AMX™ Timing Guide and Data

for

AMX MA32 Multitasking Executive

First Printing: June 1, 2001
Last Printing: November 1, 2002

KADAK Products Ltd.
206 - 1847 West Broadway Avenue
Vancouver, BC, Canada, V6J 1Y5
Phone: (604) 734-2796
Fax: (604) 734-8114

DISCLAIMER

TRADEMARKS

AMX TIMING GUIDE and DATA

Table of Contents

This page left blank intentionally.

1. AMX Timing Issues

The current release of KADAK's AMX™ Multitasking Executive is the fifth in a long series spanning more than 20 years of development. With AMX coded in C and with the increasing use of RISC processors, kernel timing metrics have become increasingly difficult to define and measure.

It is the purpose of this AMX Timing Guide to provide actual AMX timing information and to introduce the factors which affect these performance figures. Since instruction counts and cycle counts can no longer be used to measure execution times, it is necessary to provide specific measurements and to define, in detail, the conditions under which they were made. The remainder of this chapter will introduce the major factors which make timing measurement so difficult.

Chapter 2 introduces the timing tests used to measure typical task switching times in commonly occurring task to task synchronization scenarios. Chapter 3 defines the processing overhead necessary to support AMX timing services. Chapter 4 introduces the topic of interrupt latency and illustrates typical interrupt service timing.

Timing data sheets are provided in Chapter 5. The data sheets provide specific timing figures for all of the measurements described in Chapters 2 to 4. The data sheets identify the exact characteristics of the hardware on which the measurements were made and describe the C compiler options used to compile AMX.

Processor Effects

Today's RISC and superscalar microprocessors use many techniques to boost instruction execution speeds. Internal instruction queues are used to allow instruction fetch and decode operations to proceed while previously decoded instructions are executing. Multiple execution units are often employed to permit simultaneous, parallel execution of instructions. Register scoreboarding may occur so that execution only stalls when a particular instruction needs a result from another incomplete instruction.

Branch prediction may be used to force the processor to assume a particular execution path which is the expected norm. A time penalty is then incurred only in the rare cases when the prediction is in error.

Processors may use multiple register sets or register windowing schemes to improve procedure parameter passing and to minimize the register preservation requirements of procedures.

Memory Effects

The processor's interface to memory is the single greatest impediment to full speed instruction execution. Unless instructions can be fetched from memory with zero delay, instruction execution will stall.

Various techniques are used to improve memory access. The most obvious is the use of high speed memory devices which are matched to the processor clock speed. Unfortunately, cost often precludes this preferred solution.

Most high performance processors provide an autonomous memory control unit capable of independent execution. High speed, multi-ported memory buffers may be used to allow burst transfers of small blocks of data to or from memory in anticipation of the processor's needs. In some cases, the block transfer size is configurable. The manner in which instructions straddle these block boundaries can directly affect performance. Minor shifts of code up or down in memory may noticeably affect execution time.

The processor can minimize the effects of slow data memory if instructions with no external memory references can be executed while previous memory accesses complete. However, this technique depends on judicious ordering of instructions to meet the memory access constraints, a requirement which cannot always be met. The instruction ordering burden falls on the assembly language programmer or on the high level language code generator.

Caches

Another common technique used to improve execution speed is instruction and/or data caching. The instruction cache is high speed memory internal or external to the processor which is used to hold copies of instructions as they are fetched from memory. The data cache is high speed memory internal or external to the processor which is used to buffer data on its way to or from external memory.

The entire address space maps to the cache in some processor dependent fashion. The cache includes control information which identifies whether each instruction or data element in the cache is valid. When the processor fetches an instruction or data element that is already marked as valid in the cache, the cache copy is used with no time penalty and no external memory interference.

The instruction cache is usually internal to the processor and hence fixed in size. Some processors also include an internal data cache. When an external cache is used, the size of the cache is usually determined by the manner in which it is interfaced to the processor.

Some processors allow the sizes of the internal instruction and data caches to be adjusted to meet the needs of specific applications. Some processors also allow specific blocks of instructions to be copied to the cache and locked in place for guaranteed fast access to particular pieces of code.

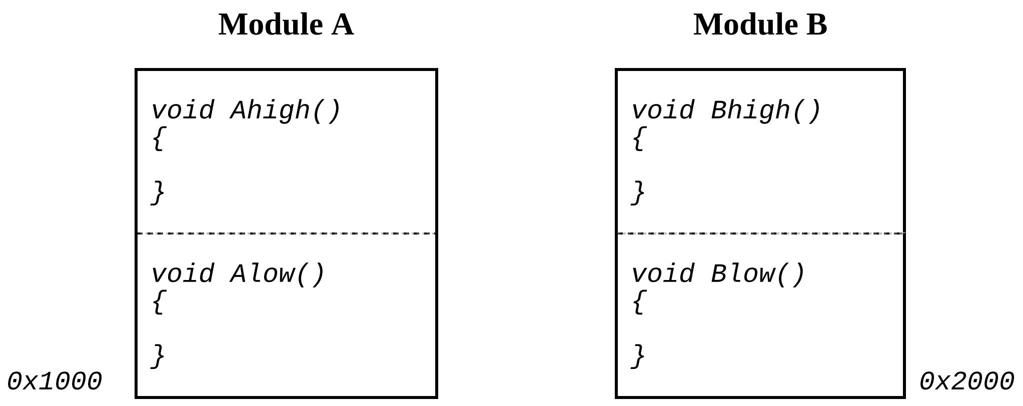
Finally, most processors allow the application to dynamically enable or disable the instruction and/or data caches. It is also usually possible to invalidate all or portions of the instruction and/or data caches.

Code and Data Location

Instruction and data caching can show dramatic improvement in the execution time of many applications in which the path of execution is single threaded. Much research has been done to show that for many applications, both instruction and data accesses are frequently very localized and hence benefit immensely from caching.

However, in a multitasking environment, the thread of execution is not so linear. Task switches require stack switching and context saving and restoral. These switches, often in response to interrupt activity, interfere with the seqential instruction execution process thereby negating some of the benefits of instruction caching.

The position of code and data in memory can also affect the caching performance as the following example illustrates. Assume that two code modules, A and B, are located in memory at addresses *0x1000* and *0x2000* respectively as shown in the diagram below. Assume that each module occupies less than 1024 bytes, the size of the instruction cache. Also assume that addresses map directly to the cache using the least significant 10 bits of the address.



If procedure `Bhigh` in module B repeatedly calls procedure `Alow` in module A, both procedures will end up in cache and benefit accordingly.

However, if procedure `Bhigh` in module B repeatedly calls procedure `Ahigh` in module A, the procedures will compete with each other for residence in the cache and performance will suffer.

Few applications are small enough to permit the fine tuning necessary to achieve optimal code location for maximum cache benefit. It is more likely that your code, and the procedures which it calls, will simply end up in memory at locations which vary with every minor code change. Move any block of code up or down in memory with respect to the other blocks of code with which it interacts and you can expect execution timing differences with caching enabled.

Compiler Effects

Earlier releases of AMX for many CISC processors were coded in assembly language. The reasons were historical. The most significant benefits were execution speed and code invariability. For these older products, the AMX code executed by the processor today is exactly the same code tested by KADAK and in use by developers world wide since the particular AMX product was first released. Only documented code changes to eliminate faults or add enhancements have been made.

Today, AMX is coded in C to satisfy the demands of developers not familiar with assembly language code and to ease the porting of AMX by KADAK to new and diverse processor architectures. However, the benefits of C have not been without a cost.

The greatest cost has been the sacrificing of AMX code invariability. The quality and efficiency of the AMX kernel is now dependent upon the code generation capabilities of the C compiler with which AMX is compiled. It is unlikely that any two C compilers will produce identical AMX code sequences. In fact, different releases of the same C compiler may very well produce different copies of AMX code. Fortunately, the testing methods developed by KADAK over many years now lead to almost fault free product releases.

There are other factors introduced by the C compiler which also may affect execution time. The parameter setup and procedure entry and exit code sequences can vary greatly among C compilers. The register preservation requirements of a particular C compiler, over and above those dictated by the processor, may affect AMX overhead.

Finally, beware of code optimizations touted by the compiler vendors. AMX source code does not include the kind of C code sequences which often confuse optimizing C compilers. Despite this fact, several C compilers have, with the simplest level of optimization enabled, generated invalid instruction sequences for perfectly valid C statements.

2. Task Switching Measurements

2.1 Task Wait/Wake Synchronization

AMX allows tasks to synchronize using a very simple wait/wake mechanism. The following example uses this feature to measure the time taken to start and end a task and to switch between tasks.

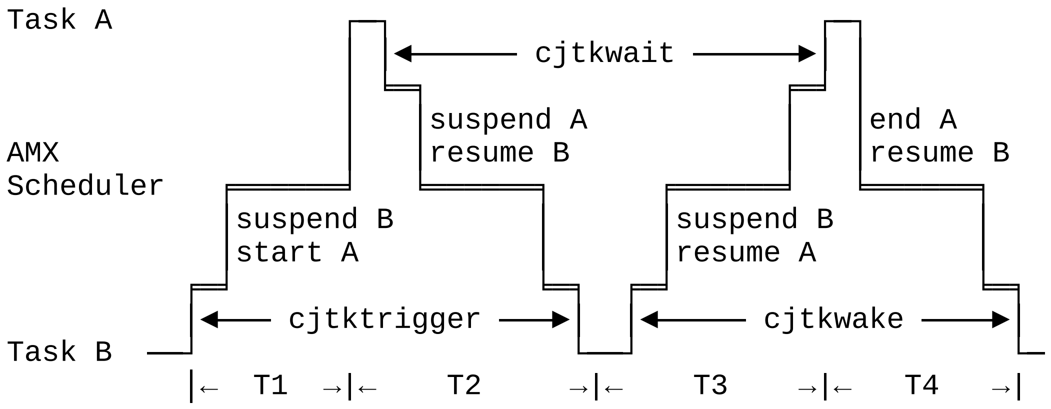
Assume that low priority task B is running and triggers higher priority task A. Since task B is preempted by the trigger, AMX suspends task B and starts task A. Task A then unconditionally waits. AMX suspends task A and resumes task B which then wakes task A. AMX suspends task B and resumes task A which then ends. AMX sets task A idle and resumes task B which also ends.

The code for these two tasks is as follows.

```
void CJ_CCPP taskA(void)
{
   cjtkwait();
   }

void CJ_CCPP taskB(void)
{
   cjtktrigger(taskAid);
   cjtkwake(taskAid);
   }
```

On a time line, the thread of execution appears as illustrated in the following diagram (not to scale). Measurements for times *Ti* are provided on separate AMX timing data sheets.



2.2 Task Mailbox Synchronization

AMX allows tasks to synchronize by sending messages to a mailbox. The following example uses this feature to measure the time taken to send and receive AMX messages and to switch between tasks.

Assume that low priority task B is running and triggers higher priority task A. Since task B is preempted by the trigger, AMX suspends task B and starts task A. Task A then unconditionally waits at a mailbox for a message to arrive. AMX suspends task A and resumes task B which then sends a message to the mailbox. Since task B is preempted by the message transmission, AMX suspends task B, delivers the message from the mailbox to task A and resumes task A which then ends. AMX sets task A idle and resumes task B which also ends.

The code for these two tasks, ignoring message construction and interpretation, is as follows.

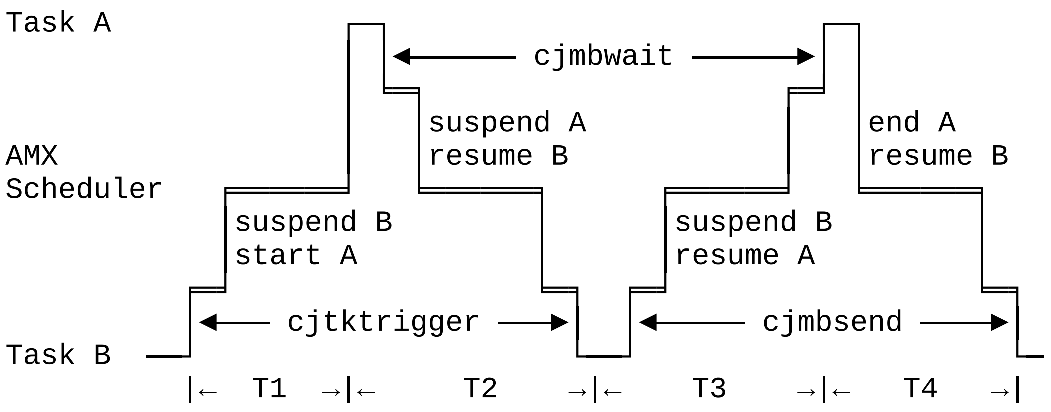
```
void CJ_CCPP taskA(void)                 void CJ_CCPP taskB(void)
{                                        {
   struct cjxmsg recvmsg;                   struct cjxmsg sendmsg;

                                            cjtktrigger(taskAid);
   cjmbwait(mailboxid,                      cjmbsend(mailboxid,
            &recvmsg, 0, 0);                         &sendmsg, CJ_NO);
}                                        }
```

Note

The default AMX message size is 12 bytes. Task A receives a local copy of the 12 byte message ready for processing.

On a time line, the thread of execution appears as illustrated in the following diagram (not to scale). Measurements for times *Ti* are provided on separate AMX timing data sheets.



3. Clock Service Measurements

3.1 Clock Interrupt and Timer Service

AMX is delivered with a clock driver for one or more counter/timer devices. The clock driver responds to clock interrupts, dismisses the interrupt request and calls the AMX Clock Handler. If an active AMX timer expires, the AMX Clock Handler preempts the interrupted task and forces the AMX Kernel Task to service the application timer.

The following example measures the clock overhead for each AMX tick and for servicing an expired periodic application timer.

Assume that the AMX clock tick is set to exactly match the hardware clock interrupt rate. Assume that task A creates and starts a periodic AMX timer with a period of two AMX ticks. The Timer Procedure associated with the timer does absolutely nothing. Task A goes compute bound forever in order to illustrate task preemption for clock service.

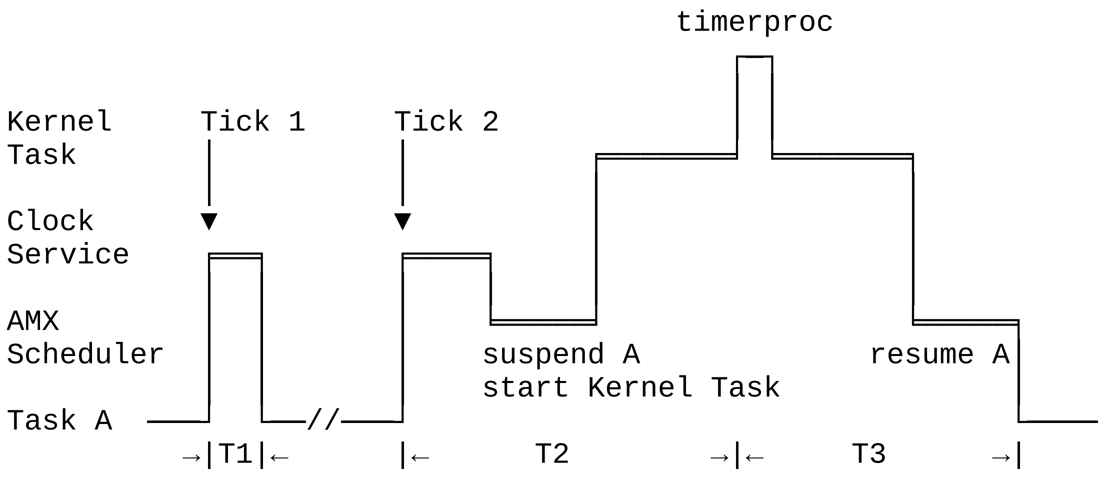
The code for the Timer Procedure and task A is as follows.

```
void CJ_CCPP timerproc(CJ_ID timerid, void *unused)
{
   }

void CJ_CCPP taskA(void)
{
   CJ_ID    timerid;

   cjtmcreate(&timerid, "TIMR", (CJ_TMRPROC)timerproc, 2, CJ_NULL);
   cjtmwrite(timerid, 2);
   for (;;) ;
   }
```

On a time line, the thread of execution appears as illustrated in the following diagram (not to scale). Measurements for times *Ti* are provided on separate AMX timing data sheets.



3.2 Task Wait (Delay) for Timed Interval

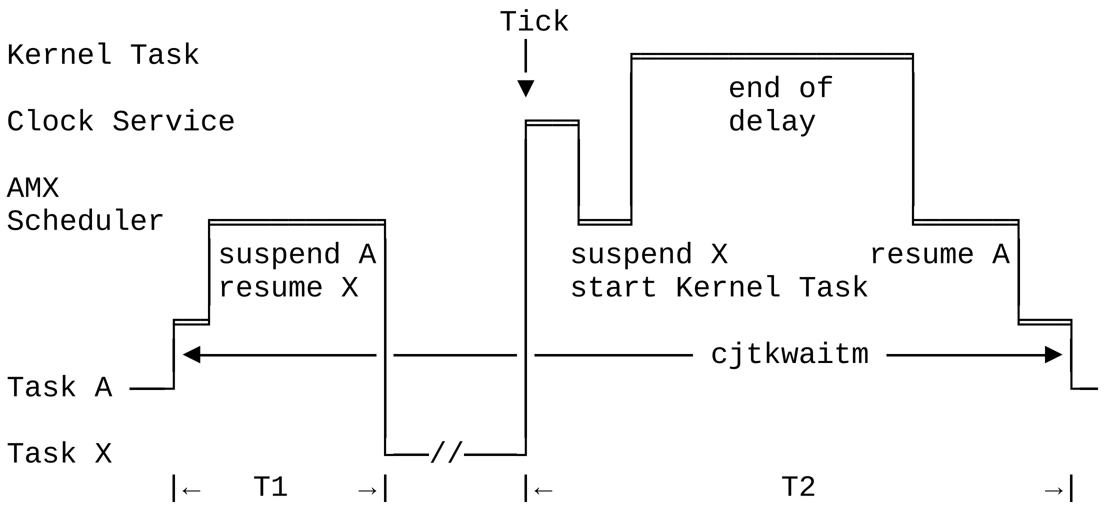
AMX allows tasks to delay for one or more AMX clock ticks. The following example uses this feature to measure the time taken for a task to resume execution after a timed delay.

Assume that the AMX clock tick is set to exactly match the hardware clock interrupt rate. Task A calls AMX to wait for one AMX tick. AMX suspends task A and resumes some other lower priority task X. When the next clock tick occurs, the clock driver dismisses the interrupt request and calls the AMX Clock Handler. The AMX Clock Handler forces AMX to suspend task X and start the AMX Kernel Task to service the expired task delay. AMX then resumes task A.

The code for task A is as follows.

```
void CJ_CCPP taskA(void)
{
   cjtkwaitm(1);
   }
```

On a time line, the thread of execution appears as illustrated in the following diagram (not to scale). Measurements for times *Ti* are provided on separate AMX timing data sheets.



4. Interrupt Response Measurements

4.1 Interrupt Latency

Interrupt latency, sometimes called interrupt response time, means different things to different people. Within the AMX environment, KADAK uses the term interrupt latency to refer to the time taken from the leading edge of an external interrupt request signal to the processor to the fetch of the first interrupt service instruction by the processor.

Interrupt latency consists of the sum of two components: processor interrupt latency and software interrupt latency.

Processor Latency

Many factors affect interrupt latency. The first is the delay generated by the processor itself in its recognition of the external interrupt request. This delay may be fixed for some processors and variable for others. For RISC and superscalar processors, the delay may range from 3 or 4 processor clock cycles to tens or even hundreds of clock cycles.

The longest delays are usually associated with interrupt requests which occur while the processor is executing complex instructions which, for processor reasons, must be allowed to complete (or at least reach some internal state at which the instruction can be resumed) before the interrupt request can be acknowledged. Multiply and divide are instructions of this type, as are looping instructions such as block moves and string compares.

Additional delay may stem from instruction and/or data caching operations. Response will be faster if the interrupt service instructions happen to already be in the cache as a consequence of a previous interrupt.

Since the processor's interrupt response always requires the saving of some processor state information, the processor architecture can dramatically affect latency. Expect longer delays if the processor automatically saves some or all registers or switches to alternate register sets. The delays will be even longer if the saving is done to memory.

Software Latency

Most processors permit some or all internal and/or external device interrupts to be inhibited under software control. The longest interval for which any software module disables interrupts is called software latency.

The AMX MA32 kernel uses this feature from time to time to briefly inhibit all interrupt sources while it executes some critical section of code. The longest interval for which AMX disables interrupts is called the **AMX interrupt latency**.

KADAK has identified and measured the worst case (longest) intervals during which AMX inhibits interrupts. AMX interrupt latency measurements are provided on separate AMX timing data sheets.

4.2 Handler Latency

Of equal importance to interrupt latency is what KADAK calls handler latency, the time from the processor's first response to an interrupt request signal through to the first useful instruction in the interrupt service procedure.

The term useful is nebulous but can be characterized as follows. Useful instructions are those which deal with the interrupting device and its related application. Useful instructions can not usually be executed until a few processor registers have been saved freeing them for use. In some cases, a switch to an alternate stack may be necessary. Once these housekeeping operations have been completed, useful work can be accomplished.

Handler latency is therefore the sum of the processor latency and the housekeeping execution time.

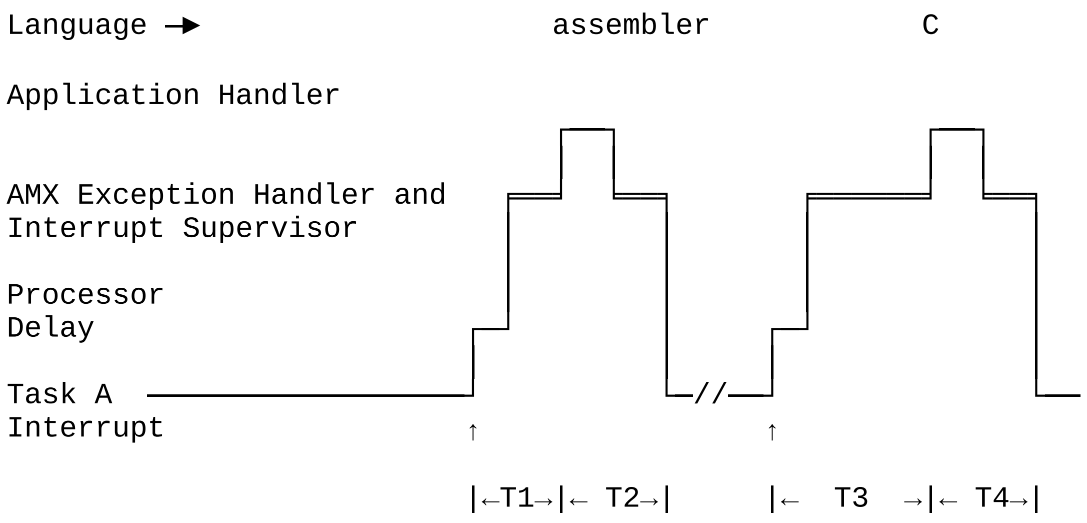
Interrupt Service

In the AMX MA32 environment, handler latency is easily measured since the AMX exception handler and Interrupt Supervisor perform all housekeeping actions.

The following example measures the interrupt overhead for two AMX application Interrupt Service Procedures (ISPs): an ISP coded in assembly language and an ISP coded in C. In each case, an external device interrupt is used to produce the interrupt. Each ISP simply clears the device interrupt request and dismisses the interrupt.

On entry to an AMX ISP coded in assembly language, only a few registers are free for use. On entry to an AMX ISP coded in C, all registers required by C are setup and ready for use.

On a time line, the thread of execution appears as illustrated in the following diagram (not to scale). Measurements for times `Ti` are provided on separate AMX timing data sheets.



AMX™ MA32 Multitasking Executive
Timing Data

for

80 MHz MIPS 4Kc Core and MetaWare C

June 1, 2001

Processor: MIPS 4Kc processor on the MIPS Malta Development Platform operating at 80 MHz.

Memory: 32 Mbyte of SDRAM with an effective zero wait state for read and write.

Cache: The on-chip 16 Kb instruction cache and 16 Kb data cache are always enabled, unless otherwise indicated, so that times recorded are typical of real time applications. External (secondary) instruction and/or data caches are not present.

Code/Data Location: No attempt is made to locate the AMX kernel code and data or the application code and data for best performance. Code and data reside in memory according to the order in which modules are encountered by the linker, again typical of real applications.

C Compiler: MetaWare Incorporated MIPS32 High C/C++ Compiler v4.3 with:

- no debug information in code modules
- absolute addresses for all code and data references
- register based argument passing conventions
- no global flow optimizations
- no instruction scheduling optimizations
- no function in-lining optimizations
- no optimizations using run-time profiling
- no processor specific code; use generic MIPS32 code only
- AMX MA32 procedure parameter validation enabled
- no AMX application scheduler hooks installed

Task Switch: An AMX MA32 task switch, excluding the events leading to the switch, takes 4.4 µs. This value was derived by averaging 100 task switch measurements made with instruction and data caches enabled. This task switch time rises to 20.7 µs if the caches remain enabled but are invalidated prior to each task switch measurement. With the caches disabled, the task switch time rises to 55.4 µs and is completely determined by the access constraints of the memory system.

AMX™ MA32 Multitasking Executive Timing Data

for

80 MHz MIPS 4Kc Core and MetaWare C

June 1, 2001

Measurement (note 1)	**Timing Diagram**	**T1 µs**	**T2 µs**	**T3 µs**	**T4 µs**
Synchronization (note 2)					
Wait/Wake	(page 5)	6.7	6.4	6.2	3.3
Mailbox	(page 6)	6.6	13.9	14.4	3.7
Clock Service (note 2)					
Timers	(page 7)	10.7	26.3	6.3	
Task Delay	(page 8)	11.5	38.0		
Interrupt Service (note 3)					
Handler Latency	(page 10)	4.8	11.6	6.6	12.9

Interrupt Latency: The worst case AMX MA32 interrupt latency is 21.3 µs.

The worst case handler latency for an AMX ISP coded in assembly language is 4.8 µs.

The worst case handler latency for an AMX ISP coded in C is 6.6 µs.

Note 1: Refer to the referenced page in the AMX Timing Guide for the applicable timing diagram to which measurements T1, T2, T3 and T4 apply.

Note 2: Instruction and data caches are enabled. Each test is performed 100 times and the average of all measured values is reported.

Note 3: Instruction and data caches are enabled so that the ISP benefits from caching. However, the caches are invalidated before each measurement so that WORST case handler latencies are measured. Each test is performed 100 times and the average of all measured values is reported.