

ICL XINU

C/C++ Software Development Kit Reference Manual

Revision D
Library compatibility: V1.20
P/N 60012004
revised 11/24/97

INDUSTRIAL CONTROL LINKS, INC.
13620 Lincoln Way Auburn, CA 95603
Tel: (916) 888-1800 FAX: (916) 888-7017
www.iclinks.com

© 1997 Industrial Control Links, Inc. All rights reserved

Table of Contents

OVERVIEW	1
Target Platform Support	3
License Information.....	5
INSTALLATION	7
COMPILING AND LINKING WITH ICL XINU	9
ARCHITECTURE AND OPERATION	11
Threads.....	11
Scheduling, Preemption and Context Switching.....	12
Priorities	13
Blocking.....	13
POTENTIAL PITFALLS	14
Reentrancy	14
Code You Create	14
Borland Run-Time Library	15
DOS/BIOS Software Interrupts	17
Shared Resource Access	19
Disabling Preemption	19
Using Semaphores	20
Other Methods	21

Stack Overflow.....	22
"Unsupported BIOS call"	23
INITIALIZATION.....	25
THREAD CREATION, DELETION AND STARTUP	27
TIMING.....	31
INTER-THREAD MESSAGING.....	33
CONTROLLING PREEMPTION	37
SEMAPHORES.....	39
STACK USAGE	41

Overview

ICL XINU is a small preemptive multi-threaded operating system (or micro-kernel, if you prefer) that is well suited for embedded systems development. Using a multithreaded operating system can in many ways simplify system design and improve performance. Without a multithreaded operating system, often the embedded system developer must come up with tricky methods to perform multiple concurrent operations, such as complex state machines and interrupt service routines. A multithreaded operating system gives you the ability to naturally divide logically unique tasks into separate threads, which appear to execute concurrently.

The ICL XINU operating system implementation is derived from the original work by Douglas Comer as presented in his book OPERATING SYSTEM DESIGN - THE XINU APPROACH. The ICL implementation is pared down to its essentials as applicable to embedded systems. In keeping with its intended use (embedded systems), ICL XINU does not have its own command line interpreter, but can be run on top of DOS. The API (Application Programming Interface) has been made to adhere to more current style than the original version, and new typedefs have been introduced to improve compiler independence and clarity. Stack overflow checking and peak usage tracking were added for increased robustness.

Features:

- single executable, multiple thread model
- fully preemptive operation
- configurable preemption rate
- blocked threads take no CPU time
- synchronized communications between threads
- semaphores to control multiple access to shared resources
- minimal system overhead required (less than 8k code)
- allows efficient inter-thread communications through shared memory

Target Platform Support

The current version of ICL XINU supports the ICL PC on a Stick (PCOS) single board computer and derived products, such as the ICL-4300 controller. The PCOS is an Intel 386EX based single board computer in the same form factor as a standard SIMM memory module. When running ICL XINU on the PCOS, ICL XINU uses the 386EX internal watchdog timer for its thread preemption mechanism. The timer tick occurs at a 1000Hz rate. The preemption frequency is selectable by the application program (such as once every 10 ticks).

Current compilers supported are:

- Turbo C++ 3.0 for DOS
- Borland C++ 3.1
- Borland C++ 4.5
- Borland C++ 5.0

Your compiler must create a DOS executable (.exe) file. The current implementation requires DOS on the target system in order to load the .exe file and begin execution. Multiple threads may then be spawned from within this single executable. Only the large memory model / real mode is supported. Other platform and compiler support will be most likely be added in the future.

License Information

As required, the following license statement is reproduced here:

(1) Regarding the software known as XINU and documented in OPERATING SYSTEM DESIGN - THE XINU APPROACH (Copyright 1984 Prentice Hall Inc.) Redistribution and use in source and binary forms are permitted provided that this notice is preserved and that due credit is given to the copyright holders. The names of the copyright holders may not be used to endorse or promote products derived from this software without specific prior written permission. The software is provided AS IS. Douglas Comer and Prentice Hall, Inc. make no warranty of the software and/or documentation as to merchantability and fitness for a particular purpose. Douglas Comer does not warrant or guarantee the use of, or the results of the use of, the software in terms of correctness, accuracy, reliability, currentness or otherwise. In no event will Douglas Comer or Prentice Hall Inc., be liable to LICENSEE or any other person for loss or damage, including time, goodwill, and consequential damages which may arise or be incurred by LICENSEE or anyone else from the use of the software.

(2) In consideration of the obligations of the parties, the nonexclusive royalty-free, irrevocable right to use, copy, modify, incorporate or create derivatives of, & distribute all or portions of software known as XINU as documented in OPERATING SYSTEM DESIGN THE XINU APPROACH is hereby granted to LICENSEE.

The above rights are granted subject to the condition that LICENSEE's charge for product or application does not include any charge for XINU or any unmodified portion thereof.

Since ICL XINU is a substantial modification of the original version, Industrial Control Links charges a reasonable fee for the ICL XINU software development kit. This fee authorizes the licensee to create and distribute applications built from ICL XINU without paying any royalty fees to Industrial Control Links, *as long as those applications are executed only on ICL hardware.*

Installation

To install ICL XINU, run the `INSTALL.EXE` program from the distribution diskette. The default installation directory is `C:\XINU`. During the installation process, you choose which compiler you would like support for. The installer will decompress and copy the appropriate files to the specified directory.

If you are installing a newer version of ICL XINU, it is best to delete the older version first. For this reason, it is not recommended that you store any of your application code under the ICL XINU directory.

If you want to support more than one compiler, run the installer again and choose another compiler option. The same main directory may be chosen for different compilers, since compiler-specific files for each compiler are installed in a unique subdirectory.

Compiling and Linking with ICL XINU

To create an application program that uses ICL XINU, you must compile your source code and then link it with the proper ICL XINU and PCOS or ICL-4300 libraries to create the final DOS executable file. The following compiler/linker options are required:

Target: standard, real-mode DOS, 16-bit
 Memory model: large
 Stack overflow detection: off (ICL XINU provides this itself)

In any C/C++ source file in a project where you use any ICL XINU functions or Borland run-time library (RTL), you must include the main header file `XINU.H`. `XINU.H` has macros that re-map some of the non-reentrant RTL functions to ICL XINU functions. These functions are required for proper operation (see the section on the Borland RTL). `XINU.H` should come *after* any Borland header files are included.

You may compile and link your application code using either the Borland IDE (Integrated Development Environment) or using the Borland `make` utility. Some sample files are provided in the `SAMPLE` subdirectory to get you started.

There are different sets of ICL XINU object code files and libraries, each corresponding to a different Borland compiler. You must link with the set of files that corresponds to the compiler you used to compile your source code. The files are kept in subdirectories by compiler version, as shown:

Compiler	File	Description
Turbo C++ 3.0	C:\XINU\TC30\XINU.LIB	Main XINU API library
	C:\XINU\TC30\RTL.LIB	RTL support library
	C:\XINU\TC30\PCOSTARG.LIB	PCOS Target library
	C:\XINU\TC30\43TARG.LIB	ICL-4300 Target library
	C:\XINU\TC30\HEAP.OBJ	Thread-safe new/delete operators
Borland C++ 3.1	C:\XINU\BC31\XINU.LIB	Main XINU API library
	C:\XINU\BC31\RTL.LIB	RTL support library
	C:\XINU\BC31\PCOSTARG.LIB	PCOS Target library
	C:\XINU\BC31\43TARG.LIB	ICL-4300 Target library
	C:\XINU\BC31\HEAP.OBJ	Thread-safe new/delete operators
Borland C++ 4.5	C:\XINU\BC45\XINU.LIB	Main XINU API library
	C:\XINU\BC45\RTL.LIB	RTL support library
	C:\XINU\BC45\PCOSTARG.LIB	PCOS Target library
	C:\XINU\BC45\43TARG.LIB	ICL-4300 Target library
	C:\XINU\BC45\HEAP.OBJ	Thread-safe new/delete operators
Borland C++ 5.0	C:\XINU\BC50\XINU.LIB	Main XINU API library
	C:\XINU\BC50\RTL.LIB	RTL support library
	C:\XINU\BC50\PCOSTARG.LIB	PCOS Target library
	C:\XINU\BC50\43TARG.LIB	ICL-4300 Target library
	C:\XINU\BC50\HEAP.OBJ	Thread-safe new/delete operators

The Main XINU API library file (`XINU.LIB`) contains the functions that are not target-specific. The PCOS Target library file (`PCOSTARG.LIB`) contains those functions that are specific to supporting the PCOS. The ICL-4300 Target library file (`43TARG.LIB`) contains those functions that are specific to supporting the ICL-4300. You do not need to link this unless you are using an ICL-4300. The RTL support library (`RTL.LIB`) contains thread-safe versions of run-time library functions. The `HEAP.OBJ` file contains thread-safe versions of the new/delete C++ operators, which must be linked in *before* the Borland runtime library. To create an ICL XINU application for the PCOS, you must link these XINU files, as well as the PCOS API library file(s). For the ICL-4300, link these XINU files and the ICL-4300 library files.

Architecture and Operation

Threads

The primary feature of ICL XINU that makes it so useful is the support of threads. A thread in ICL XINU is a task that can execute concurrently with other threads. An application that is created using ICL XINU consists of a single executable (a DOS .exe file) that spawns one or more threads. Threads have the following characteristics:

- Threads can be dynamically created and destroyed (“killed”) as needed.
- Each thread has its own separate stack space.
- A thread is created from a conventional function.
- Access to variables from within a thread follow the same scope rules as if the thread were just a normal function in a single tasking program.
- If a thread returns, it is automatically killed.
- When a thread is killed, its stack space is automatically freed.
- Any memory that a thread dynamically allocates is *not* automatically freed on termination.
- A thread can send a message to another thread.
- A thread can block (stop executing) while waiting for time to pass, a message to be received or a semaphore to be signaled.
- A blocked thread consumes no CPU time.
- Each thread has an associated priority.
- More than one thread can execute the same code, if the code is reentrant, since each thread has its own stack.

Scheduling, Preemption and Context Switching

The thread-scheduling algorithm determines when each thread will execute. This algorithm is simple to explain and implement, yet quite effective:

At any given time, the highest priority, ready-to-run thread will be executing. If more than one thread fits this criteria (priorities are the same), then ICL XINU preemptively schedules the threads, giving each thread an equal slice of CPU time in cyclical sequence.

A “Null Thread” exists so that if no user thread is ready to run, ICL XINU runs the Null Thread, which essentially does nothing.

This preemption scheme evenly distributes the available CPU time and normally prevents one thread from monopolizing the CPU. As you can see, ICL XINU does not truly execute threads concurrently (there is only one CPU, after all), however, if preemption happens with sufficient frequency, the threads will appear to execute concurrently. This preemption rate is configurable by the application programmer, and thus can be optimized for the application. If preemption is made to happen too frequently, the CPU will unnecessarily waste time performing lots of context switches. If preemption is not frequent enough, the illusion of concurrency becomes strained. A preemption period of about 10 milliseconds (100 Hz) is usually sufficiently frequent for most applications without introducing significant overhead.

When the scheduler determines that conditions are right to switch to another thread, it performs what is referred to as a “context switch.” ICL XINU stores information about each thread in a thread table. The context of the current thread (which consists simply of the values in all the CPU registers) is stored when the thread is preempted. The scheduler then loads the CPU registers with the previously stored values of the thread that is to execute next. Finally, ICL XINU resumes execution of the new thread at exactly the same place in exactly the same state that the thread was in when its execution was interrupted.

When a thread is preempted, it doesn’t “notice the difference” when it is resumed, except that:

- Time has passed.
- Global or static variables may have changed.

Its stack and register values will be the same as before it was interrupted (which means all local variables will be unchanged).

Priorities

Each ICL XINU thread has an associated priority. Since the highest priority ready-to-run thread is always the one that will execute, you must use priorities with caution. If you have a thread with a higher priority, and you intend for other threads to also execute, you must arrange for the higher priority thread to block on occasion. For “normal” threads that you would like to have executed equally with other similar threads, use the priority of `XINU_NRM_PRIO`, which is defined in the ICL XINU header file `XINU.H`.

Blocking

When a thread is “blocked,” it is no longer considered “ready-to-run” and will not be scheduled. A blocked thread uses no CPU time. A thread can be made to block for the following reasons:

- Waiting for a specified period of time to elapse. When the time elapses, thread execution is resumed.
- Waiting for a message to be sent from another thread. When a message is received, thread execution is resumed.
- Waiting for a semaphore to be signaled. Semaphores can be used to control access to a shared resource. When the semaphore is signaled, the thread resumes execution.
- A thread can be explicitly suspended by another thread. The thread can begin to execute again when another thread explicitly tells it to resume.

For best system performance, carefully consider whether a given thread can be made to block when it is not really “doing anything.” Examples of threads that could be made to block are a user interface thread that is waiting for input, or a communications thread that is waiting for a message to arrive.

Potential Pitfalls

Even though a multi-threaded application can in some ways be a conceptually simpler, more elegant and smaller solution than a single tasking approach, there are some potential pitfalls that you need to be aware of. Especially if you have not programmed in a multi-threaded environment before, pay attention to these pitfalls and you will avoid some headaches.

Reentrancy

Code You Create

Any piece of code that will be concurrently executed by more than one thread needs to be “reentrant.” Code that is reentrant suffers no ill effects when concurrently executed. The problem with reentrancy is best explained through the use of an example. Bear in mind in this discussion that a thread can be interrupted between any two machine instructions and another thread resumed. Note that a single line of C/C++ code may be compiled into many machine instructions.

Suppose that two threads were created from the following function:

```
void MyThread( void )
{
    char *str = "hello";
    static char *cp;

    while ( 1 ) // keep the thread running
    {
        // Send the string out:
        for ( cp = str; *cp; cp++ ) // advance through each character
            Output( str[cp] ); // send one character out

        ThreadSleepSec( 5 ); // sleep 5 seconds
    }
}
```

The code in `MyThread` would be executed concurrently by the two different threads, but there is only one copy of the static variable `cp`. Each thread will try to set and increment `cp` asynchronously. If both threads are executing the `for` loop concurrently, the results will be erratic. There is also a potential danger of `cp` accessing the wrong area of memory if the compiler generates more than one machine instruction to increment `cp`. One thread could be in the middle of incrementing `cp`, then get preempted by the other thread, which tries to use `cp`. At that point, `cp` could contain half old and half new information, resulting in access outside of the memory allocated to `str`. This would especially be a problem if `cp` was being used to *write* to memory, as the write would occur to an unintended memory location.

Note that you are not necessarily safe from reentrancy problems if the main body of two threads do not share the same code, because they may very well both call some of the same functions.

Writing code that *is* reentrant, and thus safe in a multithreaded application, is normally not difficult. Simply avoid using any global or static variables or resources. Make sure that variables are local (automatic) variables and thus stored on the stack. Since each thread gets its own stack, these variables will not “collide.” Global resources that you should be careful about accessing from multiple threads include hardware resources, such as displays, timers and serial ports.

If you truly *need* to access a global or static variable or a global resource from multiple threads (legitimate reasons do exist), there are safe ways to do so, as explained in the section Shared Resource Access.

Borland Run-Time Library

In the current ICL XINU implementation, several versions of the Borland C++ compiler are supported. This support has some restrictions, however. The Borland compilers are designed to create *single-threaded* DOS executables. As a result, some functions in the Borland run-time library (RTL) are not thread-safe, because they are not reentrant. How much you are affected by this problem depends on how extensively you need to use the RTL in your application, and which functions you need to use. Some of the RTL functions have already been made thread-safe. Others have not been tested and therefore should be used with caution. The list below shows functions that have been made thread-safe as part of ICL XINU.

<u>Stream I/O:</u>		<u>Floating point emulation:</u>	<u>Dynamic memory allocation:</u>	
fclose	fseek	all functions	calloc	new operator
fflush	ftell		free	delete operator
fgetc	fwrite		malloc	
fgets	gets			
fopen	printf			
fprintf	puts			
fputc	rename	<u>String formatting:</u>		
fputs	rewind			
fread	unlink	sprintf		

Additional RTL functions will be made thread-safe in future releases of ICL XINU. If you have particular functions that you need, contact ICL and we can place a higher priority on those functions.

In the meantime, if you really need to use an RTL function that you believe to be non thread-safe, you can simply disable preemption around the call to the function, and then you should be safe to call the function. See the section on Shared Resource Access for a description of disabling preemption.

The Stream I/O functions listed above are all protected with a shared semaphore. When a thread calls one of these functions, it uses this semaphore to determine if a Stream I/O function is already in process. If so, the current thread blocks until the first thread completes its Stream I/O function and the semaphore is signaled. The second thread is then safe to execute the function. This all happens transparently to the threads that are using these functions, except that they may block occasionally.

Knowing how reentrant code is written, you can many times predict with reasonable accuracy which RTL functions are likely to be reentrant or not reentrant.

The following types of functions probably ARE NOT reentrant:

- functions that access a global, shared resource, such as a file or the console
- functions that remember anything between calls (such as the function strtok)

Functions that receive all the information that they need as arguments, do not access any global resources, and do not remember any information between calls, in most cases *are* reentrant and therefore safe to use without any special protection. Some functions that would fit this description are `strlen`, `strcpy` and `atoi`.

Important Note:

One technique that is used to replace some non-reentrant RTL calls with equivalent ICL XINU calls is name substitution using macros. This causes a protected version of the function to be called instead of the normal RTL function being called directly. In order to assure that this happens, you must include `XINU.H` in each of your application source files. `XINU.H` should come *after* any Borland header files are included. In other cases, some functions are replaced completely by XINU. In order to assure that this happens, you must link the Borland RTL (`cl.lib`) *after* the XINU libraries.

DOS/BIOS Software Interrupts

Since the original IBM PC running BIOS and DOS was intended only to be a single tasking system, most BIOS/DOS software interrupt calls are not reentrant. In ICL XINU, however, this problem is taken care of for you. ICL XINU intercepts non-reentrant BIOS/DOS software interrupts and disables thread preemption for the duration of the call. The following table lists the status of the software interrupts as they relate to ICL XINU for PCOS. The software interrupts that are listed as “fixed” are those that ICL XINU protects from preemption.

BIOS Software Interrupts

Interrupt	Type	Reentrant	Comment	Fixed
INT05	printscreen	n/a	unsupported	
INT10	video	yes	only subfunction 0E is implemented on PCOS/ICL-4300	
INT11	equip list	yes		
INT12	mem size	yes		
INT13	disk driver	n/a	unsupported	
INT14	serial driver	no		yes
INT15	various	no		yes
INT16	keyboard driver	no		yes
INT17	printer driver	n/a	unsupported	
INT18	BIOS Boot fail	n/a	not meant to be called by application	
INT19	Boot BIOS Interrupt	n/a	not meant to be called by application	
INT1A	clock services	no		yes
INT1B	keyboard break	n/a	not meant to be called by application	
INT1C	clock tick	n/a	not meant to be called by application	
INT1D-INT1F		n/a	unsupported	
INT40-INT5C		n/a	unsupported	

DOS Software Interrupts

Interrupt	Type	Reentrant	Comment	Fixed
INT20	program terminate	n/a	not to be called directly by a XINU application	
INT21	general services	no		yes
INT22	terminate address	n/a	not meant to be called by application	
INT23	ctrl-break exit address	n/a	not meant to be called by application	
INT24	fatal error vector	n/a	not meant to be called by application	
INT25	absolute disk read	no		yes
INT26	absolute disk write	no		yes
INT27	terminate, stay resident	n/a	not to be called by a XINU application	
INT28	idle	n/a	not meant to be called by application	
INT29	tty output	no		yes
INT2A	MS net services	no		yes
INT2B	reserved	n/a		
INT2C	reserved	n/a		
INT2D	reserved	n/a		
INT2E	run command	no		yes
INT2F	multiplex	no		yes
INT30	long jump	n/a	not meant to be called by application	
INT31	long jump address	n/a	not meant to be called by application	
INT32	reserved	n/a		
INT33	mouse functions	no		yes
INT34- INT3E	floating point emulation	n/a	not meant to be called by application	
INT3F	overlay	no		yes
INT67	EMM functions	no		yes

Shared Resource Access

There are two recommended ways to prevent multiple threads from accessing a shared resource concurrently. These are:

- disable preemption
- use semaphores

These techniques which are explained in the following sections use the term “critical code section.” A critical code section is a section of code, which manipulates the shared resource that you are trying to protect from concurrent access.

Disabling Preemption

The way to use this approach is to disable preemption before entering a critical code section, then restore preemption at the end of the critical code section. It is better to *restore* the state than to explicitly *enable* it, because it may have already been disabled earlier (perhaps by the calling function). The following code segment illustrates the approach:

```
tPreemptState prevState;
.
.
.
prevState = ThreadPreemptDisable(); // turn off preemption
// critical code section
ThreadPreemptRestore( prevState ); // restore preemption to previous setting
.
.
.
```

When you call `ThreadPreemptDisable`, it turns off the preemption mechanism. No interrupts are disabled by this function, so system timers and other interrupt driven mechanisms can continue to operate. The function `ThreadPreemptRestore` is used to restore the preemption mechanism to its previous state. The method described here is fairly easy to use, but it does have the drawback of preventing any other threads from executing, not just the thread(s) that might have access to the shared resource. This could be significant if you need to disable preemption for very long.

Using Semaphores

Semaphores can also be used to prevent concurrent access to shared resources. This approach is a bit more sophisticated than completely disabling preemption, since it can be used to block only those threads that are actually trying to access the shared resource. It is only slightly more complex than the method of disabling preemption. The ICL XINU implementation of semaphores uses the following functions:

- SemCreate - create a semaphore
- SemDelete - delete a semaphore
- SemWait - wait on a semaphore
- SemSignal - signal (release) a semaphore

To control access to a shared resource, create an associated semaphore with the `SemCreate` function. Before you enter a critical code section, call `SemWait`. When leaving the critical code section, call `SemSignal`. The first thread which calls `SemWait` on a given semaphore will cause subsequent threads which call `SemWait` on the same semaphore to block (suspend execution). When the first thread calls `SemSignal`, it will allow the second thread to proceed. Multiple threads can be queued up on the same semaphore. As `SemSignal` is called, the next waiting thread will be allowed to execute. Unlike the method of explicitly disabling preemption, the use of semaphores allows other threads (ones not trying to access the shared resource) to continue to execute normally. The code excerpts below illustrate the use of a semaphore:

```
// initialization code:
bufSem = SemCreate( 1 ); // semaphore to control access to shared resource

// thread 1 code:
.
.
.
SemWait( bufSem ); // wait until we have access to shared resource
// critical code section
SemSignal( bufSem ); // signal that we are done with shared resource
.
.
.

// thread 2 code:
.
.
.
SemWait( bufSem ); // wait until we have access to shared resource
// critical code section
SemSignal( bufSem ); // signal that we are done with shared resource
.
.
.
```

Make sure that for every `SemWait` call you have a corresponding `SemSignal` call, or you could permanently block other threads. The value that is passed to `SemCreate` is the initial count that is stored in the semaphore. If you want the first call to `SemWait` to *not block*, use an initial count of 1. If you want the first call to `SemWait` to *block*, use an initial count of 0.

Other Methods

There are two additional methods of protecting critical sections of code, but these are not recommended:

- disable all interrupts (not a very good idea)
- disable the timer interrupt (a little better, but still not that good)

The primary reason these methods are mentioned here is because you may already be aware of them and may be tempted to use them.

Before entering a critical code section, interrupts are disabled, then restored at the end of the critical code section.

The reason this works is that the thread preemption mechanism is based on a hardware timer interrupt. If all interrupts or the timer interrupt is disabled, then preemption will not occur. The most significant drawback with this approach is that you may miss interrupts that otherwise would have been serviced if interrupts had not been disabled, especially if your critical code section takes much time to execute. If, for example, an interrupt driven serial port driver is being used, and a stream of data is being received on the serial port, the driver may not be able to service the serial port interrupt quickly enough to prevent the loss of incoming data (in the case when all interrupts are disabled). Or, if only the timer interrupt is disabled, you may start to miss system timer interrupts. For this reason, you are encouraged to use one of the other methods instead of disabling interrupts.

Stack Overflow

An event that can cause unexpected system crashes and can be difficult to track down is stack overflow. Each thread is allocated its own stack space. The stack size is specified when you create the thread using the `ThreadCreate` function. If you do not allow enough stack space, the stack for the thread will overflow, and most likely crash the target. ICL XINU does perform some stack overflow checking, but only when it is preparing to switch to a thread, not when the thread is actually running. Therefore it may not catch the overflow. When in doubt, allocate more stack space than you think you need to a thread, and then you can trim it down later after all the code for the thread is fully implemented and is unlikely to change further. Note that 512 bytes of the stack space that you specify are used up in storing the state of the floating point emulator. Therefore, you must take this into account. For a thread that does very little, you will probably want to start with at least 1000 bytes of stack. ICL XINU provides a function `StkPeakUsage` for determining the peak stack usage of a thread, which can be used to fine tune the stack space allocated.

"Unsupported BIOS call"

An "Unsupported BIOS call" message on the PCOS console port means that something in your program has caused execution to jump to the handler for the associated interrupt. The PCOS BIOS installs a default handler for all interrupts that simply dumps a message to the console port. If the application program does not install a different handler, then the default handler may be executed under certain conditions. In some cases it may be an actual hardware interrupt. In other cases it could be a software interrupt. In many cases it simply means that your program has "crashed" in some unknown manner, and has ended up running the handler. A crash could be caused by a number of different situations, including:

- stack overflow
- stack corruption
- storing data using an uninitialized pointer
- calling a function via an uninitialized function pointer
- buffer/array overflow
- non-reentrant code called from multiple threads
- non-reentrant code called from a thread and an interrupt service routine

If you get an "Unsupported BIOS call" message and you think it is due to a program crash, here are some hints as to how to find the problem:

- What is the last change you made to the code?
- Can you back the change out and have the problem go away?
- Try to narrow the problem down by simplifying the program - remove sections until the problem goes away (the last code you removed is most likely the problem).
- Step through the code with the debugger - does it get to a certain point and "lock up?" Be aware that when you are running the Paradigm Debug/EPC debugger, when you free run to a breakpoint, several threads may get executed before you get to the breakpoint, since preemption is enabled when free running (as opposed to single stepping).

Initialization

ICL XINU is initialized and started by calling the function `XinuStart`, which will then spawn the thread `MainThread`. `XinuStart` does not normally return unless all threads are killed.

```
void XinuStart( UINT16 ticksPerPreempt, INT32 argc, char *argv[] );
```

ticksPerPreempt

Specifies how frequently ICL XINU should attempt to preempt/reschedule in terms of system timer ticks.

argc

Argument count, interpreted in the same way as the usual `argc` parameter in a normal C main function. Specifies the number of arguments to be found in the `argv` array. `argc` is passed on by ICL XINU to `MainThread`.

argv

Argument value(s), interpreted in the same way as the usual `argv` parameter in a normal C main function. Consists of an array of pointers to character strings. `argv` is passed on by ICL XINU to `MainThread`.

Example:

If the timer interrupt rate is 1000Hz, then `ticksPerPreempt` might reasonably be set to 10, providing a preemption rate of 100Hz:

```
int main( int argc, char *argv[] )
{
    .
    .
    .
    // hardware and other system initialization
    .
    .
    .
    // Initialize and start ICL XINU:
    XinuStart( 10, argc, argv ); // pass argc and argv on to MainThread
}
```

When `XinuStart` executes, it starts up the thread defined by the function `MainThread`. You must provide `MainThread` as part of your application program. `MainThread` is free to spawn other threads.

```
void MainThread( INT32 argc, char *argv[] );
```

argc

Argument count, passed in directly from the `argc` supplied to `XinuStart`.

argv

Argument value(s), passed in directly from the `argv` supplied to `XinuStart`.

Thread Creation, Deletion and Startup

The function `ThreadCreate` is used to create a single thread. Each thread executes concurrently as an independent task. When you create a thread with this function, it does not immediately start to execute, however. To start execution of the thread, call the `ThreadResume` function. To delete a thread, call `ThreadKill`.

```
tThreadID ThreadCreate( void (*thrAddr)(), UINT16 stkSize,  
                       UINT16 priority, char *name,  
                       INT32 argCount, INT32 args, ... );
```

return value

This function returns the thread ID of the created thread. This thread ID can be used to reference the thread. If the function fails, returns `kXinulllegalThrID`.

thrAddr

Address of the function that the thread is to be created from.

stkSize

The amount to stack space, in bytes, to allocate for the thread. A good starting point for a thread with minimal stack usage would be 2000 bytes. For a thread that you expect to use a fair amount of stack space, you might start with 8000 bytes. Be aware that certain Borland run-time library functions might take quite a bit of stack space when called.

priority

The priority to assign to the thread. Priority 1 is the lowest, 65535 is the highest. Priority 0 is reserved for the system. If you want all your threads to be preempted equally, consider using `XINU_NRM_PRIO` as a normal thread priority level. See the discussion of priorities in the Architecture section.

name

A name to be associated with the thread, in the form of a null terminated string up to 9 characters long (plus the null).

argCount

Argument count, to be passed to the thread. If no arguments are provided, set `argCount` to 0.

args, ...

Variable number of arguments, passed to the thread. If no arguments are needed, pass a single `args` value of 0. Note that each argument must be passed as a `UINT32` type. If the argument is not already a `UINT32`, you must typecast it as one to ensure it is passed properly to `ThreadCreate`.

```
void ThreadResume( tThreadID id );
```

Start (resume) a thread that has been created with ThreadCreate.

id

The thread ID of the thread to start executing (returned from ThreadCreate). It is safe to pass a value of kXinuIllegalThrID to ThreadResume (it will do nothing).

```
tXinuRslt ThreadKill( tThreadID id );
```

Stop a thread and remove it from the ICL XINU thread table. When a thread is stopped with ThreadKill, the stack space that was allocated to it is freed. Any other memory that was dynamically allocated by the thread is not automatically freed. A thread can kill itself by simply returning.

id

The thread ID of the thread to kill (returned from ThreadCreate). It is safe to pass a value of kXinuIllegalThrID to ThreadResume (it will do nothing).

Example (simple thread creation, execution, deletion):

```
void UserIntefaceThread(void)
{
    while ( 1 ) // must have a loop in thread if it is to continue to execute
    {
        ProcessKeyboard();
        UpdateDisplay();
    }
}

void MainThread( INT32 argc, char *argv[] )
{
    tThreadID uifThreadID;

    // Create the user interface thread from a function:
    uifThreadID = ThreadCreate( UserIntefaceThread, 8000, XINU_NRM_PRIO,
                              "uifThread", 0, 0 );

    // Start the thread running:
    ThreadResume( uifThreadID );

    .
    .
    .

    // Kill the thread:
    ThreadKill( uifThreadID );

    .
    .
    .
}
```

Example (passing arguments to a thread):

```
void CommunicationsThread( UINT32 port, UINT32 baud )
{
    ComInitialize( port, baud );

    while ( 1 ) // must have a loop in thread if it is to continue to execute
    {
        Receive( port );
        Transmit( port );
    }
}

void MainThread( INT32 argc, char *argv[] )
{
    tThreadID comThreadID;

    // Create the user interface thread from a function:
    comThreadID = ThreadCreate( CommunicationsThread, 8000, XINU_NRM_PRIO,
                               "comThread", 2, (UINT32)3, (UINT32)9600 );

    // Start the thread running:
    ThreadResume( comThreadID );

    .
    .
    .
}
```


Timing

ICL XINU provides two functions that may be used for timing/delaying purposes. The functions work in the same way except that their resolutions differ. The functions are `ThreadSleepSec` and `ThreadSleepTicks`. Each function suspends the calling thread for the specified period of time. While the thread is suspended, it consumes no CPU time, making this a highly efficient mechanism. You are encouraged to use one of the `ThreadSleep` functions as opposed to using the Borland `delay` function or a delay loop of your own invention. Using the `delay` function or your own delay loop continue to use CPU time while the delay is being executed. The accuracy of the `ThreadSleep` functions is quite good, being based on a CPU hardware timer. However, if you have two threads that are scheduled to wake up at exactly the same time, one of them will be awakened slightly later than the other.

```
void ThreadSleepSec( UINT16 time );
```

Suspend the calling thread for the given number of seconds. While the thread is suspended, the thread uses no CPU time.

return value
None.

time
The amount of time, in seconds, to suspend the current thread for.

Example:

```
void TestThread(void)
{
    // Print "hello" once a second:
    while ( 1 )
    {
        ThreadSleepSec( 1 );
        printf( "hello\n" );
    }
}
```

```
void ThreadSleepTicks( UINT16 time );
```

Suspend the calling thread for the given number of timer ticks. While the thread is suspended, the thread uses no CPU time.

return value
None.

time
The amount of time, in system timer ticks, to suspend the current thread for. The system timer tick period for the PC on a Stick is 1 millisecond.

Example:

```
void TestThread(void)
{
    BOOL toggle = 0;

    // Toggle output every 100 system timer ticks:
    while ( 1 )
    {
        SetOutput( toggle );
        toggle = !toggle;
        ThreadSleepTicks( 100 ); // suspend for 100 ticks
    }
}
```

Inter-thread Messaging

The ICL XINU facilities for sending and receiving messages can be used to pass information between threads, and also as a thread synchronization mechanism. A message consists of a 32-bit signed integer value. This value, however, can be used to represent non-integer values through the use of type casting. Thus floating point values or pointers to a C struct can be sent as messages. The receiving thread can be implemented to wait on the arrival of a message, thus providing synchronization between two threads. There is room for only one pending message for any given thread - messages are not queued up. If a message has already been posted but not received by the intended thread, a new message cannot be sent to the same thread unless it is “forced,” in which case it replaces the older message and the older message is lost.

```
tThreadMsg ThreadMsgRcv( void );
```

Receive an inter-thread message. If a message is available, the function returns immediately. If no message is available, the thread suspends until a message is received (or until the thread is killed). While the thread is suspended, the thread uses no CPU time.

return value

The received message as a `tThreadMsg`. A `tThreadMsg` is equivalent to an `INT32`.

Example:

```
void ReceivingThread(void)
{
    tThreadMsg message;
    .
    .
    message = ThreadMsgRcv();
    .
    .
}
```

```
tXinuRslt ThreadMsgRcvTm( UINT16 timeout, tThreadMsg *msg );
```

Receive an inter-thread message, waiting up to the specified timeout. If a message is available, the function returns immediately. If no message is available, the thread suspends until a message is received or the timeout occurs (or until the thread is killed). If the thread must wait for a message, the thread uses no CPU time during the time it is waiting.

return value

kXinuErr - a message was not received before the timeout

kXinuOk - a message was successfully received

msg

The received message as a tThreadMsg, if a message was received. A tThreadMsg is equivalent to an INT32.

timeout

The maximum time, in system timer ticks (milliseconds for PCOS), to wait for a message. If timeout is 0, then the function will return immediately with or without a message.

Example:

```
void ReceivingThread(void)
{
    tThreadMsg message;
    tXinuRslt result;
    .
    .
    .
    result = ThreadMsgRcvTm( 1000, &message ); // wait up to 1000 ticks

    if ( result == kXinuOk ) // valid message?
    {
        if ( message == 123 )
        .
        .
        .
    }
}
```

```
tXinuRslt ThreadMsgXmt( tThreadID id, tThreadMsg msg, BOOL resched );
```

Transmit an inter-thread message to the specified thread.

return value

kXinuErr - thread already has a message, or bad thread ID

kXinuOk - message sent successfully

msg

The message to send. The message type must be tThreadMsg, but this is equivalent to an INT32. You can typecast your message value if you wish to send something other than an INT32.

resched

TRUE - reschedule (switch to) the receiving thread after delivering the message.

FALSE - do not reschedule until the next system clock tick.

Example:

```
static tThreadID receiverID;

void MainThread( INT32 argc, char *argv[] )
{
    receiverID = ThreadCreate( ReceivingThread . . .
    ThreadResume( ReceivingThread );
    ThreadCreate( TransmittingThread . . .
    ThreadResume( TransmittingThread );
    .
    .
    .
}

void TransmittingThread(void)
{
    tThreadMsg message;
    .
    .
    .
    message = 17L;
    ThreadMsgXmt( receiverID, message, TRUE );
    .
    .
}
}
```

```
tXinuRslt ThreadMsgXmtForce( tThreadID id, tThreadMsg msg );
```

Function

Transmit an inter-thread message to the specified thread, forcing delivery if a message has already been posted but not yet received by the destination thread. The previously posted message will be lost.

Return Value

kXinuErr - bad thread ID
kXinuOk - message sent successfully

msg

The message to send. The message type must be tThreadMsg, but this is equivalent to an INT32. You can typecast your message value if you wish to send something other than an INT32.

Example:

```
static tThreadID recID;

void TransThread(void)
{
    float msg;
    .
    .
    .
    msg = 1.23;
    ThreadMsgXmtForce( recID, (tThreadMsg)msg ); // cast float to tThreadMsg
    .
    .
    .
}

void RecThread(void)
{
    float message;

    message = (float)ThreadMsgRcv(); // cast tThreadMsg to float
    .
    .
    .
}
```

Controlling Preemption

There are two functions provided in ICL XINU to disable/enable thread preemption. These functions can be used as a simple means of protecting a critical section of code that accesses a shared resource. For more details on why you might want to use these functions, see the section on Shared Resource Access. Note that when you have preemption disabled, if you call any functions that would cause ICL XINU to switch to another thread, the usual effect will not occur. For example, if you call a `ThreadSleep` or `ThreadMsgRcv` function, the current thread will not suspend.

```
BOOL ThreadPreemptDisable( void );
```

Function

Disables thread preemption. The current thread will continue to execute, excluding other threads from executing.

Return Value

The previous state of thread preemption. This value is suitable for passing to `ThreadPreemptRestore`.

Example:

```
BOOL preemptState;  
.  
.  
.  
preemptState = ThreadPreemptDisable(); // disable preemption  
// critical code section  
ThreadPreemptRestore( preemptState ); // put it back the way it was  
  
void ThreadPreemptRestore( BOOL prevState );
```

Function

Restores thread preemption to its previous state.

Return Value

None.

prevState

The previous state of thread preemption. This value should have been returned from an earlier call to `ThreadPreemptDisable`.

Semaphores

Semaphores provide a convenient mechanism to control access to a shared resource. Semaphores in ICL XINU can be created and deleted and are identified by a variable of the type `tSem`. A semaphore contains a count value that results in its behavior. When the function `SemWait` is called, the value is decremented. If the resultant value is negative, the calling thread will be suspended and placed in a queue of threads that are waiting on that semaphore. When the function `SemSignal` is called, the semaphore value is incremented. If a thread is queued up waiting on the semaphore, the first thread on the queue is resumed. Thus multiple threads can wait on a semaphore, each resuming execution in turn as the semaphore is signaled. When a semaphore is created, you specify its initial value. The value used will determine whether the first `SemWait` call on the semaphore will suspend or not.

```
tSem SemCreate( INT16 initCount );
```

Function

Creates a semaphore.

Return Value

Returns `kXinuErr` if no more semaphores can be created, or on success returns a value that is used to identify the semaphore in other function calls.

initCount

Initial count that should be stored to the semaphore. A value of 0 will cause the first call to `SemWait` to suspend. A value of 1 will allow the first call to `SemWait` to proceed.

Example:

```
tSem accessControl;
.
.
.
if ( ( accessControl = SemCreate( 0 ) ) != kXinuErr )
{
.
.
.

tXinuRslt SemDelete( tSem sem );
```

Function

Deletes a semaphore that was created earlier with `SemCreate`.

Return Value

Returns `kXinuErr` if a bad semaphore was passed in.

sem

Identifies the semaphore to delete.

Example:

```
tSem sem;
.
.
.
if ( ( sem = SemCreate( 0 ) ) != kXinuErr )
{
.
.
.
SemDelete( sem );

tXinuRslt SemWait( tSem sem );
```

Function

Waits on a semaphore - decrements the semaphore value and if the value goes negative, suspends the current thread. If the semaphore value is non-negative, returns to the calling thread immediately.

Return Value

Returns `kXinuErr` if a bad semaphore was passed in.

sem

Identifies the semaphore to wait on.

Example:

```
tSem sem; // semaphore to control access to a shared resource
.
.
.
SemWait( sem ); // wait for resource to become available
// access the resource
SemSignal( sem ); // signal that we are done with the resource

tXinuRslt SemSignal( tSem sem );
```

Function

Signals a semaphore - increments the semaphore value and if a thread is waiting on the semaphore, releases the waiting thread.

Return Value

Returns `kXinuErr` if a bad semaphore was passed in.

sem

Identifies the semaphore to signal.

Stack Usage

The ICL XINU implementation has instrumentation to help you to empirically determine the amount of stack space used by a thread. You will probably find this to be a useful feature, since it can be difficult to calculate the amount of stack space that a thread requires, especially since you may be calling functions that you do not have control over. The way this mechanism works is that ICL XINU fills the stack space for the thread with a marker value of 0xAAAA when the thread is created. When you call `StkPeakUsage`, it scans the stack area for the current thread and looks for the lowest (since stacks grow down) memory location where the marker value has been overwritten. If, at its point of greatest stack usage, your thread pushes/pops one or more 0xAAAA values, the reported usage could be off slightly, but in practice for normal threads it gives an accurate indication. You can use this information to tune the stack sizes of your threads so that a reasonable margin of safety is maintained, and yet stack space is not wasted excessively. This only will capture the maximum stack that was actually used, however, not the maximum that could be used, if some of your thread's code has not been exercised.

```
UINT16 StkPeakUsage( void );
```

Function

Returns the peak stack usage of the current thread, since that thread was created.

Return Value

The maximum stack usage that has been detected thus far, in bytes.

Example:

```
max = StkPeakUsage(); // find out how much stack we are using
```