

The C Kernel

Application Note

Binding Implementation Guide

Filename : The C Kernel Binding Guide.doc/pdf
Modified : September 10, 2004
Author : P.K. van der Vlugt
e-mail : peter@ckernel.nl

Introduction

This short application note is a step-by-step guide for implementing binding projects for **The C Kernel**. Successfully implemented and tested bindings can be published on **The C Kernel** website, for more information contact support@ckernel.nl.

The binding implementation guide is a helpful guide for getting started into the details of creating new bindings. In fifteen steps this application note explains what knowledge should be gathered, what should be done, and how the binding can be implemented and tested.

Binding implementation steps

1. Get an in-depth knowledge of the micro controller or processor architecture. Main focus points for the kernel binding implementation is the interrupt architecture. Find out what happens in the hardware when an interrupt occurs; check if registers are saved automatically by the interrupt mechanism, and what exactly happens to the stack. Also get detailed knowledge on how the stack mechanism of the platform works. All this information is typically documented in hardware or user manuals of the micro controller or processor platform. Make sure to have this documentation available at all times.
2. Get an in-depth knowledge of the compiler environment or IDE for the hardware target. The compiler should at least be able to generate either reentrant functions or have a possibility to protect function parameter passing by means of for example a 'critical' keyword which disables interrupts before passing the parameters and calling the function. When reentrant functions are supported by the compiler, parameter passing will use the processor registers and/or the stack and local function variables have to reside in registers or have to be stack based as well. All this information is typically documented in the user manual of the compiler environment or IDE. Make sure to have this documentation available at all times.
3. Find out what kind of memory models are supported by the compiler environment and hardware architecture. Do not choose a too small model, it works better and easier when there is enough RAM available in the system. Extra RAM does not add up much to the final system cost, but if it is not possible due to cost or PCB size restrictions try to choose a micro controller or a processor with enough internal RAM. If RAM is however limited the configuration for the kernel can be chosen to limit the number of resources used. It is for example possible to run a task only model without using the other resources, and thus saving memory space.
4. How does the compiler handle interrupts in C ? Analyze a C interrupt routine (see also under 9) in the LST or SRC file result after compilation what kind of assembler statements are generated by the compiler before inserting the actual C code of the handler. It is sometimes necessary to turn on a compiler option to generate such statements in the output list file. A useful source of information regarding interrupt handling is in most cases also the user manual of the compiler.
5. The actual porting and creation of a binding needs to be done in only 3 files. The general file *stddefs.h* and the binding module files *ckbind.h* (binding.h for V1.01) and *ckbind.c* (binding.c for V1.01).
6. To start with, file stddefs.h should be setup for the processor architecture and compiler environment. Implement the HiByte and LoByte macros and additionally the HiWord and LoWord macros. Determine how a multi-byte variable is stored in memory; Intel or Motorola like. In other words find out if the lower byte is stored at the low address or at the high

address. Furthermore a definition needs to be implemented for `_enable` and `_disable` if this is not defined by the compiler environment.

7. In file `ckbind.h` only the section definitions and macros needs to be worked on. Setup the definitions for `START_CRITICAL` and `END_CRITICAL`, usually these will be equal to `_disable` and `_enable`. Furthermore the internal 'ck_' data types need to be specified, this can be particularly useful when the compiler for example supports fast internal memory types or bit types, by doing so the kernel internal variables can be optimized for fast access, speed and size optimized.
8. In file `ckbind.c` all the functions need to be implemented. It is strongly advised to checkout other binding implementations and use that knowledge when starting a new implementation. To start with, function `KernelTickInit` has to setup the kernel hardware timer (also called tick) interrupt and make sure that the initialization uses the definition of `KERNEL_TICKCOUNT` from the kernel configuration file `ckconfig.h`. The kernel configuration file is used at compile time to configure the kernel resources and hardware at compile time. The `KERNEL_TICKCOUNT` defines the internal kernel tick in microseconds. A good internal tick basis for the kernel is somewhere between 1 and 20 milliseconds depending on the needs of the application.
9. The next function in `ckbind.c` to implement is the kernel tick interrupt function. This interrupt function must call the `KernelTickHandler` function as declared in the core module `ckernel`. Analyze the C interrupt function prolog and see if all necessary registers are saved. Make sure that before calling the `KernelTickHandler` the complete context of registers and state is saved ! This is the most important mechanism that the kernel relies on. When switching task contexts due to a (tick) interrupt event, half of the work is already done by the interrupt handler's saving scheme. That is why the normal context switching uses a same switching scheme.
10. Now that the layout of the stack during interrupts and the interrupt handling is known, the function `SetupStack` in `ckbind.c` can be implemented. This function is called from the `CK_CreateTask` function for setting up an initial stack image for the task that is being created. The initial stack image should look exactly the same as happens when entering a C interrupt function; setup the same order of function return address, processor status and all registers.
11. Implement function `ContextSwitch` in `ckbind.c`. In this function the stack of the running task needs to be saved and then the stack of the new running task needs to be restored. Depending on the platform this may be simply setting the contents of the stack pointer register from one task stack to the other task stack, or in case of for example internal system stacks, copying the contents of the stack to the task stack and restoring vice versa.
12. Finally implement function `ISRContextSwitch` in `ckbind.c`. This is a 'half' context switch since this function is called from within an interrupt context meaning that the task state has already been saved by the interrupt handler. The function `ISRContextSwitch` is called by the kernel when a task switch should occur (preemption) from any interrupt context, the function itself only restores the stack for the new running task.
13. After having implemented all the above items, debugging and testing can start. Create a small application with only two tasks running and have both tasks call for example the `CK_Sleep` function just after having executed something 'recognizable' such as toggling an output or sending a character to an output stream. A good strategy is to start analyzing the very first task context switch when the system enters the `CK_Run` function (it is advised to keep interrupts off in this stage of testing). The task with the highest priority has to become the running task. When this task is really becoming active after the first switch, then the next step is to move on to the second context switch (in the example through `CK_Sleep`), still with the interrupts turned off. Analyze if the context of the first task was saved properly and see if the second task becomes active. When this all succeeds and the second task enters `CK_Sleep` the last task which is the `IdleTask` in the core module `ckernel` will become active and remain active forever since the interrupts were off.

14. The next step is to follow a different scheme as in 14 and now with interrupts on. Make sure that one task remains running and active all time without switching. Check if the tick interrupts are running and if these are being handled correctly. The interrupted task must at all time resume execution after the interrupt handling without any problems or corruption of the stack. Carefully monitor what happens to the stack of the interrupted task and if this is being restored in the correct way.
15. When task switching and interrupt handling works correct start a testing scheme as described in 13 with two application tasks and check if all runs together with the interrupts turned on. The two application tasks that are running and using the CK_Sleep function should be switched to active using the correct timing as specified in the CK_Sleep function.

Final tips

When using compiler environments or IDE's check if there is a simulator debugger with the software. These simulators are very useful in analyzing all the steps described above, and most of the time they work quite well. In particular the environment of for example Keil has a very good simulator for different targets built-in. The main advantage of these simulators is that most of them are capable of debugging at the source code level allowing the user to check registers, stacks, memory, output windows etc. It allows the user to test the system on the development platform before going into the real target hardware.