dBOS Readme

dBOS Real Time Operating System

Revision 1.0

Contents

1 Introduction

Embedded software for deeply embedded applications is still predominantly written in plain old C, the most popular RTOSs are written in C.
This project was undertaken as an experiment in object oriented software design combined with writing C++ for an embedded target specifically the Cortex M3/M4F microcontroller.

C++ obviously enables object oriented programming, inheritance and polymorphism and therefore allows easy implementation of common and proven software design patterns which can only be done to a limited extent using manual techniques (i.e. structures full of function pointers) in C. For example http://www.state-machine.com/doc/AN_Simple_OOP_in_C.pdf .

This project has resulted in a full featured OS for embedded applications, it has nearly all the common features implemented by the well known RTOSs and unlike many of the RTOSs available it comes with a full set of unit tests.

Some links to information on the web about the use of C++ in embedded environments,

http://www.artima.com/shop/effective_cpp_in_an_embedded_environment

http://www.embedded.com/design/programming-languages-and-tools/4438660/3/Modern-C--in-embedded-systems---Part-1--Myth-and-Reality

Section 4 of this document,

http://www.state-machine.com/doc/Building_bare-metal_ARM_with_GNU.pdf

2 Features

2.1 Application Programming Interface

The API is slightly unusual, instead of calls like,

SemaphoreOne->Take() or MutexOne->Lock()

there is,

pK->Object_Wait(SemaphoreOne, &SignalData, TIMEWAITFOREVER)

The first argument is a pointer to a semaphore, it maybe a pointer to any type of synchronisation object such as a mutex or flag.

The calling thread will wait to take the semaphore, lock the mutex etc.

Similarly,

pK->Object_Signal(SemaphoreOne, 8, SIGNAL_AUTO, TRUE)

Will cause the calling thread to give the semaphore or unlock a mutex etc.

A thread may wait on multiple objects of differing types,

```
dBOS_SYNCINTERFACE_P_t pObjects[] = { MutexOne, SignalOne, SemaphoreOne };
SNATIVECPU_t Index;

pK->Object_WaitMultiple(pObjects, 3, TIMEWAITFOREVER, &Index, &SignalData);
```

Index will be the index in pObjects of the object that was signalled, unlocked etc.

Only one of the objects maybe a mutex, support for waiting on more than one mutex at a time significantly increases the complexity of the priority inheritance mechanism.

pK in the above examples is the pointer to the kernel. The unit tests contain dozens of code examples.

2.2 Unit Tests

dBOS includes a full set of unit tests based on a simple test framework. The tests obviously verify the functionality is as intended but just as importantly provide examples of how to use the OS and the API. The source for the tests is located here dBOS/dBOSpp/Tests/

2.3 Priority Inheritance

dBOS includes a full priority inheritance mechanism meaning priority is inherited through multiple mutex owner task pairs. For example,

Task One takes Mutex A
Task Two takes Mutex B and is then set waiting on attempting to take Mutex A.
Task Three is set waiting on Mutex B.

In the above case Task One inherits the priority of Task Three (if task three has the highest priority).

2.4 Designed for Cortex-M Exception Priorities

The ARM Cortex M cores support assigning priorities to interrupt and exception handlers, a mask register allows handlers of a specified and lower priority to be disabled.
The asynchronous context switching is performed by the OS in the PendSV handler, this is assigned the lowest possible priority.

OS critical sections are formed by disabling the PendSV handler, the OS raises the CPU's priority mask to the that of the PendSV handler, this leaves all other handlers enabled.
Higher priority handlers that use OS objects are disabled for only short sections of code, improving interrupt handling latency.
OS objects that are accessed from interrupt handlers must have the priority of the highest priority handler which uses the object specified when the object is created,

SemaphoreOne = pK->CreateSemaphore("Semaphore One", 0, 1, PRIORITYLIST, ISRPRIORITY);

2.5 Hierarchical State Machine

Included in the OS is an event driven Hierarchical State Machine framework, dBOS/dBOSpp/dBOS_HSM/
This runs on top of the OS and processes events from an OS Queue,

```
class dBOS_HSMTESTBASE * pHSM = new dBOS_HSMTESTDERIVED("Test HSM", HSM_DEBUG_ALL_EVENTS);

pHSM->Init();

dBOS_HSMEVENT Event;

while(1){
	pK->Queue_ReadFromFront(QueueOne, (void * )&Event, TIMEWAITFOREVER);
	pHSM->ProcessEvent(&Event);
}
```

Unit tests provide examples of how to use the framework, dBOS/dBOSpp/dBOS_HSMTest/

The HSM framework allows creation of entire state machines derived from an existing state machine enabling further or different functionality to be added to an existing design.
States can be derived from existing states to add handling of more events or change the way an event is handled, it is important to understand the distinction between a states's parent state and the state class the state is derived from.

3 Demo Projects

Demonstration projects are Atollic True Studio projectS and will run on an Olimex STM32-P103 board.

3.1 dBOS_Tests

This project runs unit tests over the OS.

3.2 dBOS_TimeSliceDemo

Demonstrates the optional time slicing feature. The OS switches between three threads of equal priority at regular intervals. These threads would otherwise not yield the CPU as they do not call any wait or yield OS methods.

3.3 dBOS_LEDFlashDemo

Demonstrates flashing a LED with thread that waits on a timer object.

3.4 dBOS_HSMTests

Runs unit tests over the Hierarchical State Machine system.

4 Project Status

Currently this project has been tested to the extent of the include unit tests, it has not been used in any commercial projects as yet.

5 Contact Details

Contact the author, David Broadhurst e-mail decbroadhurst@gmail.com