CODE TIME TECHNOLOGIES

Runtime Safe Service Creation

Whitepaper

Introduction

A common problem when implementing an application is which task will create a shared kernel service. This includes a queue, mutex, or any other service that is shared between multiple tasks.

The typical approach is to designate a "master" task that instantiates the shared service, with all other "slave" tasks left pending until the service is created. The resource is then shared using a global variable. Such a method is not very elegant or easily expandable, and is open to usability or debug issues as the application expands. It also breaks the task encapsulation model by forcing the use of shared variables.

Master/Slave Creation

Using a master/slave approach to shared service creation, one task would be responsible for creating the shared services, and all other tasks would wait until the first task completes. This could be accomplished in the `main()` function, prior to starting the RTOS. Or it could be done after the RTOS has started, in a high priority task. In either case, the services would normally be shared using global variables.

For example, consider the case where 2 shared services are created, using the above methods.

Created in `main()`

main.c:
```
#include "OS_Mutex.h"

OS_Mutex *G_Mutex;

int main(void)
{
    …
    G_Mutex = OS_MutexCreate("Lockbox");
    …
    /* Initialize the RTOS */
    OS_Init();
}
```

task1.c:
```
#include "OS_Mutex.h"

extern OS_Mutex *G_Mutex;
void Task1(void)
{
    …
    OS_MutexLock(G_Mutex, -1);
    …
}
```

task2.c:
```
#include "OS_Mutex.h"

extern OS_Mutex *G_Mutex;
void Task2(void)
{
    …
    OS_MutexLock(G_Mutex, -1);
    …
}
```

Created by Master task

main.c:
```
int main(void)
{
    …
    /* Initialize the RTOS */
    OS_Init();
}
```

master.c:
```
#include "OS_Mutex.h"

volatile OS_Mutex *G_Mutex = NULL;

void Master(void)
{
    …
    G_Mutex = OS_MutexCreate("Lockbox");
    …
    OS_MutexLock(G_Mutex, -1);
    …
}
```

slave.c:
```
#include "OS_Mutex.h"

extern volatile OS_Mutex *G_Mutex;
void Slave(void)
{
    …
    /* Has mutex been created? */
    for ( ; G_Mutex == NULL ; ) ;
    OS_MutexLock(G_Mutex, -1);
    …
}
```

Both of these methods require the use of global variables, resulting in less maintainable code, and decreased localization. In addition, the use of a "master" task requires all slave tasks to either validate that the mutex has been properly created prior to use, or the master must be guaranteed to run prior to any slave service accesses, leading to debugability and expandability issues.

Abassi Advantages

Runtime Safe Creation

Runtime safe creation is a RTOS enhancement that eliminates the typical master/slave issues when creating shared system services, such as queues and mutexes. Instead of forcing the highest priority task to create all shared services and export those as global variables, every task that wants access to those services simply "opens" them. Whichever task "open" the service first actually "creates" it, and all successive accesses simply point to the existing service, identified by its name

For example, reconsider the previous example. Here, runtime safe creation allows all tasks to "open" a shared system resource, and the kernel itself ensures that all tasks will access the same, unique resource.

Abassi

```
main.c:
int main(void)
{
    …
    /* Start the multitasking */
    OSstart();
}

task1.c:
#include "Abassi.h"

void Task1(void)
{
MTX_t *SharedMutex;
    …
    SharedMutex = MTXopen("Lockbox");
    …
    MTXLock(SharedMutex, -1);
    …
}

task2.c:
#include "Abassi.h"

void Task2(void)
{
MTX_t *SharedMutex;
    …
    SharedMutex = MTXopen("Lockbox");
    …
    MTXLock(SharedMutex, -1);
    …
}
```

It does not matter if `Task1` runs first or `Task2` runs first, they will both attempt to "open" the same mutex (named "Lockbox"), and the first one to make the attempt will create the mutex. Any future attempts to open that mutex will simply gain access to the already created one. This is because, in Abassi, service names have real runtime meaning, and are not simply used for debug purposes.

By introducing runtime safe service creation, common problems that occur as a design evolves and task priorities are modified are eliminated, and individual tasks can keep as many variables local as possible, instead of exporting all shared services as global variables.

Conclusion

By employing a greenfield design approach, Code Time Technologies has been able to create a next generation real-time kernel that vastly improves upon any available today. The Abassi real-time kernel outperforms all existing platforms by combining code size and CPU efficiency with an unrivalled set of features and usability enhancements.