Technical notes on using Analog Devices DSPs, processors and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Using VisualDSP++® Thread-Safe Libraries with a Third-Party RTOS

Contributed by Andy Millard

Rev 1 – November 10, 2006

Introduction

This document describes how to use the thread-safe C and C++ libraries provided with VisualDSP++® 4.5 with an RTOS other than VDK. This information is provided as a guide for users who have a firm understanding of multithreaded programming principles and the procedures used in their multithreaded operating system.

Thread-safe libraries are designed to work specifically with the VDK RTOS, which is the only configuration supported by Analog Devices.

Overview

The C and C++ libraries that are shipped with VisualDSP++ 4.5 are available in various flavors, several of which contain functions considered to be safe for use in a multithreaded environment. These thread-safe versions of the libraries were developed as a requisite for the VDK kernel and are not intended to be used in conjunction with any other RTOS. By replacing the VDK functions called from within the C and C++ libraries with equivalent functions from the alternative RTOS, adding the relevant support to the initialization objects and modifying the Linker Description File (`.LDF`), it is possible to integrate the libraries into another environment.

It is assumed that users wishing to use an alternative RTOS will be using initialization code provided by the RTOS developer responsible for heap and stack initialization, file I/O, and other required components. This overrides the behavior of the default initialization code provided with VisualDSP++. A customized `.LDF` file provided with the RTOS must be used to include these new objects.

When using the multithreaded C run-time support library, the application must be linked against the multithreaded C++ run-time support library. The C library requires the use of a C++ mutex class defined in the C++ library.

Sections of this Document

This document contains the following sections.

- Changes from VisualDSP++ 4.0 to VisualDSP++ 4.5
- Locking and Unlocking Critical Sections in the C and C++ Libraries
- Thread-Local Storage
- Modifying the Linker Description File
- Initializing the C and C++ Libraries

Changes from VisualDSP++ 4.0 to VisualDSP++ 4.5

The underlying implementation of the thread-safe libraries has changed significantly between the VisualDSP++ 4.0 and VisualDSP++ 4.5 releases. The previous releases used semaphores, and the new implementation uses a mutex-based system.

The following functions are no longer called from the run-time libraries that are thread safe:

- `CreateSemaphore__3VDKFUiN31`
- `DestroySemaphore__3VDKF11SemaphoreID`
- `PendSemaphore__3VDKF11SemaphoreIDUi`
- `PostSemaphore__3VDKF11SemaphoreID`
- `GetThreadID__3VDKFv`

The following functions, which are described later in this document, were added in VisualDSP++ 4.5:

- `RMutexInit__3VDKFPQ2_3VDK6RMutexUi`
- `RMutexDeInit__3VDKFPQ2_3VDK6RMutex`
- `RMutexAcquire__3VDKFPQ2_3VDK6RMutex`
- `RMutexRelease__3VDKFPQ2_3VDK6RMutex`

Locking and Unlocking Critical Sections in the C and C++ Libraries

The C and C++ runtime support libraries provided with VisualDSP++ 4.5 use several VDK functions for serialized access to critical sections of code, as follows.

void PushUnscheduledRegion__3VDKFv (void)

This function is called whenever a critical section is entered. The function ensures that the scheduler does not assign control of the process to another thread while inside a critical section. Calls to `PushUnscheduledRegion` can be nested. For details, refer to *VisualDSP++ 4.5 Kernel (VDK) User's Guide*[1].

void PopUnscheduledRegion__3VDKFv (void)

This function is called when leaving a critical section. Function calls can be nested. For details, refer to *VisualDSP++ 4.5 Kernel (VDK) User's Guide*[1].

void RMutexInit__3VDKFPQ2_3VDK6RMutexUi(void* mutex, unsigned int size)

This function constructs a mutex of size `size`. In calls from the thread-safe runtime libraries, the size is `5*sizeof(int)`. In the thread-safe libraries, the memory pointed to by `mutex` is allocated by a call to `malloc()` before the pointer is passed to this function.

void RMutexDeInit__3VDKFPQ2_3VDK6RMutex(void* mutex)

This function destructs the mutex for the given `mutex` pointer. The memory allocated for the mutex is not freed by the call to this routine and must be done independently.

void RMutexAcquire__3VDKFPQ2_3VDK6RMutex(void* mutex)

This function acquires the mutex pointed to by `mutex`.

void RMutexRelease__3VDKFPQ2_3VDK6RMutex(void* mutex)

This function releases the mutex pointed to by `mutex`.

The suggested approach for replacing these functions is to create them with the same names that will call the appropriate functions in the alternative RTOS.

Note that the VDK mutex is implemented as a recursive mutex, i.e., one for which nested acquisitions by the same thread are legal and safe.

The mutex implementation keeps track of both its current owner (if there is one) and the count of nested acquisitions by that thread. The mutex is not released until the count falls to zero.

The VDK locking mechanisms that are called from the C and C++ libraries concern themselves with disabling the scheduler only. They do not disable interrupts. This is left to the developers' discretion when calling these functions.

The C and C++ libraries are designed to be thread-safe only and are not re-entrant. As a consequence, interrupt service routines should not call any C or C++ library functions as they may cause the library to perform in an undefined fashion. More information on calling library functions from an ISR can be found in the *VisualDSP++ 4.5 C/C++ Compiler and Library Manual* [2] [3] [4].

It is possible to replace these calls with calls to functions that disable the scheduler and disable interrupts. This approach is not recommended as the library may be attempting to generate its own interrupt.

Thread-Local Storage

The C library supports and uses thread-local storage to preserve data between function calls for four sets of data:

errno

The value of `errno` is thread-specific. Each thread uses thread-local storage to preserve the value of `errno`.

strtok()

The `strtok()` function uses thread-local storage to preserve internal pointers between calls to the function.

rand()

The `rand()` function uses thread-local storage to allow a program to define a seed value on a per-thread basis.

time() and asctime()

The `asctime()` function and the `time()` family of functions use thread-local storage to allow calls to these functions on a per-thread basis.

The routines detailed above rely on three functions in the VDK. Functions with identical functionality are available in most RTOS.

bool AllocateThreadSlotEx__3VDKFPiPFPv_v(int *, void (*cleanupfunc)(void *))

This function allocates a new thread storage slot if one has not already been assigned. The VDK function has several return values. Refer to the VDK API in the *VisualDSP++ 4.5 Kernel (VDK) User's Guide*[1] so that the alternative function can emulate the behavior of `AllocateThreadSlotsEx`.

void *GetThreadSlotValue__3VDKFi (int)

This function returns a pointer to that thread's locally stored data. Refer to the VDK API in the *VisualDSP++ 4.5 Kernel (VDK) User's Guide*[1] for information on `GetThreadSlotValue`.

bool SetThreadSlotValue__3VDKFiPv (int, void *)

This function stores the data pointed to in the threads area of the slot. Refer to the VDK API in the *VisualDSP++ 4.5 Kernel (VDK) User's Guide*[1] for information on `SetThreadSlotValue`.

Modifying the Linker Description File

The Linker Description File (`.LDF`) used by VDK differs from the standard `.LDF` file because it links the user's program using the thread-safe version of the C and C++ libraries by default.

The default `.LDF` file does not contain support for multithreaded libraries. Users wishing to use an alternative RTOS must copy the standard `.LDF` file and modify it to link with the correct versions of the libraries.

The multithreaded libraries for Blackfin® and SHARC® contain an `mt` extension in the library name; for example, the ADSP-BF532 C library is named `libc532mt.dlb` with the ADSP-21060 C library named `libcmt.dlb`. The multithreaded libraries for TigerSHARC® contain an `_mt` extension in their name; for example, the ADSP-TS201 C library is named `libc_TS201_mt.dlb`.

Initializing the C and C++ Libraries

The following sections describe the areas of the supplied initialization code that must be modified to allow use of the C and C++ libraries.

It is assumed that the source code for the initialization code used by the RTOS will be available.

Several sections of the default initialization code must be replicated in the initialization code of the RTOS to ensure that the C and C++ libraries will function as expected. Details on these sections follow for each of the Blackfin, SHARC, and TigerSHARC processor targets.

Blackfin

Start-Up Code

The start-up code in `VisualDSP\Blackfin\lib\src\libc\basiccrt.s` supports several configuration options. The Blackfin installation includes numerous pre-assembled combinations. The default `.LDF` file selects the `crt*.doj` file to link with, based on user-specified defines and `ccblkfn` options.

FIOCRT

The `FIOCRT` macro is true when we wish to use file I/O inside a program. Enabling this define ensures that the `_init_devtab` routine is called to initialize your I/O method.

CPLUSCRT

The C++ `___ctorloop` routine must be called. This routine ensures that C++ objects are created and destroyed correctly. It also initializes behind-the-scenes garbage collection. The `___ctor_table` variable should also be declared in a similar manner to the declaration at the bottom of the default `basiccrt.s` file.

FP/SP Initialization

The initialization code should ensure that the `FP` and `SP` registers are set to sensible values. It would be expected than an alternative RTOS initialization routine would provide this as default.

SHARC

Start-Up Code

The start-up code's source code is in one of the following files, depending on the target architecture:

- `VisualDSP\21k\lib\src\crt_src\020_hdr.asm`
- `VisualDSP\21k\lib\src\crt_src\06x_hdr.asm`
- `VisualDSP\211xx\lib\src\crt_src\16x_hdr.asm`
- `VisualDSP\212xx\lib\src\crt_src\26x_hdr.asm`
- `VisualDSP\213xx\lib\src\crt_src\36x_hdr.asm`

The SHARC installation includes numerous pre-assembled combinations. When using the default `.LDF` file, the `.LDF` file selects the `*hdr*.doj` file to link with, based on user-specified defines and `cc21k` options.

By default, several variants of pre-built start-up code are included in the release. By convention, the file names of the pre-built start-up modules that that support C++ contain `cpp`, and the thread-safe versions contain `_mt` in their names.

__cplusplus Define

The `_lib_call_ctors` routine will be included when this macro is defined. This routine ensures that C++ objects are created and destroyed correctly. This routine is called when the `__cplusplus` define is set when assembling the `*hdr*.asm`. The `___ctors` variable should also be declared; this variable should be defined within the `.LDF` file and should point to the start of the `seg_ctdm` section.

TigerSHARC

Start-Up Code

The start-up code's source code is in the assembly source file that can be found in the following location: `VisualDSP\TS\lib\src\crt_src\ts_hdr.asm`. The TigerSHARC installation includes numerous pre-assembled combinations. When using the default `.LDF` file, the `.LDF` file selects the `ts_hdr*.doj` file to link with, based on user-specified defines and `ccts` options.

By default, several variants of pre-built start-up code are included in the release. By convention, file names for the pre-built start-up modules that support C++ contain `cpp`, thread-safe versions contain `_mt` in their names, and byte-addressing mode start-up files contain `_ba` in their names.

It is assumed that users will have the source code for the initialization code that is used by the RTOS.

The following sections of code from the default initialization code must be replicated in the initialization code of the RTOS to ensure that the C and C++ libraries will function as expected.

_CPLUSPLUS Define

The C++ routine `___ctorloop` must be called to ensure that C++ objects are created and destroyed correctly. This routine is called when the `_CPLUSPLUS` define is set while assembling `ts_hdr.asm`. This routine will also initialize the heap and the I/O library, so these must be initialized before use. The `___ctor_table` variable should also be declared in a manner similar to the declaration at the bottom of the default `ts_hdr.asm` file.

vdkMainMarker

The C variable `vdkMainMarker` is declared within VDK to notify the run-time libraries as to when the application enters a multithreaded state. `vdkMainMarker` is declared as:

```
int vdkMainMarker = 0;
```

The variable is assigned a value of 1 immediately before the program invokes the highest-priority boot thread. Until the variable is set to a value of 1, no VDK locking functions are called by the run-time libraries. This is implemented to avoid calling thread-specific functions from non-threaded code.

For use with any RTOS other than VDK, this variable should be declared with an initial value of zero and set to one just before the program invokes the user boot thread. This applies to Blackfin, SHARC, and TigerSHARC processors.

References

[1] *VisualDSP++ 4.5 Kernel (VDK) User's Guide.* Rev 2.0, April 2006. Analog Devices, Inc.

[2] *VisualDSP++ 4.5 C/C++ Compiler and Library Manual for Blackfin Processors.* Rev 4.0 April 2006. Analog Devices, Inc.

[3] *VisualDSP++ 4.5 C/C++ Compiler and Library Manual for SHARC Processors.* Rev 6.0 April 2006. Analog Devices, Inc.

[4] *VisualDSP++ 4.5 C/C++ Compiler and Library Manual for TigerSHARC Processors.* Rev 3.0 April 2006. Analog Devices, Inc.

Document History

Revision	Description
Rev 1 – November 10, 2006 by Andy Millard	Initial Release