Inter-Context Communications with the fido1100 Microcontroller

An Innovasic Semiconductor Application Note
fido1100 Application Note 160

Version 1.1

December 2006

Innovasic Semiconductor

Table of Contents

Abstract

This application note address how communications takes place between the Hardware Contexts on the fido1100 microcontroller.

Approach

Memory-Based Communications

The **fido1100** has a flat memory space. All contexts can read/write all memory except as restricted by the Memory Protection Unit (which can enforce different rules for each context). Thus, the programmer is free to use data structures in memory to communicate between contexts.

The **fido1100** instruction set includes a variety of read-modify-write instructions (test and set, add to memory, subtract from memory, AND memory, OR memory, XOR memory, etc.) which are atomic in their operation with respect to context switches. This allows the creation of a rich set of communication primitives that are safe for inter-context use.

Hardware Assisted Communications

In addition to memory-based communications, the **fido1100** provides specific hardware support for inter-context communications.

Software Interrupts

Each context has a set of registers associated with it that provide a 'software interrupt' capability. For example, if context 4 enables its software interrupt (by writing its Software Interrupt Control Register), then another context writes to context 4's Software Interrupt Actuation register, an interrupt will be generated to context 4.

This approach is especially useful because it allows a context that is not currently ready to execute (it has executed a 'sleep' instruction) to be made ready by another context, but only if the interrupt is enabled.

Claim Registers

Another hardware feature, the Claim register, provides the functionality necessary to implement mutual exclusion (mutex) semaphores for contexts. Using this feature the programmer can share resources between multiple contexts without fear that they will corrupt each other's efforts.

Context 0 does not have a claim register associated with it. This is due to the fact that the master context must be able to respond to critical events (e.g. fatal errors in other contexts) at all times, therefore it is not appropriate to have it pend on a semaphore held by a lower priority context.

Each other context (1-4) has an associated Claim register. The register includes a 12-bit Object ID field. Thus the designer can create up to 4K inter-context semaphore objects. Operation is as follows:

- When a context wishes to take a claim on an object, it writes that object id to its claim register.
- Hardware compares the object ID to the contents of all other claim registers.
    - If there is no match then the ID is written to the claim register and the context continues execution.
    - If there is a match the the ID is not written then the following sequence occurs
        - The writing context is held pending until the context that holds the claim clears its claim register
        - If the priority of the writing context is higher than the priority of the context holding the claim, then the priority of the writing context is inherited by the context holding the claim until the claim is relinquished.
- When a context is ready to relinquish a claim on an object, it writes a zero to its claim register.
    - The claim register is cleared to zero
    - If the context inherited a higher priority from another context trying to take the same claim, its priority is returned to its original level
    - Any contexts pending on the object ID just cleared are removed from the pending state and allowed to executed (if they have sufficient priority).

If the programmer wishes for the context relinquishing a claim to immediately sleep and give up the processor, he should write a value of 0xFFF to the claim register to relinquish. This operates the same as the sequence described above for a write of 0x000, except that the context is also moved to the NOT READY state. This feature can be used to avoid race conditions between relinquishing a claim and pending interrupts, etc.

The following example will help to illustrate operation of the claim register.

- Context 1, at priority 1, writes 0x12 to its claim register
- Context 1 reads a 0x12 from the claim register and continues executing, having been granted the claim for object ID 0x12
- Context 2, at priority 4, is enabled by an interrupt and takes the processor.
- Eventually Context 2 writes 0x12 to its claim register
- Context 2 is held pending, Context 1 inherits its priority of 4 and resumes execution.
- Context 3, at priority 5, is enabled and takes the processor.
- Context 3 writes 0x12 to its claim register

- Context 3 is held pending, Context 1 inherits its priority of 5 and resumes execution.
- Context 1 finishes with the protected object and writes a 0 to its claim register.
- Context 1 is dropped to its original priority (1), Contexts 2 and 3 are released from their pending state.
- Context 3 gets the processor (because it is highest priority), and reads a zero from its claim register
- Context 3 again writes a 0x12 to the Claim Register and is granted the claim.
- Context 3 reads a 0x12 from its claim register and continues execution.
- Context 3 finishes its protected operation and writes a zero to its claim register.
- Context 3 eventually finishes its current function and issues a 'sleep' instruction.
- Context 2 now gets the processor and reads a 0 from its claim register.
- Context 2 writes 0x12 to its claim register and reads back a 0x12.
- Context 2 does its processing and eventually writes a 0 to its claim register.
- Now there is not pending request for object ID 0x12.

So, a typical code fragment used to take a claim (or is it 'stake'?) looks like this:

```
int ClaimTake(int objectID)
{
  int successful = 0;
  int context = *FIDO_CURRENT_CONTEXT_REGISTER;
  int *claimRegisterp;

  if (objectID > 0xFFE)
        return 1;   // illegal object ID
  switch (context) {
  case 1: claimRegisterp = FIDO_CONTEXT_1_CLAIM;
        break;
  case 2: claimRegisterp = FIDO_CONTEXT_2_CLAIM;
        break;
  case 3: claimRegisterp = FIDO_CONTEXT_3_CLAIM;
        break;
  case 4: claimRegisterp = FIDO_CONTEXT_4_CLAIM;
        break;
  default: return 1;  // can't claim in this context
  }

  while (!successful) {
    *claimRegisterp = objectID;
        if (*claimRegisterp == objectID)
      successful = 1;
  }
  return 0;
}
```

TrapX

The **fido1100** instruction set includes a new form of trap instruction, the trapx. The syntax is 'trapx xx' where xx is a number from 0-15. Each number causes a different handler to be accessed.

When this instruction is issued in context 1-4, it is handled in the master context (context 0). It can be used in the same way that a trap instruction is used on a traditional, single-context, processor. For example, if the programmer wishes to perform some action that can only be accomplished by the master context (e.g. write to a register with restricted access), or to lock out operation of other contexts (i.e. Put a critical section in the trapX handler).

The handler itself can determine which context initiated the trap by reading the Faulted Context register.

Conclusions

A number of powerful mechanisms are provided to support multi-context applications with complex interactions. This Application Note describes some of the key concepts used in the **fido1100** architecture to enable the customer to develop powerful applications using hardware contexts.

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