Because they are monolithic, they must include full board support, and because they are load images, they must know about the load address, SDRAM size, processor frequency, etc. from the target system.

Most or all of these disadvantages can be solved using the TriMedia dynamic loader. A dynamic loader based setup allows an application to be partitioned into a number of dynamic libraries, plus a largely application-independent core that contains all board-dependent drivers.

The core board functions can be embedded in the L2 boot image, ideally by the manufacturer of the board. This is more feasible, because board definitions are more standardized in board support packages.

When the dynamic libraries are given well defined, controlled interfaces, more finely grained application upgrades can be performed, involving small groups of dynamic libraries or even single dynamic libraries. For instance, a new Java interpreter requires only one corresponding dll to be upgraded, with all other parts of the system untouched. In particular, this means:

- Smaller reserved flash space is needed for upgrading.
- Modifications by component vendors can be incorporated by application clients, without the need for cooperation by application vendors.
- Components are application- and board-independent. Applications contain less board dependencies or might even contain no board dependencies at all.

When time-critical parts of the application are linked as dynamic libraries in *immediate* mode (see Chapter 13, *Dynamic Linking API*), these libraries are loaded and linked from flash during application startup with a result that is indistinguishable from a statically linked executable. No inter-library function calls and variable references will be redirected via function stubs.

The advantages of dynamic loading must be weighed against the following disadvantages:

- Upfront loading and linking adds to startup time.
- Dynamic libraries contain more flash space than images because they are still relocatable.

Safe Upgrading Basics

Self-upgrading systems can basically choose between the following two update schemes.

Update Scheme 1

One scheme involves partial upgrading, where independent components are replaced with new versions having an unchanged or extended interface. Examples of this are:

- upgrading the entire system
- upgrading the Java virtual machine (jvm.dll)
- upgrading the entire set of Java romized classes(romjava.dll)
- upgrading a device library with a new, corrected version (libVO.dll, only internal change)
- upgrading the OS with a new version that has a more efficient implementation (psos.dll)

Essential to this type of system upgrading is that the entire system be functional with both the old and new version of the upgraded component. This means that an upgrade can be started and followed at any moment by a commit or rollback. This is best illustrated using pSOS.dll. Because the pSOS external interface is well-defined and stable, it can be readily replaced with a new version that contains either a different implementation of this interface or one that simply provides more system calls. As long as it is well tested and correct, the upgraded system will function with both the old and new pSOS version.

Using the safe flash update properties described on 226, and a variant of the hide-and-swap protocol described on page 227, a system can start a component upgrade knowing that, whatever happens, the system will remain functional.

Update Scheme 2

Where system components can be accessed at any time at standardized places on an network, systems can be designed that follow less safe upgrade protocols. Using the safe flash update properties but with unsafe, opportunistic, simple copying schemes, systems can economize on flash save space, at the risk of ending up in inconsistencies after errors during updates. Recovery from inconsistencies is then possible by keeping a small network-enabled loader in flash that collects all system components from their servers. Note

 ©1999 Philips Semiconductors 10/08/99

that when using this scheme, the system might become unusable (temporarily or permanently), when servers are down, or in the case of heavily deteriorated flash.

TriMedia Flash File System API Data Structures

EventHandler

```
typedef void (*EventHandler)(
   FlashEvent
);
typedef enum {
   FlashCleanBlock,
   FlashSwapCleanBlock,
   FlashSectorWrite,
   FlashPurgeTT,
   FlashRelocateSector,
   FlashWriteError,
   FlashFailedCleanBlock,
   FlashBecomingFull,
   FlashFull,
   FlashDangerousWriteError,
   FlashStaleBootRemoved,
   FlashStaleSTTRemoved,
   FlashStaleRootRemoved,
   FlashInterruptedMoveRepaired
} FlashEvent;
```

Fields

`FlashCleanBlock`	Info: Block dirty sector reclaim started.
`FlashSwapCleanBlock`	Info: Block dirty sector reclaim and block contents swap with a less often erased one started.
`FlashSectorWrite`	Info: Sector allocated and written.
`FlashPurgeTT`	Info: **SectorTranslatorSector** has been purged and reallocated because one of its entries became full.
`FlashRelocateSector`	Warning: Flash sector was relocated due to write error during block dirty sector reclaim.
`FlashWriteError`	Warning: Flash write error detected. Sector has been marked bad, and retried on other sector.
`FlashFailedCleanBlock`	Warning: Block dirty sector reclaim failed, usually due to an abundance of write errors resulting in a reserved pool underflow.
`FlashBecomingFull`	Warning: Starting to allocate from reserved pool.
`FlashFull`	Warning: Last flash sector write failed.
`FlashDangerousWriteError`	Serious warning: More than 7 bits were in error during an attempted uncommit of a virtual block mapping. The uncommit failed, and in case of a

	power loss during retry, or a flash full condition during retry, the file system will become inconsistent.
`FlashStaleBootRemoved`	Stale boot sector, probably caused by power loss during boot image update, has been freed
`FlashStaleSTTRemoved`	Stale **SectorTranslatorSector**, probably caused by power loss during block erase, has been freed.
`FlashStaleRootRemoved`	Stale root sector, probably caused by power loss during block erase, has been freed.
`FlashInterruptedMoveRepaired`	Intermediate file move condition, probably caused by power loss during block erase, has been repaired.

Description

Provides for installing a callback function that is called upon the events described above. The default event handler does not do anything at all. However, a more realistic handler should at least detect that the write error/write sector ratio becomes nonsignificant, and advise users to replace their flash. A **FlashFull** event is not strictly harmful. The file system remains consistent, although it is of course no longer possible to update.

The **Flash_install_event_handler** function installs the specified handler, and returns the old one:

```
EventHandler Flash_install_event_handler(
   EventHandler
);
```

TriMedia Flash File System API Functions

FlashUtil_init_filesystem

```
Bool FlashUtil_init_filesystem();
```

Description

The function creates an empty file system on the flash. Failure is generally caused by write errors.

FlashUtil_put_bootimage

```
Bool FlashUtil_put_bootimage(
   Pointer  p,
   Pointer  start_address,
   Int      size
);
```

Return

This function returns True if it succeeded and False if it failed.

Description

Function for writing a new boot image to flash. The boot image is passed via 'image', and has specified size. It is intended for copying to the specified start address during booting. Note that this start address can be chosen per boot image.

Flash_boot

```
void Flash_boot();
```

Description

Copies the boot image from flash into SDRAM, and starts it. The function never returns. Note that the start address in SDRAM where the image will be copied has been defined by the corresponding call to **FlashUtil_put_bootimage**, and that the caller is responsible that this target area is unused during **Flash_boot**. This function should be called with data cache disabled.

 ©1999 Philips Semiconductors 10/08/99

Flash Driver API

FLASH_block_erase

```
Bool FLASH_block_erase(
   UInt   ab,
   Bool   check_if_necessary
);
```

Parameters

`ab`	The logical flash block to be erased. Block 0 is the first valid logical block in flash.
`check_if_necessary`	Indicates that you should check whether erasure is really needed (the block might already have erased contents). This parameter is typically used during initialization of the flash manager to have certain blocks erased as start condition without unnecessarily adding to flash wear.

Description

The function erases the specified logical flash block.

Return

The function returns True if it succeeded and False if it failed.

FLASH_init

```
Bool FLASH_init();
```

Description

This function performs required initialization, if any.

Return

The function True if it succeeded and False if it failed.

FLASH_write

```
Bool FLASH_write(
   Pointer  address,
   UInt32   data
);
```

Parameters

`address`	The logical address at which to write. Address 0 is the first valid logical address in flash.
`data`	The data to be written.

Description

This function writes the specified 32-bit value at the specified logical flash address.

Return

The function returns True if it succeeded and False if it failed.

FLASH_read

```
UInt32 FLASH_read(
   Pointer   address
);
```

Parameters

`address`	The logical address at which to read. Address 0 is the first valid logical address in flash.

Description

The function reads the specified 32-bit value from the specified logical flash address.

Return

The function returns the 32-bit value read from flash.

 © 1999 Philips Semiconductors 10/08/99

FLASH_block_read

```
void FLASH_block_read (
   Pointer   address,
   Pointer   image,
   Int       number_of_words
   );
```

FLASH_block_write

```
Bool FLASH_block_write (
   Pointer   address,
   Pointer   image,
   Int       number_of_words
   );
```

Parameters

address	The logical address in flash at which to read or write. Address 0 is the first valid logical address in flash.
image	The address in RAM of the image to transfer.
number_of_words	The number of words to transfer.

Description

The above two functions are similar to **FLASH_read** and **FLASH_write**. These two functions are guaranteed to read data from. or write data to, a single flash block. Hence, bank selection can be shared between all transferred words.

FLASH_block_write should write the image starting with its last word, as in this pseudocode example:

```
f = address + number_of_words * sizeof(UInt32);
i = image + number_of_words * sizeof(UInt32);
do {
   result = FLASH_write( --f, *(--i) );
} while (···)
```

 © 1999 Philips Semiconductors 10/08/99

Chapter 11

General Purpose Compression API

Licensing Issues

The directory tree $TCS/examples/compression/zlib provides the public domain, general-purpose data compression library **zlib 1.1.3** by Jean-loup Gailly and Mark Adler. The library has been obtained as is from **http://www.cdrom.com/pub/infozip/zlib/**, and carries a copyright notice, which is reproduced here:

```
(C) 1995-1998 Jean-loup Gailly and Mark Adler

This software is provided "as-is", without any expressor implied warranty.
In no event will the authors be held liable for any damages arising from the
use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must
   not claim that you wrote the original software. If you use this
   software in a product, an acknowledgment in the product
   documentation would be appreciated, but is not required.

2. Altered source versions must be plainly marked as such, and must
   not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source
   distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

The header file zconf.h has been altered. The changes made are listed in the following file $TCS/examples/compression/zlib/zlib-1.1.3/CHANGESMADE. The rest of the source of zlib remains unaltered. However, the makefile was altered to produce TCS-compatible binaries.

 © 1999 Philips Semiconductors 10/08/99

Overview

The zlib compression library provides in-memory compression and decompression functions, including integrity checks of the uncompressed data. This version of the library supports only one compression method (deflation) but other methods will be added later and will have the same stream interface.

Compression can be done in a single step if the buffers are large enough (for example, an input file is mmap'ed), or can be done by repeated calls to the compression function. In the latter case, the application must provide more input and/or consume the output (providing more output space) before each call.

The library also supports reading and writing files in gzip (.gz) format with an interface similar to that of stdio.

The library does not install any signal handler. The decoder checks the consistency of the compressed data, so the library should never crash even in case of corrupted input.

Zlib Statistics

Compression statistics were computed using the high-level compression functions **compress** and **uncompress**. These functions use the default values for speed and memory requirement.

Compression of data makes more memory available to a user. However, when using a compression algorithm, a user should be aware of the cost of compression. Code size will increase because of the size of the compression or decompression source. Dynamic Memory requirements may increase as data are compressed or decompressed. Access time to the data will also increase as the data must be decompressed before it can be used.

The following measurements were carried out using the default compression level. The compression levels can vary between 0 and 9. Level 1 gives the best speed. Level 9 gives the best compression. Level 0 gives no compression at all. The default compression level is 6, which makes a compromise between speed and compression.

The performance data were collected using the Philips NAB board on a 100 MHz TM-1. They were taken with the compiler optimization set to **-O3**. The **tmcc** profiling and grafting options were not used. The use of profiling and grafting will increase the compression and decompression speed of the zlib library.

Using the zlib library increases the code size of your application. If an application compresses data only (no decompression), the code size will increase by an extra 57 kilobytes. If an application decompresses data only (no compression), the code size will increase by an extra 57 kilobytes. If an application uses both compression and decompression, the code size will increase by about 112 kilobytes.

Dynamic memory is generally required to carry out compression and decompression. When decompressing data, zlib allocates, and subsequently frees, a maximum of 47 kilobytes. When compressing data, zlib allocates, and subsequently frees, a maximum of 268 kilobytes.

The time taken to compress or decompress data depends on the type of data. The following timing measurements were done on a Philips NAB board running on a TM-1 at 100 MHz.

When compressing data, the rate of compression varies depending on the type of data. The compression/decompression tests were carried out on various DLL and Dynamic applications. The average data compression rate was 0.237 Mb/s. The average data decompression rate was 4.53 Mb/s. If **tmcc**'s profiling and grafting options are used when calculating the rate of compression or decompression, then the average compression rate is 0.74 Mb/s and the average decompression rate is 7.15 Mb/s.

When compressing data, the compression ratio varies depending on the type of data. Text usually compresses better than a binary image. The compression/decompression tests were carried out on various DLL and Dynamic applications. In the test carried out, the average compression ratio (compressed vs. original image) was 44.78 percent. The rate of compression of the original images varied from between 24.95% and 61.05%.

Endian Independence

The tests carried out verify that zlib is "endian" independent. Data can be compressed on a big endian machine and can be decompressed on a little endian machine, and vice versa. The data itself will remain as it was before compression, i.e., big endian data compressed on a big endian machine will remain big endian data, even when decompressed on a little endian machine.

 © 1999 Philips Semiconductors 10/08/99

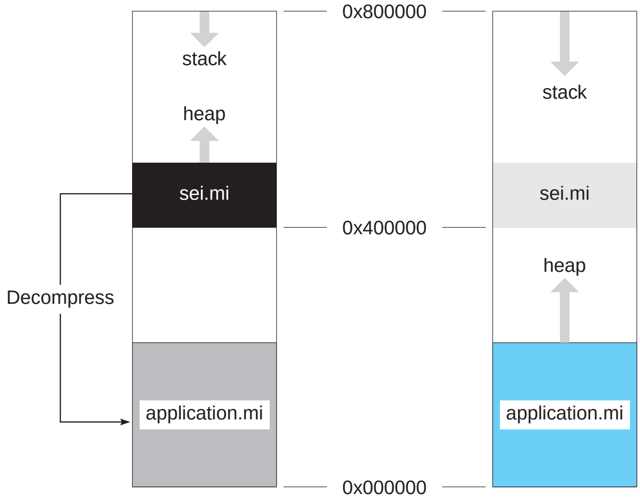
Compression Tools

The compression library forms a basis for utilities related to flash memory and flash boot that are described in the next three sections. The utilities are provided in the form of examples under $TCS/examples/compression/zlib/utilities.

The examples provided in this chapter assume that the relevant BSP is linked by default. You can do this by altering the **BOARD_LIST_EL** and **BOARD_LIST_EB** lines in **tmconfig**. Tee compression tools use the BSP as the flash file system hardware interface by default.

Because TriMedia tools place no other dependencies onto the compression library, developers are free to replace the compression library with their own version.

tmSEI: Self-Extracting Load Images




```
tmcc -host nohost ... -o application.out

tmSEI application.out -o sei.mi \
      -load 0x000000,0x800000   \
      -sei  0x400000,0x800000   \
      -tm_freq 100000000        \
      -mmio_base 0xefe00000
```

Figure 13 A Self-Extracting Load Image

tmSEI is a sample tool that can be used when the size of load images is an issue, for example, when these load images must be stored on a scarce storage medium, such as flash in an embedded system.

The tool achieves size reduction by extracting a load image from a given executable object file, subsequently compressing it using the compressor library, and finally embedding the result into an extractor application. The sole purpose of this extractor application is to decompress the original load image to its start address, and transfer control to it. This scheme reduces the net size of the load image when the size reduction gained by compression exceeds the constant overhead of the compressor library that is subsequently linked to it. Note that the extractor application is discarded after starting the original application, as shown in Figure 13.

tmSEI needs the following options and arguments:

- Name of input file. This must be a TriMedia executable object file compiled with **host=nohost**.
- The DSCPU frequency and MMIO base address of TriMedia processor, via options **-tm_freq**, and **-mmio_base**, respectively. These values are filled into the load image that is extracted from the input executable.
- The load range of extracted load image, via option **-load**.
- The load range of 'sei' image, via option **-sei**. When this option is omitted, the output file will still be an executable object file.

Because the extracted load image generally needs to be assigned the entire SDRAM, the arguments of option **-load** usually are the SDRAM base and end address. The arguments of option **-sei** must be chosen so that the extracted load image, at the start of SDRAM, does not overlap the memory range in which the unpacker itself operates. This can be achieved by loading this unpacker at the top of SDRAM, hence leaving the lower end available for unpacking. In this scheme, the unpacker should be loaded at **SDRAM_BASE+N**, where **N** is larger than the summed size of all the initialized sections as reported by **tmsize**, rounded up to the next multiple of the TM instruction cache size of 64 bytes. For executables that are not extremely large, simply load the unpacker at the middle of SDRAM, as shown in Figure 13. Note that the flash boot image writer allows the choice of a different base address for each individual boot image.

Extractor images can be used as L2 boot programs, provided that the L1 boot code loads them at the proper start address. See Chapter 7, *Bootstrapping TriMedia in Autonomous Mode* of Book 2, *Cookbook*, Part C, for a description of L1/L2 booting.

Sample Performance

The following shows the size reduction, plus unpack time, of two large sample executables. One of these executables (P1) almost exclusively consists of TriMedia instructions, while the other (P2) consists mostly of embedded Java class files. Both of them reduce to slightly below 50%. Unpacking increases the startup time by about one second, at a decompression rate of approximately 3.5 megabytes/sec on a 100 MHz TM-1. The constant overhead of the unpacker application has been observed to be around 80 kilobytes, with a size reduction break-even point around 240 kilobyte (original) image size.

 © 1999 Philips Semiconductors 10/08/99

P1

P1 is a large multimedia application.

- Contents: 1.76 megabytes of instructions, 0.43 megabytes of initialized data
- Original load image: 2.19 megabytes
- Resulting load image: 0.97 megabytes
- Size reduction: 44%
- Unpacking overhead: 0.6 seconds (on 100 MHz TM-1)

P2

P2 is a Java interpreter, with a large set of ROMized standard classes.

- Contents: 2.25 megabytes of instructions, 1.60 megabytes of initialized data.
- Original load image: 3.85 megabytes.
- Resulting load image: 1.86 megabytes.
- Size reduction: 48%.
- Unpacking overhead: 1.1 seconds (on 100 MHz TM-1).

tmSEA: Self-Extracting Archives

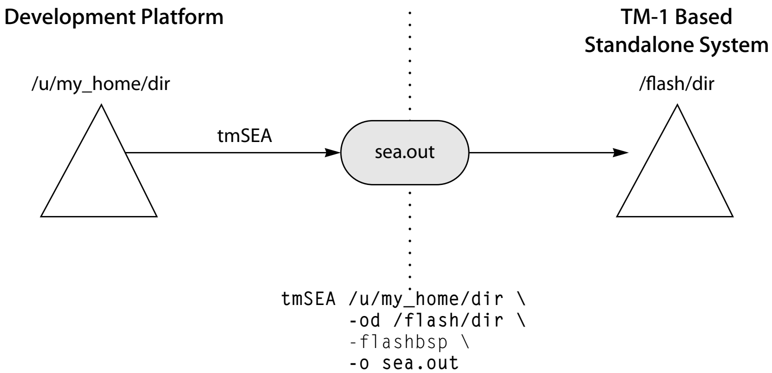


Figure 14 A Self-Extracting Archive

tmSEA is a sample tool that can be used for easy transfer of files from a development platform to a flash file system on a standalone TM-based board. Similarly to **tmSEI**, it embeds the directories in compressed form into an unpacker application, along with directives on where it should be copied into the target file system. This program can be downloaded to the standalone board (via a JTAG connection, for example) and will extract and store its embedded files to the indicated target directory.

In its default form, **tmSEA** links the TriMedia flash file system manager to the generated archive, assuming that the target directory is somewhere on /flash. **tmSEA** uses the BSP as the flash file system hardware interface by default.

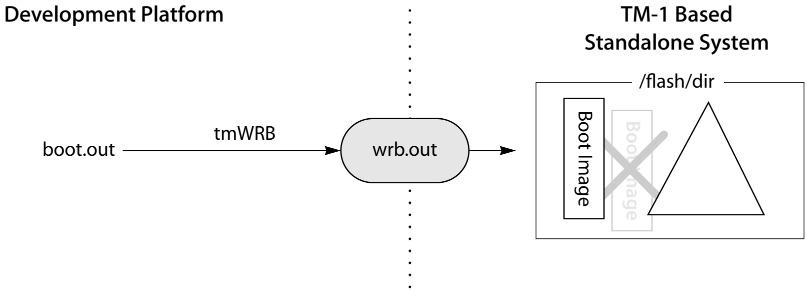
However, **tmSEA** can also be used when the target is a file system other than flash managed by the TriMedia flash file system manager, as long as this file system can be accessed using ANSI and POSIX 1.1 I/O functions. Options **-nostandard** and **-ldflags** can then be used for suppressing the use of the TriMedia flash manager, and for using another file system manager, respectively.

tmSEA supports the following options and arguments:

inputdir	Input directory.
-od *outputdir*	Name of the target directory (default "/flash")
-el | **-eb**	Specify endianness of **tmSEA** output file.
-host *type*	Specify one of the following types for the host processor: **nohost.** No host is the default. **tmsim.** The TriMedia simulator. **Win95.** Windows 95. **MacOS.** Macintosh. **WinNT.** Windows NT.
-nostandard	Disables use of TriMedia flash manager.
-flashbsp	Use BSP as the flash file system hardware interface. This is the default. To disable this option, use the **-noflashbsp** option.
-noflashbsp	Use a flash-specific driver as the flash file system hardware interface, as specified in the makefile.
-v	Specify that the **tmSEA** output file is verbose during unpacking. (Users might want to link a console I/O driver using option **-ldflags**).
-ccflags "*cc_string*"	A string of arguments to pass to **tmcc** when compiling sources for **tmSEA** application.
-ldflags "*ld_string*"	A string of arguments to pass to **tmcc** when linking **tmSEA** application.
-o *outputfile*	Name of **tmSEA** output file that will be created. Default output name is "a.out."

 ©1999 Philips Semiconductors 10/08/99

tmWRB: Boot Image Writing




```
tmcc boot.c -host nohost -o boot.out
tmWRB boot.out -o wrb.out -load 0,0x8000 -flashbsp
```

Figure 15 Boot Image Writing

tmWRB is a sample tool that can be used for easy transfer of a boot executable from a development platform to the boot image of a flash file system on a standalone TM-based board. Similarly to **tmSEI**, it extracts a load image from a specified executable and embeds it in an unpacker application. Again, this program can be downloaded to the standalone board (via a JTAG connection, for example) and will extract and install the embedded image as the boot image in flash memory, thereby overwriting the previous boot image. Updating the boot image in this way is safe, in that it either succeeds, or (in case of power failures or serious flash errors) has no effect. See the discussion in *Flash Manager Properties* on page 226 in Chapter 10.

tmWRB uses the BSP as the flash file system hardware interface by default.

tmWRB supports the following options and arguments:

inputfile	Name of input file. This must be a TriMedia executable object file compiled with **host=nohost**. Because image file size is probably an issue here, it is advisable to have this object file linked with **tmld** compaction options **-bcompact**, **-bfoldcode**, and **-bremoveunusedcode**.
-eb | **-el**	Specify endianness of **tmWRB** output file.
-o *outputfile*	Name of **tmWRB** output file that will be created. Default output name is a.out.
-tmfreq *freq*	Frequency of TriMedia processor. This value is filled into the load image that is extracted from the input executable.
-load *beginMem* , *endMem*	Specify the download memory region of the input file.
-flashbsp	Use BSP as the flash file system hardware interface. This is the default. To disable this option, use the **-noflashbsp** option.

-noflashbsp	Use a flash-specific driver as the flash file system hardware interface, as specified in the makefile.
-mmio_base *base*	MMIO base address of TriMedia processor. This value is filled into the load image that is extracted from the input executable.

Zlib API Data Structures

The **z_stream** data structure is the primary data structure in zlib. All compression and decompression make use of this structure. For definition of all types other than those described here and used by the zlib library please refer to the zconf.h and zlib.h header files.

© 1999 Philips Semiconductors 10/08/99

z_stream

```
typedef struct z_stream_s {
   Bytef                      *next_in;
   uInt                        avail_in;
   uLong                       total_in;
   Bytef                      *next_out;
   uInt                        avail_out;
   uLong                       total_out;
   char                       *msg;
   struct internal_state FAR *state;
   alloc_func                  zalloc;
   free_func                   zfree;
   voidpf                      opaque;
   int                         data_type;
   uLong                       adler;
   uLong                       reserved;
} z_stream;

typedef z_stream FAR *z_streamp;
```

Fields

next_in	Next input byte.
avail_in	Number of bytes available at **next_in**.
total_in	Total number of input bytes read so far.
next_out	Next output byte.
avail_out	Remaining free space at **next_out**.
total_out	Total number of bytes output so far.
msg	Last error message, **Null** if no error.
state	Not visible by applications.
zalloc	Pointer to the function that allocates the internal memory.
zfree	Pointer to a function that frees the internal memory.
opaque	Private data object passed to **zalloc** and **zfree**.
data_type	Best guess about the data type: ASCII or binary.
adler	Adler32 value of the uncompressed data.
reserved	Reserved for future use.

Description

The application must update **next_in** and **avail_in** when **avail_in** has dropped to zero. It must update **next_out** and **avail_out** when **avail_out** has dropped to zero. The application must initialize **zalloc**, **zfree** and **opaque** before calling the initialization function. All other fields are set by the compression library and must not be updated by the application.

The **opaque** value provided by the application will be passed as the first parameter of calls to **zalloc** and **zfree**. This can be useful for custom memory management. The compression library attaches no meaning to the **opaque** value.

zalloc must return **Z_NULL** if there is not enough memory for the object. If zlib is used in a multi-threaded application, **zalloc** and **zfree** must be thread safe.

The fields **total_in** and **total_out** can be used for statistics or progress reports. After compression, **total_in** holds the total size of the uncompressed data and may be saved for use in the decompressor (particularly if the decompressor wants to decompress everything in a single step).

 ©1999 Philips Semiconductors 10/08/99

Zlib API Functions

The following tables list the functions available to applications that use the TCS zlib library.

Basic Functions	Page
zlibVersion	257
deflateInit	258
deflate	259
deflateEnd	261
inflateInit	262
inflate	263
inflateEnd	265

High-Level Functions	Page
compress	267
compress2	268
uncompress	269

Advanced Functions	Page
deflateInit2	271
deflateSetDictionary	273
deflateCopy	275
deflateReset	276
inflateInit2	278
inflateSetDictionary	279
inflateSync	280
inflateReset	281

File Utility Functions	Page
gzopen	282
gzdopen	283
gzsetparams	284
gzread	285
gzwrite	286

Basic Compression and Decompression Functions

This section presents the functions used for basic (low-level) compression and decompression.

zlibVersion

```
const char *zlibVersion();
```

Parameters

None.

Return

The function returns a null-terminated string with the current version of zlib, for example "1.1.3."

Description

The application can compare the value of zlibVersion and **ZLIB_VERSION** for consistency. If the first character differs, the library code actually used is not compatible with the zlib.h header file used by the application. This check is automatically made by **deflateInit** and **inflateInit**.

deflateInit

```
int deflateInit(
   z_streamp  stream,
   int        level
);
```

Parameters

`stream`	Stream state reference that can be used for compression.
`level`	Compression level. See the description below for compression level options.

Return Codes

`Z_OK`	Success.
`Z_MEM_ERROR`	There was not enough memory.
`Z_STREAM_ERROR`	Level is not a valid compression level.
`Z_VERSION_ERROR`	The zlib library version (zlib_version) is incompatible with the version assumed by the caller (**ZLIB_VERSION**).

Description

Initializes the internal stream state for compression. The fields zalloc, zfree and opaque must be initialized before by the caller. If **zalloc** and **zfree** are set to **Z_NULL**, deflateInit updates them to use default allocation functions.

The compression level must be a value from 0 to 9. Level 1 gives the best speed, level 9 gives the best compression, and level 0 gives no compression at all (the input data is simply copied a block at a time). The term **Z_DEFAULT_COMPRESSION** requests a default compromise between speed and compression (currently equivalent to level 6).

msg is set to null if there is no error message. deflateInit does not perform any compression. Compression is performed by **deflate**.

 © 1999 Philips Semiconductors 10/08/99

deflate

```
int deflate (
   z_streamp  stream,
   int        flush
);
```

Parameters

`stream`	Stream state reference for compression returned by deflateInit.
`flush`	Valid flush value. See the description below for details.

Return Codes

`Z_OK`	Some progress has been made (more input processed or more output produced).
`Z_STREAM_END`	If all input has been consumed and all output has been produced (only when flush is set to **Z_FINISH**).
Z_STREAM_ERROR	If the stream state was inconsistent (for example if **next_in** or **next_out** was **NULL**).
Z_BUF_ERROR	If no progress is possible (for example **avail_in** or **avail_out** are zero).

Description

The function compresses as much data as possible, and stops when the input buffer becomes empty or the output buffer becomes full. It may introduce some output latency (reading input without producing any output) except when forced to flush.

The function performs one or both of the following actions:

- Compress more input starting at **next_in** and update **next_in** and **avail_in** accordingly. If not all input can be processed (because there is not enough room in the output buffer), **next_in** and **avail_in** are updated and processing will resume at this point for the next call to deflate().
- Provide more output starting at **next_out** and update **next_out** and **avail_out** accordingly. This action is forced if the parameter flush is non-zero. Forcing flush frequently degrades the compression ratio, so this parameter should be set only when necessary (in interactive applications). Some output may be provided even if flush is not set.

Before the call to **deflate**, the application should ensure that at least one of the actions is possible, by providing more input and/or consuming more output, and updating **avail_in** or **avail_out** accordingly; **avail_out** should never be zero before the call. The application can consume the compressed output when it wants, for example, when the output buffer is full (**avail_out==0**), or after each call of deflate(). If deflate returns **Z_OK** and with zero **avail_out**, it must be called again after making room in the output buffer because there might be more output pending.

If the parameter flush is set to **Z_SYNC_FLUSH**, all pending output is flushed to the output buffer and the output is aligned on a byte boundary, so that the decompressor can get all input data available so far. (In particular, **avail_in** is zero after the call if enough output space has been provided before the call.) Flushing may degrade compression for some compression algorithms and so it should be used only when necessary.

If flush is set to **Z_FULL_FLUSH**, all output is flushed as with **Z_SYNC_FLUSH**, and the compression state is reset so that decompression can restart from this point if previous compressed data has been damaged or if random access is desired. Using **Z_FULL_FLUSH** too often can seriously degrade the compression.

If deflate returns with **avail_out == 0**, this function must be called again with the same value of the flush parameter and more output space (updated **avail_out**), until the flush is complete (deflate returns with non-zero **avail_out**).

If the parameter flush is set to **Z_FINISH**, pending input is processed, pending output is flushed and deflate returns with **Z_STREAM_END** if there was enough output space; if deflate returns with **Z_OK**, this function must be called again with **Z_FINISH** and more output space (updated avail_out) but no more input data, until it returns with **Z_STREAM_END** or an error. After deflate has returned **Z_STREAM_END**, the only possible operations on the stream are deflateReset or deflateEnd. **Z_FINISH** can be used immediately after deflateInit if all the compression is to be done in a single step. In this case, **avail_out** must be at least 0.1% larger than **avail_in** plus 12 bytes. If deflate does not return **Z_STREAM_END**, then it must be called again as described above. deflate() sets **stream->adler** to the adler32 checksum of all input read so far (that is, **total_in** bytes).

deflate() may update **data_type** if it can make a good guess about the input data type (**Z_ASCII** or **Z_BINARY**). In doubt, the data is considered binary. This field is only for information purposes and does not affect the compression algorithm in any manner.

 © 1999 Philips Semiconductors 10/08/99

deflateEnd

```
int deflateEnd (
   z_streamp  stream
);
```

Parameters

`stream`	Stream state reference used for compression returned by deflateInit.

Return

`Z_OK`	Success.
`Z_STREAM_ERROR`	The stream state was inconsistent.
`Z_DATA_ERROR`	The stream was freed prematurely (some input or output was discarded).

Description

All dynamically allocated data structures for this stream are freed. This function discards any unprocessed input and does not flush any pending output.

In the error case, **msg** may be set but then points to a static string (which must not be deallocated).

inflateInit

```
int inflateInit (
   z_streamp  stream
);
```

Parameters

`stream`	Stream state reference that can be used for decompression.

Return

`Z_OK`	Success.
`Z_MEM_ERROR`	There was not enough memory.
`Z_VERSION_ERROR`	The zlib library version is incompatible with the version assumed by the caller.

Description

Initializes the internal stream state for decompression. The fields next_in, avail_in, **zalloc**, **zfree** and **opaque** must be initialized before by the caller. If **next_in** is not **Z_NULL** and **avail_in** is large enough (the exact value depends on the compression method), inflateInit() determines the compression method from the zlib header and allocates all data structures accordingly. Otherwise, the allocation will be deferred to the first call toinflate. If **zalloc** and **zfree** are set to **Z_NULL**, **inflateInit** updates them to use default allocation functions.

msg is set to null if there is no error message. **inflateInit** does not perform any decompression apart from reading the zlib header if present: this will be done by **inflate**. (So **next_in** and **avail_in** may be modified, but **next_out** and **avail_out** are unchanged.)

 © 1999 Philips Semiconductors 10/08/99

inflate

```
int inflate (
   z_streamp  stream,
   int        flush
);
```

Parameters

`stream`	Stream state reference for decompression returned by inflateInit.
`flush`	Valid flush value. See the description below for details.

Return

`Z_OK`	Some progress has been made (more input processed or more output produced).
`Z_STREAM_END`	The end of the compressed data has been reached and all uncompressed output has been produced.
`Z_NEED_DICT`	A preset dictionary is needed at this point.
`Z_DATA_ERROR`	The input data was corrupted (input stream not conforming to the zlib format or incorrect adler32 checksum).The application may then call inflateSync to look for a good compression block.
`Z_STREAM_ERROR`	The stream structure was inconsistent (for example if **next_in** or **next_out** was **NULL**).
Z_MEM_ERROR	There was not enough memory.
Z_BUF_ERROR	No progress is possible or there was not enough room in the output buffer when **Z_FINISH** is used.

Description

The function decompresses as much data as possible, and stops when the input buffer becomes empty or the output buffer becomes full. It may some introduce some output latency (reading input without producing any output) except when forced to flush.

The function performs one or both of the following actions:

- Decompress more input starting at **next_in** and update **next_in** and **avail_in** accordingly. If not all input can be processed (because there is not enough room in the output buffer), **next_in** is updated and processing will resume at this point for the next call of **inflate**.
- Provide more output starting at **next_out** and update **next_out** and **avail_out** accordingly. **inflate** provides as much output as possible, until there is no more input data or no more space in the output buffer (see below about the flush parameter).

Before the call to inflate(), the application should ensure that at least one of the actions is possible, by providing more input and/or consuming more output, and updating the **next_*** and **avail_*** values accordingly. The application can consume the uncompressed output when it wants, for example, when the output buffer is full (**avail_out==0**), or after each call to inflate(). If inflate returns **Z_OK** and with zero **avail_out**, it must be called again after making room in the output buffer because there might be more output pending.

If the parameter flush is set to **Z_SYNC_FLUSH**, inflate flushes as much output as possible to the output buffer. The flushing behavior of inflate is not specified for values of the flush parameter other than **Z_SYNC_FLUSH** and **Z_FINISH**, but the current implementation actually flushes as much output as possible anyway.

inflate() should normally be called until it returns **Z_STREAM_END** or an error. However if all decompression is to be performed in a single step (a single call of inflate), the parameter flush should be set to **Z_FINISH**. In this case all pending input is processed and all pending output is flushed; avail_out must be large enough to hold all the uncompressed data. (The size of the uncompressed data may have been saved by the compressor for this purpose.) The next operation on this stream must be inflateEnd to deallocate the decompression state. The use of **Z_FINISH** is never required, but can be used to inform inflate that a faster routine may be used for the single inflate() call.

If a preset dictionary is needed at this point (see inflateSetDictionary below), inflate sets **stream->adler** to the adler32 checksum of the dictionary chosen by the compressor and returns **Z_NEED_DICT**; otherwise it sets **stream->adler** to the adler32 checksum of all output produced so far (that is, **total_out** bytes) and returns **Z_OK**, **Z_STREAM_END** or an error code as described below. At the end of the stream, **inflate** checks that its computed adler32 checksum is equal to that saved by the compressor and returns **Z_STREAM_END** only if the checksum is correct.

 © 1999 Philips Semiconductors 10/08/99

inflateEnd

```
int inflateEnd (
   z_streamp  stream
);
```

Parameters

`stream`	Stream state reference used for decompression returned by **inflateInit**.

Return Codes

`Z_OK`	Success.
`Z_STREAM_ERROR`	The stream state was inconsistent.

Description

All dynamically allocated data structures for this stream are freed. This function discards any unprocessed input and does not flush any pending output.

In the error case, **msg** may be set but then points to a static string (which must not be deallocated).

High-Level Compression and Decompression Functions

The following utility functions are implemented on top of the basic stream-oriented functions. To simplify the interface, certain default options are assumed (compression level and memory usage, standard memory allocation functions). The source code of these utility functions can easily be modified if you need special options.

 © 1999 Philips Semiconductors 10/08/99

compress

```
int compress (
   Bytef       *dest,
   uLongf      *destLen,
   const Bytef *source,
   uLong        sourceLen
);
```

Parameters

dest	Destination buffer.
destLen	Length of destination buffer.
source	Source buffer.
sourceLen	Length of source buffer.

Return Codes

Z_OK	Success.
Z_MEM_ERROR	There was not enough memory.
Z_BUF_ERROR	There was not enough room in the output buffer.

Description

Compresses the source buffer into the destination buffer. **sourceLen** is the byte length of the source buffer. Upon entry, **destLen** is the total size of the destination buffer, which must be at least 0.1% larger than sourceLen plus 12 bytes. Upon exit, **destLen** is the actual size of the compressed buffer.

This function can be used to compress a whole file at once if the input file is mmap'd.

compress2

```
int compress2 (
   Bytef       *dest,
   uLongf      *destLen,
   const Bytef *source,
   uLong        sourceLen,
   int          level
));
```

Parameters

`dest`	Destination buffer.
`destLen`	Length of destination buffer.
`source`	Source buffer.
`sourceLen`	Length of source buffer.
`level`	Compression level.

Return

`Z_OK`	Success.
`Z_MEM_ERROR`	There was not enough memory.
`Z_BUF_ERROR`	There was not enough room in the output buffer.
`Z_STREAM_ERROR`	The level parameter is invalid.

Description

Compresses the source buffer into the destination buffer. The **level** parameter has the same meaning as in `deflateInit`. **sourceLen** is the byte length of the source buffer. Upon entry, **destLen** is the total size of the destination buffer, which must be at least 0.1% larger than **sourceLen** plus 12 bytes. Upon exit, **destLen** is the actual size of the compressed buffer.

 © 1999 Philips Semiconductors 10/08/99

uncompress

```
int uncompress (
   Bytef       *dest,
   uLongf      *destLen,
   const Bytef *source,
   uLong        sourceLen
);
```

Parameters

`dest`	Destination buffer.
`destLen`	Length of destination buffer.
`source`	Source buffer.
`sourceLen`	Length of source buffer.

Return Codes

`Z_OK`	Success.
`Z_MEM_ERROR`	There was not enough memory.
`Z_BUF_ERROR`	There was not enough room in the output buffer.
`Z_DATA_ERROR`	The input data was corrupted.

Description

Decompresses the source buffer into the destination buffer. **sourceLen** is the byte length of the source buffer. Upon entry, **destLen** is the total size of the destination buffer, which must be large enough to hold the entire uncompressed data. (The size of the uncompressed data must have been saved previously by the compressor and transmitted to the decompressor by some mechanism outside the scope of this compression library.) Upon exit, **destLen** is the actual size of the compressed buffer. This function can be used to decompress a whole file at once if the input file is mmap'ed.

Advanced Functions

The following functions are needed only in some special applications.

 © 1999 Philips Semiconductors 10/08/99

deflateInit2

```
int deflateInit2 (
   z_streamp stream,
   int       level,
   int       method,
   int       windowBits,
   int       memLevel,
   int       strategy
);
```

Parameters

`stream`	Stream state reference used for decompression, returned by inflateInit.
`level`	Compression level.
`method`	Compression method. It must be **Z_DEFLATED** in this version of the library.
`windowBits`	The base 2 logarithm of the window size (the size of the history buffer).
`memLevel`	Specifies how much memory should be allocated for the internal compression state.
`strategy`	Specifies the compression algorithm.

Return Codes

`Z_OK`	Success.
`Z_MEM_ERROR`	There was not enough memory.
`Z_STREAM_ERROR`	A parameter is invalid (such as an invalid method).

Description

This is another version of deflateInit with more compression options. The fields **next_in**, **zalloc**, **zfree** and **opaque** must be initialized before by the caller.

The compression level must be a value from 0 to 9. Level 1 gives best speed, level 9 gives best compression, and level 0 gives no compression at all (the input data is simply copied a block at a time). The term **Z_DEFAULT_COMPRESSION** requests a default compromise between speed and compression (currently equivalent to level 6).

The **windowBits** parameter is the base 2 logarithm of the window size (the size of the history buffer). It should be in the range 8 to 15 for this version of the library. Larger values of this parameter result in better compression at the expense of memory usage. The default value is 15 if **deflateInit** is used instead.

The **memLevel** parameter specifies how much memory should be allocated for the internal compression state.

memLevel=1	uses minimum memory but is slow and reduces compression ratio
memLevel=9	uses maximum memory for optimal speed.

The default value is 8. See zconf.h for total memory usage as a function of **windowBits** and **memLevel**.

The **strategy** parameter tunes the compression algorithm. Use the value **Z_DEFAULT_STRATEGY** for normal data, **Z_FILTERED** for data produced by a filter (or predictor), or **Z_HUFFMAN_ONLY** to force Huffman encoding only (no string match). Filtered data consists mostly of small values with a somewhat random distribution. In this case, the compression algorithm is tuned to compress them better. The effect of **Z_FILTERED** is to force more Huffman coding and less string matching; it is somewhat intermediate between **Z_DEFAULT** and **Z_HUFFMAN_ONLY**. The **strategy** parameter affects only the compression ratio but not the correctness of the compressed output (even if it is not set appropriately).

msg is set to **NULL** if there is no error message. **deflateInit2** does not perform any compression. Compression is performed by **deflate**.

 ©1999 Philips Semiconductors 10/08/99

deflateSetDictionary

```
int deflateSetDictionary (
   z_streamp    stream,
   const Bytef *dictionary,
   uInt         dictLength
);
```

Parameters

`stream`	Stream state reference used for compression, returned by deflateInit.
`dictionary`	A byte sequence that consist of strings that are likely to be encountered later in the data to be compressed.
`dictLength`	Length of the dictionary byte sequence.

Return Codes

`Z_OK`	Success.
`Z_STREAM_ERROR`	A parameter is invalid (such as **NULL** dictionary) or the stream state is inconsistent (for example if deflate has already been called for this stream or if the compression method is bsort).

Description

Initializes the compression dictionary from the given byte sequence without producing any compressed output. This function must be called immediately after **deflateInit**, **deflateInit2** or **deflateReset**, before any call of deflate. The compressor and decompressor must use exactly the same dictionary (see **inflateSetDictionary**).

The dictionary should consist of strings (byte sequences) that are likely to be encountered later in the data to be compressed, with the most commonly used strings preferably put towards the end of the dictionary. Using a dictionary is most useful when the data to be compressed is short and can be predicted with good accuracy; the data can then be compressed better than with the default empty dictionary.

Depending on the size of the compression data structures selected by **deflateInit** or **deflateInit2**, a part of the dictionary may in effect be discarded, for example if the dictionary is larger than the window size in **deflate** or **deflate2**. Thus the strings most likely to be useful should be put at the end of the dictionary, not at the front.

Upon return of this function, **stream–>adler** is set to the Adler32 value of the dictionary; the decompressor may later use this value to determine which dictionary has been used by the compressor. (The Adler32 value applies to the whole dictionary even if only a subset of the dictionary is actually used by the compressor.)

deflateSetDictionary does not perform any compression. Compression is performed by **deflate**.

 © 1999 Philips Semiconductors 10/08/99

deflateCopy

```
int deflateCopy (
   z_streamp  dest,
   z_streamp  source
);
```

Parameters

dest	Destination stream.
source	Source stream.

Return Codes

Z_OK	Success.
Z_MEM_ERROR	There was not enough memory.
Z_STREAM_ERROR	The source stream state was inconsistent (such as `zalloc` being **NULL**).

Description

Sets the destination stream as a complete copy of the source stream.

This function can be useful when several compression strategies will be tried, for example when there are several ways of pre-processing the input data with a filter. The streams that will be discarded should then be freed by calling **deflateEnd**. Note that **deflateCopy** duplicates the internal compression state which can be quite large, so this strategy is slow and can consume much memory.

msg is left unchanged in both source and destination.

deflateReset

```
int deflateReset (
   z_streamp stream
);
```

Parameters

`stream`	Stream state reference used for compression, returned by deflateInit.

Return

`Z_OK`	Success.
`Z_STREAM_ERROR`	The source stream state was inconsistent (such as **zalloc** or state being **NULL**).

Description

This function is equivalent to **deflateEnd** followed by **deflateInit**, but does not free and reallocate all the internal compression state. The stream will keep the same compression level and any other attributes that may have been set by **deflateInit2**.

 © 1999 Philips Semiconductors 10/08/99

deflateParams

```
int deflateParams (
   z_streamp  stream,
   int        level,
   int        strategy
);
```

Parameters

`stream`	Stream state reference used for compression returned by deflateInit.
`level`	New compression level.
`strategy`	New compression strategy.

Return

`Z_OK`	Success.
`Z_STREAM_ERROR`	The source stream state was inconsistent or if a parameter was invalid.
`Z_BUF_ERROR`	The field **stream–>avail_out** was zero.

Description

Dynamically update the compression level and compression strategy. The interpretation of level and strategy is as in deflateInit2. This can be used to switch between compression and straight copy of the input data, or to switch to a different kind of input data requiring a different strategy. If the compression level is changed, the input available so far is compressed with the old level (and may be flushed); the new level will take effect only at the next call of **deflate**.

Before the call of deflateParams, the stream state must be set as for a call of **deflate**, since the currently available input may have to be compressed and flushed. In particular, **stream–>avail_out** must be non-zero.

inflateInit2

```
int inflateInit2 (
   z_streamp  stream,
   int        windowBits
);
```

Parameters

`stream`	Stream state reference that can be used for decompression.
`windowBits`	The base 2 logarithm of the maximum window size (the size of the history buffer).

Return Codes

`Z_OK`	Success.
`Z_MEM_ERROR`	There was not enough memory.
`Z_STREAM_ERROR`	A parameter is invalid (e.g., a negative **memLevel**).

Description

This is another version of inflateInit with an extra parameter. The fields **next_in**, **avail_in**, **zalloc**, **zfree** and **opaque** must be initialized before by the caller.

The **windowBits** parameter is the base 2 logarithm of the maximum window size (the size of the history buffer). It should be in the range 8 to 15 for this version of the library. The default value is 15 if inflateInit is used instead. If a compressed stream with a larger window size is given as input, inflate() will return with the error code **Z_DATA_ERROR** instead of trying to allocate a larger window.

msg is set to **NULL** if there is no error message. inflateInit2 does not perform any decompression apart from reading the zlib header if present: this will be done by **inflate**. (So **next_in** and **avail_in** may be modified, but **next_out** and **avail_out** are unchanged.)

 © 1999 Philips Semiconductors 10/08/99

inflateSetDictionary

```
int inflateSetDictionary (
   z_streamp    stream,
   const Bytef *dictionary,
   uInt         dictLength
);
```

Parameters

`stream`	Stream state reference used for decompression, returned by inflateInit.
`dictionary`	A byte sequence that consist of strings that are likely to be encountered later in the data to be decompressed.
`dictLength`	Length of the dictionary byte sequence.

Return Codes

`Z_OK`	Success.
`Z_STREAM_ERROR`	A parameter is invalid (e.g., a **NULL** dictionary) or the stream state is inconsistent.
`Z_DATA_ERROR`	The given dictionary doesn't match the expected one (incorrect Adler32 value).

Description

Initializes the decompression dictionary from the given uncompressed byte sequence. This function must be called immediately after a call to **inflate** if this call returned **Z_NEED_DICT**. The dictionary chosen by the compressor can be determined from the Adler32 value returned by this call to **inflate**. The compressor and decompressor must use exactly the same dictionary (see **deflateSetDictionary**).

inflateSetDictionary does not perform any decompression. Decompression is performed by subsequent calls to **inflate**.

inflateSync

```
int inflateSync (
   z_streamp stream
);
```

Parameters

stream	Stream state reference used for decompression, returned by inflateInit.

Return Codes

Z_OK	A full flush point has been found.
Z_BUF_ERROR	No more input was provided.
Z_DATA_ERROR	No flush point has been found.
Z_STREAM_ERROR	The stream structure was inconsistent.

Description

Skips invalid compressed data until a full flush point (see above the description of deflate with **Z_FULL_FLUSH**) can be found, or until all available input is skipped. No output is provided.

In the success case, the application may save the current value of **total_in** which indicates where valid compressed data was found. In the error case, the application may repeatedly call **inflateSync**, providing more input each time, until success or end of the input data.

 © 1999 Philips Semiconductors 10/08/99

inflateReset

```
int inflateReset (
   z_streamp stream
);
```

Parameters

`stream`	Stream state reference used for decompression, returned by inflateInit.

Return

`Z_OK`	Success.
`Z_STREAM_ERROR`	The source stream state was inconsistent (such as **zalloc** or state being **NULL**).

Description

This function is equivalent to **inflateEnd** followed by inflateInit, but does not free and reallocate all the internal decompression state. The stream will keep attributes that may have been set by **inflateInit2**.

File Utility Functions

The following functions provide the support for reading and writing files in gzip (.gz) format. The interface is similar to that of stdio.

gzopen

```
gzFile gzopen (
   const char *path,
   const char *mode
);
```

Parameters

path	Path of file to open.
mode	Mode in which the file should be opened.

Return

On success, the function returns a gzFile reference.

The function returns **NULL** if the file could not be opened or if there was insufficient memory to allocate the (de)compression state; **errno** can be checked to distinguish the two cases (if **errno==0**, the zlib error is **Z_MEM_ERROR**).

Description

Opens a gzip (.gz) file for reading or writing. The mode parameter is as in fopen (**"rb"** or **"wb"**) but can also include a compression level (**"wb9"**) or a strategy: **'f'** for filtered data as in **"wb6f"**, **'h'** for Huffman only compression as in **"wb1h"**. (See the description of deflateInit2 for more information about the strategy parameter.)

The function can read a file which is not in gzip format. In such a case, **gzread** will directly read from the file without decompression.

 © 1999 Philips Semiconductors 10/08/99

gzdopen

```
gzFile gzdopen (
   int         fd,
   const char *mode
));
```

Parameters

`fd`	gzFile descriptor.
`mode`	Mode in which the file should be opened.

Return Codes

On success, the function returns a gzFile reference to the opened file.

The function returns **NULL** if there was insufficient memory to allocate the (de)compression state.

Description

The function associates a gzFile with the file descriptor **fd**. File descriptors are obtained from calls like open, dup, creat, pipe or fileno (if the file has been previously opened with fopen).

The next call to gzclose applied to the returned gzFile will also close the file descriptor **fd**, just as **fclose(fdopen(fd), mode)** closes the file descriptor **fd**. If you want to keep **fd** open, use **gzdopen(dup(fd), mode)**.

gzsetparams

```
int gzsetparams (
   gzFile  file,
   int     level,
   int     strategy
));
```

Parameters

file	A gzFile reference to an open file.
level	Compression level.
strategy	Compression strategy.

Return Codes

Z_OK	Success.
Z_STREAM_ERROR	The file was not opened for writing.

Description

Dynamically update the compression level or strategy. See the description of **deflateInit2** for the meaning of these parameters.

 © 1999 Philips Semiconductors 10/08/99

gzread

```
int gzread (
   gzFile    file,
   voidp     buf,
   unsigned  len
);
```

Parameters

`file`	A gzFile reference to an open file.
`buf`	Buffer in which to put read data.
`len`	Length of buffer.

Return

The function returns the number of uncompressed bytes actually read (0 for end of file, and –1 for error).

Description

Reads the given number of uncompressed bytes from the compressed file. If the input file was not in gzip format, **gzread** copies the given number of bytes into the buffer.

gzwrite

```
int gzwrite (
   gzFile      file,
   const voidp  buf,
   unsigned    len
);
```

Parameters

`file`	A gzFile reference to an open file.
`buf`	Buffer from which to take data.
`len`	Length of buffer.

Return

The function returns the number of uncompressed bytes actually written (0 in case of error).

Description

Writes the given number of uncompressed bytes into the compressed file.

 © 1999 Philips Semiconductors 10/08/99

gzprintf

```
int gzprintf (
   gzFile      file,
   const char *format,
   ...
);
```

Parameters

`file`	A gzFile reference to an open file.
`format`	Format string as in **fprintf**.

Return Codes

The function returns the number of uncompressed bytes actually written (0 in case of error).

Description

Converts, formats, and writes the args to the compressed file under control of the format string, as occurs in **fprintf**.

gzputs

```
int gzputs (
   gzFile     file,
   const char *s
);
```

Parameters

`file`	A gzFile reference to an open file.
`s`	Null-terminated string to be written.

Return Codes

The function returns the number of characters written, or –1 in case of error.

Description

Writes the given null-terminated string to the compressed file, excluding the terminating null character.

 ©1999 Philips Semiconductors 10/08/99

gzgets

```
char *gzgets (
   gzFile  file,
   char   *buf,
   int     len
);
```

Parameters

`file`	A gzFile reference to an open file.
`buf`	Buffer from which to read.
`len`	Maximum length of the string to be read, including null termination.

Return

The function returns **buf**, or **Z_NULL** in case of error.

Description

Reads bytes from the compressed file until **len**–1 characters are read, or a newline character is read and transferred to **buf**, or an end-of-file condition is encountered. The string is then terminated with a null character.

gzputc

```
int gzputc (
   gzFile file,
   int ch
);
```

Parameters

`file`	A gzFile reference to an open file.
`ch`	Character to be written to the file.

Return Codes

The function returns the value that was written, or –1 in case of error.

Description

Writes **ch** (as an **unsigned char**) into the compressed file.

gzgetc

```
int gzgetc (
   gzFile  file
);
```

Parameters

`file`	A gzFile reference to an open file.

Return

The function returns the byte read, or –1 in case of end-of-file or error.

Description

The function reads one byte from the compressed file.

 © 1999 Philips Semiconductors 10/08/99

gzflush

```
int gzflush (
   gzFile  file,
   int     flush
);
```

Parameters

`file`	A gzFile reference to an open file.
`flush`	Valid flush value. See the description of **deflate** for details.

Return Codes

The function returns **Z_OK** if the flush parameter is **Z_FINISH** and all output could be flushed. The return value is the zlib error number. (See function **gzerror** on page 295.)

Description

Flushes all pending output into the compressed file. The function should be called only when strictly necessary because it can degrade compression.

gzseek

```
z_off_t gzseek (
   gzFile   file,
   z_off_t  offset,
   int      whence
);
```

Parameters

`file`	A gzFile reference to an open file.
`offset`	Represents a number of bytes in the uncompressed data stream.
`whence`	Defined as in **lseek(2)**;

Return Codes

The function returns the resulting offset location, as measured in bytes from the beginning of the uncompressed stream. In the case of error, the function returns –1, particularly when the file is open for writing and the new starting position would be before the current position.

Description

Sets the starting position for the next gzread or gzwrite on the given compressed file. The **whence** parameter is defined as in **lseek(2)**; the value **SEEK_END** is not supported. If the file is open for reading, this function is emulated but can be extremely slow. If the file is open for writing, only forward seeks are supported; gzseek then compresses a sequence of zeroes up to the new starting position.

 © 1999 Philips Semiconductors 10/08/99

gzrewind

```
int gzrewind (
   gzFile file
);
```

Parameters

`file`	A gzFile reference to an open file.

Return

The function returns the resulting offset location, as measured in bytes from the beginning of the uncompressed stream. The function returns 0 on success or –1 in case of error, particularly when the file is open for writing and the new starting position would be before the current position.

Description

Rewinds the given file. This function is supported only for reading. **gzrewind(file)** is equivalent to **(int)gzseek(file,0L,SEEK_SET)**.

©1999 Philips Semiconductors 10/08/99

gztell

```
z_off_t gztell (
   gzFile file
);
```

Parameters

`file`	A gzFile reference to an open file.

Return Codes

Returns the starting position for the next **gzread** or **gzwrite** on the given compressed file.

Description

The returned position represents a number of bytes in the uncompressed data stream. **gztell(file)** is equivalent to **gzseek(file, 0L, SEEK_CUR)**.

gzeof

```
int gzeof (
   gzFile file
);
```

Parameters

`file`	A gzFile reference to an open file.

Return

The function returns 1 when end-of-file has previously been detected reading the given input stream. Otherwise, the function returns zero.

 © 1999 Philips Semiconductors 10/08/99

gzclose

```
int gzclose (
   gzFile file
);
```

Parameters

`file`	A gzFile reference to an open file.

Return Codes

The function returns the zlib error number (see function **gzerror** below).

Description

The function flushes all pending output if necessary, closes the compressed file and deallocates the entire (de)compression state.

gzerror

```
const char *gzerror (
   gzFile  file,
   int    *errnum
);
```

Parameters

`file`	A gzFile reference to an open file.
`errnum`	Address at which the zlib error number can be written.

Return Codes

The function returns the error message for the last error that occurred on the given compressed file.

Description

The function sets **errnum** to the zlib error number. If an error occurred in the file system and not in the compression library, **errnum** is set to **Z_ERRNO** and the application may consult **errno** to get the exact error code.

Checksum Functions

These functions are not related to compression but are exported anyway because they might be useful in applications using the compression library.

 © 1999 Philips Semiconductors 10/08/99

adler32

```
uLong adler32 (
   uLong         adler,
   const Bytef  *buf,
   uInt          len
);
```

Parameters

adler	Previous Adler-32 checksum.
`buf`	Buffer for which to calculate Adler-32 checksum.
`len`	Length of buffer.

Return Codes

If **buf** is **NULL**, the function returns the required initial value for the checksum. Otherwise this function returns a new Adler-32 checksum.

Description

Update a running Adler-32 checksum with the bytes in **buf** (the first **len** bytes) and return the updated checksum. If buf is **NULL**, the function returns the required initial value for the checksum.

An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much faster.

Example

```
uLong adler = adler32( 0L, Z_NULL, 0 );
while( read_buffer(buffer, length) != EOF ){
   adler = adler32( adler, buffer, length );
}
if( adler != original_adler ) error();
```

©1999 Philips Semiconductors 10/08/99

crc32

```
uLong crc32 (
   uLong       crc,
   const Bytef *buf,
   uInt        len
);
```

Parameters

`crc`	Previous CRC.
`buf`	Buffer for which to calculate the CRC.
`len`	Length of buffer.

Return

If **buf** is **NULL**, the function returns the required initial value for the checksum. Otherwise this function returns a new CRC.

Description

The function updates a running CRC from the bytes in **buf** (the first **len** bytes) and returns the updated CRC. If **buf** is **NULL**, the function returns the required initial value for the CRC. Pre- and post-conditioning (one's complement) is performed within this function; your application shouldn't do it.

Example

```
uLong crc = crc32( 0L, Z_NULL, 0 );
while( read_buffer(buffer,length) != EOF ){
   crc = crc32( crc, buffer, length );
}
if( crc != original_crc ) error();
```

 ©1999 Philips Semiconductors 10/08/99

Chapter 12

Downloader API

Downloader Library

The TriMedia downloader library is intended for extracting a relocated load image from an executable object file. This functionality is used by several of the TriMedia SDE tools, and is also available in the form of a library *libload.a* which has been compiled for a number of platforms. By this, instead of using **tmmon** or **tmrun**, users are able to use the downloader within their own application for loading and starting TriMedia programs. The downloader library can even be used from the TriMedia processor itself; an example that uses the downloader library for standalone booting is presented at the end of this section.

The following table shows where in the SDE the header file and the different libraries can be found:

Location in SDE	File Name	Description
$TCS/include/tmlib	TMDownLoader.h	header file
$TCS/lib/el	libload.a	library, TriMedia version, little endian
$TCS/lib/eb	libload.a	library, TriMedia version, big endian
$TCS/lib/<platform>	libload.a	library, for platforms Win95, MacOS, SunOS and HP-UX

The following commands show how to build an application that uses the downloader library for the TriMedia-based or the SunOS-based case, respectively:

```
tmcc main.c -lload
acc  main.c -I$TCS/include -L$TCS/lib/SunOS -lload
```

The downloader is robust, in the sense that it detects exceptional situations like memory overflow, and translates them into appropriate error codes. It is also efficient, in that it only reads those parts of the downloaded object file that are strictly necessary for downloading. For instance, unless it has been stripped, a large part of an object file generally consists of symbol and debug information; none of this information will be touched by the downloader. The downloader can be used for generating load images that use shared memory on a multiprocessor cluster. Finally, the downloader can be instructed to read from different types of object file sources such as a file or a consecutive memory area (for example, EEPROM). The memory needed for downloading can roughly be estimated by the total executable size reported by **tmsize**, plus 20 kb.

 © 1999 Philips Semiconductors 10/08/99

Downloader API Description

This interface provides the typical functions which are needed by a TM-1 downloader. Downloading here is defined as the process of getting a bootable executable on a TM-1 in reset state. This is in contrast to "dynamic loading," in which case the TM-1 itself loads an executable or library.

Examples of Downloader Use

In most situations, the downloader library is used by a monitor program or execution shell to place a relocated image of an executable object file into the SDRAM of an idle TriMedia processor. The processor is then booted by releasing it from its RESET state; this causes it to start executing instructions from the start of SDRAM (where the image has been loaded). Examples of TriMedia tools which use the downloader library in this way are **tmmon**, **tmgmon**, **tmrun**, and **tmmprun**.

Apart from use by this monitor- type of application running on a host, the downloader library can also be used in several other situations:

- The downloader library can be used in a tool that stores the load image in a file for later use, for later copying into SDRAM, for example, or for burning into EEPROM. An example of such a tool is **tmld**, when it is used with option **-mi**.
- The downloader library can be used by a resident monitor program running on a standalone TriMedia board. This monitor's load image is copied from EEPROM to the beginning of SDRAM when the processor is booted.Upon commands from a terminal, the monitor may itself download and relocate executables from a parallel port, or from flash memory. Because the monitor program already occupies memory at the beginning of SDRAM, it must place the new load image 'somewhere' at a higher position. It can choose any memory range for that, as long as this does not interfere with monitor execution. After relocation and loading, the monitor starts the loaded image simply by transferring control to its first instruction. Figure 16, following, sketches a load map containing the memory areas occupied by the monitor and by the executable that is later loaded by it.[1]
- The downloader can also be used by a standalone multiprocessor boot procedure to economize on EEPROM size when the images to be loaded on the different processors are obtained from the same executable. Especially for large executables, or when the number of processors is high, storing load images for all processors would require considerably more EEPROM than storing the (still relocatable) object file, plus a downloader-based boot program that reads and relocates the executable for each of

1. For simplicity, this example does not show that also the monitor itself is loaded in two phases by the L1/L2 boot procedure.

the processors, starts them, and in the last step overwrites itself. An example illustrating this is shown in the simple download program in Figure 17 on page 307.

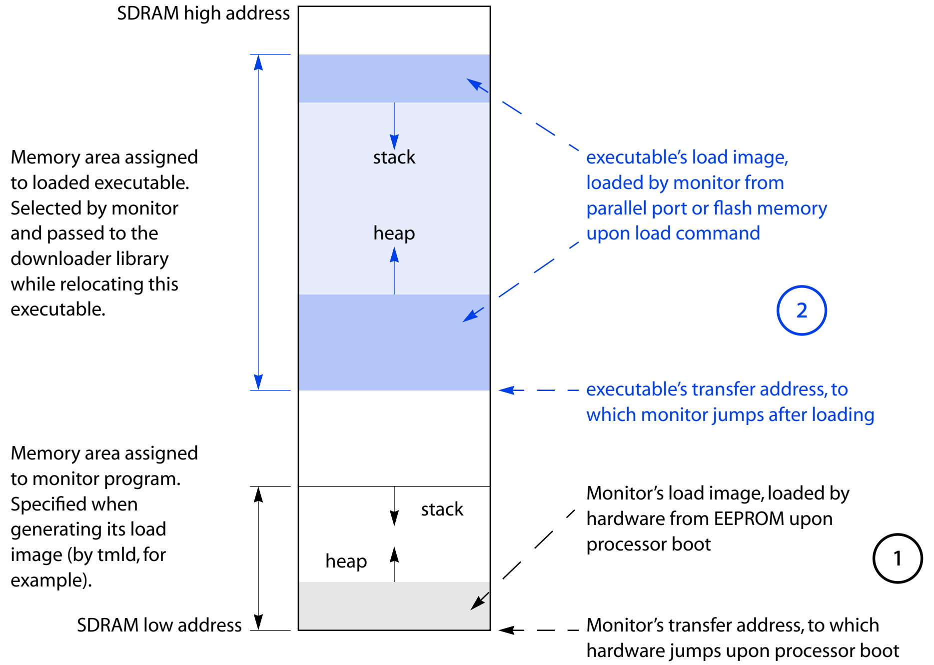


Figure 16 Downloading of Executable by Monitor Program

Phases of Downloading

The following steps describe in which phases downloading is to be performed. First, the procedure for 'normal', single processor downloading is described, followed by a description of how this should be adapted for downloading a multiprocessor cluster. The examples shown in Figure 17 on page 307 and Figure 18 on page 309 can be used as illustrations of both procedures. The individual functions of the downloader API are only listed here.

1. An executable object is *loaded*, that is, a handle is created which is to be used to refer to the object in the next steps; some internal data structures are set up, and initial data like the object's header is copied to the downloader's memory. None of this loaded information is accessible other than via calls to downloader functions while specifying the handle.

 © 1999 Philips Semiconductors 10/08/99

Different functions for loading are available, to be used for different locations of the object:

Function	Object Location
TMDwnLdr_load_object_from_file	File
TMDwnLdr_load_object_from_mem	A consecutive memory range
TMDwnLdr_load_object_from_driver	Anywhere, by constructing an appropriate **Lib_IODriver** object. Such an object is an encapsulation of user-specified callback functions which are used to access the object in a user-specified way.

2. Optionally, the image size and required image alignment can be retrieved. This size can be used for checking whether the memory area into which the image is to be placed is large enough to hold this image, or (in tools like **tmld** which construct load images) to allocate memory for temporarily storing the image before it can e.g. be written to file; the retrieved alignment can be used for checking whether the alignment of the eventual SDRAM load address matches the one that is required by the executable.

   The retrieved size is *approximately* equal to the sum of the sizes of the initialized sections reported by **tmsize**. Differences are caused by padding between section images for maintaining section alignment.

   The retrieved alignment *usually* is equal to 64 bytes, which is the TriMedia instruction cache block size. Alignments of executables will become larger when *.align* directives specifying alignments that are not divisors of 64 have been used in trees or assembly sources. Such alignments will not be generated by the TriMedia C compiler.

Function	Description
TMDwnLdr_get_image_size	Get minimal image size and minimal image alignment.

3. All download symbols present in the executable other than the reserved ones must be resolved by giving them appropriate 32-bit values, depending on their semantics. The reserved download symbols will be implicitly resolved in the next step.

   Note

   For more information on reserved symbols, refer to *Reserved Download Symbols* in Chapter 11 of Book 4, *Software Tools*, Part B.

Function	Description
TMDwnLdr_resolve_symbol	Resolve download symbol to absolute 32-bit value.

4. After all symbols other than reserved download symbols have been resolved, the executable must be relocated. Relocation does not retrieve an image yet; rather, it reads the remaining parts of the object that are needed, does some error checking, maps all

of the object's sections in the specified SDRAM memory range and prepares image extraction; as a side effect it implicitly resolves the reserved download symbols using the some basic information that is passed as arguments to the relocation function:

— SDRAM memory range into which the load image is to be eventually placed

— Host type

— MMIO base address

— Processor frequency

— A flag indicating whether caching should be automatically enabled or disabled, or left to the user

Function	Description
TMDwnLdr_relocate	Relocate an executable.

5. After the loaded object has been relocated, the load image can be extracted. The image extraction function takes a memory base address to which the image must be copied; this address must not be confused with the memory base specified to the relocation call (Refer to step 4, above). While the latter address specifies the *SDRAM address* where the load image *eventually* must be loaded, the image extraction address specifies where the result of image extraction currently must be placed. The physical SDRAM load address must be specified during relocation. In monitor programs that place the extracted image immediately into SDRAM, the virtual SDRAM load address in the monitor's address space must be specified during image extraction. Similarly, in case of a tool that writes the extracted image to a file, the address of a temporary buffer for holding the image before it can be written to file must be specified during image extraction.

Function	Description
TMDwnLdr_get_memory_image	Extract load image.

6. For the final step, the object handle, with all resources currently allocated for it, must be deallocated. A handle can *not* be reused for a new relocation, or for a new image extraction.

Function	Description
TMDwnLdr_unload_object	Free object handle with all associated resources.

In case of downloading a multiprocessor cluster, the above procedure must be repeated for loading all executables on all the processors, with a slight adaptation necessary for implementing shared sections. First, a logical numbering of the used processors must be made using numbers *0 .. N-1,* where *N* is the number of processors. Second, an alternate relocation function must be used in step 4; this function takes an array of MMIO bases of all processors as additional argument, as well as the assigned number of the 'current' processor and the total number of processors (*N*). Finally, a single *shared section table* must be passed to all calls to the relocation function. This table is necessary to record some downloader history, such as the endian of all previously loaded executables (all members of a multiprocessor cluster must have same endian), and which shared sections have been encountered in previously loaded executables: executables containing shared sections that already have been encountered in previous executables will not receive a new copy of the section, but will instead be made to refer to the already loaded one.

Function	Description
TMDwnLdr_create_shared_section_table	Create a shared section table.
TMDwnLdr_multiproc_relocate	Relocate an executable that is part of a multi-processor cluster.
TMDwnLdr_unload_shared_section_table	Free a shared section table, with all allocated resources.

Auxiliary Functions

The following functions allow some further inspection of loaded objects:

Function	Description
TMDwnLdr_get_endian	Get the object's "endian-ness."
TMDwnLdr_patch_value	Store a 32 bit value into the object, at the address of the specified symbol. The value will be stored in the 'correct' byte order, according to the object's own endian; that is, a 32-bit full word memory fetch from the specified address by the downloaded executable will result in the patched value. A patch must be performed after relocation, but before image extraction. The symbol must correspond with an address in an initialized data section.
TMDwnLdr_get_value	Get the 32-bit value from the specified address from the object. The value will be read according to the object's own endian; see above. A 'get_value' must be performed after relocation, but before memory extraction.The symbol must correspond with an address in an initialized data section.
TMDwnLdr_load_symbtab_from_object	Construct a symbol table containing the names and values of all of the object's symbols, and return a handle. Symbol table construction must be performed after relocation, but before memory extraction.
TMDwnLdr_get_address	Get a symbol's 32 bit value from a symbol table.

Function	Description
TMDwnLdr_enclosing_symbol	Return a descriptor of a symbol with largest value that is still less than or equal to a specified value.
TMDwnLdr_traverse_symbols	Call a specified callback function on all symbols in the symbol table, in either alphabetical order, or in order of the symbol's increasing value.
TMDwnLdr_unload_symboltable	Free a symbol table, with all allocated resources.
TMDwnLdr_get_last_error	This function returns a pointer to an internal buffer containing a textual representation of the status of the last call to the downloader library.

Simple Download Example

The program listed in Figure 17 on page 307, shows a **tmsim**-based shell which uses the downloader library for loading a second executable into a 4-megabyte region of the SDRAM that has been allocated from the shell's own heap. Loading is performed according to the steps described earlier in this section; after the executable has been loaded, the shell transfers control to it by means of a branch to its start address. Note that the memory map of shell and loaded program during this procedure is comparable to the one shown in Figure 16 on page 302, but with the exception that loading now is performed into the stack/heap gap instead of after the SDRAM area allocated to the shell. Checking of the error codes returned by the downloader functions has been omitted in the listed program; this is for readability only, and not advised in realistic applications.

By the standard boot code ***$TCS/lib/<endian>/reset.o*** which is added by the compiler driver **tmcc** to executables, the second executable runs completely independent from the shell, and never returns: it performs a cold software start, by setting a new processor endianness (the endianness of shell and of the executable loaded by it need not be the same), by setting up a new stack/heap area within the 4-megabyte memory range in which it was loaded, and by initializing its runtime libraries. Upon termination, similar to any executable, it will bring TriMedia into a RESET state, without returning to the shell. Without special measures for recovering this memory, all SDRAM but the memory in which the second executable has been loaded remains unused. An example of such "special measure" is looking up the actual SDRAM range from MMIO locations DRAM_BASE and DRAM_LIMIT.

Because the user is responsible for TriMedia cache coherence, it is necessary to flush the data cache, and invalidate the instruction cache after an executable has been loaded. This to force parts of the written executable that are still pending in the data cache to be written out to SDRAM, and to prevent stale contents of the instruction cache from being

 ©1999 Philips Semiconductors 10/08/99

executed after the newly loaded executable has started. This is the purpose of the calls to *_cache_copyback* and *iclr* in the example. Note the special way in which download symbols are defined: because it is not possible to cleanly define absolute symbols in C, they have been defined as external arrays.

```
#include “tm1/mmio.h”
#include “tmlib/TMDownLoader.h”
#include “tmlib/tmlibc.h”
#include “assert.h”
#include “stdio.h”
typedef void (*Func)();
custom_op void iclr(void);

/* tmsim’s download parameter, to be passed on: */
   void _syscall();
/* general download parameters, to be passed on: */
   extern Int _host_type_init[];
   extern Int _clock_freq_init[];
   extern Int _MMIO_base_init[];

main(){
   String    filename      = “second.out”;
   UInt      sdram_length  = 4000000;
   Pointer   sdram_base    = _cache_malloc( sdram_length, -1 );
   Int       alignment, minimal_image_size;
   TMDwnLdr_Object_Handle  handle;

   printf(“Loading...\n”);

/* STEP 1 */
   TMDwnLdr_load_object_from_file (filename, Null, &handle);

/* STEP 2 */
   TMDwnLdr_get_image_size   (handle, &minimal_image_size, &alignment );
   assert( (Int)sdram_base % alignment == 0 );

/* STEP 3 */
   TMDwnLdr_resolve_symbol   (handle, “__syscall”, (Int)_syscall );

/* STEP 4 */
   TMDwnLdr_relocate( handle, (tmHostType_t)_host_type_init,
                      (Address)_MMIO_base_init,
                      (UInt)_clock_freq_init,
                      sdram_base, sdram_length,
                      TMDwnLdr_LeaveCachingToDownloader);
/* STEP 5 */
   TMDwnLdr_get_memory_image (handle, sdram_base);

/* STEP 6 */
   TMDwnLdr_unload_object    (handle);

   printf(“Running...\n”);
   _cache_copyback(sdram_base, LOAD_SIZE);
   iclr();
   ((Func)sdram_base)();
   /* never come back */
}
```

Figure 17 Sample Use of Downloader

Multiprocessor Booting

Figure 18 on page 309 shows a simple extension of the downloader program from Figure 17 on page 307 to a multiprocessor downloader function. In this particular setup, the function takes a single executable object from a specified memory address, and loads this executable on a number of TriMedia processors that are specified by a few information arrays passed to the function. Error checking is still omitted for clarity of the example, but should be handled properly in a real use of this function.

If it is assumed that executable objects can be stored into an EEPROM mapped in PCI space, this function can be used for booting a standalone multiprocessor cluster, as follows:

1. One selected processor boots in standalone mode; this processor is referred to as the *boot processor.* All other processors, referred to as the *slave processors*, will boot in 'host assisted' mode, with the boot processor serving as 'host' during booting. Note that this master/slave relationship is only valid during booting, and does not correspond with any master/slave relationship of the processors during execution.
2. The boot processor goes through the L1/L2 boot stages, as described in Chapter 7, *Bootstrapping TriMedia in Autonomous Mode*, of Book 2, *Cookbook*, Part C, and starts a multiprocessor loader based on the function described in Figure 18. The loader relocates and distributes the executable object from EEPROM to all slave processors and to the boot processor.
3. The boot processor releases all slave processors from their RESET states by writing to their BUI_CTL registers in their MMIO spaces. Upon release; the slave processors start executing their copy of the loaded executable.
4. The boot processor flushes the data cache (using library function **_cache_copyback**) to make sure that the loaded executable is completely written to SDRAM; after that, it invalidates the instruction cache (using custom operation *iclr*), to make sure that the instruction cache does not contain stale contents, and it performs a jump to the start address of its own copy of the loaded executable.

The above steps perform multiprocessor booting, except for one technical detail: while loading the new executable on to itself, the boot processor should not overwrite the downloader program. This is easily solved by modifying the L1 part of the L1/L2 loader so that it loads the multiprocessor loader in the stack/heap gap of the final executable. This stack/heap gap remains unused until the loaded executable is started and the multiprocessor loader is no longer needed.

 © 1999 Philips Semiconductors 10/08/99

A full **tmsim**-based example demonstrating this, with the L1 loader simulated by a **tmsim** executable, can be found in the TCS example directory:

$TCS/examples/downloading/mp_downloading

```
LoadMPCluster(
   UInt             nrof_nodes,
   Pointer          eeprom_address,
   Int              eeprom_length,
   tmHostType_t     host_type,
   Address          mmio_bases[],
   UInt             cpu_frequencies[],
   Address          sdram_bases[],
   UInt             sdram_lengths[]
){
   Int                                node;
   TMDwnLdr_SharedSectionTab_Handle   shared_sections;

   TMDwnLdr_create_shared_section_table(&shared_sections);

   for( node = 0; node < nrof_nodes; node++ ){
      TMDwnLdr_Object_Handle   handle;
      Int                      alignment, minimal_image_size;

      TMDwnLdr_load_object_from_mem( eeprom_address, eeprom_length,
                                     shared_sections, &handle );

      TMDwnLdr_multiproc_relocate( handle, host_type, mmio_bases, node,
                                   nrof_nodes, cpu_frequencies[node],
                                   sdram_bases[node], sdram_lengths[node]
                                   TMDwnLdr_LeaveCachingToDownloader );

      TMDwnLdr_get_memory_image(handle, sdram_bases[node]);

      TMDwnLdr_unload_object(handle);
   }
   TMDwnLdr_unload_shared_section_table(shared_sections);
}
```

Figure 18 Sample multiprocessor downloader function

Downloader API Structures and Enumerations

This section presents the TriMedia Downloader API data structure and enumerations.

 © 1999 Philips Semiconductors 10/08/99

TMDwnLdr_Status

```
typedef enum {
   TMDwnLdr_OK,
   TMDwnLdr_UnexpectedError,
   TMDwnLdr_InputFailed,
   TMDwnLdr_InsufficientMemory,
   TMDwnLdr_NotABootSegment,
   TMDwnLdr_InconsistentObject,
   TMDwnLdr_UnknownObjectVersion,
   TMDwnLdr_NotFound,
   TMDwnLdr_UnresolvedSymbols,
   TMDwnLdr_SymbolIsUndefined,
   TMDwnLdr_SymbolNotInInitialisedData,
   TMDwnLdr_SDRamTooSmall,
   TMDwnLdr_SDRamImproperAlignment,
   TMDwnLdr_SymbolNotADownloadParm,
   TMDwnLdr_NodeNumberTooLarge,
   TMDwnLdr_NumberOfNodesTooLarge,
   TMDwnLdr_HandleNotValid,
   TMDwnLdr_EndianMismatch
} TMDwnLdr_Status;
```

Return Codes

All functions exported by the downloader library provide a return status. The list below describes the possible values:

Return Value	Description
TMDwnLdr_OK	Successful completion.
TMDwnLdr_UnexpectedError	An unexpected situation occurred. This status should actually never be returned, and indicates an internal error.
TMDwnLdr_InputFailed	While loading an object from file, the specified file could not be opened as an object.
TMDwnLdr_InsufficientMemory	A memory overflow occurred.
TMDwnLdr_NotABootSegment	Attempt to load an object that is either a plain object, a dynamic library, or an application segment. Use an object that has been compiled and linked with **-btype boot** or **-btype dynboot** instead.
TMDwnLdr_InconsistentObject	Attempt to load an object whose contents are possibly corrupted. Try to rebuild the object.

Return Value	Description
TMDwnLdr_UnknownObjectVersion	Attempt to load an object that has an unexpected (possibly newer) object file format version number. Try a downloader library from a newer SDE.
TMDwnLdr_UnresolvedSymbols	Unresolved (download) symbols were encountered during relocation. Usually this occurs when the object has been compiled and linked for an improper host. Inspect the object's download symbols using **tmnm**.
TMDwnLdr_SymbolIsUndefined	Attempt to patch, resolve, or lookup a symbol that is not defined in the object, or (depending on the call) that is not defined as *global* symbol in the object.
TMDwnLdr_SymbolNotInInitializedData	Attempt to patch, or get the contents of a symbol that is not in an initialized data section.
TMDwnLdr_SDRamTooSmall	The relocation function detected that the object image is too big to fit in the specified amount of SDRAM.
TMDwnLdr_SDRamImproperAlignment	The relocation function detected an improper alignment of the specified SDRAM start address; check the *.align* directives in the object's hand-coded trees and assembly sources.
TMDwnLdr_SymbolNotADownloadParm	The symbol specified to a call to the resolve function is not a download parameter. Reason for this was that it was already not a download parameter in the loaded object, or that it has already been resolved: resolution changes download parameters into absolute symbols.
TMDwnLdr_NodeNumberTooLarge	The node number specified in the multiprocessor relocation call is not within the range 0 to N-1, where N is the specified number of nodes.

 © 1999 Philips Semiconductors 10/08/99

Return Value	Description
TMDwnLdr_NumberOfNodesTooLarge	The specified number of nodes is too large for the loaded executable. The maximally allowed number of nodes is considered to be the largest number *N* for which *all* download symbols **__MMIO_base_init_***i* exist in the object for all $0 \leq i < N$.
TMDwnLdr_HandleNotValid	The object, symboltable, or shared sectiontable handle specified in a downloader call was not known.
TMDwnLdr_EndianMismatch	While relocating a member of a multiprocessor cluster (using function **TMDwnLdr_multiproc_relocate**), it had an "endian-ness" different from the first relocated member of the cluster.

Description

This enumeration type defines the possible return status values of the functions in the downloader API.

TMDwnLdr_Caching

```
typedef enum {
   TMDwnLdr_Cached,
   TMDwnLdr_Uncached,
   TMDwnLdr_CacheLocked
} TMDwnLdr_Caching;
```

Description

This enumeration is used in section descriptors, and defines the number of ways the TriMedia cache can be used on the data within specific sections.

TMDwnLdr_Symbol_Scope

```
typedef enum {
TMDwnLdr_LocalScope,
TMDwnLdr_GlobalScope,
TMDwnLdr_DynamicScope
} TMDwnLdr_Symbol_Scope;
```

Description

This enumerations used in symbol descriptors, and defines the visibility of symbols in static and dynamic linking.

 © 1999 Philips Semiconductors 10/08/99

TMDwnLdr_Symbol_Type

```
typedef enum {
   TMDwnLdr_UnresolvedSymbol,
   TMDwnLdr_AbsoluteSymbol,
   TMDwnLdr_RelativeSymbol,
   TMDwnLdr_DynamicallyImportedSymbol
} TMDwnLdr_Symbol_Type;
```

Description

This enumeration is used in symbol descriptors, and specifies the type of the symbol.

TMDwnLdr_Symbol_Traversal_Order

```
typedef enum {
   TMDwnLdr_ByAddress,
   TMDwnLdr_ByName
} TMDwnLdr_Symbol_Traversal_Order;
```

Description

This enumeration is used as a parameter in the function **TMDwnLdr_traverse_symbols**, and specifies the order in which the symbol callback function is called.

TMDwnLdr_CachingSupport

```
typedef enum {
   TMDwnLdr_CachesOff,
   TMDwnLdr_LeaveCachingToUser,
   TMDwnLdr_LeaveCachingToDownloader
} TMDwnLdr_CachingSupport;
```

Description

This enumeration is used as a parameter in the functions, **TMDwnLdr_relocate** and **TMDwnLdr_multiproc_relocate**. It specifies how the TriMedia instruction and data cache should be initialized. See also the description of **caching_support** on page 326.

 © 1999 Philips Semiconductors 10/08/99

TMDwnLdr_Section_Rec

```
typedef struct TMDwnLdr_Section_Rec{
   String   name;
   Address   bytes;
   UInt   size;
   UInt   alignment;
   Bool   big_endian;
   Bool   has_data;
   Bool   is_code;
   Bool   is_read_only;
   TMDwnLdr_Caching   caching;
   Address   relocation;
} TMDwnLdr_Section_Rec;
```

Fields

`name`	Name of the section.
`bytes`	If **has_data** is equal to True, then **bytes** is a pointer to a copy of the section's data in memory.
`size`	The size of the section in bytes.
`alignment`	Required alignment of the section in SDRAM.
`big_endian`	A flag specifying if the section contains instructions or data from a big- or little endian program.
`has_data`	A flag specifying if the section has initial contents.
`is_code`	A flag specifying if the section contains TriMedia instructions.
`is_read_only`	A flag specifying if the section's data is intended to be modified during execution.
`caching`	Specification of how the section's data should be cached during execution.
`relocation`	After a call to **TMDwnLdr_relocate**, this field is set to the SDRAM address to which the section will be loaded.

Description

This descriptor is a section representation, used in functions **TMDwnLdr_get_section**, and **TMDwnLdr_traverse_sections**. Its fields are described under Section Attributes in Chapter 11, *Linking TriMedia Object Modules*, of Book 4, *Software Tools*, Part B.

Downloader API Functions

This section presents the TriMedia Downloader API functions.

Category	Name	Page
Shared Section Table Functions	TMDwnLdr_create_shared_section_table	319
	TMDwnLdr_unload_shared_section_table	320
Object Loading Functions	TMDwnLdr_load_object_from_file	321
	TMDwnLdr_load_object_from_mem	323
	TMDwnLdr_load_object_from_driver	324
	TMDwnLdr_get_image_size	325
	TMDwnLdr_relocate	326
	TMDwnLdr_multiproc_relocate	328
	TMDwnLdr_get_memory_image	331
	TMDwnLdr_patch_value	332
	TMDwnLdr_resolve_symbol	333
	TMDwnLdr_get_value	334
	TMDwnLdr_unload_object	335
	TMDwnLdr_get_section	336
	TMDwnLdr_traverse_sections	337
	TMDwnLdr_get_endian	338
	TMDwnLdr_load_symbtab_from_object	339
Symbol Table Functions	TMDwnLdr_get_address	340
	TMDwnLdr_get_enclosing_symbol	341
	TMDwnLdr_traverse_symbols	342
	TMDwnLdr_unload_symboltable	343
	TMDwnLdr_get_last_error	344

 © 1999 Philips Semiconductors 10/08/99

TMDwnLdr_create_shared_section_table

```
TMDwnLdr_Status TMDwnLdr_create_shared_section_table(
   TMDwnLdr_SharedSectionTab_Handle   *result
);
```

Parameters

`result`	Returned handle, or undefined when result unequal to **TMDwnLdr_OK**.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_InsufficientMemory`	A memory overflow occurred.
`TMDwnLdr_UnexpectedError`	An unexpected situation occurred. This status should actually never be returned, and indicates an internal error.

Description

Create an empty shared section table, for use in multiprocessing downloading.

Side effects: Memory to hold the result is allocated via malloc.

TMDwnLdr_unload_shared_section_table

```
TMDwnLdr_Status TMDwnLdr_unload_shared_section_table(
   TMDwnLdr_SharedSectionTab_Handle   handle
);
```

Parameters

`handle`	Handle of loaded table to unload.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_HandleNotValid`	Returned when handle becomes invalid.

Description

Unload shared section table. Postcondition: handle becomes invalid.

 © 1999 Philips Semiconductors 10/08/99

TMDwnLdr_load_object_from_file

```
TMDwnLdr_Status TMDwnLdr_load_object_from_file(
   String                            path,
   TMDwnLdr_SharedSectionTab_Handle  shared_sections,
   TMDwnLdr_Object_Handle           *result
);
```

Parameters

path	Name of executable file to be loaded.
shared_sections	Table that stores the addresses of shared sections in case of multiple downloads of executables in a multiprocessor system. Null is allowed when this facility is not used, for instance in single processor downloads.
*result	Returned handle, or undefined when result unequal to **TMDwnLdr_OK**.

Return Codes

TMDwnLdr_OK	Success.
TMDwnLdr_FileNotFound	While loading an object from file, the specified file could not be opened. **Note** This status will also be returned in case of a file read protection violation. Provide the proper filename, or allow read access.
TMDwnLdr_NotABootSegment	Attempt to load an object that is either a plain object, a dynamic library, or an application segment. Use an object that has been compiled and linked with -btype boot or -btype dynboot instead.
TMDwnLdr_UnknownObjectVersion	Attempt to load an object that has an unexpected (possibly newer) object file format version number. Try a downloader library from a newer SDE.
TMDwnLdr_InsufficientMemory	A memory overflow occurred.
TMDwnLdr_InconsistentObject	Attempt to load an object whose contents are possibly corrupted. Try to rebuild the object.
TMDwnLdr_UnexpectedError	An unexpected situation occurred. This status should actually never be returned, and indicates an internal error.

Description

Read an executable from file into memory, and return a handle for subsequent operations. Side effects: Memory to hold the result is allocated using **malloc**.

 © 1999 Philips Semiconductors 10/08/99

TMDwnLdr_load_object_from_mem

```
TMDwnLdr_Status TMDwnLdr_load_object_from_mem(
   Address                              mem,
   Int                                  length,
   TMDwnLdr_SharedSectionTab_Handle     shared_sections,
   TMDwnLdr_Object_Handle              *result
);
```

Parameters

`mem`	Start of memory image of executable.
`length`	Length of image.
`shared_sections`	Table remembering the addresses of shared sections in case of multiple downloads of executables in a multiprocessor system. Null is allowed when this facility is not used, for instance in single processor downloads.
`result`	Returned handle, or undefined when result unequal to **TMDwnLdr_OK**.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_NotABootSegment`	Attempt to load an object that is either a plain object, a dynamic library, or an application segment. Use an object that has been compiled and linked with -btype boot or -btype dynboot instead.
`TMDwnLdr_UnknownObjectVersion`	Attempt to load an object that has an unexpected (possibly newer) object file format version number. Try a downloader library from a newer SDE.
`TMDwnLdr_InsufficientMemory`	A memory overflow occurred.
`TMDwnLdr_InconsistentObject`	Attempt to load an object whose contents are possibly corrupted. Try to rebuild the object.
`TMDwnLdr_UnexpectedError`	An unexpected situation occurred. This status should actually never be returned, and indicates an internal error.

Description

Reads an executable from a memory area into memory, and returns a handle for subsequent operations. Memory to hold the result is allocated using **malloc**.

TMDwnLdr_load_object_from_driver

```
TMDwnLdr_Status TMDwnLdr_load_object_from_driver(
   Lib_IODriver                      driver,
   TMDwnLdr_SharedSectionTab_Handle  shared_sections,
   TMDwnLdr_Object_Handle           *result
);
```

Parameters

`driver`	Driver controlling object access.
`shared_sections`	Table that stores the addresses of shared sections in case of multiple downloads of executables in a multiprocessor system. Null is allowed when this facility is not used, for instance in single processor downloads.
`result`	Returned handle, or undefined when result unequal to **TMDwnLdr_OK**.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_NotABootSegment`	Attempt to load an object that is either a plain object, a dynamic library, or an application segment. Use an object that has been compiled and linked with **-btype** boot or **-btype** dynboot instead.
`TMDwnLdr_UnknownObjectVersion`	Attempt to load an object that has an unexpected (possibly newer) object file format version number. Try a downloader library from a newer SDE.
`TMDwnLdr_InsufficientMemory`	A memory overflow occurred.
`TMDwnLdr_InconsistentObject`	Attempt to load an object whose contents are possibly corrupted. Try to rebuild the object.
`TMDwnLdr_UnexpectedError`	An unexpected situation occurred. This status should actually never be returned, and indicates an internal error.

Description

Reads an executable from a previously created driver, and returns a handle for subsequent operations. Memory to hold the result is allocated using **malloc**.

 ©1999 Philips Semiconductors 10/08/99

TMDwnLdr_get_image_size

```
TMDwnLdr_Status TMDwnLdr_get_image_size(
   TMDwnLdr_Object_Handle   handle,
   Int                     *minimal_image_size,
   Int                     *alignment
);
```

Parameters

`handle`	Handle of loaded exec to be queried.
`minimal_image_size`	Pointer to minimal size of image alignment.
`alignment`	Pointer to required alignment of the download area in terms of TM-1's address space.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_HandleNotValid`	Returned when the handle is invalid.

Description

Gets the extracted image size, and its required alignment in SDRAM.

TMDwnLdr_relocate

```
TMDwnLdr_Status TMDwnLdr_relocate(
   TMDwnLdr_Object_Handle  handle,
   tmHostType_t            host_type,
   Address                 MMIO_base,
   UInt                    TM1_frequency,
   Address                 sdram_base,
   UInt                    sdram_length,
   TMDwnLdr_CachingSupport   caching_support
);
```

Parameters

`handle`	Handle of loaded executable to be relocated.
`host_type`	Value that the object might want to know.
`MMIO_base`	Value that the object might want to know.
`TM1_frequency`	Value that the object might want to know.
`sdram_base`	Base of download area in TM's address space.
`sdram_length`	Length of download area.
`caching_support`	Specification of responsibility of setting the cacheable limit and the cachelocked regions: **TMDwnLdr_LeaveCachingToUser** Cacheable limit and cachelocked regions are entirely under control of the user, the downloader/boot code will not touch it. **TMDwnLdr_LeaveCachingToDownloader** Cachelocked regions and cacheable limit are entirely under control of the downloader, which will use this control to intelligently map the different cached/uncached/ cachelocked sections within the specified sdram, partitioned in different caching property regions, and let the downloaded program set cacheable limit and cachelocked regions accordingly. **TMDwnLdr_CachesOff** Cachelocked regions and cacheable limit are entirely under control of the downloader, which will let the downloaded program run with cache "off."

 © 1999 Philips Semiconductors 10/08/99

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_UnresolvedSymbols`	Unresolved (download) symbols were encountered during relocation. Usually this occurs when the object has been compiled and linked for an improper host. Inspect the object's download symbols using **tmnm**. (Refer to *Reserved Download Symbols* in Chapter 11 of Book 4, Software Tools, Part B.)
`TMDwnLdr_SDRamTooSmall`	The relocation function detected that the object image is too big to fit in the specified amount of SDRAM.
`TMDwnLdr_SDRamImproperAlignment`	The relocation function detected an improper alignment of the specified SDRAM start address; check the .align directives in the object's hand-coded trees and assembly sources.
`TMDwnLdr_HandleNotValid`	Returned when the handle is invalid.
`TMDwnLdr_InsufficientMemory`	A memory overflow occurred.
`TMDwnLdr_InconsistentObject`	Attempt to load an object whose contents are possibly corrupted. Try to rebuild the object.
`TMDwnLdr_UnexpectedError`	An unexpected situation occurred. This status should actually never be returned, and indicates an internal error.

Description

Relocate the loaded executable into a specified TM1 address range, with specified values for **MMIO_base** and **TM1_frequency**. The specified TM1 address base must be aligned according to the value returned by **TMDwnLdr_get_image_sizes**. Also the length of this range must be larger than the minimal length returned by **TMDwnLdr_get_image_sizes**.

Cachelocked regions and cacheble limit are entirely under control of the downloader, which will let the downloaded program run with cache "off."

TMDwnLdr_multiproc_relocate

```
TMDwnLdr_Status TMDwnLdr_multiproc_relocate(
   TMDwnLdr_Object_Handle   handle,
   tmHostType_t             host_type,
   Address                 *MMIO_bases,
   UInt                     node_number,
   UInt                     number_of_nodes,
   UInt                     TM1_frequency,
   Address                  sdram_base,
   UInt                     sdram_length,
   TMDwnLdr_CachingSupport   caching_support
);
```

Parameters

`handle`	Handle of loaded executable to be relocated.
`host_type`	By this, the host that downloads the executable makes itself known to the executable.
`MMIO_bases`	Array specifying for each node in the range 0 to **number_of_nodes**–1 of its mmio base address.
`node_number`	'Current' node number, that is, the processor number on which the relocated code is to run; its value is required to be in the range 0 to **number_of_nodes**–1.
`number_of_nodes`	Number of TM-1s available.
`TM1_frequency`	Processor frequency [MHz].
`sdram_base`	Base of download area in TM's address space.
`sdram_length`	Length of download area.
`caching_support`	Specification of responsibility of setting the cacheable limit and the cachelocked regions: **TMDwnLdr_LeaveCachingToUser** Cacheable limit and cachelocked regions are entirely under control of the user, the downloader/boot code will not touch it **TMDwnLdr_LeaveCachingToDownloader** Cachelocked regions and cacheable limit are entirely under control of the downloader, which will use this control to intelligently map the different cached/uncached/cachelocked sections within the specified sdram, partitioned in different caching property regions, and let the downloaded program set cacheable limit and cachelocked regions accordingly.

© 1999 Philips Semiconductors 10/08/99

	TMDwnLdr_CachesOff
	Cachelocked regions and cacheable limit are entirely under control of the downloader, which will let the downloaded program run with 'cache off.'

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_UnresolvedSymbols`	Unresolved (download) symbols were encountered during relocation. Usually this occurs when the object has been compiled and linked for an improper host. Inspect the object's download symbols using **tmnm**. (Refer to Reserved Download Symbols in Chapter 11, *Linking TriMedia Object Modules*, of Book 4, *Software Tools*).
`TMDwnLdr_SDRamTooSmall`	The relocation function detected that the object image is too big to fit in the specified amount of SDRAM.
`TMDwnLdr_SDRamImproperAlignment`	The relocation function detected an improper alignment of the specified SDRAM start address; check the.align directives in the object's hand-coded trees and assembly sources.
`TMDwnLdr_HandleNotValid`	Returned when the handle is invalid.
`TMDwnLdr_InsufficientMemory`	A memory overflow occurred.
`TMDwnLdr_InconsistentObject`	Attempt to load an object whose contents are possibly corrupted. Try to rebuild the object.
`TMDwnLdr_NodeNumberTooLarge`	The node number specified in the multiprocessor relocation call is not within the range 0 to N-1, where N is the specified number of nodes.
`TMDwnLdr_NumberOfNodesTooLarge`	The specified number of nodes is too large for the loaded executable. The maximum number of nodes is considered to be the largest number *N* for which all download symbols **__MMIO_base_init_***i* exist in the object for all $0 \le i < N$.
`TMDwnLdr_UnexpectedError`	An unexpected situation occurred. This status should actually never be returned, and indicates an internal error.

Description

Relocates the loaded executable into a specified TM1 address range, with specified values for **MMIO_base** and **TM1_frequency.** The specified TM1 address base must be aligned according to the value returned by **TMDwnLdr_get_image_sizes**. Also the length of this range must be larger than the minimal length returned by **TMDwnLdr_get_image_sizes**.

This relocation function is intended for use in multiprocessor TM1 environments; the all processors are numbered from 0 to number_of_nodes-1, and their SDRAMs and MMIO spaces are cross accessible.

 © 1999 Philips Semiconductors 10/08/99

TMDwnLdr_get_memory_image

```
TMDwnLdr_Status TMDwnLdr_get_memory_image(
   TMDwnLdr_Object_Handle   handle,
   Address                  base
);
```

Parameters

`handle`	Handle of loaded executable to be queried.
`base`	Base of download area in the address space of the caller of this function.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_HandleNotValid`	Returned when the handle is invalid.

Description

Copy the section’s memory images of the specified loaded executable into a buffer in the current (for example, the downloader’s) address space. NB: This function is destructive on the loaded object. It cannot further be used and must be deallocated after this call.

TMDwnLdr_patch_value

```
TMDwnLdr_Status TMDwnLdr_patch_value(
   TMDwnLdr_Object_Handle   handle,
   String                   symbol,
   UInt32                   value
);
```

Parameters

handle	Handle of loaded exec to be patched.
symbol	Name of symbol in null-terminated string.
value	Value to assign.

Return Codes

TMDwnLdr_OK	Success.
TMDwnLdr_SymbolIsUndefined	Attempt to patch, resolve, or lookup a symbol that is not defined in the object, or (depending on the call) that is not defined as global symbol in the object.
TMDwnLdr_SymbolNotInInitialisedData	Attempt to patch, or get the contents of a symbol that is not in an initialized data section.
TMDwnLdr_HandleNotValid	Returned when the handle is invalid.

Description

Assign a 32-bit value to a symbol with specified name in an initialized data section.

Note

The symbol must have dynamic scope.

 © 1999 Philips Semiconductors 10/08/99

TMDwnLdr_resolve_symbol

```
TMDwnLdr_Status TMDwnLdr_resolve_symbol(
   TMDwnLdr_Object_Handle  handle,
   String                  symbol,
   UInt32                  value
);
```

Parameters

`handle`	Handle of loaded exec to be patched.
`symbol`	Name of symbol in null-terminated string.
`value`	Value to assign.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_SymbolIsUndefined`	Attempt to patch, resolve, or lookup a symbol that is not defined in the object, or (depending on the call) that is not defined as global symbol in the object.
`TMDwnLdr_HandleNotValid`	Returned when the handle is invalid.
`TMDwnLdr_SymbolNotADownloadParm`	The symbol specified to a call to the resolve function is not a download parameter. Reason for this is because it was not a download parameter in the loaded object, or it had already been resolved: resolution changes download parameters into absolute symbols.

Description

Define a 32-bit absolute value for the still unresolved symbol with specified name (this must then be an unresolved symbol of type **download_parm**). This function must be used to resolve all download parameters before any call to **TMDwnLdr_relocate**.

Note
The symbol must have dynamic scope, but this is automatically the case when it has been created by the TriMedia object library, or by **tmld**.

TMDwnLdr_get_value

```
TMDwnLdr_Status TMDwnLdr_get_value(
   TMDwnLdr_Object_Handle   handle,
   String                   symbol,
   UInt32                  *result
);
```

Parameters

`handle`	Handle of loaded executable.
`symbol`	Name of symbol in null-terminated string.
`result`	Pointer to result location.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_SymbolIsUndefined`	Attempt to patch, resolve, or lookup a symbol that is not defined in the object, or (depending on the call) that is not defined as global symbol in the object.
`TMDwnLdr_SymbolNotInInitialisedData`	Attempt to patch, or get the contents of a symbol that is not in an initialized data section.
`TMDwnLdr_HandleNotValid`	Returned when the handle is invalid.

Description

Retrieves a 32-bit value from a symbol with specified name in an initialized data section.

 © 1999 Philips Semiconductors 10/08/99

TMDwnLdr_unload_object

```
TMDwnLdr_Status TMDwnLdr_unload_object(
   TMDwnLdr_Object_Handle  handle
);
```

Parameters

`handle`	Handle of loaded exec to unload.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_HandleNotValid`	Returned when handle is invalid.

Description

Unload loaded executable; all resources allocated for the executable will be freed, but extracted section group images and extracted symbol tables will be unaffected. Postcondition: handle becomes invalid.

TMDwnLdr_get_section

```
TMDwnLdr_Status TMDwnLdr_get_section(
   TMDwnLdr_Object_Handle   handle,
   String                   name,
   UInt32                  *section
);
```

Parameters

`handle`	Handle of loaded exec to get section from.
`name`	Name of requested section.
`section`	Pointer to buffer which will be set as a result of this function.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_NotFound`	While loading an object from file, the specified file could not be opened. **Note** This status will also be returned in the case of a file read protection violation. Provide the proper filename, or allow read access.
`TMDwnLdr_HandleNotValid`	Returned when handle is invalid.

Description

Look up a user section by name.

 © 1999 Philips Semiconductors 10/08/99

TMDwnLdr_traverse_sections

```
TMDwnLdr_Status TMDwnLdr_traverse_sections(
   TMDwnLdr_Object_Handle   handle,
   TMDwnLdr_Section_Fun     fun,
   Pointer                  data
);
```

Parameters

`handle`	Handle of loaded executable to traverse.
`symbol`	Function to apply.
`value`	Additional data argument.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_HandleNotValid`	Returned when the handle is invalid.

Description

Apply function **fun** to all sections in the specified loaded object, in download order and Side effects: fun has been applied to all sections, in the order in which they will be downloaded.

Note
The TMDownLoader will place all the cached sections at the beginning of SDRAM and the uncached (data) sections at the end. Although the function will traverse in download order, there might be a hole in between. The section buffers used in the calls to **fun** will not survive this function call.

TMDwnLdr_get_endian

```
TMDwnLdr_Status TMDwnLdr_get_endian(
   TMDwnLdr_Object_Handle   handle,
   Endian                  *endian
);
```

Parameters

`handle`	Handle of loaded executable.
`endian`	Pointer to result location.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_HandleNotValid`	Returned when the handle is invalid.

Description

Get the endianness of the specified loaded object.

 © 1999 Philips Semiconductors 10/08/99

TMDwnLdr_load_symbtab_from_object

```
TMDwnLdr_Status TMDwnLdr_load_symbtab_from_object(
   TMDwnLdr_Object_Handle    object,
   TMDwnLdr_Symbtab_Handle  *result
);
```

Parameters

`object`	Object extracts from this handle.
`result`	Returned handle, or undefined when result unequal to **TMDwnLdr_OK**.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_HandleNotValid`	Returned when the handle is invalid.
`TMDwnLdr_InsufficientMemory`	A memory overflow occurred.
`TMDwnLdr_InconsistentObject`	Attempt to load an object whose contents are possibly corrupted. Try to rebuild the object.
`TMDwnLdr_UnexpectedError`	An unexpected situation occurred. This status should actually never be returned, and indicates an internal error.

Description

Construct a symbol table from a previously loaded object, and return a handle for subsequent operations. Memory to hold the result is allocated using **malloc**.

Note
The information in this symbol table becomes meaningless upon subsequent relocation of the object, but remains valid when the object is unloaded.

TMDwnLdr_get_address

```
TMDwnLdr_Status TMDwnLdr_get_address(
   TMDwnLdr_Symbtab_Handle  handle,
   String                   symbol,
   String                  *section,
   Address                 *address
);
```

Parameters

`handle`	Handle of executable's symbol table.
`symbol`	Name of symbol in null-terminated string.
`section`	Returned name of symbol's section.
`address`	Returned symbol address.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_SymbolIsUndefined`	Attempt to patch, resolve, or lookup a symbol that is not defined in the object, or (depending on the call) that is not defined as global symbol in the object.
`TMDwnLdr_HandleNotValid`	Returned when handle is not valid.

Description

Return the address of the symbol in terms of the loaded object's memory space given its current relocation state.

 © 1999 Philips Semiconductors 10/08/99

TMDwnLdr_get_enclosing_symbol

```
TMDwnLdr_Status TMDwnLdr_get_enclosing_symbol(
   TMDwnLdr_Symbtab_Handle   handle,
   Address                   address,
   String                   *section,
   String                   *symbol,
   Address                  *symbol_address
);
```

Parameters

handle	Handle of executable's symbol table.
address	Input address.
section	Section name of matching symbol.
symbol	Name of matching symbol.
symbol_address	Address of matching symbol.

Return Codes

TMDwnLdr_OK	Success.
TMDwnLdr_SymbolIsUndefined	Attempt to patch, resolve, or lookup a symbol that is not defined in the object, or (depending on the call) that is not defined as global symbol in the object.
TMDwnLdr_HandleNotValid	Returned when handle is not valid.

Description

Return info on the symbol with the highest address less than or equal to the specified address. All addresses in terms of the loaded object's memory space given its current relocation state.

TMDwnLdr_traverse_symbols

```
TMDwnLdr_Status TMDwnLdr_traverse_symbols(
   TMDwnLdr_Symbtab_Handle           handle,
   TMDwnLdr_Symbol_Traversal_Order   order,
   TMDwnLdr_Symbol_Fun               fun,
   Pointer                           data
);
```

Parameters

handle	Handle of symbol table to traverse.
order	Parameter to guide order of traversal.
fun	Function to apply.
data	Additional data argument.

Return Codes

TMDwnLdr_OK	Success.
TMDwnLdr_HandleNotValid	Returned when handle is invalid.

Description

Apply function **fun** to all symbols in the specified symbol table, with additional data argument. **fun** has been applied to each symbol, in the order specified by **order**.

TMDwnLdr_unload_symboltable

```
TMDwnLdr_Status TMDwnLdr_unload_symboltable(
   TMDwnLdr_Symbtab_Handle   handle
);
```

Parameters

`handle`	Handle of loaded symbol table to unload.

Return Codes

`TMDwnLdr_OK`	Success.
`TMDwnLdr_HandleNotValid`	Returned when handle is invalid.

Description

Unload loaded symbol table; all resources allocated for the symbol table will be freed. Postcondition: Handle becomes invalid.

TMDwnLdr_get_last_error

```
String TMDwnLdr_get_last_error(
   TMDwnLdr_Status   status
);
```

Parameters

`status`	Last returned error code.

Return

The function return is (a pointer to) the string.

Description

Returns a string that contains the last error message. The string is owned by the library.

 © 1999 Philips Semiconductors 10/08/99

Chapter 13

Dynamic Linking API

Overview

This section describes the explicit dynamic loader programming interface. The dynamic loader is a TriMedia library that is part of the system library **libam.dll**, and is automatically available for TriMedia programs that have been compiled for dynamic linking (that is, for code segments that have been linked with **tmcc** option **-btype dynboot**, **-btype dll**, or **-btype app**).

Functions are provided for loading and unloading code segments, and for binding and unbinding them. These and other dynamic loader concepts are described in are described in Chapter 11, *Linking TriMedia Object Modules*, of Book 4, *Software Tools*, Part B.

The function prototypes and typs of the dynamic loader API are defined in the include file tmlib/DynamicLoader.h.

Dynamic Linking API Types

This section presents the TriMedia Dynamic Linking API types.

Category	Name	Page
enum	DynLoad_Status	347
struct	DynLoad_Code_Segment_Handle	348
function	DynLoad_MallocFun	349
	DynLoad_FreeFun	349
	DynLoad_ErrorFun	349

© 1999 Philips Semiconductors 10/08/99

DynLoad_Status

```
typedef enum {
   DynLoad_OK                      =  0,
   DynLoad_FileNotFound            =  1,
   DynLoad_InsufficientMemory      =  2,
   DynLoad_InconsistentObject      =  3,
   DynLoad_UnknownObjectVersion    =  4,
   DynLoad_WrongEndian             =  5,
   DynLoad_WrongChecksum           =  6,
   DynLoad_NotUnloadable           =  7,
   DynLoad_UnresolvedSymbol        =  8,
   DynLoad_NotADll                 =  9,
   DynLoad_NotAnApp                = 10,
   DynLoad_NotPresent              = 11,
   DynLoad_StillReferenced         = 12,
   DynLoad_StackOverflow
} DynLoad_Status;
```

Description

Result status values for the exported functions.

DynLoad_Code_Segment_Handle

```
typedef struct {
   String    name;
   Pointer   start;
} *DynLoad_Code_Segment_Handle;
```

Fields

`name`	Segment name.
`start`	Start address.

Description

Representation of loaded code segment.

© 1999 Philips Semiconductors 10/08/99

DynLoad_MallocFun

```
typedef Pointer (*DynLoad_MallocFun)(
   UInt
);
```

Description

Pointer to the **DynLoad_MallocFun** function.

DynLoad_FreeFun

```
typedef void (*DynLoad_FreeFun)(
   Pointer
);
```

Description

Callback to **DynLoad_FreeFun** function.

DynLoad_ErrorFun

```
typedef void (*DynLoad_ErrorFun)(
   DynLoad_Status,
   String
);
```

Description

Callback to the ***DynLoad_ErrorFun** function.

Dynamic Linking API Functions

This section presents the TriMedia Dynamic Linking API functions.

 © 1999 Philips Semiconductors 10/08/99

DynLoad_load_application

```
DynLoad_Status DynLoad_load_application(
   String                        path,
   DynLoad_Code_Segment_Handle   *result
);
```

Parameters

`path`	Name of executable file to be loaded.
`*result`	Returned handle, or undefined when result unequal to `DynLoad_OK`.

Return Codes

`DynLoad_OK`	Success.
`DynLoad_FileNotFound`	File was not found.
`DynLoad_InsufficientMemory`	There was insufficient memory.
`DynLoad_InconsistentObject`	Incorrect path specification.
`DynLoad_UnknownObjectVersion`	Unknown object version.
`DynLoad_WrongEndian`	Wrong endian.
`DynLoad_WrongChecksum`	Wrong checksum.
`DynLoad_UnresolvedSymbol`	Unresolved symbol.
`DynLoad_NotADll`	There is no dll.

Description

Read an application segment from file into memory, and return a handle for subsequent operations. Contrary to dlls, location of application object files is not subject to any lookup mechanism; for this reason a path must be used for specifying the application file. The *path* here is the text string which could be used in calls to **open** in order to open the application object file. Also contrary to dlls, duplicate copies of apps are allowed, and therefore subsequent load calls with the same path value result in different, independent loaded applications. The transfer address of a loaded application can be found in the **start** field of the returned module descriptor. Loaded applications can be unloaded by means of a call to **DynLoad_unload_segment**.

Memory to hold the result is allocated using either **malloc** or the user-specified memory manager (Refer to function **DynLoad_swap_mm** on page 100).

DynLoad_unload_application

```
DynLoad_Status DynLoad_unload_application(
   DynLoad_Code_Segment_Handle   segment
);
```

Parameters

`segment`	Descriptor of application to unload.

Return Codes

`DynLoad_OK`	Success.
`DynLoad_NotAnApp`	Does not correspond with an application segment.
`DynLoad_StillReferenced`	The application's code is still in use.

Description

Unload specified application from memory. This function will fail if the segment does not correspond with an application segment, or if the application's code is still in use (e.g. when it contains a still installed interrupt handler, or when other tasks are still executing its code, or when it has been bound by a call to **AppModel_bind_codeseg**.

 © 1999 Philips Semiconductors 10/08/99

DynLoad_bind_dll

```
DynLoad_Status DynLoad_bind_dll(
   String                        name,
   DynLoad_Code_Segment_Handle   *result
);
```

Parameters

`name`	Name of dll to be loaded. No path specification is allowed.
`result`	Returned handle, or undefined when result unequal to **DynLoad_OK**.

Return Codes

`DynLoad_OK`	Success.
`DynLoad_FileNotFound`	File was not found.
`DynLoad_InsufficientMemory`	There was insufficient memory.
`DynLoad_InconsistentObject`	Incorrect path specification.
`DynLoad_UnknownObjectVersion`	Unknown object version.
`DynLoad_WrongEndian`	Wrong endian.
`DynLoad_WrongChecksum`	Wrong checksum.
`DynLoad_UnresolvedSymbol`	Unresolved symbol.
`DynLoad_NotADll`	There is no dll.

Description

Locate specified dll, load it into memory when not already loaded, and return a handle for subsequent operations. The dll is marked as being in use, preventing it from being unloaded, until a matching call to **DynLoad_unbind_dll**.

Contrary to applications, which must be loaded with complete path specification, dlls are subject to a lookup mechanism; for this reason, no path specification is allowed. Instead, just the base file name need be given.

DynLoad_bind_dll and **DynLoad_unbind_dll** maintain a reference count. Memory to hold the result is allocated using **malloc** or the user-specified memory manager (Refer to the function **DynLoad_swap_mm** on page 100).

DynLoad_unbind_dll

```
DynLoad_Status DynLoad_unbind_dll(
   String   name
);
```

Parameters

`name`	Name of DLL to unbind.

Return Codes

`DynLoad_OK`	Success.
`DynLoad_NotPresent`	Returned if the DLL is not present.

Description

Remove usage mark from DLL.

DynLoad_unload_dll

```
DynLoad_Status DynLoad_unload_dll(
   String   name
);
```

Parameters

`name`	Name of DLL to unload.

Return Codes

`DynLoad_OK`	Success.
`DynLoad_NotPresent`	It is not present.
`DynLoad_StillReferenced`	Returned when other tasks are still executing its code.

Description

Unload specified DLL from memory, together with all other dlls that make "immediate" use of it. Unloading will fail if any of these DLLs contain code that is still in use (for example, when it contains a still installed interrupt handler, or when other tasks are still executing its code, or when it has been bound by a call to **AppModel_bind_codeseg** or **DynLoad_bind_dll**).

 © 1999 Philips Semiconductors 10/08/99

DynLoad_unload_all

```
DynLoad_Status DynLoad_unload_all(
   String   name
);
```

Parameters

name	Name of the DLL to unload.

Return Codes

DynLoad_OK	Success.

Description

Unload all currently unused DLLs.

DynLoad_bind_codeseg

```
DynLoad_Code_Segment_Handle DynLoad_bind_codeseg(
   Address   code
);
```

Parameters

`code`	Code address (for example, a function pointer).

Return Codes

`DynLoad_OK`	Success.

Description

Mark the code segment that contains the specified code address as used. **DynLoad_bind_codeseg** maintains a reference count.

DynLoad_unbind_codeseg

```
DynLoad_Code_Segment_Handle DynLoad_unbind_codeseg(
   Address   code
);
```

Parameters

`code`	Code address (for example, a function pointer).

Return Codes

`DynLoad_OK`	Success.

Description

Mark the code segment that contains the specified code address as no longer used. **DynLoad_bind_codeseg** maintains a reference count.

 © 1999 Philips Semiconductors 10/08/99

DynLoad_swap_mm

```
void DynLoad_swap_mm(
   DynLoad_MallocFun   *perm_malloc,
   DynLoad_FreeFun     *perm_free,
   DynLoad_MallocFun   *temp_malloc,
   DynLoad_FreeFun     *temp_free
);
```

Parameters

perm_malloc	Functions for allocating storage for loaded code segments.
perm_free	Functions for allocating storage for loaded code segments.
temp_malloc	Functions for allocating storage for working memory during dynamic loading.
temp_free	Functions for allocating storage for working memory during dynamic loading.

Return Codes

DynLoad_OK	Success.

Description

Swap the permanent and temporary storage manager currently in use by the dynamic loader.

Note
This function is for system's purposes only, and should be called before the dynamic loader has been active.

DynLoad_swap_stub_error_handler

```
void DynLoad_swap_stub_error_handler(
   DynLoad_ErrorFun   *stub_error_handler
);
```

Parameters

`stub_error_handler`	Function stub error handler.

Return Codes

`DynLoad_OK`	Success.

Description

Swap the (global) error handler that is to be called upon failure in implicit dll loading from function stubs.

Note
Generally, this error handler has three options:

1. Abort the application by calling exit, optionally after printing a diagnostic.

2. Clean up the global application state (e.g., free up memory after a load failure due to memory overflow), and return, so that the dynamic loader can retry its failing load.

3. Raise an exception (or perform a longjmp in C) so that the implicit load failure can be handled by the application at a higher level.

As long as the error handler returns, the dynamic loader will retry the load.

 © 1999 Philips Semiconductors 10/08/99

Chapter 14

TriMedia Manager API for Windows

Introduction

The new unified version of the TriMedia Manager (TMMan) was developed as a more portable refinement of the Windows 95 TriMedia Manager. The new TMMan runs on Windows NT, Windows 95, Windows 98, and Windows CE. The new interfaces are very similar to the old interfaces of TMMan for Windows 95. Both interfaces are designed to support communication between TriMedia and a host processor. For an architectural overview, see Chapter 3, *Host Windows Interfaces*, of Book 3, *Software Architecture*, Part A.

Implementation Notes

Throughout this chapter, function descriptions refer to the following notes:

Synchronization Handle on page 360
Scatter Gather Buffer Locking on page 361
Status Codes on page 362
Speculative Load Fix on page 363
WinCE Issues on page 364
Object Names on page 361
Debug Buffer Pointers on page 362
SDRAM Mapping on page 362
Big Endian Execution on page 363
Synchronization Flags on page 364

Synchronization Handle

The caller creates these handles via calls to the operating system specific functions like **CreateEvent** or **AppSem_create**. The caller is also responsible for freeing these handles. The following sections list the operating systems supported by TMMan, the functions used by the caller to allocate these handles and the functions TMMan uses to signal these handles.

Win95 Kernel Mode

Task	Function
Creation	CreateEvent
Signaling	VWIN32_SetWin32Event
Closing	CloseHandle

 © 1999 Philips Semiconductors 10/08/99

WinNT/98 KernelMode

Task	Function
Creation	CreateEvent
Signaling	KeSignalEvent
Closing	CloseHandle

pSOS

Task	Function
Creation	AppSem_create + AppSem_p
Signaling	AppSem_v
Closing	AppSem_delete

Stand-alone (no operating system)

Task	Function
Creation	AppSem_create + AppSem_p
Signaling	AppSem_v
Closing	AppSem_delete

Note
Under Windows operating systems (WinNT, Win95, Win98, or WinCE) this event has to be created as an Auto Reset Event.

Object Names

Every TMMan object has a name associated with it. This name is used to form a binding between the host and target counterparts of the objects. This object name is a unique user supplied name that can be 12 characters long (maximum). The names are case sensitive. The host has to create the named object before the target can find it otherwise the named object creation on the target will fail. These names do not have to be unique across objects—an event and a message channel can use the same name.

Scatter Gather Buffer Locking

The **tmmanSGBuffer***xxx* functions are applicable to systems in which the host processor supports virtual memory. If an application running on the host allocates a buffer which the target processor needs to access (read from or write to), scatter gather locking has to be performed. Scatter Gather locking a buffer ensures that the memory allocated to that buffer does not get paged out. This locking operation also generates a scatter gather list that is used by the target to access the memory allocated to the buffer which is fragmented in physical address space.

Debug Buffer Pointers

The **tmmanDebug*xxx*Buffers** functions return 2 sets of pointers and sizes. These pointers point to a circular buffer in SDRAM or in PC memory. The pointers that track the current state of the debug buffers are constantly changing. These functions returns a snapshot of the pointers. The contents of the circular wrap-around buffer must be accessed in two parts via the **FirstHalfPtr** and the **SecondHalfPtr**. If the buffer has not wrapped around at the instant a call is made, the **FirstHalfPtr** will be Null. And only the **SecondHalfPtr** will point to valid contents. To print the contents of the buffer, the code should be like this.

```
if( FirstHalfPtr ){
   Print( FirstHalfPtr, FirstHalfBufferSize );
}
Print( SecondHalfPtr, SecondHalfBufferSize );
```

Status Codes

All the TMMan API functions return statusSuccess on successful completion. Callers can retrieve a textual description of the failure codes by calling **tmmanGetErrorString**. All error codes are documented in the file TMManErr.h.

SDRAM Mapping

Due to limited amount of virtual address space on some Windows platforms, simultaneous mapping of SDRAM and MMIO spaces of multiple TM processors may fail.

One solution for this problem is to defer SDRAM mapping until it is needed and immediately follow it with unmapping to free Virtual Address Space for mapping other TriMedia processor SDRAMs in the machine.

The following two functions have been added to perform SDRAM mapping and unmapping:

- **tmmanDSPMapSDRAM**
- **tmmanDSPUnmapSDRAM**

To disable SDRAM mapping at initialization (default is to map all of SDRAM), the following registry key has to be set:

```
HKLM\SOFTWARE\PhilipsSemiconductors\TriMedia\TMMan
MapSDRAM=0
```

 ©1999 Philips Semiconductors 10/08/99

When SDRAM mapping is disabled, calls to **tmmanDSPDSPInfo** return invalid values in the **tmmanMemoryBlock.SDRAM.MappedAddress** field. Other calls that make use of this field will also fail. For example:

```
tmmanMappedToPhysical
tmmanPhysicalToMapped
tmmanValidateAddressAndLength
tmmanTranslateAdapterAddress
tmmanDebugDPBuffers
tmmanDebugTargetBuffers
```

To avoid failures, calls to the SDRAM mapping/unmapping functions must to be wrapped with calls to **tmmanDSPMapSDRAM** and **tmmanDSPUnmapSDRAM**.

Note
The C Run Time library server DLL (tmcrt.dll) will not work anymore since it depends on the entire SDRAM to be accessible all the times. So target executables have to be compiled with **-host nohost** or a dummy version of the host_comm.o have to be linked if **-host Windows** is used. Also note that TriMedia DLLs will not work since loading DLLs require file I/O.

Speculative Load Fix

The TriMedia compiler supports speculative loading—values in registers are used as pointers and are dereferenced to load data from memory in advance.

Speculative loading can happen from SDRAM MMIO or over the PCI bus. On certain Intel 440LX and 440BX Pentium II machines, load access to PC memory, adapter memory (like VGA frame buffer), or unclaimed PCI physical address space, across the PCI bus causes the machine to lock up.

The TriMedia processor has a hardware feature that allows all PCI access (expect SDRAM and MMIO accesses) to be disabled. This feature can be enabled and disabled at run time.

The TriMedia Manager disables PCI accesses at startup. When the TriMedia Manager, however, needs to access PC memory for host communication functions, it enables PCI accesses, performs the accesses and then disables PCI accesses.

Similarly, user programs that allocate and use shared memory should use the **pciMemoryReadUIntXX** and **pciMemoryWriteUIntXX** functions to access shared memory on the host, instead of reading and writing memory by de-referencing pointers directly. These functions are documented in TCS\include\tm1\tmPCI.h.

The **SpeculativeLoadFix** option is turned OFF by default. It can be turned ON by the following registry entry.

```
[HKEY_LOCAL_MACHINE\\SOFTWARE\\PhilipsSemiconductors\\TriMedia\\TMMan]
"SpeculativeLoadFix"=dword:00000001
```

WARNING
Current applications using shared memory will break if this fix is turned on.

Big-Endian Execution

The TriMedia processor can execute in both big-endian and little-endian modes. TMMan supports execution of little endian as well as big endian executables on the TriMedia processor.

If a PC-hosted TriMedia processor executes in big endian mode, every access made to shared memory must be swapped to maintain data consistency because the PC's host processor always executes in little endian mode.

The TriMedia processor always accesses data in native endianess (big endian or little endian mode). It is the host application's responsibility to swap data types if the target is running in a different endianess from that of the host. The sample programs: memory, message, tmapi, and sgbuffer, take care of endianess issues using macros for swapping data.

To execute big endian examples, set the following flag in %windir%\\tmman.ini:

```
[TMMan]
Endian=1 ; LITTLE Endian - This is the default
Endian=0 ; BIG Endian
```

The machine does not have to be rebooted after this change.

WinCE Issues

Under other Win32 platforms like WinNT, Win95, and Win98, Kernel Mode drivers can signal user mode events via handles.

Under WinCE, however, TMMan Drivers run in user mode within the driver .exe process. The only way to set an event in user mode is by obtaining a handle to the event via the same name that the user application used to create the event. For this reason, applications that use TMMan Event and Messaging Interfaces and run under WinCE have to take the following into considerations:

- All Win32 events must be named events.
- All event names must be unique, even across applications because events are created in the global Win32 namespace.

Note
These restrictions do *not* apply to other Win32 platforms.

Synchronization Flags

`constTMManModuleHostKernel`	Indicates that the Host module calling the required function is running in Kernel Mode. If this flag is specified TMMan interprets the `SynchronizationHandle` parameter as a handle to a Win32 Synchronization Object. Typically WinNT/Win9X Device Drivers.

`constTMManModuleHostUser`	Indicates that the Host module calling the required function running in User Mode. Typically WinNT/Win9X/WinCE DLLs or Applications.
`constTMManModuleTargetKernel`	On the target there is no distinction between user and kernel mode. If this flag is specified TMMan interprets **SynchronizationHandle** as an **AppSem** type of synchronization object.
`constTMManModuleTargetUser`	On the target there is no distinction between user and kernel mode. If this flag is specified TMMan interprets **SynchronizationHandle** as an **AppSem** type of synchronization object.

Porting Guidelines

The TriMedia Manager API has changed considerably from the previous release. The new interface is currently available under Windows NT only. In future releases, it will be available under other platforms like Windows 95, Windows 98, and Windows CE. Due to the changes to the TriMedia Manager API, existing applications that use the existing Windows 95 TMMan API will have to be ported to the new TMMan interface.

This document describes the issues involved in porting applications from the old to the new interface.

Inter-processor Messaging and Event API

Under the old TMMan API, the tmMessageCreate function, both on the host and the target, required the caller to specify a callback function. On the host this callback would be called from a high priority thread (within TMMan) when a message arrived from the target. On the target TMMan would invoke this callback from within the ISR. This callback function was called with a pointer to the packet that arrived. This was a push mechanism and in most applications the caller had to insert the packet into a temporary queue from within the callback and process the packet later.

Under the new TMMan API, the tmmanMessageCreate function, both on the host and the target, expects the caller to pass in a handle to an operating system object that can be signaled. On the host callers allocate a Win32 event and pass a handle to that event. When a message arrives from the target this event is signaled. If the caller was blocked on this event, it would unblock. On the target side the caller creates an AppSem with a semaphore count of 1. The caller then claims the Semaphore via AppSem_p reducing the count to 0 and then passes the handle to this Semaphore to tmmanMessageCreate. The caller then does an AppSem_p again and blocks. On the arrival of a new packet from the host TMMan calls AppSem_v to increment the semaphore count to 1 unblocking the call to AppSem_p. The caller then makes repeated calls to tmmanMessageReceive to retrieve the incoming packets until such time that the tmmanMessageReceive call returns an error. At this point it can call the OS specific blocking function i.e. (WaitForSingleObject or AppSem_p) and wait for it to be signaled when the next message(s) arrive.

Inter-processor events are new to the TMMan API. They follow the same guidelines for blocking and signaling as the messaging APIs.

Object ID

The old TMMan API used numeric IDs to form co-relation between objects created on the host and the target. For example the host would create a message channel with an ID of 2. Similarly the target would create a message channel with an ID of 2. When the host would send a message to the target, it is would be received by the callback installed for message channel with the ID of 2.

Under the new TMMan API ASCII strings are used instead of IDs. These strings can be a maximum of 12 characters long. This decreases the probability of collisions when multiple applications, using the TMMan API, create multiple objects. Using this interface, the host will pass a unique ASCII string, e.g., **"MyObject123"**, while creating any TMMan object. The target side will use exactly the same string, **"MyObject123",** while creating the corresponding TMMan object on the target.

C Run Time

Under the old interface TMCons.exe was use as a C Run Time Server on the host. TMCons would handle and satisfy all requests that the target makes to POSIX level 2 calls like create, read, write, seek, fcntl, isatty, setmode, mktemp, and so forth. Also TMCons was automatically invoked by virtue of the host application calling tmDSPExecutableRun.

Under the new interface all C Run Time Dependencies has been removed from TMMan. There is a separate DLL TMCrt.dll that provides the C Run Time Server functionality. TMRun uses this DLL, TMMPRun and all other host applications that need to provide C Run Time support for the target executable. TMMon and TMGMon explicitly invoke TMRun for providing support to executables on the target. Host side applications written users do not need to call the TMCRT interface if the corresponding target executable is compiled with –host nohost linker flags.

Argument Passing

The host passes command Line arguments to the target executables.

Under the old TMMan API the host application passed these arguments to the tmDSPExecutableRun function which would internally pass it on to the target executable.

Under the new TMMan API the host application passes these arguments via the cruntimeXXX set of functions exported by TMCRT.dll.

Data Type Changes

The old TMMan interface used Windows specific data types like VOID, DWORD, WORD, and so forth, that made the header files non-portable to other platforms.

The new TMMan interface uses TriMedia standard Data types such as **UInt32**, **UInt16**, **UInt8**, and so forth, which are defined in tmtypes.h.

Shared Memory Allocation

The tmShmemAllocate function, of the old TMMan interface, would return the Physical address of the shared memory. This address would be passed to the target side using messages or the **tmParameterDWORDSet** function. The target could access the shared memory directly.

The new TMMan API requires a object name to be assigned to every shared memory buffer allocated. The host does not receive a physical address. The target side can retrieve the address of the shared memory block by using the same object name and calling the tmmanSharedMemory*xxx* set of calls.

Scatter Gather Locking

The tmBufferPrepare function, of the old TMMan interface, would return a physical address that the host would communicate to the target via messages or via the tmParameterDWORDSet function. The target would pass this physical address directly to **tmSG*xxx*** set of calls.

Under the new interface the **tmmanSGBufferCreate** call requires an unique object name at the time of page locking the buffer on the host. By the same token the **tmmanSGBufferOpen** call on the target requires the same object name. The **tmmanSGBufferOpen** function returns a handle which is then passed to the **tmmanSGBuffer***xxx* set of calls on the target.

Dynamic Task Downloading

The Dynamic Task Downloading API has been removed from the new TMMan interface as this was introducing Operating System specific functionality within TMMan. Host initiated dynamic task downloading can be implemented using shared memory and message passing functionality. An example program that demonstrates how to do this will be provided in a later release.

Get/Set Parameters

The **tmParameterDWORD*xxx*** Parameter APIs have been removed from the new interface. The introduction of the object name space and the shared memory API obviates the need for these APIs.

TMManager Data Structures

This section presents the TMManager data structures.

 © 1999 Philips Semiconductors 10/08/99

tagtmmanPacket

```
typedef struct tagtmmanPacket{
   UInt32   Argument[constTMManPacketArgumentCount];
   UInt32   Reserved;
} tmmanPacket;
```

Fields

`Argument`	Array containing application specific arguments that TMMan does not modify or interpret.
`Reserved`	Do not use this field. TMMan overwrites it.

Description

Specifies the packet to be used by TMMan.

tagtmmanVersion

```
typedef struct tagtmmanVersion{
   UInt32   Major;
   UInt32   Minor;
   UInt32   Build;
} tmmanVersion;
```

Fields

`Major`	Major version number of the specified TMMan component.
`Minor`	Minor version number of the specified TMMan component.
`Build`	Build version number of the specified TMMan component.

Description

Specifies the version (Major, Minor, or Build) of TMMan.

© 1999 Philips Semiconductors 10/08/99

tagtmmanMemoryBlock

```
typedef struct tagtmmanMemoryBlock{
   UInt32   MappedAddress;
   UInt32   PhysicalAddress;
   UInt32   Size;
} tmmanMemoryBlock;
```

Fields

`MappedAddress`	Operating System Mapped Address corresponding to **PhysicalAddress**.
`PhysicalAddress`	Physical address of the SDRAM or MMIO Window.
`Size`	Size of the SDRAM or MMIO Window.

Description

Specifies the Operating System Mapped Address, and the SDRAM or MMIO Window Physical Address.

tagtmmanDSPInfo

```
typedef struct tagtmmanDSPInfo{
   tmmanMemoryBlock  SDRAM;
   tmmanMemoryBlock  MMIO;
   UInt32            TMClassRevisionID;
   UInt32            TMSubSystemID;
   UInt32            DSPNumber;
   UInt32            TMDeviceVendorID;
   UInt32            BridgeDeviceVendorID;
   UInt32            BridgeClassRevisionID;
   UInt32            BridgeSubsystemID;
   UInt32            Reserved[8];
} tmmanDSPInfo;
```

Fields

`SDRAM`	Address information about SDRAM.
`MMIO`	Address information about MMIO.
`TMClassRevisionID`	TriMedia PCI Class and Revision ID for CPU version.
`TMSubSystemID`	TriMedia PCI Subsystem & Subsystem Vendor ID—same as Board Revision.
`DSPNumber`	DSP Number that depends on the order this device was found on the PCI bus.
`TMDeviceVendorID`	TriMedia PCI Device and Vendor ID—TM1*xxx* / TM2*xxx* support.
`BridgeDeviceVendorID`	Bridge PCI Device and Vendor ID—non-transparent bridge support.
`BridgeClassRevisionID`	Bridge PCI Class and Revision ID for CPU version.
`BridgeSubsystemID`	Bridge PCI Subsystem & Subsystem Vendor ID—non-transparent bridge support.
`Reserved`	Reserved for future use.

Description

This structure contains PCI-specific information about the TriMedia device of the bridge device.

 © 1999 Philips Semiconductors 10/08/99

TMManager General Functions

This section describes the general TMManager functions.

Category	Name	Page
General type	tmmanGetErrorString	374
	tmmanNegotiateVersion	375
	tmmanNegotiateVersion	375
	tmmanMappedToPhysical	376
	tmmanPhysicalToMapped	376
	tmmanValidateAddressAndLength	377
	tmmanTranslateAdapterAddress	378
DSP Interfaces	tmmanDSPGetNum	379
	tmmanDSPGetInfo	379
	tmmanDSPGetStatus	380
	tmmanDSPMapSDRAM	381
	tmmanDSPUnmapSDRAM	382
	tmmanDSPGetEndianess	383
	tmmanDSPOpen	384
	tmmanDSPClose	385
	tmmanDSPLoad	386
	tmmanDSPStart	387
	tmmanDSPStop	388
	tmmanDSPReset	389

tmmanGetErrorString

```
Int8* tmmanGetErrorString(
   TMStatus   StatusCode
);
```

Parameters

`StatusCode`	Status code that needs to be converted to a string.

Return

`Int8*`	(Pointer to a) Null-terminated string describing the error.

Description

Returns the string corresponding to the specified error code.

 © 1999 Philips Semiconductors 10/08/99

tmmanNegotiateVersion

```
TMStatus tmmanNegotiateVersion(
   UInt32          ModuleID,
   tmmanVersion   *Version
);
```

Parameters

`ModuleID`	Module Identification of the TMMan component whose version needs to be verified. Possible values of this parameter are:
	`constTMManModuleHostKernel`
	`constTMManModuleHostUser`
	`constTMManModuleTargetKernel`
	`constTMManModuleTargetUser`

`Version`	Pointer to the TMMan version structure with its Major and Minor fields filled up.

Return Codes

`statusMajorVersionError`	Caller provided a major version that is less than the major version of the given module.
`statusMinorVersionError`	Caller provided a minor version that is less than the minor version of the given module.
`statusUnknownComponent`	Caller-provided **ModuleID** was outside the range supported on this platform such as when this function is called on the host with **constTMMan-ModuleTargetKernel** or **constTMManModule-TargetUser**.

Description

Called by the application to perform a version negotiation with the different components of TMMan. The application should fill up the fields of the version structure with the constTMManDefaultVersion*xxx* constants defined in this file, before calling the function. If TMMan cannot handle the version passed in this structure it will return an error. Otherwise it will return a success status. Note that both in case of a failure or success TMMan will write its current version information in the structure pointed to by the version parameter. An application can restrict itself to run with a *specific version* of TMMan by doing either of the following:

- *Not proceeding* if this function returns failure.
- *Not proceeding* based on the TMMan version returned by this function.

tmmanMappedToPhysical

```
UInt32 tmmanMappedToPhysical(
   tmmanMemoryBlock   *MemoryBlock,
   UInt32              MappedAddress
);
```

Parameters

`MemoryBlock`	Pointer to a SDRAM or MMIO memory block structure, that will be used for translating the address. The contents of this structure can be retrieved by calling **tmmanDSPGetInfo**.
`MappedAddress`	The platform specific translated (mapped) address.

Description

Translates an Operating System Mapped SDRAM or MMIO address to a physical address and returns it (the physical address). This function translates SDRAM and MMIO addresses only.

tmmanPhysicalToMapped

```
UInt32 tmmanPhysicalToMapped(
   tmmanMemoryBlock   *MemoryBlock,
   UInt32              PhysicalAddress
);
```

Parameters

`MemoryBlock`	Pointer to a SDRAM or MMIO memory block structure, that will be used for translating the address. The contents of this structure can be retrieved by calling **tmmanDSPGetInfo**.
`PhysicalAddress`	The platform-specific MMIO or SDRAM physical address.

Description

Translates a SDRAM or MMIO physical address to an Operating System mapped address and returns it.

 © 1999 Philips Semiconductors 10/08/99

tmmanValidateAddressAndLength

```
Bool tmmanValidateAddressAndLength(
   tmmanMemoryBlock   *MemoryBlock,
   UInt32              Address,
   UInt32              Length
);
```

Parameters

`MemoryBlock`	Pointer to a SDRAM or MMIO memory block structure, that will be used for translating the address. The contents of this structure can be retrieved by calling **tmmanDSPGetInfo**.
`Address`	Physical address that needs to be checked.
`Length`	Length of the block that needs to be checked.

Return Codes

`True`	If the address and length describes a block lying *within* the range specified by **MemoryBlock**.
`False`	If the address and length describes a block lying *outside* the range specified by **MemoryBlock**.

Description

Checks if the given physical address and length lies within the definable limits of the given memory block. This function works for SDRAM and MMIO addresses only.

tmmanTranslateAdapterAddress

```
Bool  tmmanTranslateAdapterAddress (
   UInt32   MappedAddress,
   UInt32   Length,
   UInt32  *PhysicalAddressPtr
);
```

Parameters

`MappedAddress`	OS Mapped memory address that needs to be translated.
`Length`	Length of the block that needs to be translated. Does not need to encompass the entire memory range.
`PhysicalAddressPtr`	Address of the memory location where the translated physical address will be stored.

Return Codes

`True`	Address and length translated successfully to an adapter physical address.
`False`	Address translation failed.

Description

Uses the TMMan Kernel Mode Driver to translate an adapter-mapped address to a physical address that can be accessed by the TM processor.

NOTE
This function can only be used to translate physical adapter memory addresses (physical memory that is guaranteed to be page locked and contiguous). Because the address range is assumed to be contiguous, the length of memory range passed to this function does not have to be the entire range of memory that needs to be accessed.

 © 1999 Philips Semiconductors 10/08/99

tmmanDSPGetNum

```
UInt32 tmmanDSPGetNum( void );
```

Parameters

None.

Description

Returns the number of TriMedia processors installed in the system.

tmmanDSPGetInfo

```
TMStatus tmmanDSPGetInfo(
   UInt32          DSPHandle,
   tmmanDSPInfo   *DSPInfo
);
```

Parameters

DSPHandle	Handle to the DSP returned by **tmmanDSPOpen**.
DSPInfo	Pointer to the structure where the TriMedia processor-related information will be returned.

Return Codes

statusInvalidHandle	Handle to the DSP is corrupted.

Description

Retrieves the properties of the specified TriMedia processor.

tmmanDSPGetStatus

```
TMStatus tmmanDSPGetStatus(
   UInt32   DSPHandle,
   UInt32  *StatusFlags
);
```

Parameters

`DSPHandle`	Handle to the DSP returned by **tmmanDSPOpen**.
`StatusFlags`	Pointer to the location where the status flags will be stored. The status flags can be one of the following: **constTMManDSPStatusUnknown**: TMMan cannot determine the state of the TriMedia processor. **constTMManDSPStatusReset**: TriMedia processor is in a reset state, so it is not running. **constTMManDSPStatusRunning**: TriMedia processor is in a running state.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted.
`statusUnsupportedOnThisPlatform`	If this function is called on the target.

Description

Returns the current state of the specified TriMedia Processor.

 ©1999 Philips Semiconductors 10/08/99

tmmanDSPMapSDRAM

```
TMStatus  tmmanDSPMapSDRAM (
   UInt32   DSPHandle
);
```

Parameters

`DSPHandle`	Handle to the DSP returned by **tmmanDSPOpen**.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted.
`statusOutOfVirtualAddresses`	There are no more free Page Table Entries to map this memory.
`statusUnsupportedOnThisPlatform`	If this function is called on the target.

Description

Maps SDRAM into the Operating System and Process virtual address space.

tmmanDSPUnmapSDRAM

```
TMStatus  tmmanDSPUnmapSDRAM (
   UInt32   DSPHandle
);
```

Parameters

`DSPHandle`	Handle to the DSP returned by **tmmanDSPOpen**.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted.
`statusUnsupportedOnThisPlatform`	If this function is called on the target.

Description

Unmaps SDRAM from Process virtual address space. If all instances of SDRAM for this processor have been unmapped, the OS mapping is also undone.

Note
tmmanDSPMapSDRAM and **tmmanDSPUnmapSDRAM** must be called in pairs.

 ©1999 Philips Semiconductors 10/08/99

tmmanDSPGetEndianess

```
TMStatus  tmmanDSPGetEndianess (
   UInt32    DSPHandle,
   UInt32   *EndianessFlags
);
```

Parameters

`DSPHandle`	Handle to the DSP returned by **tmmanDSPOpen**.

`EndianessFlags`	Pointer to the location where the endianess flags will be stored.
	The endianess flags can be one of the following.
	`constTMManEndianessUnknown`
	`constTMManEndianessLittle`
	`constTMManEndianessBig`

Return

`statusInvalidHandle`	Handle to the DSP is corrupted.
`statusUnsupportedOnThisPlatform`	If this function is called on the target.

Description

Gets the current endianess of the specified TriMedia Processor.

tmmanDSPOpen

```
TMStatus tmmanDSPOpen(
   UInt32    DSPNumber,
   UInt32*   DSPHandlePointer
);
```

Parameters

`DSPNumber`	Number of the TriMedia processor that needs to be opened. Note that this count reflects the order in which the TriMedia processor was detected by tmman. This is generally dependent on the PCI slot in which the TriMedia board is sitting.
`DSPHandlePointer`	Address of the memory location where the handle to the DSP will be stored. All future references to the board have to be made via the handle.

Return Codes

`statusDSPNumberOutofRange`	The DSPNumber parameter does not lie within **0** and **tmmanDSPGetNum**–1.

Description

Opens the given TriMedia Processor. This call simply increments an internal reference count. It does not perform physical detection of the processor. All TriMedia processors are detected when TMMan is loaded.

 ©1999 Philips Semiconductors 10/08/99

tmmanDSPClose

```
TMStatus tmmanDSPClose(
   UInt32   DSPHandle
);
```

Parameters

`DSPHandle`	Handle to the TriMedia processor returned by **tmmanDSPOpen**.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted.

Description

Closes the given handle to the TriMedia processor. This call decrements an internal reference count. The caller will be able to use the handle even after closing it. The handle to the DSP remains valid as long as the TriMedia processor to which the handle refers exists in the system.

tmmanDSPLoad

```
TMStatus tmmanDSPLoad(
   UInt32   DSPHandle,
   UInt32   LoadAddress,
   UInt8*   ImagePath
);
```

Parameters

`DSPHandle`	Handle to the TriMedia processor returned by **tmmanDSPOpen**.
`LoadAddress`	Address of SDRAM where the executable should be downloaded. To use the default values use **constTMManDefault**.
`ImagePath`	Path to the executable file image. This image should have a boot image, not a task.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted or DSP has already been closed.
`statusUnsupportedOnThisPlatform`	If this function is called on the target.
`statusOutOfVirtualAddresses`	There are no more free Page Table Entries to map SDRAM for image download.
`statusExecutableFileWrongEndianness`	The endianess of the executable file is not the same as that specified in the INI file or registry.
`statusDownloaderXXX`	Range of TMDownloader error codes. For explanation of these error codes refer to TMDownloader.h.

Description

Loads a boot image on to the DSP. This image must be compiled with the **-btype boot** flag.

 © 1999 Philips Semiconductors 10/08/99

tmmanDSPStart

```
TMStatus tmmanDSPStart(
   UInt32   DSPHandle
);
```

Parameters

`DSPHandle`	Handle to the TriMedia processor returned by **tmmanDSPOpen**.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted.
`statusUnsupportedOnThisPlatform`	If this function is called on the target.

Description

Unresets the DSP. The DSP starts executing code at SDRAM base. See *C Run Time* on page 366.

tmmanDSPStop

```
TMStatus tmmanDSPStop(
   UInt32   DSPHandle
);
```

Parameters

`DSPHandle`	Handle to the TriMedia processor returned by **tmmanDSPOpen**.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted.
`statusUnsupportedOnThisPlatform`	If this function is called on the target.

Description

Puts the CPU into a reset state. Resets all the peripherals via MMIO registers. Resets shared data structures that TMMan uses across the bus.

 ©1999 Philips Semiconductors 10/08/99

tmmanDSPReset

```
TMStatus tmmanDSPReset(
   UInt32   DSPHandle
);
```

Parameters

`DSPHandle`	Handle to the TriMedia processor returned by **tmmanDSPOpen**.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted.
`statusUnsupportedOnThisPlatform`	If this function is called on the target.

Description

Initializes the TriMedia Processor after it has been manually reset by the reset button or other means.

This function can additionally preform a hardware reset of the TriMedia processor provided the necessary hardware modifications have been made to the IREF board.

TMManager Message Interface Functions

This section presents the Message Interface TMManager functions.

 © 1999 Philips Semiconductors 10/08/99

tmmanMessageCreate

```
TMStatus tmmanMessageCreate(
   UInt32   DSPHandle,
   UInt8    Name,
   UInt32   SynchronizationHandle,
   UInt32   SynchronizationFlags,
   UInt32  *MessageHandlePointer
);
```

Parameters

`DSPHandle`	Handle to the TriMedia processor returned by **tmmanDSPOpen**.
`Name`	Unique caller-supplied name for this message channel. See *Object Names* on page 361.
`SynchronizationHandle`	Pointer to OS-specific synchronization object. See *Synchronization Handle* on page 360.
`SynchronizationFlags`	This parameter describes how TMMan should interpret the **SynchronizationHandle** parameter. See *Synchronization Flags* on page 364.
`MessageHandlePointer`	Address of the location where the pointer to the message channel will be stored in tmmanapi.h.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted.
`statusObjectAllocFail`	Object memory allocation failed.
`statusObjectListAllocFail`	No more message channels free.
`statusNameSpaceNoMoreSlots`	Out of name space slots; internal error.
`statusNameSpaceLengthExceeded`	Name is more than 12 characters.
`statusNameSpaceNameConflict`	The user-assigned name already exists in TMMan name space.
`statusSynchronizationObjectCreateFail`	The synchronization flags were invalid or memory could not be allocated for the Synchronization object.
`statusQueueObjectCreateFail`	Creation of the queue to buffer incoming packets failed.

Description

Creates a bidirectional message channel between the host and the target processor. This message channel can be used to send fixed size packets of type **tmmanPacket** from one processor to another. The message packets are copied across the PCI bus via shared mailboxes. Every message channel has its own private queue where incoming packets from the other processor are temporarily buffered.

When a packet arrives from the other processor the caller supplied OS synchronization object will be signalled. The caller can use native OS primitives to block on this object or on multiple objects as required. Note, however, that due to the relative speed of the two processors there may not be a one to one correspondence between the number of times the object is signalled and the number of packets in the incoming queue.

 © 1999 Philips Semiconductors 10/08/99

tmmanMessageDestroy

```
TMStatus  tmmanMessageDestroy(
   UInt32   MessageHandle
);
```

Parameters

`MessageHandle`	Handle to the message channel returned by **tmmanMessageCreate**.

Return Codes

`statusInvalidHandle`	Handle to the message channel is corrupted or has already been closed.

Description

Closes the message channel handle returned by **tmmanMessageCreate.** Only the message channel and the queue are freed. The caller is responsible for freeing the OS synchronization object that was supplied at the time of **tmmanMessageCreate**.

tmmanMessageSend

```
TMStatus  tmmanMessageSend(
   UInt32   MessageHandle,
   void    *DataPointer
);
```

Parameters

`MessageHandle`	Handle to the message channel returned by **tmmanMessageCreate**.
`DataPointer`	Pointer to the **tmmanPacket** data structure. Once this call returns successfully the data structure can be reused.

Return Codes

`statusInvalidHandle`	Handle to the message channel is corrupted or has already been closed.
`statusChannelMailboxFullError`	The interprocessor mailbox is temporarily full, this is a temporary condition. The user is supposed to retry the call only when this error code is returned. See the Implementation Notes below.

Description

This function sends a fixed size data packet of type **tmmanPacket** to the peer processor. This functions returns an error if there is no space in the interprocessor mailbox to send packets. However this may be a temporary condition and caller should retry sending the packet after a timeout. Packets on a certain message channel are guaranteed to arrive in order on the peer processor.

Implementation Notes

Regarding the error code, **statusChannelMailboxFullError**, the caller should not do the following:

```
while ( tmmanMessageSend(Handle,&Packet) != statusSuccess ){}
```

Rather, the caller should do the following:

```
while( tmmanMessageSend(Handle,&Packet) ==
                          statusChannelMailboxFullError){}
```

 © 1999 Philips Semiconductors 10/08/99

tmmanMessageReceive

```
TMStatus  tmmanMessageReceive(
   UInt32   MessageHandle,
   void    *DataPointer
);
```

Parameters

`MessageHandle`	Handle to the message channel returned by **tmmanMessageCreate**.
`DataPointer`	Pointer to the **tmmanPacket** data structure. If this call succeeds the **tmmanPacket** structure contains a valid packet.

Return Codes

`statusInvalidHandle`	Handle to the message channel is corrupted or has already been closed.
`statusInvalidHandle`	Handle to the message channel is corrupted or has already been closed.
`statusMessageQueueEmptyError`	There are no pending packets in the incoming message queue for this message channel.

Description

This function retrieves a packet from the incoming packet queue. This is a non-blocking function, so if there are no packets in the queue this function returns immediately with an error code. A synchronization object may be signalled one for multiple packets. The caller should call this function repeatedly, until it fails, to retrieve all packets that have arrived.

TMManager Event Functions

This section presents the TMManager Event functions.

 © 1999 Philips Semiconductors 10/08/99

tmmanEventCreate

```
TMStatus tmmanEventCreate(
   UInt32   DSPHandle,
   UInt8*   Name,
   UInt32   SynchronizationHandle,
   UInt32   SynchronizationFlags,
   UInt32  *EventHandlePointer
);
```

Parameters

`DSPHandle`	Handle to the TriMedia processor returned by **tmmanDSPOpen**.
`Name`	Unique caller-supplied name for this event. See *Object Names* on page 361.
`SynchronizationHandle`	Pointer to OS-specific synchronization object. See Synchronization Handle on page 360.
`SynchronizationFlags`	Describes how TMMan should interpret the **SynchronizationHandle** parameter. See Synchronization Flags on page 364.
`EventHandlePointer`	Address of the location where the pointer to the event will be stored.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted.
statusObjectAllocFail	Object memory allocation failed.
`statusObjectListAllocFail`	No more events free.
`statusDeviceIoCtlFail`	Internal Error.
`statusNameSpaceNoMoreSlots`	Out of name space slots, internal error.
`statusNameSpaceLengthExceeded`	Name is more than 12 characters.
`statusNameSpaceNameConflict`	The user-assigned name already exists in TMMan name space.
`statusSynchronizationObjectCreateFail`	The synchronization flags were invalid or memory could not be allocated for the Synchronization object.

Description

Events provide an interprocessor signalling mechanism. It enables one processor to signal an event that will cause another processor to unblock if it is waiting for that event. The caller of this function should use the native OS dependent Synchronization primitives to create a OS synchronization object, and pass the pointer to that object to this function. Due to the relative speeds of the two processors, there may not be one-to-one correspondences between the number of times one processor signals the event and the number of times the event gets signalled on the peer processor.

 © 1999 Philips Semiconductors 10/08/99

tmmanEventSignal

```
TMStatus tmmanEventSignal(
   UInt32   EventHandle
);
```

Parameters

`EventHandle`	Handle to the event returned by **tmmanEvent-Create**.

Return Codes

`statusInvalidHandle`	Handle to the event is corrupted or has already been closed.

Description

This function signals the event object causing the OS synchronization object on the peer processor to be signaled.

tmmanEventDestroy

```
TMStatus tmmanEventDestroy(
   UInt32   EventHandle
);
```

Parameters

`EventHandle`	Handle to the event returned by **tmmanEvent-Create**.

Return Codes

`statusInvalidHandle`	Handle to the object is corrupted or has already been closed.

Description

Closes the **EventHandle** parameter and frees up the resources allocated by TMMan for this object. It is the caller's responsibility to free the OS synchronization object that was passed to the **tmmanEventCreate**function.

© 1999 Philips Semiconductors 10/08/99

tmmanSharedMemoryCreate

```
TMStatus tmmanSharedMemoryCreate(
   UInt32   DSPHandle,
   UInt8*   Name,
   UInt32   Length,
   UInt32  *AddressPointer,
   UInt32  *SharedMemoryHandlePointer
);
```

Parameters

`DSPHandle`	Handle to the TriMedia processor returned by **tmmanDSPOpen**.
`Name`	Unique caller-supplied name for this object. See *Object Names* on page 361.
`Length`	Length of the required shared memory block in bytes.
`AddressPointer`	Address of the memory location where the pointer to the shared memory will be stored. This pointer can be used by the host directly to access the allocated memory.
`SharedMemoryHandlePointer`	Address of the location where the handle to the shared memory will be stored. This handle is required to free this resource.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted.
statusObjectAllocFail	Object memory allocation failed.
`statusObjectListAllocFail`	No more shared memory slots free.
`statusNameSpaceNoMoreSlots`	Out of name space slots, internal error.
`statusNameSpaceLengthExceeded`	Name is more than 12 characters.
`statusNameSpaceNameConflict`	The user-assigned name already exists in TMMan name space.
`statusMemoryUnavailable`	No more shared memory available.
`statusUnsupportedOnThisPlatform`	If this function is called on the target.

Description

Allocates a block of shared memory and returns a pointer to the memory block. This memory is allocated out of contiguous, page locked memory on the host processor. Shared memory can only be allocated on the host but can be accessed from the target. Note that this is a very expensive system resource and should be used sparingly. The memory block returned is always aligned on a 32-bit boundary. TMMan allocates a region of shared memory for every board present in the system (at startup) and then sub-allocates blocks from this region when this function is called.

 © 1999 Philips Semiconductors 10/08/99

tmmanSharedMemoryDestroy

```
TMStatus tmmanSharedMemoryDestroy(
   UInt32   SharedMemoryHandle
);
```

Parameters

`SharedMemoryHandle`	Handle to the shared memory block returned by **tmmanSharedMemoryCreate**.

Return Codes

`statusInvalidHandle`	Handle to the object is corrupted.
`statusUnsupportedOnThisPlatform`	If this function is called on the target.

Description

Closes the **SharedMemoryHandle** parameter, and frees up the shared memory that was allocated via the call to **tmmanSharedMemroyCreate**. This function should be called by the host processor only.

tmmanSharedMemoryOpen

```
TMStatus tmmanSharedMemoryOpen(
   UInt32   DSPHandle,
   UInt8*   Name,
   UInt32  *LengthPointer,
   UInt32  *AddressPointer,
   UInt32  *SharedMemoryHandlePointer
);
```

Parameters

`DSPHandle`	Handle to the TriMedia processor returned by **tmmanDSPOpen**.
`Name`	Unique caller-supplied name for this object. See *Implementation Notes* following.
`LengthPointer`	Address of the memory location where the Length of the shared memory block identified by name will be stored.
`AddressPointer`	Address of the memory location where the pointer to the shared memory will be stored. This pointer can be used by the target directly to access the allocated memory.
`SharedMemoryHandlePointer`	Address of the location where the handle to the shared memory will be stored. This handle is required to free references to this resource.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted or DSP has already been closed.
`statusObjectAllocFail`	Object memory allocation failed.
`statusObjectListAllocFail`	No more shared memory slots free.
`statusNamesPacelengthExceeded`	Name is more than 12 characters.
`statusNameSpaceNameNonexistent`	The user-provided name does not exist in TMMan name space.
`statusUnsupportedOnThisPlatform`	If this function is called on the host.

Description

This function opens a handle to a shared memory resource created on the host. This function does not actually allocate any memory, it returns a handle to an existing shared memory block, that has been already allocated on the host. This function should be called on the target processor only.

 © 1999 Philips Semiconductors 10/08/99

Implementation Notes

1. **Name:** The counterpart of this object on the host/target should use the same name. TMMan uses this name internally to set up the shared data structures between the host and the target. The name should not exceed 12 characters. The name is case-sensitive. Names do not have to unique across objects; for example, an event and a message channel can use the same name.

tmmanSharedMemoryClose

```
TMStatus tmmanSharedMemoryClose(
   UInt32   SharedMemoryHandle
);
```

Parameters

`SharedMemoryHandle`	Handle to the event returned by **tmmanShared-MemoryOpen**.

Return Codes

`statusInvalidHandle`	Handle to the object is corrupted or has already been closed.
`statusUnsupportedOnThisPlatform`	If this function is called on the host.

Description

Closes the **SharedMemoryHandle** and frees up the resources allocated by TMMan for this object. This function does not free the shared memory. The shared memory has to be freed by the host. This function should be called from the target processor only.

 ©1999 Philips Semiconductors 10/08/99

TMManager Buffer Locking Functions

This section describes the Buffer Locking TMManager functions. The Buffer Locking functions are applicable to systems that support virtual memory. If a user allocates a buffer on one processor and needs the peer processor(s) to access this memory, the memory can not be paged out. Also, the peer processor needs to know the manner in which the memory is fragmented in the physical address space. These functions handle the above issues.

Category	Name	Page
Scatter Gather Buffer Locking	tmmanSGBufferCreate	408
	tmmanSGBufferDestroy	410
	tmmanSGBufferOpen	411
	tmmanSGBufferClose	412
	tmmanSGBufferFirstBlock	413
	tmmanSGBufferNextBlock	414
	tmmanSGBufferCopy	415

tmmanSGBufferCreate

```
TMStatus tmmanSGBufferCreate(
   UInt32   DSPHandle,
   UInt8    Name,
   UInt32   MappedAddress,
   UInt32   Size,
   UInt32   Flags,
   UInt32  *BufferHandlePointer
);
```

Parameters

DSPHandle	Handle to the TriMedia processor returned by **tmmanDSPOpen**.
Name	Unique caller-supplied name for this object. See Implementation Note 1, following.
MappedAddress	Host address of the block of memory that needs to be page locked. This parameter is typically the return value of **malloc**.
Size	Size of the memory in bytes.
Flags	See Implementation Note 2, following.
BufferHandlePointer	Address of the location where the handle to the page-locked memory will be stored. This handle is required to unlock the page-locked memory.

Return Codes

statusInvalidHandle	Handle to the DSP is corrupted.
statusObjectAllocFail	Object memory allocation failed.
statusObjectListAllocFail	No more shared memory slots free.
statusDeviceIoCtlFail	Internal Error.
statusNameSpaceNoMoreSlots	Out of name space slots, internal error.
statusNameSpaceLengthExceeded	Name is more than 12 characters.
statusNameSpaceNameConflict	The user-assigned name already exists in TMMan name space.
statusMemoryUnavailable	No more shared memory available to copy the page frame table.
statusUnsupportedOnThisPlatform	If this function is called on the target.

 ©1999 Philips Semiconductors 10/08/99

Description

Page locks the specified memory and generates a page frame table that can be used by the target to access the page-locked memory. This function is only supported on hosts that have virtual memory, and can only be called by the host processor.

Implementation Notes

1. **Name:** The counterpart of this object on the host/target should use the same name. TMMan uses this name internally to set up the shared data structures between the host and the target. The name should not exceed 12 characters. The name is case-sensitive. Names do not have to unique across objects; for example, an event and a message channel can use the same name.
2. **Flags** can have one or more the following values:

`constTMManSGBufferRead`	Buffer is going to read into (Incoming Data).
`constTMManSGBufferWrite`	Buffer is going to be written from (Outgoing Data).

tmmanSGBufferDestroy

```
TMStatus tmmanSGBufferDestroy(
   UInt32   BufferHandle
);
```

Parameters

`BufferHandle`	Handle to the event returned by **tmmanSharedMemoryOpen**.

Return Codes

`statusInvalidHandle`	Handle to the object is corrupted or has already been closed.
`statusUnsupportedOnThisPlatform`	If this function is called on the target.

Description

Closes the handle of the page-locked memory, unlocks the memory, and frees up the page frame tables. This function should be called by the host processor only.

 © 1999 Philips Semiconductors 10/08/99

tmmanSGBufferOpen

```
TMStatus tmmanSGBufferOpen(
   UInt32   DSPHandle,
   UInt8*   Name,
   UInt32  *EntryCountPointer,
   UInt32  *SizePointer,
   UInt32  *BufferHandlePointer
);
```

Parameters

`DSPHandle`	Handle to the TriMedia processor returned by **tmmanDSPOpen**.
`Name`	Unique caller-supplied name for this object. See *Implementation Notes* following.
`EntryCountPointer`	Address of the memory location where the count of the PTE entries is stored by this function.
`SizePointer`	Address of the memory location where the size of the buffer is stored.
`BufferHandlePointer`	Address of the location where the handle to the scatter gather buffer will be stored. This handle is required to free references to this resource.

Return Codes

`statusInvalidHandle`	Handle to the DSP is corrupted.
`statusObjectAllocFail`	Object memory allocation failed.
`statusObjectListAllocFail`	No more shared memory slots free.
`statusNameSpaceLengthExceeded`	Name is more than 12 characters.
`statusNameSpaceNameNonExistent`	The user-provided name does not exist in TMMan name space.
`statusUnsupportedOnThisPlatform`	If this function is called on the host.

Description

Opens a handle to the block of memory that was page-locked on the host. This function should only be called by the target processor.

Implementation Notes

1. **Name:** The counterpart of this object on the host/target should use the same name. TMMan uses this name internally to set up the shared data structures between the host and the target. The name should not exceed 12 characters. The name is case-sensitive. Names do not have to unique across objects; for example, an event and a message channel can use the same name.

tmmanSGBufferClose

```
TMStatus tmmanSGBufferClose(
   UInt32   BufferHandle
);
```

Parameters

`BufferHandle`	Handle to the buffer returned by **tmmanSGBufferOpen**.

Return Codes

`statusInvalidHandle`	Handle to the object is corrupted.
`statusUnsupportedOnThisPlatform`	If this function is called on the host.

Description

Closes the reference to the scatter gather page-locked memory. This function does not unlock the memory pages.This function should be called from the target processor only.

 © 1999 Philips Semiconductors 10/08/99

tmmanSGBufferFirstBlock

```
TMStatus tmmanSGBufferFirstBlock(
   UInt32   BufferHandle
   UInt32  *OffsetPointer,
   UInt32  *AddressPointer,
   UInt32  *SizePointer,
);
```

Parameters

`BufferHandle`	Handle to the buffer returned by **tmmanSGBuffer-Open**.
`OffsetPointer`	Address of the memory location (where the offset of the block from the beginning of the memory that was page-locked on the host) will be stored.
`AddressPointer`	Address of the memory location where the pointer to the memory block will be stored.
`SizePointer`	Address of the memory location where the size of the block will be stored.

Return Codes

`statusInvalidHandle`	Handle to the object is corrupted.
`statusUnsupportedOnThisPlatform`	If this function is called on the host.

Description

Returns the description of the first contiguous run of the page locked memory on the host. The description consists of the offset of the block from the beginning of the memory, pointer of the block that the target processor can use to access the memory, and the size of the block. A call has to be made to the **tmmmanSGBufferNextBlock** to get description of subsequent blocks.

tmmanSGBufferNextBlock

```
TMStatus tmmanSGBufferNextBlock(
   UInt32   BufferHandle,
   UInt32  *OffsetPointer,
   UInt32  *AddressPointer,
   UInt32  *SizePointer
);
```

Parameters

`BufferHandle`	Handle to the buffer returned by **tmmanSGBuffer-Open**.
`OffsetPointer`	Address of the memory location (where the offset of the block from the beginning of the memory that was page locked on the host) will be stored.
`AddressPointer`	Address of the memory location where the pointer to the memory block will be stored.
`SizePointer`	Address of the memory location where the size of the block will be stored.

Return Codes

`statusInvalidHandle`	Handle to the object is corrupted.
`statusSGBufferNoMoreEntries`	There are no more entries in the page frame table. To restart parsing of the page frame table, call **tmmanSGBufferFirstBlock** followed by calls to **tmmanSGBufferNextBlock**.
`statusUnsupportedOnThisPlatform`	If this function is called on the host.

Description

Returns the description of consecutive runs of contiguous memory from the page frametable referred to by **BufferHandle**. Note that **tmmanSGBufferFirstBlock** functions should be called at least once prior to calling this function.

 © 1999 Philips Semiconductors 10/08/99

tmmanSGBufferCopy

```
TMStatus tmmanSGBufferCopy(
   UInt32   BufferHandle,
   UInt32   Offset,
   UInt32   Address,
   UInt32   Size,
   UInt32   Direction
);
```

Parameters

`BufferHandle`	Handle to the buffer returned by **tmmanSGBuffer-Open**.
`Offset`	Offset from the beginning of memory where the copying has to start.
`Address`	Pointer to the buffer on the target processor where it will be copied to/from.
`Size`	Number of bytes to copy.
`Direction`	Direction of copy. For example, if TRUE, copy from host memory to target memory; if FALSE, copy from target to host memory.

Return Codes

`statusInvalidHandle`	Handle to the object is corrupted.
`statusSGBufferOffsetOutOfRange`	The offset supplied to this function is out of range of the page-locked host buffer.
`statusSGBufferSizeOutOfRange`	The size passed to this function is greater than the amount of page-locked memory available from the given offset.
`statusUnsupportedOnThisPlatform`	If this function is called on the host.

Description

The function copies the contents of the page-locked memory on the host to/from another block of memory on the target. It uses the C run time routine **memcpy** to perform the actual copying operation. If the caller needs the copying to be done via DMA transfer, then the **tmmanSGBufferFirstBlock** and **tmmanSGBufferNextBlock** should be used instead.

TMManager Debugging Functions

This section presents the debugging TMManager functions.

 © 1999 Philips Semiconductors 10/08/99

tmmanDebugDPBuffers

```
TMStatus tmmanDebugDPBuffers(
   UInt32    DSPHandle,
   UInt8   **FirstHalfPtr,
   UInt32   *FirstHalfSizePtr,
   UInt8   **SecondHalfPtr,
   UInt32   *SecondHalfSizePtr
);
```

Parameters

`DSPHandle`	Handle to the DSP returned by **tmmanDSPOpen**.
`FirstHalfPtr`	Address of the memory location where the pointer to the first half of the buffer will be stored.
`FirstHalfSizePtr`	Address of the memory location where the size of the first half buffer will be stored.
`SecondHalfPtr`	Address of the memory location where the pointer to the second half of the buffer will be stored.
`SecondHalfSizePtr`	Address of the memory location where the size of the second half buffer will be stored.

Return Codes

`statusInvalidHandle`	Handle to the object is corrupted.
`statusDebugNoPeerDebugInformation`	This function scans through the entire SDRAM to search for a magic header that identifies valid debug information. This error code denotes that the magic header does not exist or had been corrupted.
`statusUnsupportedOnThisPlatform`	If this function is called on the host.

Description

This function retrieves pointer to the circular wrap around buffers, where the TriMedia processor dumps debug messages. This function is current callable only from the host and it retrieves debug information generated by the TriMedia processor. Debug information printed via the DP macros are retrieved via this function. See *Debug Buffer Pointers* on page 362.

tmmanDebugHostBuffers

```
TMStatus tmmanDebugHostBuffers(
   UInt8   **FirstHalfPtr,
   UInt32   *FirstHalfSizePtr,
   UInt8   **SecondHalfPtr,
   UInt32   *SecondHalfSizePtr
);
```

Parameters

`FirstHalfPtr`	Address of the memory location where the pointer to the first half of the buffer will be stored.
`FirstHalfSizePtr`	Address of the memory location where the size of the first half buffer will be stored.
`SecondHalfPtr`	Address of the memory location where the pointer to the second half of the buffer will be stored.
`SecondHalfSizePtr`	Address of the memory location where the size of the second half buffer will be stored.

Return Codes

`statusNotImplemented`	This function will be implemented in a future release. Currently all TMMan (host) debug messages are printed to the host debugger (WinDBG or NTIce).
`statusUnsupportedOnThisPlatform`	If this function is called on the target.

Description

This function retrieves pointer to the circular wrap around buffers, where the host processor dumps debug messages. This function is current callable only from the host and it retrieves debug information generated by the host component of TMMan. The are no application callable functions that can dump data into these buffers. TMMan(host) uses this buffer to print internal debug information. See *Debug Buffer Pointers* on page 362.

 © 1999 Philips Semiconductors 10/08/99

tmmanDebugTargetBuffers

```
TMStatus tmmanDebugTargetBuffers(
   UInt32    DSPHandle,
   UInt8   **FirstHalfPtr,
   UInt32   *FirstHalfSizePtr,
   UInt8   **SecondHalfPtr,
   UInt32   *SecondHalfSizePtr
);
```

Parameters

`DSPHandle`	Handle to the DSP returned by **tmmanDSPOpen**.
`FirstHalfPtr`	Address of the memory location where the pointer to the first half of the buffer will be stored.
`FirstHalfSizePtr`	Address of the memory location where the size of the first half buffer will be stored.
`SecondHalfPtr`	Address of the memory location where the pointer to the second half of the buffer will be stored.
`SecondHalfSizePtr`	Address of the memory location where the size of the second half buffer will be stored.

Return Codes

`statusInvalidHandle`	Handle to the object is corrupted.
`statusDebugNoPeerDebugInformation`	This function scans through the entire SDRAM to search for a magic header that identifies valid debug information. This error code denotes that the magic header do not exist or has been corrupted.
`statusUnsupportedOnThisPlatform`	If this function is called on the target.

Description

This function retrieves pointer to the circular wrap around buffers, where the target processor dumps debug messages. This function is current callable only from the host and it retrieves debug information generated by the target component of TMMan. Applications running on the target can call the tmmanDebugPrintf function to print information into these buffers. TMMan(target) uses this buffer to print internal debug information. See *Debug Buffer Pointers* on page 362.

tmmanDebugPrintf

```
UInt32 tmmanDebugPrintf(
   UInt8   *Format,
   ...
);
```

Parameters

`Format`	**printf** style format specifier.
`...`	**printf** style arguments.

Return

Number of items printed.

Description

This function is used to print formatted strings via the debugging subsystem of TMMan. The implementation of this function is platform specific. On the host this functions prints out strings to the debug windows. On the target this function prints strings to the debug trace buffers. The maximum length of the string can be 1024 bytes. Applications on the TriMedia processor should use the DP macros to print debugging information.

©1999 Philips Semiconductors 10/08/99

TMManager C Runtime Server

This section describes the structures and functions of the tmcrt.h header file.

©1999 Philips Semiconductors 10/08/99

tagCRunTimeParameterBlock

```
typedef struct tagCRunTimeParameterBlock{
   UInt32   OptionBitmap;
   UInt32   StdInHandle;
   UInt32   StdOutHandle;
   UInt32   StdErrHandle;
   UInt32   WindowSize;
   UInt32   CRTThreadCount;
   UInt32   SynchronizationObject;
   UInt32   VirtualNodeNumber;
} CRunTimeParameterBlock;
```

Fields

`OptionBitmap`	Options Flags (see Table 1 following).
`StdInHandle`	Handle to the standard input device. Not interpreted if **constCRunTimeFlagsNoConsole** is set. Has to be a valid Win32 handle, *not* a **FILE***, and *not* a file handle returned by **open**.
`StdOutHandle`	Handle to the standard output device. Has to be a valid Win32 handle, *not* a **FILE***, and *not* a file handle returned by **open**.
`StdErrHandle`	Handle to the standard *input* device. Has to be a valid Win32 handle, *not* a **FILE***, and *not* a file handle returned by **open**.
`WindowSize`	Number of lines in the console window. Interpreted only if **constCRunTimeFlagsUseWindowSize** is set.
`CRTThreadCount`	Number of threads that are created to serve this node. Currently *not* used.
`SynchronizationObject`	Handle to the Win32 Event that is signalled when the target exits normally. Interpreted only if **constCRunTimeFlagsUseSynchObject** is set.
`VirtualNodeNumber`	Should be **0** for the first call to **cruntimeCreate** in a process context. The value of this parameter should be incremented for each subsequent call.

 © 1999 Philips Semiconductors 10/08/99

Table 1 Options Flags

Flag	Description
constCRunTimeFlagsIgnoreParams 0x0001	Ignore all fields of the CRunTimeParameterBlock.
constCRunTimeFlagsAllocConsole 0x0002	Create a new console. This has to be used by Windows GUI applications only.
constCRunTimeFlagsNoConsole 0x0004	stdXXX should be ignored. No console windows will popup.
constCRunTimeFlagsUseWindowSize 0x0008	interpret the WindowSize field of the CRunTimeParameterBlock structure.
constCRunTimeFlagsUseSynchObject 0x0010	Signal the Event whose handle is in SynchronizationObject—when target completes execution.
constCRunTimeFlagsNonInteractive 0x0020	TMCRT prints some status messages to stdout, this flag disables write to stdout and reads from stdin. So stdin / stdout /stderr accesses are performed only when requested by the target application.

cruntimeCreate

```
UInt32 cruntimeCreate(
   UInt32                  DSPNumber,
   UInt32                  ArgumentCount,
   UInt8                  *ArgumentVector[],
   CRunTimeParameterBlock  *Parameters,
   UInt32* CRTHandlePointer );
```

Parameters

`DSPNumber`	DSP Number that this server needs to serve. Should be the same value that is passed to **tmmanDSPOpen**.
`ArgumentCount`	Should include that target image name.
`ArgumentVector[]`	Pointer to an array of pointers pointing to arguments. The first argument has to be the name of the target executable.
`Parameters`	Pointer to a **CRunTimeParameterBlock** structure defining how the server should behave.
`CRTHandlePointer`	Address of the memory location where the handle to this instance of the server will be stored.

Returns

`True`	If the function succeeds.
`False`	If the functions fails, which may be due to one of the following reasons: • Server is already running for this node. • Node could not be opened. • The Win32 event creation failed.

Description

Allocates resources for the specific TriMedia processor. This function has to be called once for every TriMedia processor that TMCRT needs to serve.

 ©1999 Philips Semiconductors 10/08/99

cruntimeDestroy

```
Bool   cruntimeDestroy (
   UInt32   CRTHandle,
   UInt32  *ExitCodePointer );
```

Parameters

`CRTHandle`	Handle to the C Runtime server instance that has to be closed.
`ExitCodePointer`	Address of the memory location where the exit code for this node will be stored. The exit code is valid only if the target has exited normally. Otherwise the target execution has been terminated abnormally and the exit code is invalid.

Returns

`True`	Target has exited normally.
`False`	If the functions fails, which may be due to one of the following reasons: • Target execution has been stopped. • Abnormal Termination. • Exit Code is invalid.

Description

Closes the server instance for this instance of the TriMedia processor.

cruntimeInit

```
Bool cruntimeInit ( void );
```

Parameters

None.

Returns

`True`	Success.
`False`	Thread creation failed.

Description

Initializes the C Run Time server to serve multiple nodes.

cruntimeExit

```
void cruntimeExit ( void );
```

Parameters

None.

Returns

Node.

Description

Terminates all the C Runtime Server threads.

 ©1999 Philips Semiconductors 10/08/99

TriMedia Manager Registry Entries

The TriMedia Manager reads all its initialization settings from the Windows Registry. The settings are read from the following key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\PhilipsSemiconductors\TriMedia\TMMan
```

Table 2 describes the initialization settings which apply to all TriMedia devices installed in the system.

Table 2 Initialization Settings

Name	**Type**	**Default**	**Description**
HostTraceBufferSize	REG_DWORD	0x1000	Controls the size of the host debug trace buffer. This parameter is not used currently.
HostTraceLevelBitmap	REG_DWORD	0x00000001	Controls which internal (TMMan host component) debug levels are enabled. There are 32 different levels. 0x00000001 — All failure conditions are enabled. 0xffffffff — All levels are enabled.
HostTraceType	REG_DWORD	0	Controls the destination of internal (TMMan host component) debug messages. Currently the only valid destination is the kernel debugger. 0: **constTMManDebugTypeNULL** 2: **constTMManDebugTypeOutput** To view the TMMan debug output a Windows kernel mode debugger like SoftICE or WinDBG or WDEB386 is required.
TargetTraceBufferSize	REG_DWORD	0x1000	Controls the size of the target debug trace buffer. This parameter is for TMMan's internal use and does not affect the size of the DP buffer.
TargetTraceLevelBitmap	REG_DWORD	0x00000001	Controls which internal (TMMan target component) debug levels are enabled. There are 32 different levels. 0x00000001 — All failure conditions are enabled. 0xffffffff — All levels are enabled.

Table 2 Initialization Settings

TargetTraceType	REG_DWORD	0	Controls the destination of internal (TMMan target component) debug messages. Currently the only valid destination is the trace buffer. 0: **constTMManDebugTypeNULL** 1: **constTMManDebugTypeTrace** The trace buffer can be examined via the TMMon "DT" command.
MemorySize	REG_DWORD	0x10000	Controls the amount of page locked contiguous memory allocated at startup for shared memory allocations. Note that this type of memory is a very expensive system resource and should be used sparingly.
MailboxCount	REG_DWORD	0x40	Controls the number of inter-processor mailboxes that are allocated. Increase this parameter if the peak packet transfer rate is very high, to prevent packets from being dropped.
ChannelCount	REG_DWORD	0x10	Controls the number of inter-processors channels that can be allocated simultaneously.
VIntrCount	REG_DWORD	0x04	Controls the number of inter-processor interrupt channels that can be allocated simultaneously.
MessageCount	REG_DWORD	0x10	Controls the number of inter-processor messages that can be allocated simultaneously. Note that the number of messages cannot exceed the number of inter- processor channels.
EventCount	REG_DWORD	0x10	Controls the number of inter-processor events that can be allocated simultaneously.
StreamCount	REG_DWORD	0x10	Controls the number of streams that can be allocated simultaneously. This parameter is not used currently.

© 1999 Philips Semiconductors 10/08/99

Table 2 Initialization Settings

NameSpaceCount	REG_DWORD	0x40	Controls the number of name space entries that can be allocated simultaneously. Note that some components like messages use multiple name space entries for a single instance.
MemoryCount	REG_DWORD	0x40	Controls the number of shared memory blocks that can be allocated simultaneously.
SGBufferCount	REG_DWORD	0x20	Controls the number of scatter gather buffers that can be allocated simultaneously.
SpeculativeLoadFix	REG_DWORD	0	Controls the enabling/disabling of the PCI memory apertures on the TriMedia device. 1: PCI aperture disabled. Speculative loads generated by the compiler/scheduler will not generate PCI bus accesses. TriMedia needs to perform accesses to memory on the PCI bus via the PCI device library (libpci). 0: PCI aperture is enabled. Speculative loads may generate PCI bus transactions. On some Pentium II PCI chipsets this can cause a bus lockup.
PCIInterruptNumber	REG_DWORD	0	Controls the PCI interrupt used by the TriMedia device to interrupt the host. Note this is not a software-controlled value. This value of this key has to match the interrupt pin routing on the TriMedia board. 0: PCI INT#A 1: PCI INT#B 2: PCI INT#C 3: PCI INT#D
MMIOInterruptNumber	REG_DWORD	28	Controls the interrupt number (IPENDING bit) that is set by the host to interrupt the TriMedia device.

Table 2 Initialization Settings

MapSDRAM	REG_DWORD	1	Controls the SDRAM mapping state during initialization. Some systems run out of Virtual Address Space (Page Tables Entries) while mapping the SDRAM when multiple TriMedia devices are plugged in the system. As a result of this TMMan cannot activate all the TriMedia Devices in the system. This flag has been introduced tp work around this problem. When this flag is 0 the user application should wrap all accesses to SDRAM with calls to **tmmanMapSDRAM** and **tmmanUnmapSDRAM**. 0: Disables automatic SDRAM mapping 1: Enables automatic SDRAM mapping.
TMRunWindowSize	REG_DWORD	25	Controls the size of the TMRun Window. This value indicates the number of lines that the TMRun windows will have when spawned from TMMon or TMGMon.
DefaultEndianness	REG_DWORD	1	Controls the expected endianess of TriMedia executables. 1: Little Endian (Intel format) 0: Big Endian (Alien format)
TMGMonDDraw	REG_DWORD	1	Controls TMGMon's usage of Direct Draw APIs for retrieving the VGA Frame Buffer Information. 0: DirectDraw APIs will not be used. 1: Direct Draw APIs will be used.

 © 1999 Philips Semiconductors 10/08/99

Table 2 Initialization Settings

TMCRTDebug	REG_DWORD	0	Controls the debug output of TMCRT. 0: Disabled. 1: Enabled. TMCRT prints all its output via Win32 function **OutputDebug-String**.
TCSPath	REG_SZ	NULL	appshell.out is loaded from the directory formed by appending lib\el\WinNT or lib\eb\WinNT to TCSPath. This is required for running dynamic files (*.app) from tmrun, tmmon or tmgmon. If this entry is not there, then appshell.out will be loaded from the current directory.
DLLPath	KEY	NULL	Contains values for the TriMedia Dynamic Link Libraries (DLLs) search paths used by the target dynamic loader. For example: 0: c:\trimedia\bin 1: c:\TriMedia\bin\lib A maximum of 32 different values can be specified.

In addition to reading the above keys, TMMan also reads device specific subkeys. The settings from these subkeys apply to individual devices rather than all of them. These subkeys are called "*DeviceX*", where X is the number of the TriMedia device. The following values are read from the *DeviceX* subkey:

Name	**Type**	**Default**	**Description**
ClockSpeed	REG_DWORD	100,000,000	Sets the clock frequency that **procGetCapabilities** returns on the target.
CacheOption	REG_DWORD	2	This value is passed to the function **TMDwnLdr_relocate**. It controls the way the downloader deals with the caching. Look at TMDownLoader.h for more info. A value of **2** indicates **TMDwnLdr_LeaveCachingTo-Downloader**.
SystemBaseAddress	REG_DWORD	0x10000000	
SDRAMBaseAddress	REG_DWORD	0x00000000	
MMIOBaseAddress	REG_DWORD	0xEF000000	

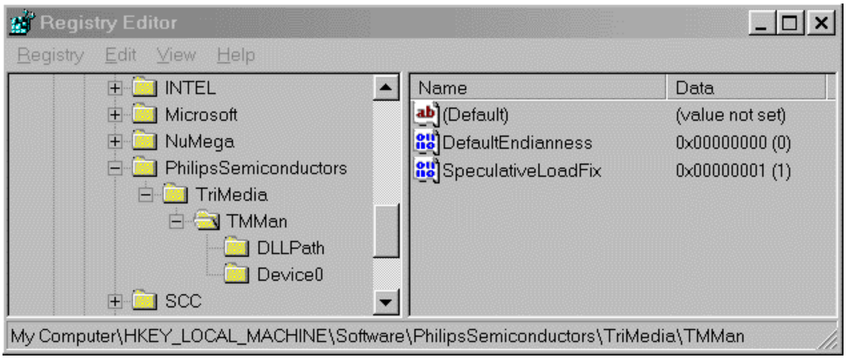


Figure 19 Registry Editor

Configuration settings for the ref3 card have to be specified in the Windows registry for SystemBaseAddress (default 0x10000000), SDRAMBaseAddress (default 0x00000000), and MMIOBaseAddress (default 0xEF000000). They must be defined as **DWORD** values under

 © 1999 Philips Semiconductors 10/08/99