Innovasic Semiconductor
fido1100
Application Note 110

Time and Space Partitioning on the fido1100 Microcontroller

Version 1.0
December 2006

Table of Contents

Index of Tables

Introduction

Any time that a microprocessor is used for more than one discrete function or application, the problem of developing and verifying applications becomes dramatically more complex and expensive. Whether there are intentional interactions between the applications or not, a change to one application generally means some degree of regression testing on all of the others. In a high reliability environment this problem is even more severe.

The problem of one application impinging on another is also exacerbated when there are applications with varying degrees of reliability or safety requirements, or when various developers are responsible for different applications (e.g. the OEM develops a set of basic applications, the end user adds some additional programming).

Approach

One approach to mitigating these issues is to use what is commonly referred to as 'time and space partitioning'. Space partitioning refers to some sort of hardware-enforced limitations on memory access (memory being the 'space') to keep the various applications from inadvertently corrupting each other's data. Time partitioning refers to a mechanism that guarantees that each application will get some predetermined portion of processor throughput.

The usual mechanism for implementing a time and space partitioned system is to use one of a handful of operating systems that support it. They use a Memory Management Unit (MMU) to provide space partitioning, and typically provide time partitioning by using a time-sliced kernel, with each slice of time preallocated to a particular application. While these operating systems can be very effective in high-end systems, when cost is a factor they have drawbacks:

- The operating systems are expensive.
- Require high end processors with a full MMU and the throughput/clock speed to handle the additional overhead.
- Overhead incurred with time partitioning is substantial, requiring more throughput.
- Typically, all time is allocated, no ability to recover slack time for background tasks.

Preemptive Kernel

The fido1100 includes a priority-based preemptive scheduler in hardware which supports up to 5 'contexts'. In addition to the basic scheduler, the processor includes a system timer to support periodic task operation, a Memory Protection Unit (MPU) to control memory accesses on a per-context basis, and a set of timers that can be used to guarantee that no context uses more execution time than is allocated to it. The remainder of this application note will describe how these features can be used to provide time and space partitioning on a low-cost microcontroller.

Hardware Kernel Basics

The fido1100 supports up to 5 hardware contexts. Each context includes user registers, program counter, etc., context-specific timers, and MPU controls. Contexts are numbered 0 through 4, where context 0 is also known as the master context. The master context (context 0) differs from the others in the following ways:

- Context 0 always has the highest priority – if it is ready to run, it will run.
- Context 0 has write access to a number of configuration registers and fields of registers that can not be written by any other context.
- Context 0 handles certain classes of critical events (double bus fault, double MPU fault, etc.).
- Context 0 does not have a claim register (hardware semaphore).
- The fido1100 TRAPX instruction traps to context 0.

Because the master context has special privileges and serves as the arbiter of last resort for critical failures in other contexts, it is typically reserved for the most critical code in a given application (or reserved to only manage critical failures after setup is complete). The only distinction between context 0 and the other contexts is that a second-level priority is based on their context number. That is, if two contexts are set to the same priority, the lower-numbered context gets the processor.

Each context has an associated context control register. This register has two fields: State and Priority, these are described below.

All contexts are always in one of three states:

- Halted: Context is not ready to run – this is the state of all contexts except the master following a reset, it can be put back into the halted state by the master context (a write to the context control register), or by experiencing a critical fault (double bus fault, double MPU fault, context timer timeout).
- Waiting: The context is not ready to run, it can be made ready to run by any interrupt event that is associated with that context. A context can transition itself into this state, from the Ready state, by using the fido1100 SLEEP instruction.
- Ready: The context is ready to run, if it is the highest priority context in this state, then it will take the processor and execute.

Each context has an associated priority. Priority is a number from 0-7, where 7 is the highest priority. Note that context 0's context control register will always return a 7 in this field. There are other elements of the fido1100 that can dramatically impact code execution performance, these are the Direct Memory Access (DMA) controller and the external bus controller interface. Code execution is slowed when the DMA controller is active because they are *generally* sharing the same bus, when an external bus controller has been granted the bus all external accesses by the fido1100 are halted, again causing a dramatic slowdown in execution rate.

For this reason, these other elements (the DMA controller and external master interface) are provided priority fields. For example, if the DMA controller has a higher priority than the currently executing process, then the DMA controller will typically use the majority of bus bandwidth. If however, a higher priority context goes active, then the DMA engine stops processing the original

context and allows the processor to execute at full speed. A similar scheme exists for the external bus master interface.

When defining a system that uses time partitioning and DMA or an external bus master, their impact on performance must be taken into account and priorities set appropriately.

Another feature of the fido1100 that supports time and space partitioning is the association of interrupts with specific contexts. Any interrupt (external interrupts, I/O interrupts, timer interrupts, etc.) can be associated with any context. Thus, any time spent processing a given interrupt is accounted to the appropriate context; any failures that occur in that processing (e.g. Illegal memory access, stack overflow, etc.) are isolated to that context.

Memory Protection Unit Basics

The Memory Protection Unit (MPU) provides resources to define up to sixteen regions of memory with various attributes (Read/Write, Read-only, No Access). Multiple sets of rules can be applied to a single range of memory. For example, memory can be designated Read/Write for one context and Read-only for all other contexts.

The sixteen regions of memory are not context-specific. Each context has a 16-bit register that defines which set of rules apply to it (this register, like most configuration registers, can only be written by the master context). A region can be sized from 64 bytes to Giga bytes. The following table shows an example setup. Notice block 0 – because it covers the entire address space and is not assigned to any particular context, it will cause an MPU fault if a context accesses an area not included in a block assigned to it. Without that entry, any part of the address range that is not covered by any MPU block is read/write accessible to any context.

A typical implementation would define an MPU handler in each context that logs an error ("this context made an illegal access to this memory location ...") then trap to the master context. Alternatively, the program could try to recover within that context (or the master context can re-start the faulted context from scratch). If a context goes way out of bounds (e.g. its stack pointer runs outside the bounds defined by the MPU) it will generally cause a double fault (for example, if the stack goes out of bounds and causes an MPU fault, the fault will double up when the exception processing tries to stack the processor state). This double fault will trap to the master context which can handle the problem (e.g. re-start the faulting application).

MPU Block	*Address Range*	*Description*	*Rules*
15		Spare	Not Enabled
14	0x010XXXXX - 0x010XXXXX	Context Z I/O space	Context Z read/write
13	0x010XXXXX - 0x010XXXXX	Context Y I/O space	Context Y read/write
12	0x010A0XXX - 0x010A0XXX	Context X I/O space	Context X read/write
11	0x01004000 - 0x01005FFF	Context 1 fast RAM	Context 1 Read/Write
10	0x01002000 - 0x01003FFF	Context 4 fast RAM	Context 4 Read/Write
9	0x01000000 - 0x01001FFF	Context 3 fast RAM (on-chip)	Context 3 Read/Write
8	0x00300000 - 0x0030FFFF	Context 3,4 shared	Context 3 Read/Write, Context 4 Read-only
7	0x00280000 - 0x002FFFFF	Context 4 Local RAM	Context 4 Read/Write
6	0x00240000 - 0x0027FFFF	Context 3 Local RAM	Context 3 Read/Write
5	0x00230000 - 0x0023FFFF	Context 2 Local RAM	Context 2 Read/Write
4	0x00220000 - 0x0022FFFF	Context 1 Local RAM	Context 1 Read/Write
3	0x00210000 - 0x0021FFFF	Context 0 Local RAM	Context 0 Read/Write
2	0x00200000 - 0x0020FFFF	Global RAM	Read/Write, all contexts
1	0x00100000 - 0x001FFFFF	Code (flash)	Read-only – all contexts
0	0x00000000 - 0x7FFFFFFF	Catch any accesses outside of defined region	No context

Table 1: Example Setup Table for Memory Regions

Context Timer Basics

The fido1100 includes a set of timers that measure run-time on a per-context basis. That is, when a context is executing, its associated context timer is incremented once for each 16 system clock periods. Each context also has a max time register, when the context timer is equal to the max time register, the corresponding context is halted and a fault is generated to the master context. This setup allows the programmer to define how much processor time a given context is allowed.

Context Timer Example

Suppose that processing in a set of applications is defined to operate at the following rates:

Context	*Period*	*Priority*	*Max Time*	*Comments*
0	sporadic	7	10 msec	Handles interrupts and faults – no particular frequency
1	sporadic	5	15 msec	Also a sporadic interrupt handler
2	10 msec	4	30 msec	100 Hz periodic processing
3	30 msec	2	20 msec	33.3 Hz periodic processing
4	80 msec	1	40 msec	12.5 Hz periodic processing
5	background	0	N/A	uses any available processor time. (at least 240 – 115 = 125 msec)

Table 2: Context Timing Rate Example

A typical approach might be to make the highest-frequency context the highest priority of periodic contexts.

Given the periods defined above in Table 2, you would choose an overall period that works out evenly with all of the periodic processes, in this case 240 msec (24 iterations of context 2, 8 iterations of context 3, and 3 iterations of context 4). Next break up the available 240 msec of run-time into the portions allowed for each context (in the Max Time column).

Now you guarantee that if one of the higher priority contexts crashes, the lower priority contexts will still be able to execute. For example, it may be that the application running at 33.3 Hz is of less consequence in terms of criticality than the application running at 12.5 Hz, or even the background processes. By using the context timers to guarantee time partitioning you can guarantee that the lower-priority applications will get an appropriate amount of processor time.

For this to work, the context timers must be cleared periodically (in this example, every 240 msec). This can be accomplished in software, by a write to the context timer clear register from the master context. It can also be accomplished with no software intervention by programming the System Timer to have a 240 msec period on one of its channels with automatic context timer clear enabled. When using the system timer to clear the individual context timers, the system can enforce strict time limitations on any set of contexts with no periodic software overhead.

Conclusion

After the MPU and context timers have been set up, applications running in different contexts can be guaranteed not to interfere with each other in terms throughput (hogging the processor) or memory (stepping on each other's data or I/O) with no software overhead beyond initializing the timers and MPU. This provides the following:

- Higher reliability.
- Reduced verification costs.
- Mix software certified to various levels.
- Reduced system debug time (problems contained in a single application).

Thank You

Thank you for taking the time to review this application note. We hope you have found the information included in this application useful and east to understand. Please feel free to contact the Innovasic Support Team any time with questions or comments.

Innovasic Support Team
3737 Princeton NE
Suite 130
Albuquerque, NM, 87107

(505) 883-5263

support@innovasic.com
http://www.innovasic.com