Sample "kernelBuilder" - A platform to build your RTOS for e200z4

Peter Vranken

Table of Contents

1. About this sample

1.1. Overview

This sample offers a pair of IVOR #4 (External Interrupt) and IVOR #8 (System Call) handlers, which are designed to support the build of a single core operating system kernel for the NXP MPC5643L (e200z4). The operation is entirely connected to the handling of asynchronous (External) and software interrupts.

The e200z4 core has one common handler each for External Interrupts and for system calls. kernelBuilder implements these handlers such that they branch into sub-handlers, which are implemented in the client code. By configuration the client code can assign different handlers to different External Interrupt sources and to different system calls. [1: The first function argument of a system call is the index into the configuration table of handlers.] The sub-handlers in the client code can communicate with the common code inside kernelBuilder in order to let it start, suspend, resume, and terminate different contexts.

The complete software, kernelBuilder plus client code, can thus be understood as an event driven state machine. The different External Interrupts and the different system calls denote events, which are recognized and processed by the various handlers and which lead to an according sequence of context switches.

The IVOR #4 handler for the External Interrupts supports ordinary interrupt service routines (ISR) and kernel relevant ISRs. When registering its ISRs, the client code will specify which category the handlers belong to.

The ordinary ISRs behave as usual, they preempt an execution context at arbitrary code location, do their job and continue the same context at the location of preemption. This is exactly the behavior of the IVOR #4 handler in the common startup code of the other samples.

The kernel relevant ISR is new. It has a return value, which permits to demand the switch to another execution context. Preemption happens identical to the ordinary ISR at arbitrary code location. After servicing the interrupt source, on return, it decides to either behave like the ordinary ISR and to continue the preempted context or it can decide to continue any other execution context. "Any other execution context" normally is one of those, which had been left by a similar, earlier decision, i.e. which had *not* been continued at that time.

Our simplest sample code, alternatingContexts, is a "scheduler", which implements a regular timer interrupt as kernel relevant ISR. Unconditionally in each tick, it demands a context switch between two existing execution contexts. The CPU executes both program paths alternatingly and pseudo-concurrent.

The second IVOR handler implements the concept of system calls. These are software interrupts, which preempt a context at the particular location, at which the code contains the system call instruction. The difference to an ordinary branch instruction is that the same CPU mechanisms are used as for asynchronous interrupts; in particular the CPU returns from user mode back to supervisor mode and it disables the servicing of External Interrupts. The latter ensures that system calls and External Interrupts can be serialized, that coherency of access to data they both share can

be safely implemented.

As for asynchronous interrupts, there are two kinds of system calls. The so called simple system calls correspond with the ordinary ISRs. The will usually implement kernel unrelated operations. They can not command a context switch and will always return to the calling context. With respect to behavior, such a simple system call is like an ordinary function call, it takes arguments and returns a result. The major difference to an ordinary function is its execution in supervisor mode, which makes it suitable for the implementation of I/O drivers.

The other kind of system calls behave like kernel relevant ISRs and the function signature is similar. On return, such system calls can decide to either return to the calling context or they can demand a context switch. In the former case, the system call behaves exactly like a simple system call. In the latter case, the calling context is suspended to the advantage of another, instead continued context. The typical use case is that the calling context needs to wait for completion of something done by one or more other contexts.

The two IVOR implementations work together. They share the same C data structure definitions for execution contexts and context switches. If a context had been preempted and left (i.e. suspended) by an External Interrupt ISR then the same context can later become the target for a context switch demanded by a system call and this way it'll be resumed. In the same way can a system call suspend a context and this context is later target of an IVOR #4 ISR, which decides on return for a context switch to that context. Furthermore, both IVORs can be commanded to create a new context on the fly and to continue with that.

In total, eight paths are possible for kernel relevant ISRs and system calls:

- The ISR can continue the preempted code, like an ordinary ISR
- The ISR can suspend the preempted context and create and enter a new context
- The ISR can suspend the preempted context and resume a context, which had been suspended earlier by an ISR
- The ISR can suspend the preempted context and resume a context, which had been suspended earlier by a system call
- The system call can return to the calling context, like a simple system call or an ordinary function call
- The system call can suspend the calling context and create and enter a new context
- The system call can suspend the calling context and resume a context, which had been suspended earlier by an ISR
- The system call can suspend the calling context and resume a context, which had been suspended earlier by a system call

Whenever the ISR/system call handler decides on return to do a context switch and if it commands to resume a context that had been suspended by a context-switching system call, then the handler can provide a uint32_t value, which is returned to the resumed context as function result of the system call, which had made it suspended. Consequently, from the perspective of the system calling context a system call always looks like an ordinary function call — it is invoked with a number of arguments and on return a function result is got. The only observable difference to a normal

function call is the (arbitrarily long) time span between entry into and exit from the system call.

The added value of this TRK-USB-MPC5643L sample is that all of this functionality is offered through a C API. The IVOR handlers and the described mechanisms are implemented in assembler code but the client code can be implemented in sheer C code. A complete, cooperative or preemptive operating system kernel can be built on this platform without any additional assembler code. No inline assembler spoils your kernel implementation or makes it compiler dependent.

Furthermore, all context synchronization is done in the assembler code and the kernel implementation in C is a race condition free development environment. Once you've understood the C API of kernelBuilder, writing your own RTOS becomes really simple. (And you may have a look at the samples, a tiny RTOS is present, too.)

1.2. Registration of ISRs and system calls

ISRs — ordinary and kernel relevant — are dynamically defined by the client code using the known mechanisms from the common startup code. A change has been made in the call for registering an ISR: A Boolean argument makes the distinction between ordinary and kernel relevant ISRs and the type of the ISR function pointer depends on this. Kernel relevant ISRs can no longer be of type `void (*)(void)` — they require a more complex signature, which permits commanding the context switch on return.

System call handlers are collected in two static, constant tables of addresses of those. There is a table for kernel relevant system calls and a second one for simple system calls.

The distinction between the two types of system calls has been made although the kernel relevant handlers can emulate the same behavior in most situations. Wherever the simple handlers can be applied they have the following advantages:

- They offer to change the machine state in which the calling context is executed. The principal use case is a pair of system calls to suspend and resume handling of External Interrupts
- They produce less overhead
- They are not serialized with other system calls (neither simple nor kernel relevant) and nor with ISRs. Therefore, they barely impact the real time behavior of a kernel

Dynamic adding of table entries is not possible for system calls at run-time; the set of system calls is considered a finalized design decision for the aimed scheduler/kernel/RTOS. The tables are declared extern to the assembler code and the client code is in charge to compile them.

kernelBuilder offers the API `init_systemCall(idxSysCall, ⋯)` to invoke the system call handler at given index in the table. The normal design decision for the client code will be to provide a preprocessor macro or an inline function for each system call, which wraps the generic call `init_systemCall(idxSysCall, ⋯)` with a meaningful name. [2: Meanwhile it appears that at least for the simple system calls a run-time table configuration could be the better choice: Most I/O drivers will require to register some of these system calls in order to provide their APIs. The current, centralized constant table requires careful code design in order to achieve proper separation of the implementation of kernel and of the different drivers. The samples demonstrate how this can be done.]

1.3. What is a context in kernelBuilder?

In kernelBuilder, a context is represented by an object of type `int_contextSaveDesc_t`. Not the register contents, which constitute an execution context, are stored in this object but the address of where they are stored. [3: kernelBuilder stores the register contents on the stack, so storing the address of where they are stored actually means storing the current stack pointer value in the context object.] This information is maintained and updated by the IVOR handlers when it comes to a context switch.

By principle, the execution of a context starts with entry into a C function. Two typical use cases exist for contexts: forever running contexts and single-shot contexts.

The former enter the entry function once but never leave it by return; there will be an infinite loop implemented in the function, which controls the tasks implemented in the context.

The latter execute their tasks implemented in the entry function and return from it. Returning from the entry function means terminating the context. These contexts typically expect that the entry function is repeatedly executed, from beginning till end, and either regularly or triggered by some kind of event.

To support the initial and repeated start of a context, the entry function is element of the context object. Finally, the execution mode is specified in the object. A context can be executed in either supervisor or user mode.

Both kinds of contexts can be suspended and later resumed - at any point in time and as often as suitable. kernelBuilder makes no difference between both kinds with respect to suspend and resume (i.e. normal context switches). Only start and termination make a difference. See next sections.

1.4. Creation and start of execution contexts

We saw how to switch between different execution contexts. But where do they initially come from?

One particular execution context is always there. It's the execution context from the startup code, passed on to C function `main`. To make use of the context switching capabilities of the IVOR handlers, we need a context descriptor object for the startup context (to be able to safely suspend it) and at minimum one other context.

kernelBuilder offers the service to create a new context. Three helper functions exist:

- `ccx_createContextSaveDescOnTheFly()` expects a C entry function, the execution mode (supervisor or user) and a stack area as arguments. It initializes a context object such that the context can be created and started (not resumed!) later, when a kernel relevant handler commands a context switch on return. (This way to start a context is called on the fly.) In case of single-shot tasks, the context object can be reused as often as suitable to re-start the same single-shot context [4: It needs to be granted that the previous shot has properly terminated prior to restart a single-shot context.]
- `ccx_createContextSaveDescShareStack()` is nearly the same, but the stack specification is made

indirectly by reference to another, already initialized context object: The two contexts will use the same stack area

- `ccx_createContextSaveDesc()` expects the same arguments as the first function. It initializes the context object and, additionally, it prepares the contents of the specified stack area such as if the context were already running and had then been suspended again - immediately before entering the C entry function. The context doesn't need to be started any more

`ccx_createContextSaveDescOnTheFly()` can be used for both, infinitely running and single-shot contexts.

`ccx_createContextSaveDescOnTheFly()` can be used for creating a context descriptor for the already existing startup context, in order to safely suspend it to the advantage of other, newly created contexts.

`ccx_createContextSaveDescShareStack()` is useful only for single-shot contexts because of the stack sharing; a context, which inherits the stack from another one needs to terminate before the other one may be resumed again.

The use case for `ccx_createContextSaveDesc()` are RTOS designs, where all or some of the contexts are declared and created prior to starting the kernel. The contexts are created in started but then suspended state and the scheduler doesn't need to take any care when commanding a context switch to one of these contexts. When using `ccx_createContextSaveDescOnTheFly()` the scheduler needs to distinguish between starting a context (first context switch to it) and resuming it (subsequent context switches to it)

TIP	Typical RTOS design: The kernel initialization routine calls function `ccx_createContextSaveDesc()` a number of times to create the requested number of tasks beforehand. From the system timer interrupt, when the particular due times are reached, these contexts are resumed.

The motivation of having `ccx_createContextSaveDescOnTheFly()` although `ccx_createContextSaveDesc()` can do the same, and even more convenient, is overhead. Using `ccx_createContextSaveDescOnTheFly()` is much cheaper in terms of CPU instructions and the slightly increased complexity of the scheduler logic will surely pay off for frequently started single-shot contexts.

1.5. Context switching

External Interrupts and system calls are considered events, which may yield a context switch. Most prominent example is the timer interrupt of a typical RTOS. The handler will count the occurrences and compare with the due time of different configured tasks. If the due time of a task is reached then the context, which implements the task, will be started or resumed.

The concept of kernelBuilder is that handlers for these events, which are implemented in the client code, do all the organizational work, which is required to keep track of which context should be served next and on return they will tell kernelBuilder's underlying assembler code what to do.

The return value of a handler indicates whether or not to do a context switch. If a context switch is

wanted then it'll further indicate whether to either suspend or terminate the left context and whether to start or resume the entered context.

All of this requires the specification of two context objects, one for the left context and in order to say where to store the information about the left context and the second one for the entered context. These two objects are returned by reference by the handler.

The "organizational work" inside the handlers, e.g. update of task lists, priority decisions, maintenance of pointer to active task, etc., happens necessarily before (inside the handler) the taken decision, i.e. the yielded context switch, can be performed (after return from the handler). This is no issue because of the race condition free implementation paradigm for the client scheduler code. All kernel relevant handler invocations, External Interrupts and system calls, are serialized. A handler will never be preempted by another one and even less by a context under control of the scheduler.

kernelBuilder applies the priority ceiling protocol for serializing the handler invocations, which means that kernel unrelated External Interrupts can still preempt all the scheduler code. (Therefore they must not make use of scheduler functions without additional, explicit synchronization code.)

NOTE	Because of the serialization of all kernel relevant handlers, any system call handler can be sure that the calling context always is the very one, which had been last recently started or resumed by the scheduler.

A handler, which requests a context switch on return can furthermore specify a uint32_t result value for the entered context; if this context is started then the value is the function argument of the context entry function, if it had suspended in a system call and is now resumed then the value is the return value from the system call. Otherwise the value won't have an effect.

1.6. Termination of execution contexts

On return from a system call or kernel relevant interrupt, and if it comes to a context switch, the handler can not only decide to suspend the left context but it can let it terminate, too.

The context termination service offered by kernelBuilder has nothing to do with deletion or destruction of contexts or stacks, it only adds a subtle thing to the demanded context switch: It reinitializes the stack of the terminating context. The use cases are single-shot contexts and stack sharing. Only with reinitialized stack it is safely possible to re-start a single-shot context later. And if several contexts share one stack and if one of them terminates and properly cleans up its portion of the stack then the others using the same stack can be safely resumed.

Note, context start and termination will most likely be applied to the implementation of single-shot tasks. In which case the context descriptor object remains valid even after context termination: The same object can be used just like that to command a start-of-context at next due time of the single-shot task.

Note, if a context has been terminated on return from a handler then the according context cannot be resumed again but it can be re-started.

1.6.1. Return from the context entry function

The context entry function can be left with return. It can even return a uint32_t value. Leaving the entry function is a totally different thing than commanding context termination at return from a kernel relevant IVOR handler but both things are logically connected and this connection needs to be understood for an actual scheduler implementation.

When the entry function is left then code execution branches into a callback, a global notification function, which is named `int_fctOnContextEnd()`. Its argument is the value returned from the entry function. This function is executed still in the same context as the left entry function and executing `int_fctOnContextEnd()` is the virtually last thing a context can do. [5: Returning from the end-of-context notification callback `int_fctOnContextEnd()` surely means a crash.] However, this function is not an IVOR handler, it is not executed in the scheduler context, it can not command context termination on return. Instead, the implementation of the callback in the client code will likely contain a system call which has the meaning "signal end of task". The system call implementation — again an IVOR handler — will update the scheduler's data structures to reflect the changed task state and command the context termination on return in order to do the stack cleanup.

Note, the callback is reentrant and shared by all contexts. Regardless, the client code doesn't need to implement a mechanism for signaling, which particular context invoked it and is about to terminate: The scheduler is as said race condition free and if we get into the hypothetic system call "signal end of task" then we can be sure that it is always the currently active task, which is the calling one. The scheduler knows of course, which one that is.

1.7. Stack sharing

Basically, any context will have its own stack area. This enables arbitrary switching between all contexts, any one can be suspended to the advantage of any other. The only drawback is the memory consumption. For the capacity of the stack of a context one needs to consider not only the consumption of the context's entry and all its sub-functions — there needs to be an additional headroom for preemptions by asynchronous interrupts.

The e200z4 core uses the normal stack pointer on entry into an ISR and it has up to 15 levels of preemption by External Interrupts. For sake of performance and simplicity, our IVOR #4 handler creates on entry a worst case stack frame, which already considers the space for a possible context switch on return (as opposed to enlarging the stack frame in case of an actually happening context switch). This stack frame has a size of about 170 Byte. If all 15 interrupt levels are in use then this would sum up to a required headroom of about 2.5 kByte — even if you will never be able to create a test case, which proves this.

This headroom has to be spent for any stack. Certain sub-sets of context can use one and the same stack and the headroom applies only once to all contexts in the set. This denotes the possible memory saving.

Note: Stack sharing is not at all a performance improvement in terms of execution speed. It just saves the stack headroom memory.

The support of stack sharing is enabled or disabled at compile time by configuration macro

`INT_USE_SHARED_STACKS`. A kernelBuilder project will copy file `int_interruptHandler.config.h.template` as `int_interruptHandler.config.h` into the build set and adjust the contained configuration items.

This is kernelBuilder's concept of stack sharing: Our stacks grow downwards. If a context A is preempted and for now suspended then another context B can safely use the stack area below the stack area currently in use by A. The current stack use of A is known through its stack pointer value at time of suspension. As soon as A is resumed it can make arbitrary use of the whole stack area — so B needs to have left the shared stack. Only suspending B would mean leaving B's context information on the stack for later resume. It would be overwritten by a resumed A and B would crash on an attempt to resume it. Therefore B needs to enter the scene by on-the-fly context creation and needs to leave it by termination — and all of this while A is suspended.

NOTE

Two contexts A and B can share the stack, if the scheduling strategy ensures that

- B becomes active only when and while A is suspended and
- B has terminated before A is resumed again.

This comes normally down to single-shot contexts of different priority, which do not suspend voluntarily, but this is not a must. A could well be an infinitely spinning context, which cyclically suspends. And even B may voluntarily suspend if only the scheduler keeps track that it must not activate A during the time B is suspended (but it may resume C, D, E, ...).

The standard use case of stack sharing is a simple, priority controlled RTOS not offering event passing between its tasks. This is often referred to as tasks of Basic Conformance Class. The tasks A, B, C, ..., have rising priorities. B can preempt A but never vice versa, C can preempt A and B but never become preempted by them and so on. None of the tasks needs to suspend voluntarily — there's no event to wait for — so the conditions above are fulfilled for all pairs of contexts and all of them can safely use the same stack. These considerations include even the startup context, which will become the never terminating idle task — and the entire RTOS implementation can use the ordinary, normal stack from the startup code.

With kernelBuilder, stack sharing is implemented through initialization of context descriptor objects. When initializing the object one either specifies the initial stack pointer value for the new context or another, already initialized context object — now the second context inherits the stack from the first one. This can be chained to share the stack with more contexts. In the BCC example we would start initializing the idle task's descriptor and then pass it for stack sharing to the initialization call of all the tasks' context objects.

1.8. Typical context life cycles

There are typical scenarios for contexts and context descriptor objects.

1. All tasks are declared beforehand. The initialization code will use `ccx_createContextSaveDesc()` an according number of times to create all context descriptor objects. The new contexts are created in suspended state and can be resumed by the scheduler on whatever event.

   The context entry function is never left, the tasks are implemented as forever spinning loops,

each cycle likely connected to a real-time event: The loop body makes a system call as first or very last statement that waits for the event of interest.

2. The maximum number of tasks is specified beforehand. A pool of tasks with individual stack areas is created once. A context descriptor object is created for each, preliminarily stating `NULL` as entry function.

   A system call is offered to start a task. The task entry function is argument to the call. It is stored in an otherwise ready to use context object taken from the pool. The system call handler is left with commanding the switch to the new context.

   The task is ended by making a dedicated system call. The system call handler returns the context object into the pool and on return it commands the termination of the context and the switch to any other context (maybe the idle task). The termination request ensures that the stack area specified in the context object remains properly reusable for future cycles.

   Note, it doesn't matter whether the system call for termination is still inside the context entry function or if this function is left and the system call is instead placed in the end-of-context callback `int_fctOnContextEnd()` — the former solution saves a few instructions but moves the responsibility of making the system call to the user.

3. Task pool without end-of-task notification. Scenario 2. can be implemented without applying kernelBuilder's context termination support, too. A scheduler can offer a system call to end a task and it implements it by only putting the context object back into the pool. It'll simply never consider it again for resume. What differs is the code required when later reusing a context object from the pool: Since we didn't do the stack cleanup, we need now to reinitialize the context object entirely, e.g. using `ccx_createContextSaveDescOnTheFly()`.

   Choosing scenario 2. or 3. doesn't make a significant difference in performance. If the system call is placed into the end-of-context callback then 2. is maybe a bit more elegant and less error-prone. 2. basically permitts using stack sharing for certain sub-sets of contexts, while this would be inhibited in 3.

4. The scenarios can be mixed. A number of tasks can be predefined, others can be pooled. Some tasks can be implemented by never left, forever spinning entry functions, others can be implemented as single-shot contexts, which terminate by returning from the context entry function.

1.9. Deletion of contexts

The implementation of an operating system kernel will have to deal with task creation and deletion. Our kernelBuilder doesn't do. It has no concept of memory allocation, new and free, pools of objects, etc. Therefore you will not find any support of context object deletion. For the IVOR handlers this is simply irrelevant; a no longer required context will just never be commanded again as target for resume. Whether the client code uses a free method to release the memory connected to a no longer used context or whether it returns it into an object pool for re-use is out of scope and fully in the design-sphere of the client code. [6: Even context termination is not connected to pooling and memory allocation. It just means to leave the stack of a no longer used context in a well defined state to maintain it usable for re-starting the same or resuming other, stack-sharing

contexts.]

2. Tools

2.1. Environment

2.1.1. Command line based build

The makefiles and related scripts require a few settings of the environment in the host machine. In particular, the location of the GNU compiler installation needs to be known and the PATH variable needs to contain the paths to the required tools.

For Windows users there is a shortcut to PowerShell in the root of this project (not sample), which opens the shell with the prepared environment. Furthermore, it creates an alias to the appropriate GNU make executable. You can simply type `make` from any location to run MinGW32 GNU make.

The PowerShell process reads the script `setEnv.ps1`, located in the project root, too, to configure the environment. This script requires configuration prior to its first use. Windows users open it in a text editor and follow the given instructions that are marked by TODO tags. Mainly, it's about specifying the installation directory of GCC.

Non-Windows users will read this script to see, which (few) environmental settings are needed to successfully run the build and prepare an according script for their native shell.

2.1.2. Eclipse for building, flashing and debugging

Flashing and debugging is always done using the NXP CodeWarrior Eclipse IDE, which is available for free download. If you are going to run the application build from Eclipse, too, then the same environmental settings as described above for a shell based build need to be done for Eclipse. The easiest way to do so is starting Eclipse from a shell, that has executed the script `setEnv.ps1` prior to opening Eclipse.

For Windows users the script `CW-IDE.ps1` has been prepared. This script requires configuration prior to its first use. Windows users open it in a text editor and follow the given instructions that are marked by TODO tags. Mainly, it's about specifying the installation directory of CodeWarrior.

Non-Windows users will read this script to see, which (few) environmental and path settings are needed to successfully run the build under control of Eclipse and prepare an according script for their native shell.

Once everything is prepared, the CodeWarrior Eclipse IDE will never be started other than by clicking the script `CW-IDE.ps1` or its equivalent on non-Windows hosts.

See project overview and GitHub Wiki for more details about downloading and installing the required tools.

2.2. Compiler and makefile

Compilation and linkage are makefile controlled. The compiler is GCC (MinGW-powerpc-eabivle-4.9.4). The makefile is made generic and can be reused for other projects, not only for a tiny "Hello World" with a few source files. It supports a number of options (targets); get an overview by typing:

```
cd <projectRoot>/LSM/kernelBuilder
mingw32-make help
```

The main makefile `GNUmakefile` has been configured for the build of sample "kernelBuilder". By default, the sample client application is `alternatingContexts` and the instruction set is Book E. Type:

```
mingw32-make -s build
mingw32-make -s build CONFIG=PRODUCTION
```

to produce the flashable files `bin\ppc-BookE\alternatingContexts\DEBUG\TRK-USB-MPC5643L-kernelBuilder.elf` and `bin\ppc-BookE\alternatingContexts\PRODUCTION\TRK-USB-MPC5643L-kernelBuilder.elf`.

To select the compilation of kernelBuilder with another sample client application add a term like `APP=code/samples/chainedContextCreation/` to the command line of mingw32-make.

To select the compilation for the other instruction set add `INSTR=VLE` to the command line of mingw32-make. For example, type:

```
mingw32-make -s build APP=code/samples/simpleRTOS/ INSTR=VLE CONFIG=PRODUCTION
```

to build our simple demo RTOS in VLE and PRODUCTION configuration. The flashable file is `bin\ppc-VLE\simpleRTOS\PRODUCTION\TRK-USB-MPC5643L-kernelBuilder.elf`.

NOTE The makefile requires the MinGW port of the make processor. The Cygwin port will fail with obscure, misleading error messages. It's safe to use the `make.exe` from the compiler installation archive. The makefile is designed to run on different host systems but has been tested with Windows 7 only.

Note, the Eclipse project configuration in the root folder of this sample supports the build of only a sub-set of the possible configurations. kernelBuilder can be compiled with a few sample applications only, each of them in DEBUG and PRODUCTION compilation and for either instruction set. To build the other sample applications with Eclipse you would have to duplicate the existing build configurations and adapt the make command lines in the build settings according to the explanations and examples above.

2.3. Flashing and debugging

The sample code can be flashed and debugged with the CodeWarrior IDE.

To flash the `*.elf` file, open the CodeWarrior IDE, go to the menu, click "Window/Show View/Other/Debug/Debugger Shell". In the debugger shell window, type for example:

```
cd <rootFolderOfSample>/makefile/debugger
source flashAlternatingContextsDEBUG.tcl
```

or

```
source flashAlternatingContextsPRODUCTION.tcl
```

(Or the according scripts for the other samples.) As of writing, the named flash scripts have been prepared for the Book E compilation artifacts only. The VLE binaries can be flashed only with the generic flash scripts, which take the name of the sample application and the instruction set as arguments. These are the scripts `flashDEBUG.tcl` and `flashPRODUCTION.tcl`. The arguments are APP and INSTR and they are implemented as global TCL variables, which have to be set prior to the run of the script. Type for example:

```
cd <rootFolderOfSample>/makefile/debugger
set APP simpleRTOS
set INSTR VLE
source flashDEBUG.tcl
```

Open the TCL script in a text editor to get more details.

The debugger is started by a click on the black triangle next to the blue icon "bug", then click "Debug Configurations.../CodeWarrior/kernelBuilder (simpleRTOS, VLE, DEBUG)". Confirm and start the debugger with a last click on button "Debug".

(Or select the according debug configuration for another sample application or the other instruction set.)

You can find more details on using the CodeWarrior IDE at https://github.com/PeterVranken/TRK-USB-MPC5643L/wiki/Tools-and-Installation.

3. Code architecture

kernelBuilder consists of the source code folders `code\startup` and `code\kernelBuilder`. Folder `startup` combines the code known from the other samples "startup" and "startup-VLE", only the standard IVOR #4 handler has been removed. Please refer to LSM/startup/readMe.adoc for details.

The sub-folders of folder `code\samples` contain a sample client implementation each. [7: With the exception of `common`, which contains common code of all or some of the samples.] Folder `code\serial` is the known implementation of `printf` and only used by the client code. Package `serial` was extended by a wrapper around the driver API so that it is available to contexts running in user mode. The wrapper implements the same API as system calls.

The build and debug scripts are a bit different to what you know from the other samples. They take an argument to select a client code sample; kernelBuilder itself is an infra-structure only, it is not a self-contained, flashable executable, you always need to compile it together with some client code.

The samples demonstrate preemptive and cooperative scheduling.

To see how a sample works you need to open a terminal software on your host machine. You can find a terminal as part of the CodeWarrior Eclipse IDE; go to the menu, "Window/Show View/Other/Terminal/Terminal".

Open the serial port, which is offered by the TRK-USB-MPC5643L. (On Windows, open the Computer Management and go to the Device Manager to find out.) The Baud rate has been selected as 115200 Bd in file `code\samples\*\mai_main.c`, 8 Bit, no parity, 1 start and stop Bit. The sequence \r\n is used as end of line character. The terminal should print the messages, which are regularly sent by the sample code running on the evaluation board.

3.1. Book E versus VLE

kernelBuilder is written in both, Book E and VLE assembler. The build scripts and the Eclipse configuration support both instruction sets.

The makefile takes an additional switch on the command line, state `INSTR=BOOK_E` (default) or `INSTR=VLE` to build the software in the wanted instruction set.

In the Eclipse project, all build and debug configurations have been duplicated, once for each instruction set. The TCL scripts, which can be used in CodeWarrior's debugger shell window to flash the software, have got another argument to select the instruction set, too.

3.2. API

kernelBuilder offers a C API for using it. This API is an extension to the API offered by the startup code, which is still required, too. This section outlines, which functions and data structures are available and how to use them. More detailed information is found as source code comments.

3.2.1. Registering an ISR

This modified function from the startup API lets your application define a handler for all needed External Interrupt sources.

```
#include "ihw_initMcuCoreHW.h"
void ihw_installINTCInterruptHandler( int_externalInterruptHandler_t interruptHandler
                                    , unsigned short vectorNum
                                    , unsigned char psrPriority
                                    , bool isPreemptable
                                    , bool isKernelInterrupt
                                    );
```

interruptHandler

`interruptHandler` is the C function implemented in your application, that serves a device when it raises the interrupt. The function argument's type `int_externalInterruptHandler_t` denotes a union of the two possible actual types `int_ivor4SimpleIsr_t` and `int_ivor4KernelIsr_t`.

isKernelInterrupt

`true` if `interruptHandler` is a kernel relevant ISR, `false` if it is an ordinary ISR.

In comparison to our startup sample, the signature of the function has changed to differentiate ordinary and kernel relevant ISRs. This affects the two explained arguments, all others are as they used to be, please refer to sample startup for details.

3.2.2. The ordinary ISR

The type of an ordinary ISR, which cannot command a context switch, and which will always continue the preempted context after return, is unchanged: `void (*)(void)`.

3.2.3. The kernel relevant ISR

The signature of a kernel relevant handler is:

```
#include "int_interruptHandler.h"
uint32_t (*)(int_cmdContextSwitch_t *pCmdCtxtSw);
```

Return value

On return from the handler you can command a command switch by return value and provide more details by writing to the function argument:

- Return bit `int_rcIsr_switchContext` to command a context switch
- Binary OR bit `int_rcIsr_createEnteredContext` to the return value if you want to start a new context on the fly
- Do *not* binary OR bit `int_rcIsr_createEnteredContext` to the return value if you want to resume an already created but later suspended context
- Binary OR bit `int_rcIsr_terminateLeftContext` if you want to do a cleanup of the stack of the left context. Note: Now this context is destroyed and can never be resumed but its context descriptor object is still valid and can be used to re-create the context again later on the fly
- Do *not* binary OR bit `int_rcIsr_terminateLeftContext` if you want to suspend the left context so that it can be resumed later
- Return zero (or `int_rcIsr_doNotSwitchContext`, which is the same) to not switch context. The ISR continues the preempted context like an ordinary ISR always does. `*pCmdCtxtSw` doesn't care

pCmdCtxtSw

If the return value is non zero then `*pCmdCtxtSw` needs to be filled with information about the two affected contexts. For both contexts, the pointer to the descriptor object is specified. Additionally, a uint32_t value can be set, which is signaled to the resumed or created context as result of a system call or as argument of the entry function, respectively. Setting the value is optional; it would have no effect if the entered context had been preempted and suspended by an External Interrupt.

3.2.4. Creating context descriptors

All context switches, all context suspend and resume operations or commanded and performed with help of the context descriptor objects. A context descriptor is not equivalent with a context; any context has a related descriptor but — in case of single-shot contexts — a descriptor can be related to an infinite series of contexts. (However, only one at a time.)

Suspended context

To create the descriptor of a context, which is already created and suspended, so that it can immediately be used for a context resume command, use:

```
#include "ccx_createContextSaveDesc.h"
void ccx_createContextSaveDesc( int_contextSaveDesc_t *pContextSaveDesc
                              , void *stackPointer
                              , int_fctEntryIntoContext_t fctEntryIntoContext
                              , bool privilegedMode
                              );
```

pContextSaveDesc

The context descriptor object by reference. Its contents are written by the function.

stackPointer

The top address of the aimed stack area. Points to the first address beyond the reserved space. Memory allocation for the stack is in the responsibility of the calling client code.

fctEntryIntoContext

The context's entry function. An ordinary C function `uint32_t (*)(uint32_t)`.

privilegedMode

`true` for supervisor or privileged mode, `false` for user or problem mode. This is the execution mode for the new context. Each context can use its individual mode.

On-the-fly created context and startup context

To create the descriptor for a context, which is not created yet and which requires on-the-fly creation, use `ccx_createContextSaveDescOnTheFly()`. The same function is applied to create a descriptor for the always created and existing startup context:

```
#include "ccx_createContextSaveDesc.h"
void ccx_createContextSaveDescOnTheFly
                        ( int_contextSaveDesc_t *pNewContextSaveDesc
                        , void *stackPointer
                        , int_fctEntryIntoContext_t fctEntryIntoOnTheFlyStartedContext
                        , bool privilegedMode
                        );
```

The function arguments are identical to `ccx_createContextSaveDesc()`.

On-the-fly created context with shared stack

To create the descriptor for a context, which will be created later on the fly and which shares the stack with another context, use:

```
#include "ccx_createContextSaveDesc.h"
void ccx_createContextSaveDescShareStack
                        ( int_contextSaveDesc_t *pNewContextSaveDesc
                        , const int_contextSaveDesc_t *pPeerContextSaveDesc
                        , int_fctEntryIntoContext_t fctEntryIntoContext
                        , bool privilegedMode
                        );
```

pPeerContextSaveDesc

An already created context descriptor object, which the new context will share the stack with.

The other function arguments are identical to `ccx_createContextSaveDesc()`.

3.2.5. The handler for the simple system call

The handler, which implements the behavior of a simple system call is a function, which gets a variable list of arguments from the client code plus the calling context's (modifiable) machine status:

```
#include "int_interruptHandler.h"
uint32_t (*int_simpleSystemCallFct_t)(uint32_t * const pMSR, ...)
```

Return value

The value returned by the handler is the return value of the system call for the calling client code.

pMSR

The machine status word relates to the context, which makes the system call. It is an input/output argument. On return from the handler, the returned word `*pMSR` will be used when continuing the system call making context.

Note, the system call handler itself is always executed in supervisor mode and with External Interrupt handling enabled, regardless of the state of the corresponding bits in `*pMSR`. This means in particular, that some client code can not span a critical section across a simple system call by means of suspending all interrupts.

(...)

The subsequent function arguments are those passed from the client code. Please, see Section 3.2.8 for details and constraints.

3.2.6. The handler for the kernel relevant system call

The handler, which implements the normal, kernel relevant system call is a function, which gets a variable list of arguments from the client code. The returned values are identical to those of the kernel relevant ISR:

```
#include "int_interruptHandler.h"
int_retCodeKernelIsr_t (*int_systemCallFct_t)( int_cmdContextSwitch_t *pCmdCtxtSw
                                             , ...
                                             );
```

Return value

On return from the handler and alike the kernel relevant ISR, you can command a command switch by return value and provide more details by writing to the function argument `*pCmdCtxtSw`. See Section 3.2.3 for details.

pCmdCtxtSw

See return value and Section 3.2.3 for details.

Note, the system call handler is always executed in supervisor mode and with External Interrupt handling enabled. However, the handling of all kernel relevant ISRs is disabled. It is generally impossible for some client code to span a critical section across a system call by means of suspending all interrupts.

(...)

The subsequent function arguments are those passed from the client code. Please, see Section 3.2.8 for details and constraints.

3.2.7. Configuration of system calls

The configuration is made by two static, compile-time defined tables:

```
#include "int_interruptHandler.h"
const SECTION(.rodata.ivor) int_simpleSystemCallFct_t
int_simpleSystemCallHandlerAry[];
const SECTION(.rodata.ivor) int_systemCallFct_t int_systemCallHandlerAry[];
```

The tables are external to the implementation of kernelBuilder; the client code will define them. The constant objects are filled by an initializer expression, which lists the function pointers to the handlers for all actual system calls.

If the compilation configuration is `DEBUG` then there are two more external declarations of kernelBuilder that need to be satisfied by the client code. The sizes of the two arrays are specified; kernelBuilder contains assertions that double-check at run-time that the table index specified when making a system is in bounds:

```
#include "int_interruptHandler.h"
extern const uint32_t int_noSystemCalls;
extern const uint32_t int_noSimpleSystemCalls;
```

NOTE

There is a significant difference between configuring ISRs and system calls. The former can be registered at run-time, while the latter require a less flexible constant initializer expression. This can be considered a bad design decision; the original intention where kernel relevant system calls into the scheduler, so system calls from a limited scope only, which a simple centralized table is appropriate for. Soon it turned out that simple system calls are required, too, for the implementation of kernel unrelated I/O drivers. I/O drivers are typically designed as independent compilation units and due to the centralized configuration table they are now forced into an unwanted relationship.

For now, the samples propose a stringent way of using header files and preprocessor definitions so that the I/O drivers can still be kept self-contained and independent of one another. They export only their individual contribution to the required initializer expressions but do not care about the table object. A scheduler owned module can define the table but referring to all the I/O driver's contributions.

On the long term, we may need a dynamic configuration alike the ISRs even if this is on cost of additional RAM usage.

3.2.8. Making a system call

From the client source code, a system call is made using:

```
#include "int_interruptHandler.h"
uint32_t int_systemCall(int32_t idxSysCall, ...);
```

idxSysCall

The index of the system call. For simple system calls this is at the same time the index into configuration table `int_simpleSystemCallHandlerAry`. For kernel relevant system calls it is at the same time the one's complement of the index into configuration table `int_systemCallHandlerAry`. (The latter use the negative numeric range of `idxSysCall`; the first entry into table `int_systemCallHandlerAry` would be addressed to by index -1, the second one by -2, and so forth.)

(...)

The subsequent function arguments are not interpreted by `int_systemCall()` but passed on to the system call handler, i.e. the function found in the configuration table at given index.

Caution, the assembler code, which implements `int_systemCall` doesn't fully implement the C ellipsis. It only supports the simple but common situation, where each function argument is conveyed in the next GPR of the CPU, beginning with r3 and till r10. The assembler code will fail to pass the system call arguments to the handler if it has more than seven arguments (r3 holds the

system call index) or if the arguments are not simple types of no more than 32 Bit length.

Note, system calls are solely made from the task body of the aimed RTOS. From inside the kernel implementation it is not allowed and useless to make the system call.

3.2.9. Mutual exclusion, critical sections

In any multi-threaded environment, which can be designed with kernelBuilder, there will be the need for well-controlled mutual exclusion of contexts. Code in different contexts, which accesses the same, shared resources (mostly shared memory) needs to form a "critical section".

Our startup code offers some typical mechanisms to implement mutual exclusion. The offered mechanisms are:

- Unconditional interrupt disable: `ihw_suspendAllInterrupts()`/`ihw_resumeAllInterrupts()`
- Nestable interrupt disable: `ihw_enterCriticalSection()`/`ihw_leaveCriticalSection()`
- Lock-free data exchange using memory barriers: `atomic_thread_fence()`

Please find the details in the other sample startup.

These mechanisms may be used from the code that implements a kernelBuilder context, too, but some restrictions apply.

The two pairs of interrupt disable functions make use of privileged instructions and require supervisor mode. They must not be used in contexts, which have been started in user mode. An exception would result.

`atomic_thread_fence()` can be applied in user mode, too. (Not proven.)

A typical kernelBuilder application, which wants to run contexts in user mode, will offer a pair of simple system calls to enter and leave a critical sections. The would just return the wanted, modified machine status for continuation of the calling context without or with handling of External Interrupts.

3.2.10. Consistency of interface assembler/C

kernelBuilder is written in assembler but it exposes a C API. This is possible due to the EABI specification, which contains a model of how a C compiler needs to interfere with machine code. A risk still arises from mixing C and assembler. An interface has at least two sides. If one side is changed without awareness and according modification of the other side then the use of this interface will fail. In a sheer C program the compiler is able to check this since both sides use the same header file. However, if our assembler code changes without careful update of the C header or if there's a revision mismatch then neither the assembler nor the compiler will report a problem and the likelihood of crashing code is high.

In the C header, there's a macro defined, `INT_STATIC_ASSERT_INTERFACE_CONSISTENCY_C2AS`, which wraps a (lengthy) assertion that double-checks a lot of assumptions, the assembler code is based on. It's mostly about size of data structures and size and offset of their fields. An according change of the assembler code without a change of the C header, would be detected by the macro.

It is strongly recommended putting the macro somewhere in the C code of every kernelBuilder application. The macro belongs as a statement into a function body. Any C module, which anyway includes the header `int_interruptHandler.h` is fine.

The macro expands to a `_Static_assert` so it'll not produce any machine instruction in the binary artifacts; it'll just let the compilation abort if there's a mismatch.

4. Simple sample code

The simplest possible kernelBuilder application is simpleSampleFromReadme. The main implementation file is included here as a short yet complete sample. At the linked location you will find all the sub-ordinated remaining files as a buildable, flashable and executable project.

```
/**
 * This kernelBuilder sample implements the most simple RTOS. There is one task
besides the
 * idle task. This task is a real time task in that it is executed every 100ms. Both
tasks
 * regularly print a hello world message. (Serial port at 115200 Bd, 8 Bit, 1 Start, 1
Stop
 * bit)
 */
(...)

/*
 * Defines
 */

/* System call index: Terminate context. */
#define IDX_SYS_CALL_TERMINATE_TASK  (-1)
(...)

/*
 * Data definitions
 */

/** We have two tasks, there are two context descriptors. */
static int_contextSaveDesc_t _contextSaveDescIdle, _contextSaveDescTask100ms;

/** The scheduler always keeps track, which context is the currently active one. */
static bool _isTask100msRunning = false;

/** Overrun counter for task activation. */
volatile unsigned int rms_cntOverrunTask100ms = 0;

/** The table of C functions, which implement the kernel relevant system calls. */
const SECTION(.rodata.ivor) int_systemCallFct_t int_systemCallHandlerAry[] =
    { [~IDX_SYS_CALL_TERMINATE_TASK] = (int_systemCallFct_t)sc_terminateTask,
    };
```

```
(...)

/*
 * Function implementation
 */

/**
 * This is the RTOS system timer, called once a 100 ms.
 */
static uint32_t isrRTOSSystemTimer(int_cmdContextSwitch_t *pCmdContextSwitch)
{
    /* Acknowledge the timer interrupt in the causing HW device. */
    PIT.TFLG0.B.TIF = 0x1;

    /* Create task context if (already) possible, otherwise report overrun. */
    if(_isTask100msRunning == false)
    {
        /* No race conditions inside scheduler: We can use ordinary variables to
           maintain our state. */
        _isTask100msRunning = true;

        /* Command a context switch from idle to task100ms. */
        pCmdContextSwitch->pSuspendedContextSaveDesc = &_contextSaveDescIdle;
        pCmdContextSwitch->pResumedContextSaveDesc = &_contextSaveDescTask100ms;
        pCmdContextSwitch->signalToResumedContext = (uint32_t)rms_cntOverrunTask100ms;
        return int_rcIsr_switchContext | int_rcIsr_createEnteredContext;
    }
    else
    {
        ++ rms_cntOverrunTask100ms;
        return int_rcIsr_doNotSwitchContext;
    }
} /* End of isrRTOSSystemTimer */



/**
 * Start the interrupt which clocks the RTOS.
 */
static void enableRTOSSystemTimer(void)
{
    /* Disable all PIT timers during configuration. */
    PIT.PITMCR.R = 0x2;

    /* Install the interrupt handler for cyclic timer PIT 0. */
    ihw_installINTCInterruptHandler
                ( (int_externalInterruptHandler_t){.kernelIsr = &isrRTOSSystemTimer}
                , /* vectorNum */ 59 /* Timer PIT 0 */
                , /* psrPriority */ 1
                , /* isPreemptable */ true
                , /* isKernelInterrupt */ true
```

```
                );

    /* Peripheral clock has been initialized to 120 MHz. Set value for a 100ms tick.
*/
    PIT.LDVAL0.R = 12000000-1;

    /* Enable interrupts by this timer and start it. */
    PIT.TCTRL0.R = 0x3;
    PIT.PITMCR.R = 0x1;

} /* End of enableRTOSSystemTimer */



/**
 * The implementation of our system call to terminate the task (to keep the context
 * descriptor usable for the next creation).
 */
static uint32_t sc_terminateTask(int_cmdContextSwitch_t *pCmdContextSwitch)
{
    /* No race conditions inside scheduler: We can use ordinary variables to maintain
       our state. */
    assert(_isTask100msRunning);
    _isTask100msRunning = false;

    /* Command a context switch from task100ms to idle. */
    pCmdContextSwitch->pSuspendedContextSaveDesc = &_contextSaveDescTask100ms;
    pCmdContextSwitch->pResumedContextSaveDesc = &_contextSaveDescIdle;
    return int_rcIsr_switchContext | int_rcIsr_terminateLeftContext;

} /* End of sc_terminateTask */



/**
 * Our 100ms single-shot task. This function is invoked every 100 ms in user mode.
 *   @param taskParam Data provided at creation of task context. Here: Number of lost
 * activations.
 */
static _Noreturn uint32_t task100ms(uint32_t taskParam)
{
    static unsigned int cnt_ = 0;
    printf("%s: %us, %lu lost activations so far\r\n", __func__, cnt_++/10,
taskParam);

    /* We terminate explicit in order to keep the sample one function shorter. */
    int_systemCall(IDX_SYS_CALL_TERMINATE_TASK);
    assert(false);

} /* End of task100ms */
```

```
/**
 * Main entry point into the scheduler. There are two tasks. The idle task, which
 * inherits the startup context and one real time task. The latter is a single-shot
 * task, which is called every 100ms and which shares the stack with the idle task.
 */
void _Noreturn rms_scheduler(void)
{
    /* Create a context descriptor of the startup context (idle task). */
    ccx_createContextSaveDescOnTheFly( &_contextSaveDescIdle
                                     , /* stackPointer */ NULL
                                     , /* fctEntryIntoOnTheFlyStartedContext */ NULL
                                     , /* privilegedMode */ true
                                     );

    /* Create a context descriptor for the other task: Single-shot, share stack. */
    ccx_createContextSaveDescShareStack
                                ( &_contextSaveDescTask100ms
                                , /* pPeerContextSaveDesc */ &_contextSaveDescIdle
                                , /* fctEntryIntoOnTheFlyStartedContext */ task100ms
                                , /* privilegedMode */ false
                                );

    /* All contexts are ready for use, we can start the RTOS system timer. */
    enableRTOSSystemTimer();

    /* We continue in the idle context. */
    while(true)
    {
        volatile unsigned long u;
        for(u=0; u<2500000; ++u)
            ;
        printf("%s: This is the idle task\r\n", __func__);
    }
} /* End of rms_scheduler */
```

5. Known issues

Debugging the interrupt controller

In the Code Warrior debugger, if the view shows the interrupt controller (INTC0) register set then the debugger harmfully affects program execution and the code fails: The write to INTC_EOIR_PRC0, which normally restores the current priority level INTC_CPR_PRC0, now fails to do so.

This effect can be observed with other samples, too.

Workaround: Don’t open the view of the INTC0 in the debugger when debugging an RTOS application. Then the INTC and the code work fine.