A Guide to The Precise/MQX Solution

Document No. 0101x

Intermetrics, Inc.
733 Concord Ave.
Cambridge, Massachusetts
# Table of Contents

- Kernel Log Services ................................................................. 38
- Naming Service ................................................................. 39
- MQX+M Multiprocessing Kernel Overview .......................... 40

## THE CODING PHASE ................................................................. 41
- IDE .................................................................................. 41
- The C Compiler Toolkit ....................................................... 43

## THE DEBUG PHASE ................................................................... 47
- Host Simulation ................................................................. 47
- Target-based Debug ........................................................... 48
  - The PassKey/MQX Kernel-Aware Debugger .................. 48
- Target Communication ...................................................... 52
  - BDM ............................................................................ 52
  - SmartMON .................................................................. 52
  - MPD ........................................................................... 53
  - ICE ............................................................................. 53

## THE INTEGRATION & TEST PHASE ......................................... 55
- Porting MQX ................................................................. 55
- Performance Analysis ....................................................... 56

## THE PRODUCTION PHASE ....................................................... 57

## OTHER CONSIDERATIONS ..................................................... 59

## FOR MORE INFORMATION .................................................. 63
Preface

Chances are, you're reading this guide because you need a real-time kernel and development environment that you can use for your company's embedded systems development. You need an environment you can trust—one that will be good not only for your upcoming project, but for your company's future projects as well.

We at Intermetrics think that you've come to the right place. We have the solution you need—The Precise Solution—and we've put this guide together to tell you about it.

We need about an hour of your time to guide you through The Precise Solution. After reading this guide, you will be familiar with:

- Intermetrics, a profitable public company, which has been providing quality solutions to embedded systems designers for over a quarter century
- The leading-edge MQX kernel (the heart of The Precise Solution), which is based on over twelve years of evolving kernel technology
- The integrated suite of tools that we provide to support you through the entire software development life cycle
- The technical and financial benefits of The Precise Solution which make your kernel and development tools selection an easy one.

All you have to do is turn the page and keep reading...
Intermetrics, The Solution Supplier

Our mission is simple—to offer a high quality kernel and integrated development environment at an affordable price!

Embedded Software Expertise for a Quarter Century

Intermetrics has provided software development tools and engineering services to satisfied customers since 1969. As a pioneer in embedded systems programming, we set the standard for quality cross-development tools. We’ve been a leader in compiler technology and host- and target-based development and debug tools. It’s only natural that we have broadened our tool suite to include a state-of-the-art, real-time kernel.

A public company with strong financial resources and hundreds of software professionals, Intermetrics provides the stability and security that you and your development team should require from your suppliers. You can be confident that we will be here through your development life cycle, and that we’ll be ready to support your next project with our latest technology.

Our thousands of installed bases assure your proven success.

Key Partnerships

Intermetrics’ broad support of microprocessors and microcontrollers with quality development tools has resulted in the forging of a strong relationship between Intermetrics and major semiconductor companies. For
example, Intermetrics is a Platinum member of the Motorola Third Party Developers program and is therefore one of the select few software suppliers to be recognized for its outstanding and consistent support of the Motorola silicon family.

Intermetrics works to ensure that our development path follows that of silicon technology. Our relationship with key semiconductor manufacturers allows us to be there early with full tool sets and kernels to support new silicon introductions.

Precise/MQX Technology

The MQX technology at the core of The Precise Solution represents a long-standing history of kernel excellence. The technology foundation was originally built over a decade ago by the National Research Council of Canada and Carleton University in Ottawa and was known as the Harmony Real-time Operating System. The Harmony technology was subsequently commercialized by real-time software engineers from DY-4 Systems, Bell-Northern Research, and Robotechnik GmbH. This commercialization resulted in the introduction of the Precise Software Technologies, Inc. (PSTI) kernels which today include MQX and MQX+M for single and multiprocessing applications, respectively.
The People Behind the Solution

The Precise Solution, which is sold, marketed, and supported by Intermetrics, comes from a team of some of the world's foremost experts in compiler, debugger, and kernel technology. Highly qualified technical experts have joined forces to bring embedded systems programmers a Solution to count on. Intermetrics, Inc. and Precise Software Technologies, Inc. have collectively invested over ten person years to develop The Precise Solution.

Tech Support—A Single Point of Contact

Make Progress NOT Phone Calls

You now have a single point of contact to guide you through installation and development. You don’t need to chase many different vendors, only to hear from each one that your problems are caused by “the other guy.” We take full responsibility for all of our products, from the editor to the compiler to the debugger to the kernel to the communications component. Your team should expect no less!
Introducing
The Precise Solution

In a word, the Precise/MQX Solution is complete. The Precise Solution consists of a suite of fully integrated products designed to support you through the entire development cycle.

The Precise Solution runs on MS Windows and on UNIX workstations. It is comprised of the following:

- **Professional Interactive Development Environments (IDEs)** that provide an integrated, GUI point-and-click interface to our InterTools compilers; the PassKey/MQX source level debugger, and the Precise/MQX real-time kernels.
- **InterTools optimizing C cross compiler** that runs on PC and UNIX hosts, with specific optimization for each member of the M68000 family.
- **PassKey/MQX**, our UNIX and MS Windows-based, kernel-aware, cross debugger, is a host-resident, source-level debugger. It lets you set task-qualified breakpoints, display/modify kernel data structures, and execute MQX services and C functions interactively from your UNIX or Windows environment.
- **Precise/MQX**, our real-time executive, is based on over a decade of proven kernel technology; it allows you to create an application that is reliable, structured, easily documented and maintained.
- **Precise/MQX+M**, our multiprocessing kernel, provides seamless migration to multiprocessing.
• **MQX/Sim host-based simulation tools** that enable you to create and debug MQX applications for single and multiple processor applications using powerful and familiar tools native to the UNIX and Windows environments so that you can develop and test your code even before hardware is available.

• **Embedded I/O components** that are a family of data communications and networking protocols specifically adapted and implemented for real-time and embedded applications. They support: SDLC, LAPB, MIL-STD 1553 bus, TCP/IP, FDDI, GPIB, and File I/O.

• **MQX hardware/board support packages** that support popular, off-the-shell, single-board computers (including Motorola, DY-4 Systems, Force, Synergy, GreenSpring, Radstone, and Tadpole). You can easily port MQX to your custom hardware using our extensive library of C and assembly language, chip-level drivers. In addition, custom integration services are available for user-specific hardware solutions.

• **MQX In-Circuit Emulator support**. The Precise Solution supports the use of popular in-circuit emulators for non-intrusive debug of real-time software.

• **Training**. Receive training on any or all of the components of the Precise Solution. Training is available at our facility or yours.

• **Support**. One call does it all. Get support for all components of the Precise Solution from Intermetrics.

Intermetrics is the first company to offer a fully integrated, premium quality, design and development system for real-time kernel development at a price you can actually afford.

Other solutions come with a high price to pay—not only in the form of expensive development licenses but significant production royalties as well. With the Precise Solution, you **pay no royalties no matter how many copies you ultimately ship**. You also avoid the long-term administrative burden of royalty reporting.

We believe in protecting your investment. We provide full source code to the kernel and embedded I/O components. Since MQX is portable, it is currently available on many microprocessor architectures.
Introducing The Precise Solution

The Precise Solution
Task-Aware Debug Environment

Host Computer with GUI
The Precise Solution—Full Life-Cycle Support

The Precise Solution is a suite of products and tools that a real-time system designer can apply throughout a product’s life cycle. To help protect your investment, the Precise Solution presents an open environment in which third party tools may be used.
Because many of the tools included with the Precise Solution are useful for multiple phases of the design cycle, you can apply them (and amortize their cost) throughout your product’s development life cycle. For example, consider requirements for field beta testing and how they will be included in the end product. Our MPD monitor or Motorola’s Background Debug Mode connector can be included in the final product, allowing you to debug your MQX application on-site using the powerful PassKey/MQX debugger.

The sections that follow present the components of the Precise Solution as they might be used during each phase of a product life cycle.
The Design Phase—the MQX Kernel

MQX Kernel Highlights

- POSIX compliant components to promote portability
- Portable architecture promotes code reuse
- Provides a stable, reliable foundation on which to build your application
- Reduces time to market
- Reduces documentation and maintenance costs
- ANSI C Source code supplied for all MQX components
- Built upon over a decade of proven kernel technology

MQX Design Philosophy

The MQX real-time executive is the heart of your product design. It allows you to create high performance, real-time applications that are structured, easily documented, and easily maintained. Your design may consist of MQX or MQX+M (our multiprocessor executive) plus any of our industry standard I/O components.

MQX, based on over 12 years of proven kernel technology, was created with the following goals in mind:
Reentrant Operation
MQX is completely reentrant (any system service can be interrupted) which makes them responsive to the demands of real-time systems. This also reduces the time required to service high priority events in your system.

High Performance Operation
MQX performance is outstanding. All time-critical services have been optimized for both speed and determinism. Context switch times and interrupt latencies are minimal. Our I/O drivers are tuned for maximum data throughput.

Deadlock (and Starvation) Detection and Prevention
The MQX Watchdog component prevents deadlock and starvation by automatically detecting if a task has run longer than intended. If a task runs longer than its allotted time, a user-provided error handler is invoked.

Source and Binary Delivery
All target-based components are available in both object library format and ANSI C source. With MQX, you have complete control over and confidence in all the code embedded in your application.

Scaleable Components
To support a wide range of applications from small, memory-restrained products to large, inter-networked, multi-processor systems, you need only include the calls that you actually use in your design. As a result, code requirements are greatly reduced.

Rock-Solid, Reliable Operation
Many mission critical products depend on MQX. We have installations world-wide with applications ranging from: satellites to aircraft to instrumentation.

Comprehensive Kernel Services
MQX provides a full complement of real-time services, including deterministic priority-based scheduling, time management, buffer and pool management, and communications and synchronization mechanisms for use between interrupt service routines, device drivers, and tasks.
Resource Reclamation
When MQX tasks and their children are deleted or destroyed, all variable-sized memory and resources (task control block, stack, etc.) are automatically reclaimed by the kernel, promoting long-term reliability and robustness.

Safe Memory Management
MQX is the safe choice when it comes to memory management. MQX has services to help fight memory leaks, memory fragmentation, memory errors, and memory and stack overwrites.

Scaleable to Multiple Processors
Single processor applications can be scaled to a multiprocessor configuration without source code changes. Processors can be connected via any media using our unique, variable-sized message-passing paradigm. MQX even supports heterogeneous configurations with built-in message routing and endian-order swapping.

Priority Inversion Free
MQX supports Mutexes with priority ceiling and semaphores with Priority inheritance to help avoid priority inversion in your real-time design

Portability/Investment Protection
Over 95% of MQX is in ANSI C this protects your investment because you are never locked into a processor family. MQX can be ported to virtually any processor.

MQX Kernel Overview
MQX lets you organize your programs into functional groups called tasks, interrupt service routines (ISRs), and device drivers, thereby making your code easier to design, test and maintain.

Tasks
In MQX, tasks are individual units of code that can compete for system resources in a real-time system. These resources may include the CPU, memory, or I/O. MQX schedules (decides when to run) each task based on a software priority that you assign. In MQX, tasks look like C functions, so if
you know C you will find that MQX provides a natural multitasking extension to the language.

**ISRs**

Interrupt service routines allow the processor to service real-time requests from external devices quickly and efficiently. With MQX, your ISRs can call any non-blocking system call allowing you to benefit from the rich set of services that MQX provides. In addition, if your application requires very high speed interrupt response, approaching the maximum hardware interrupt latency of the processor, MQX lets you service these critical interrupts outside the kernel.

**Device drivers**

In MQX, device drivers are server tasks that typically provide access to an external device for I/O. Standard drivers are provided for TCP/IP, SDLC, LAPB and others. You can also write your own device drivers that take advantage of all the features of the MQX kernel.

**Categories of Services**

The MQX kernel is comprised of fast and deterministic kernel services, which we will discuss in the following order:

- Task management services
- Message passing services
- Event services
- Mutual exclusion services
- Semaphore services
- Storage allocation services
- Time management and clock services
- Naming services
- Deadlock detection and prevention services
- I/O services
- Interrupt services
- Error handling services
- Kernel log services
Task Management Services

### Task Management Highlights

- Supports dynamic creation and deletion of tasks
- There is no practical limit on the number of objects tasks that can be created—you are limited only by memory.
- All critical calls execute in constant time allowing you to create a deterministic, predictable system
- The MQX scheduler is POSIX compliant.
- Deadlock and task starvation detection are built into the MQX kernel
- Resources are reclaimed on task destruction, including those of child tasks
- Task priorities can be changed dynamically
- Task stack sizes can be set on a per-task basis

---

<table>
<thead>
<tr>
<th>Task Management Services</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Add_ready</td>
<td>Makes a task ready to run.</td>
</tr>
<tr>
<td>_Block</td>
<td>Blocks task execution.</td>
</tr>
<tr>
<td>_Convert_to_td</td>
<td>Converts a task ID to a task descriptor address.</td>
</tr>
<tr>
<td>_Create</td>
<td>Allocates and starts a new task</td>
</tr>
<tr>
<td>_Destroy</td>
<td>Deallocates a task.</td>
</tr>
<tr>
<td>_Father_id</td>
<td>Gets the task ID of the parent task.</td>
</tr>
<tr>
<td>_Home_processor</td>
<td>Gets the processor number where a specified task is running.</td>
</tr>
<tr>
<td>_MQX</td>
<td>Initializes and starts the MQX kernel on the current processor.</td>
</tr>
<tr>
<td>_MQX_Exit</td>
<td>Exits MQX and returns to the environment that called _MQX.</td>
</tr>
<tr>
<td>_My_id</td>
<td>Gets the calling task's ID.</td>
</tr>
<tr>
<td>_Reschedule</td>
<td>Places the currently running task at the end of its priority queue.</td>
</tr>
<tr>
<td>_sched_get_priority</td>
<td>Obtains the maximum task priority level.</td>
</tr>
</tbody>
</table>
__max
_sched_get_priority
_min
_sched_get_rr
_interval
_sched_getparam
_sched_setparam
_sched_setscheduler
_sched_yield
_Set_task_error_code
_Task_error_code

Obtains the minimum task priority level.

Gets the current round robin interval time.

Gets the current scheduling policy parameters.

Modifies the current scheduling policy parameters.

Sets the current scheduling policy.

Relinquishes control of the CPU.

Sets the task error code.

Returns the task error code for the calling or active task.

---

**Task Template Structure**

The task template structure is a C structure that is used to provide parameters required by MQX to create an instance of a task. The parameters include:

- Task Index: an index into the task template struct
- Start Address of Task: the address where task code begins
- Task Priority: the relative software priority level of the task
- Task Stack Size: the size of the stack for a task
- Task Debug Name: symbol name used by the task-aware debugger
- Task Attribute: if the task is to be created at system startup
- Task Creation Parameter: a pointer to data element passed to task upon creation

A task is a unique instance of a task template. At any time, several tasks created from the same task template may be running. Every task has a unique task ID. MQX maintains each instance by saving the task's program counter, register usage, and stack. Any task can create another task, by calling the `Create` primitive. The newly readied task does not become active until the kernel runs (does a dispatch).

When you create an MQX task, you can optionally pass it a 32-bit parameter. This is useful if you wish to create multiple tasks that all use the
same code but require different initialization parameters. (As an example, you could use the parameter as a pointer to some initialization data for an I/O channel.)

In MQX, a task's stack size can be set on a task-by-task basis. This provides you with maximum flexibility when configuring your system.

**Kernel Initialization Tasks**

Internal system tasks (the idle task and local task manager) are automatically created by the kernel during initialization. The idle task runs when there are no user tasks ready to run. The local task manager creates and deletes MQX tasks.

**Task Priorities**

MQX allows you to set a software priority level for each of your tasks. Task priorities help you to design a system that ensures that the most important task is always the one that runs at any given time. MQX supports $2^{32}$ priorities.

Task priorities are set when a task is created and can be changed (raised or lowered) at any time with the scheduler primitives.

**Task Scheduling**

One of the most important functions of a real-time multitasking executive is to support dynamic scheduling of task execution. The MQX scheduler is designed to be POSIX compliant and employs a policy known as *Natural Break Preemptive Scheduling*. This means that the highest priority, ready-to-run task will always execute.

The MQX scheduler also allows round-robin scheduling with a time-slice option. This scheduling policy can be selected by an individual task, or as the default for all created tasks. In this policy, each task can select a “time period” to indicate how long it should run once it gains control of the processor. When this time period is over, the kernel removes the running task, and places it at the end of its priority level queue.

A task will continue to execute until one of the following happens:

- It voluntarily gives up the processor by calling `Block`
- The task's time-slice has expired
The Design Phase—The MQX Kernel

- The task blocks waiting on a resource.
- An interrupt occurs that needs to be serviced.
- A higher priority task has become ready to run.

If any of the above conditions occurs, the active task may become blocked, and the kernel must do what is known as a dispatch. When MQX performs a dispatch, the highest priority, ready-to-run task is made the active task. If more than one task is ready to run at the same priority level, then the task that has been ready to run the longest is made the active task. MQX is designed so that the time to dispatch the highest priority task is fixed, regardless of the number of tasks in the system.

Task States
A task can be in one of the following states:

- **Ready**: A ready task is one that could run, but is not the active task, and is not executing. Either it has a lower priority than the active task or it has the same priority as the active task, but became ready after the active task.
- **Blocked**: The task has issued a system call and is attempting to gain access to a resource that is not available. Tasks can wait on: events, queues, semaphores etc. Tasks can wait forever or until a specified time-out expires. A task can also become blocked by calling the **Block** primitive.
- **Active**: The task has control of the processor and is executing its code.
- **Dying**: The task is in the process of being destroyed. All of its resources are being released. The task is not destroyed until after the resources have been released. Note that one task can destroy any other, as long as it knows that task’s ID. At task destruction, the task’s stack, task control blocks, variable-size memory, and message buffers are reclaimed.
Inter-Task Communications

- Message queues are owned by the creating task.
- Tasks may wait on multiple queues.
- Message queues are owned by the task that creates them.
- Tasks may wait on multiple message queues.
- Flexible message pool management—**named** (fast) and **general** (efficient)
- Variable size message are supported (up to 64K bytes)

<table>
<thead>
<tr>
<th>Intertask Communication Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Create_pool</td>
</tr>
<tr>
<td>_Create_named_pool</td>
</tr>
<tr>
<td>_Destroy_named_pool</td>
</tr>
<tr>
<td>_Free_buffer</td>
</tr>
<tr>
<td>_Get_buffer</td>
</tr>
<tr>
<td>_Get_named_buffer</td>
</tr>
<tr>
<td>_Open_queue</td>
</tr>
<tr>
<td>_Convert_to_qid</td>
</tr>
<tr>
<td>_Message_pending</td>
</tr>
<tr>
<td>_Send_message</td>
</tr>
<tr>
<td>_Receive_message</td>
</tr>
<tr>
<td>_Receive_message_queue</td>
</tr>
<tr>
<td>_Close_queue</td>
</tr>
</tbody>
</table>

Tasks in MQX communicate by sending and receiving messages. Messages are sent to and received from a task’s messaging queues. Message queues are mechanisms that are used by MQX tasks to deposit and retrieve messages. Tasks communicate by sending and receiving messages to and from message queues. Any task can send a message to any queue in the system, but only the task that opened a queue may receive messages from that queue.
Queues are used to transfer messages between tasks. Queue size may be fixed or unbounded. Closing a queue deletes all entries on the queue and frees the message buffers.

Messages are created in message buffers. Message buffers are allocated from message pools. MQX supports two types of message pools: general pools and named pools.

**Message Pools**

General message pools are considered to be resources of the system and not of any specific task. This means that any task can allocate a message from this pool. Thus, there are no restrictions on which tasks can access a general pool. When a task requests a message buffer, MQX allocates it the best available buffer. Once created, a general pool cannot be destroyed, and it will exist for the life of the system.

When named message pools are created, MQX returns a pool ID to the task requesting the pool. When a task requests a message buffer from this type of pool, it must supply the pool ID. Since a task requests buffers from a specific, named pool, allocating message buffers from named pools is fast and deterministic. Unlike general pools, a named pool's lifespan is determined when all of its buffers have been returned.

When preparing to send a message to another task, a task must first request a message buffer of an appropriate size from the kernel. In response, MQX allocates a free buffer from an appropriate message pool. (A free buffer is one that is not currently allocated to any task.)

When a task is finished with a message buffer, it should return the buffer to its pool. Message buffers are typically returned after they have been received from an input queue.

**Message Passing Sequence**

After a task has created a message using an allocated message buffer, it can then send the message to another task by using the _Send_message primitive.

To receive messages, a task issues _Receive_message for a queue with messages waiting on it. If there are messages queued, the request is satisfied immediately and the task continues to run without interruption. If there are
no messages waiting, the task will block until either an appropriate message is received or a specified time-out value expires.
**Event Services**

**Event Primitives**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_event_mkevent</td>
<td>Create an instance of an event.</td>
</tr>
<tr>
<td>_event_destroy</td>
<td>Destroy an instance of an event.</td>
</tr>
<tr>
<td>_event_open</td>
<td>Open (gain access to) an event.</td>
</tr>
<tr>
<td>_event_close</td>
<td>Close (relinquish access to) an event.</td>
</tr>
<tr>
<td>_event_wait_all</td>
<td>Wait for all specified events.</td>
</tr>
<tr>
<td>_event_wait_any</td>
<td>Wait for any specified event.</td>
</tr>
<tr>
<td>_event_set</td>
<td>Set the specified event bits in an event.</td>
</tr>
<tr>
<td>_event_clear</td>
<td>Clear the specified event bits in an event.</td>
</tr>
<tr>
<td>_event_value</td>
<td>Obtain the current event bits that have been set.</td>
</tr>
<tr>
<td>_event_task_wait_count</td>
<td>Obtain the number of tasks waiting for an event.</td>
</tr>
</tbody>
</table>

The MQX event component is designed to provide a signaling mechanism between MQX tasks. Events are unowned system objects that are used to synchronize multiple tasks or tasks and ISRs. Events consist of a 32-bit pattern which can be modified by using the _event_set (set any bit to “1”) and _event_clear (set any bit to “0”) system calls. Tasks can wait at an event for a particular bit pattern. Tasks may wait or wait with time-out in FIFO or priority order. When a particular bit pattern is satisfied, the blocked task will be made ready. The value of any event can be read at any time using the _event_value system call. The number of the task currently

The Design Phase—The MQX Kernel
waiting at any event can be obtained using the \_event_task_wait_count system call.

**Queuing Options**

If the flag event_priority_queuing is set, then when a task is queued, waiting for an event, it will be queued according to its software task priority.

Each instance of an event can be opened by multiple tasks, each of which can wait for a specified event(s). The events are identified by the 32 bit bit-mask. When the events arrive, the queue of waiting tasks is searched to determine which tasks should be activated. If the event priority queuing flag is not set, the default queuing mechanism is FIFO.

**Event Creation**

An instance of an event identified by the string name will be created.

**Opening a New Event**

The event identified by the string name will be opened on the specified processor. If a processor number is not provided, the event will be opened on the local processor. The EVENT_PTR value returned is to be used in all functions used to access the event.

**Setting Events**

This function causes the specified bits (all bits of value 1) to be set in the event.

**Clearing Events**

This function causes the specified bits (all bits of value 1) to be cleared in the event.

**Waiting for Events**

This function is wait for all events in a specific set of events to be set. This function is wait for any one event in a specific set of events to be set.

**Reading an Event’s Value**

This function returns the current value of the event.
Closing an Event
This function causes the event_ptr used by the task to access a event, to no longer be valid. Any use of this handle with other event package functions, will cause an error.

Event Destruction
The event identified by the string name will be destroyed on the specified processor. If a processor number is not provided, the event will be destroyed on the local processor.

If the force_destroy flag is FALSE then the destruction of the event will occur only when the last waiting task has finished waiting. If the flag is TRUE, then all waiting tasks will be activated, and the event destroyed.

Testing Services
This function returns the number of tasks waiting for an event.
Mutual Exclusion

Features and Benefits of the Mutual Exclusion Services

- POSIX compliant
- Fast, deterministic, and indivisible access
- Supports priority ceiling
- Supports priority inheritance
- Tasks can wait, wait with time-out, or poll
- Tasks can wait spin locked, limited spin lock, and priority-based

Mutex Primitives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_mutex_create_component</td>
<td>Install the mutex component</td>
</tr>
<tr>
<td>pthread_mutex_destroy</td>
<td>Destroy a mutex.</td>
</tr>
<tr>
<td>pthread_mutexattr_getprioceiling</td>
<td>Get a mutex's priority ceiling value.</td>
</tr>
<tr>
<td>pthread_mutex_init</td>
<td>Initialize a mutex.</td>
</tr>
<tr>
<td>pthread_mutex_lock</td>
<td>Lock (obtain) a mutex.</td>
</tr>
<tr>
<td>pthread_mutex_task_wait_count</td>
<td>Get number of tasks waiting for mutex.</td>
</tr>
<tr>
<td>pthread_mutex_getprioceiling</td>
<td>Get a mutex's priority ceiling value.</td>
</tr>
<tr>
<td>pthread_mutex_try_lock</td>
<td>Lock (obtain) a mutex without waiting</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>Release (free) a mutex.</td>
</tr>
<tr>
<td>pthread_mutexattr_destroy</td>
<td>Destroy a mutex attributes object.</td>
</tr>
<tr>
<td>pthread_mutexattr_getprotocol</td>
<td>Obtain a mutex attributes priority inheritance protocol values.</td>
</tr>
<tr>
<td>pthread_mutexattr_init</td>
<td>Initialize a mutex attributes structure to the default values.</td>
</tr>
<tr>
<td>pthread_mutexattr_setprioceiling</td>
<td>Set a mutex's priority ceiling value.</td>
</tr>
<tr>
<td>pthread_mutexattr_t</td>
<td>The definition of what a mutex attribute's object is.</td>
</tr>
<tr>
<td>pthread_mutexattr_setprotocol</td>
<td>Initialize a mutex attribute's priority inheritance protocol value.</td>
</tr>
</tbody>
</table>
The MQX mutex package is designed to provide mutual exclusion to a shared resource, between MQX tasks, on a single processor. The mutex functions are compliant with those defined in the POSIX threads standard: P1003.4a/D7, April 1993. This implementation allows for priority inheritance and priority ceiling mutexes.

- **Priority inheritance** means that when a task owns the mutex, its software priority will be the maximum of its own software priority, or that of the highest priority task waiting for the mutex.
- **Priority ceiling** means that when a task acquires the mutex, its software priority will be the maximum of its own software priority, or the priority of the mutex, specified when the mutex was created.

If a mutex is not available, a number of waiting protocols are provided:

- **Spin waiting** means that the task sits in a while loop trying to acquire the mutex. This method is applicable to a time-sliced task.
- **Limited spin** is the same as spin, but with a counter used to time-out the waiting.
- **Queued** means that the task puts itself on a waiting queue of tasks. When the mutex is freed, the first available task is scheduled to run.
- **Priority queued** is the same as queued, except that when the task is queued, it is put onto the queue in order of task priority. Thus the highest priority task waiting will get the mutex next.
Semaphore Services

Semaphore Services:

- Fast and deterministic operation
- POSIX compatible
- Supports priority inheritance to prevent priority inversion
- May be created and deleted dynamically for maximum flexibility with minimum memory usage
- Tasks may wait queue in FIFO or priority order with optional time-out

Semaphore Primitives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_sem_create_component</td>
<td>Install the semaphore component into the kernel.</td>
</tr>
<tr>
<td>_sem_mksem</td>
<td>Create an instance of a semaphore</td>
</tr>
<tr>
<td>_sem_open</td>
<td>Open or gain access to a semaphore</td>
</tr>
<tr>
<td>_sem_close</td>
<td>Close or relinquish access to a semaphore</td>
</tr>
<tr>
<td>_sem_wait</td>
<td>Wait for a semaphore activation.</td>
</tr>
<tr>
<td>_sem_destroy</td>
<td>Destroy a semaphore</td>
</tr>
<tr>
<td>_sem_post</td>
<td>Return a semaphore activation.</td>
</tr>
<tr>
<td>_sem_value</td>
<td>Return the number of semaphore activations available.</td>
</tr>
<tr>
<td>_sem_task_wait_count</td>
<td>Return the number of tasks waiting for semaphore activation.</td>
</tr>
</tbody>
</table>

The semaphore component of MQX provides a means for a task to synchronize access to some shared resource. The semaphores provided by MQX are named, counting semaphores: To create, destroy, or replace a named semaphore, you need to know its name. For a counting semaphore, you need to know its value. When a task waits on a counting semaphore, the counter is decremented, and the task is allowed to proceed. When the counter reaches 0, the task is suspended until a semaphore is posted (returned).
MQX semaphores are strictly monitored for safety and long-term robustness—a task posting a semaphore must have previously waited on it.

**Semaphore Creation and Waiting**

`_sem_mksem` creates an instance of a semaphore which has allows, in `sem_count`, a maximum number of task activations. Further tasks requiring activation are queued waiting for the return of semaphore.

Tasks can be queued at a semaphore according to their task priority when waiting for the semaphore. If the `flag` is set to 0, the tasks will be queued in a first-come, first-served basis when waiting for the semaphore. (A task may wait forever for the semaphore by specifying a time-out of 0.)

When a semaphore activation is returned, the first task in the semaphore queue is activated.

The function `_sem_task_wait_count` returns the number of tasks waiting at a semaphore. The `_sem_value` returns the number of semaphore activations available.

A semaphore can only be destroyed if it was previously created by `sem_mksem`.

If the `force_destroy` isn’t used, this semaphore will destruct only when the last waiting task has relinquished it. Otherwise, all waiting tasks will be activated.
Storage Allocation Services

Storage Highlights

- Choose between fast, fixed-size memory or more efficient, variable-sized memory blocks
- Track your allocation and freeing of memory so that you know where you stand in relation to potential memory leaks
- Reclaim all variable-sized memory on task deletion, including that used by a task’s children
- Use either of two garbage collector options so that you can have deterministic or non-deterministic access to control over memory fragmentation.
- Use a memory-testing facility that is designed to be run in the background to constantly test memory integrity for the whole system.

Variable-Size Memory Manager Primitives

- _Getvec: Allocates bytes for a memory block
- _Zeromem: zeroes the memory block
- _Get_zerovec: Allocates and zeroes a memory block
- _Freevec: Returns the memory block to the pool

Fixed-Size Memory Manager Primitives

- _Create_Named_Pool: Creates pool of fixed-size memory buffers
- _Destroy_named_pool: Deletes pool of fixed-size memory buffers
- _Get_named_buffer: Allocates fixed-size memory block
- _Free_buffer: Returns the memory block to a named pool

Related Memory Manager Utility Primitives

- _Copy: Copies one buffer to another
- _Hightestmem: Returns highest allocated memory position to date
- _Memory_block_in_error: Returns addresses of blocks with failed checksums
- _Sizevec: determines the exact size of a memory block
- _Testmem: checks internal checksums for memory violations
The MQX kernel provides functions to dynamically allocate and free variable-sized memory, much like the `malloc` and `free` functions in most C run-time libraries. MQX also supports fixed-sized memory for deterministic operations. The memory management functions divide memory up and give pieces of it to a task when it asks for it. Memory is a resource of the task that allocated it. If the task is destroyed, its variable-sized memory resources are automatically freed and returned to the free memory pool.

### Variable Size Memory

When a memory request is received, the memory manager allocates a block of the appropriate size to the requesting task. If the next free block is too large, then it is broken into two parts: one is returned to the calling task, the other is put onto the memory manager’s free list of memory blocks.

When a memory block is returned, the memory blocks physically located before and after this block are checked to determine whether or not they are currently free. If they are, the blocks are coalesced to form as large a block as possible and this block is placed on the free list.

Precise/MQX uses an algorithm known as **nearest neighbor coalescing** when memory blocks are deallocated. This is a deterministic method of garbage collection that is bounded in execution time. The intent is to minimize memory fragmentation with an algorithm that also addresses the requirements of a real-time system which are low overhead and predictable execution time. Essentially, the kernel tries to coalesce a newly-freed block with the block previous to it, or the next block after it. If one of these is free, the returning block is added to it to form a single free block.

In addition to built-in memory management, MQX provides utilities to the user that provide another level of control over memory resources. These utilities allow the application to perform the following functions:

- Determine the exact size of memory allocated by MQX
- Return unused or unneeded memory to MQX
- Copy contents of one buffer to another
The Design Phase—The MQX Kernel 31

- Initialize a buffer to “0”
- Determine the highest memory address allocated by MQX
- Check internal checksums and determine integrity of memory pool

The user also can set memory management configuration options at compile time. These options:

- Cause MQX to print error conditions and progress indications
- Cause MQX to count the number of _Getvec and _Freevec primitives used. This provides a mechanism to track allocation of memory and thereby prevent memory leaks.
- Determine the memory reclamation algorithm (Nearest Neighbor or Insertion Sort) used by MQX. Nearest Neighbor is the default memory management policy for MQX. You can override this policy with the Insertion Sort algorithm, which is more efficient in a system with a mix of small and large memory buffers.

Fixed Size Memory

Fixed size memory management is accomplished through the use of primitives designed for message pool allocation. Named pools can be used as general purpose memory pools. Since the user specifies the number and the size of the memory partitions when a named pool is created, no best fit algorithm is used when a memory buffer is requested from this pool. Without this overhead, memory allocation is fast and deterministic.
Time Management and Clock Services

Time Primitives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Get_time</td>
<td>returns system up-time</td>
</tr>
<tr>
<td>_Set_time</td>
<td>sets time to an arbitrary value</td>
</tr>
<tr>
<td>_Timeout</td>
<td>suspends operation for some milliseconds</td>
</tr>
</tbody>
</table>

The Precise/MQX kernel provides some basic time services for use by applications. The clock component provides a means for a task to obtain periodic notification via messages. It also provides functions that map the system time (seconds and milliseconds) to a date and time format (year, month, day, hour, minute, second, millisecond).

Delay Services

A task can choose to suspend its operation for a specified number of milliseconds. When they have expired, the task is automatically placed on the ready-to-run queue and must wait until it is the highest priority ready task before it becomes active again.

Clock Server

Clock Server Primitives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK_Cancel_alarm</td>
<td></td>
</tr>
<tr>
<td>CLK_Time_from_struct</td>
<td></td>
</tr>
<tr>
<td>CLK_Time_to_structt</td>
<td></td>
</tr>
</tbody>
</table>

The Clock component of MQX provides a means for a task to obtain periodic notification via messages. It also provides functions that map the system time (seconds and milliseconds) to a date and time format (year, month, day, hour, minute, second, millisecond).

A task can use CLK_Alarm to request alarms, and CLK_Cancel_alarm to cancel them.
I/O Services

Formatted I/O System Primitives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>Open an I/O stream, obtaining a handle to it</td>
</tr>
<tr>
<td>getchar, fgetc</td>
<td>Get a character from an input source</td>
</tr>
<tr>
<td>getline, fgetline</td>
<td>Get a line of characters from an input source</td>
</tr>
<tr>
<td>gets, fgets</td>
<td>Get a line of characters from an input source and put it in a string</td>
</tr>
<tr>
<td>printf, sprintf, fprintf</td>
<td>Do ANSI C compatible formatted output to a device or string</td>
</tr>
<tr>
<td>putchar, fputc</td>
<td>Put a character to an output device or stream</td>
</tr>
<tr>
<td>puts, fputs</td>
<td>Puts a string of characters to an output device or stream</td>
</tr>
<tr>
<td>scanf, sscanf, fscanf</td>
<td>Formatted input from and input device or string</td>
</tr>
<tr>
<td>status, fstatus</td>
<td>Get status of input source</td>
</tr>
<tr>
<td>ungetc, fungetc</td>
<td>Put character back on an input source</td>
</tr>
<tr>
<td>_Get_stdio</td>
<td>Get address of default standard I/O stream</td>
</tr>
<tr>
<td>_Set_stdio</td>
<td>Set the standard I/O handle streams</td>
</tr>
</tbody>
</table>

MQX provides non-interrupt-driven and interrupt driven I/O that can be configured at initialization time to provide XON/XOFF protocol flow, to translate carriage returns and handle back-spacing, and to echo input characters. In addition, MQX provides standard I/O functions that:

- Open an I/O stream and gets its address
- Get characters and lines from input sources
- Put characters and strings to and output device or stream
- Do ANSI C compatible formatted I/O to a device or string
- Get formatted input from input device or string
Interrupt Handling

Interrupt Services Primitives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Enable</td>
<td>Enable hardware interrupts</td>
</tr>
<tr>
<td>_Disable</td>
<td>Disable hardware interrupts</td>
</tr>
<tr>
<td>_Install_first_level_notifier</td>
<td>Initialize H/W interrupt vector for an interrupt handler outside of MQX</td>
</tr>
<tr>
<td>_Install_notifier</td>
<td>Initialize second level interrupt handler</td>
</tr>
<tr>
<td>_Flint</td>
<td>First level interrupt H/W handler</td>
</tr>
<tr>
<td>_Open_notifier_queue</td>
<td>Open a queue for a notifier task</td>
</tr>
<tr>
<td>_Close_queue</td>
<td>Close notifier queue</td>
</tr>
<tr>
<td>_receive_message_queue</td>
<td>Non-blocking receive at notifier queue</td>
</tr>
</tbody>
</table>

Description of Interrupt Handling

MQX is designed to catch all hardware interrupts and save the context of the currently active task. MQX is specifically optimized for minimal interrupt latency.

A notifier is the kernel’s method of handling hardware interrupts. A notifier is not a task but a small, high-speed routine that reacts quickly to hardware interrupts. Notifiers are typically written in C. Its duties include resetting the device, obtaining its data, and signaling the appropriate task. A notifier is referred to as a second-level interrupt handling routine in MQX.

Notifier Queues

The MQX kernel supplies a special communication mechanism designed for, but not limited to, interrupt handling. Notifier queues are in some ways similar to message queues described earlier. Unlike message queues, notifier queues are not owned by a task. When installed(created), a function and data can be associated to the queue so that when sent to, cause the function(possibly a notifier) to execute. This is a useful way for an application to send a software interrupt to a notifier.

Interrupts Outside MQX

In some applications, event handling exceeds the overhead for even a low overhead context switch. For these cases, we have provided a mechanism
to bypass the kernel, thereby giving the problem the full attention of the microprocessor. This mechanism is called the **First Level Notifier**. Installing a First Level Notifier basically assigns a hardware interrupt to a function that will handle the interrupt outside of MQX. Since these interrupts do not pass through the MQX first level interrupt handler, interrupts can be handled more quickly.
Deadlock Detection and Prevention Services

Watchdog Primitives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_watchdog_create_component</td>
<td>installs component into kernel</td>
</tr>
<tr>
<td>_watchdog_start</td>
<td>starts/restarts watchdog</td>
</tr>
<tr>
<td>_watchdog_stop</td>
<td>stops/disables watchdog</td>
</tr>
</tbody>
</table>

The watchdog component prevents deadlocks. It allows for the monitoring of tasks, so that they are not abandoned if another task tries to consume an excessive amount of computer use.

Deadlock or deadly embrace potentials between two tasks are solved by the watchdog function.

Starvation can also be solved with the watchdog function.

A task starts a watchdog timer by calling the _watchdog_start function and specifying an absolute maximum time limit for the task to execute. If the task does not call _watchdog_stop before the time expires, then a user-supplied error function will be called, passing the address of the offending task descriptor.
Error Handling

Task Error Code Interface Primitives

| _Set_task_error_code | clears or resets the code |
| _Task_error_code     | returns the calling task's current code |

Each task has an error code associated with its context. Each time the task executes, the kernel sets its error code. If an error is ignored, additional errors will occur.

Any further errors that are detected will not change the error code until the task explicitly clears it, since in most cases the first detected error provides the best indication of the problem.

Usually, the application task uses OK as the input parameter to _Set_task_error_code to retrieve the current task error code and then clear it. If the input parameter is any other value than OK, and the current task error code is OK, the task error code will be set to the input value. If the current task error code is not OK, the input value is discarded.
Kernel Log Services

Log Component Primitives

```
_log_create_component _log_destroy _log_reset
_log_disable _log_enable _log_write
_log_read
```

The data log component allows for the storage and retrieval of data from a log. This log component allows for the creation of up to 16 separately configurable logs. The data in a log may be used as part of the program algorithm, or to store debugging information, or in the display of task or program performance. Each data element in the log is time-stamped and sequenced.
Naming Service

<table>
<thead>
<tr>
<th>Naming Services</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_naming_add_name</td>
<td>Adds a name to the global name database</td>
</tr>
<tr>
<td>_naming_create_component</td>
<td>Installs naming services into kernel</td>
</tr>
<tr>
<td>_naming_delete_name</td>
<td>Deletes name from global name database</td>
</tr>
<tr>
<td>_naming_find_name</td>
<td>Looks up name in global name database</td>
</tr>
</tbody>
</table>

The MQX naming service is one of the kernel components. It is designed to provide for a mapping between string names and a binary value. A simple database is kept of these mappings.
MQX+M Multiprocessing Kernel Overview

Multiprocessor Support

_SWAP_ENDIAN swaps byte order

Precise/MQX was specifically designed to address, in a general manner, the complexities of running a distributed application with different types of processors. This includes the ability to mix Motorola and Intel processors where the byte ordering of information in one is the reverse of the other. These differences are referred to as little endian and big endian format.
The Coding Phase

One of the objectives of the Precise Solution is to provide a set of tools for real-time embedded systems designers to make their task of developing real-time software a speedy one. To that end, we offer an Integrated Development Environment (IDE) for both PC and UNIX users. The IDE provides a single environment from which a user can easily and quickly edit, compile, manage files and projects, and debug.

IDE

Intermetrics has integrated popular IDEs like Codewright and Crisp to provide professional programming interfaces to Intermetrics' compiler family and debugger. These IDEs work with the MQX kernels to provide fast solutions to the edit, compile, and debug cycle of development.

- You can use a command shell window to issue commands to compilers, version control systems, the operating system, and more.
- You can switch between projects using different compilers—for example, a Motorola 68360 project using the InterTools 68000 family compiler and PassKey/MQX source level debugger can be exchanged for a different Motorola-based project using other compilers and debuggers.
- You can rely on features like auto-save, smart-indenting, language templates, and compile/find errors.

Project configuration and workspace saving features simplify project organization and switching between groups of related files. You can choose
settings for a wide array of features to match your project requirements, whether you’re programming alone or in a workgroup.

You can edit text or binary files in ASCII or Hex and check differences or merge files. The IDEs don’t truncate lines, “fix” the nulls, or remove special characters from a file. You can even insert or delete bytes from a binary file.

ChromaCoding highlights different items in the source, including comments and keywords, strings, numbers, braces, and preprocessor commands. ChromaCoding has a side-by-side, color markup, difference analysis to show what changed between two revisions of a file. You can also apply color coding to printed files in a variety of differing text styles to enhance the readability of printed source code.

The selective text display capability makes it easy to focus on the source code that is important right now. By hiding unwanted portions of the text, you can more easily understand files containing conditional code. You can also display only the first line of each function in a file and then selectively open and close the functions. In addition, you can choose to see code as the compiler will see it by processing #ifdefs.

Special error parsers are included with the IDEs. By clicking on an error within the error file or using the Select Next Error command, you can automatically load a source file with the cursor placed on the offending line of code.

Command Completion for Precise/MQX kernel calls simplifies the creation of MQX tasks and removes any uncertainty of syntax. As a Precise/MQX kernel call is typed into a source file, your typing is monitored and can be expanded to the full kernel call.

Extension through Dynamic Link Libraries (DLLs) is designed into Codewright. You can use any compiler or assembler that can produce Windows DLLs. Along with more than 700 functions in our extension library for customizing the environment, you can link functions from other libraries or allow them to be called from existing DLLs.
The C Compiler Toolkit

For developers designing with the Motorola 680x0, 68EC0x0 or 683xx family of microprocessors, Intermetrics offers a stable suite of C compilers. The Intermetrics’ optimizing compilers are 100% ANSI compatible. The current version of the 68000 compilers represents the culmination of 35 man years of development.

InterTools 68000 ANSI C Compilers apply powerful optimization algorithms which, combined with target-specific code generation, produce very efficient code that is highly optimized and can be sent directly to the linker, avoiding an assembler step, or output as assembler source for use in-line in C source. The compilers also produce complete symbol information for use with the PassKey/MQX debugger, emulators, or third party debuggers.

Important ANSI features include:

- **C++ Preprocessor support** that allows you to use the toolkit in conjunction with a number of ANSI C-compatible C++ preprocessors, so that you can write C++ source code, use the preprocessor to generate ANSI C source, and compile it with your C source and any in-line assembly language.

- **Full ANSI C Compliance**: The InterTools 68000 Compiler is fully compliant with the ANSI C standard. Important C language features that are implemented in the toolkit include the ANSI features shown below.

<table>
<thead>
<tr>
<th>Important ANSI Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>function prototypes</td>
</tr>
<tr>
<td>void data type</td>
</tr>
<tr>
<td>identical member names in different structures</td>
</tr>
</tbody>
</table>

**In-Line Assembly Language** allows you to access all of the features of the M68000 family processors by defining and calling in-line assembly language macros. The _CASM method makes it easy to transform calls to out-of-line assembly language routines into in-line code. The _ASM method makes it easy for you to control in-line expansion depending on the kind of
arguments (register, memory or constant) that are passed in each _ASM invocation.

**Position Independence** produces PC-relative code that you can even move with its data from its original address to another address after it has been downloaded. You can make code and data a single position independent unit, or you can make code and data independently relocatable. Options include 32-bit AS-relative addressing, position-independent string literals and function pointers, PC-relative addressing, and separate code and data address spaces.

**Optimization features** include:

- **Performance Optimizations** to take full advantage of the 680x0, 68EC0x0 and 683xx series processors
- **All the addressing modes** available for the M68000 family processors, so that more complex addressing modes can be replaced by a larger, but faster sequence of simpler operations (unless directed to optimize for code size.
- **Automatically allocation of variables to registers**—the single most effective method for generating efficient code for processors that have large register sets, this saves a load instruction at every reference, and a store at every assignment and can reduce the code size by as much as 30%.
- **Common subexpression recognition**, which identifies common subexpressions (those that always yield the same value), computes their values once, saves the results in either registers or memory, and uses it instead of recomputing the expression. The optimizer can decide if a common subexpression will produce improved code, and it then transforms the program appropriately.
- **Strength reduction optimization**, which replaces indexing operations (traditionally done through multiplication) with pointer operations handled by addition.
- **Common computation recognition**, which lifts a computation outside of a loop if its value is the same for every loop iteration, even if contained in differing levels of nested loops.
Additional Optimizations

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algebraic Simplification</td>
<td>Copy Propagation</td>
</tr>
<tr>
<td>Constant Folding</td>
<td>Entry/Exit Optimization</td>
</tr>
<tr>
<td>Leaf Function Optimization</td>
<td>Multiplication Optimization</td>
</tr>
<tr>
<td>Short Branch Instructions</td>
<td>Subscript Optimization</td>
</tr>
<tr>
<td>Target Path Computation</td>
<td>Unreachable Code</td>
</tr>
</tbody>
</table>

**InterTools 68000 Assemblers** are 100% compatible with Motorola's assembly language specifications for each supported processor. They provide full macro facilities, structured and conditional assembly, an unlimited number of relocatable, absolute, and combinable segments, and they produce relocatable object modules ready for linking.*

The **InterTools Linking Locator**, during the link, combines relocatable object modules into a composite module after resolving references to functions and variables, and after searching through libraries for run-time routines. (The linker also supports incremental linking.) The locator phase gives you complete control over the placement of relocatable code and data segments in target memory through a user-defined command file.

**C Run-time Libraries** provide for every combination of 68000 processor and hardware or software floating point. These libraries are fully compliant with ANSI C and virtually all include source code so that you can tailor them to your application.

**Programming support utilities** include:

- **A formatter** that generates standard ASCII and binary download formats. (IEEE695, COFF and other symbol formats are available)
- **A librarian** that helps you create and maintain library files
- **A symbol list utility** that lists debugging information on all types, and global and local symbols, with target locations for source lines of input code

* Special features include the ability to select 32-bit addressing (for the 68020 and 68030) or 8-bit addressing for forward branches. For the 68020, you can force fullword alignment for improved system performance. In addition, you can use an assembler switch (-MMU) for access to special 68851 paged memory management instructions.
• A **global symbol mapper** that displays addresses for all global symbols and segments within the current memory configuration.
The Debug Phase

The Precise Solution supports a variety of debug environments. To speed up code development, many users choose to begin code debug using some sort of simulated target environment. This is typically found in one of the following forms:

- Host Simulation
- Resource mapping via In-Circuit Emulator (ICE)
- Target emulation via commercially available Single Board Computer (SBC)

The Precise Solution supports each of the environments listed above. In this section we will present other components of the Precise Solution as they relate to code debug for each of these simulated environments, in addition to debugging code in your target environment.

Host Simulation

Intermetrics provides as part of the purchase price of the Precise Solution, host-based simulation of MQX. Very simply, this is a version of MQX that runs in native mode on your host development system. This provides a means by which you can do high level functional testing of much of your MQX application code. Native simulation allows you to develop and debug MQX applications on your host using familiar and powerful native development tools.
Typically, a large portion of non-real-time hardware-independent code can be developed and debugged quickly and efficiently then recompiled and integrated with the hardware specific environment. MQX/SIM provides the following benefits:

**Use Familiar Host-based tools**
Build and test MQX applications using familiar and powerful native tools (e.g., C and C++ compilers by Borland, Microsoft, and GNU).

**Identical C-level MQX API**
Provides painless migration from Host to Target. Simply recompile with processor specific files to move tested applications to the target hardware.

**Powerful Task-Aware System Level Debug Tool**
Host tools can work in concert with the MPD task-aware debug monitor to provide MQX-aware debug information.

**Supports Single and Multi-Processor Development.**
Develop complete MQX and MQX+M applications on the PC. Your host PC can be linked to other PCs and even 68xxx hardware (running MQX+M) for complete multi-processor design and simulation.

**Hardware Independent Development**
With MQX/PC, you can verify the functionality of your MQX application before target hardware is available. Applications programmers can begin developing MQX code even before a specific processor has been selected.

**Target-based Debug**
Debugging MQX-based applications would be the same from a user standpoint regardless of the state of your target hardware (actual or emulated). Communications to the target hardware could be over a variety of media. The human interface to the debugger provided with the Precise Solution is provided by PassKey/MQX.

**The PassKey/MQX Kernel-Aware Debugger**
Intermetrics' development tools have traditionally concentrated on supporting embedded processing solutions.
That's why we have developed our debugger as we have—so that your application code can execute under its control in an execution device such as an application board, an in-circuit emulator, or a ROM monitor.

PassKey/MQX is a Windows-based, source-code-level cross-debugger designed to allow for multi-task and multi-processor debugging. PassKey/MQX uses knowledge MQX or MQX+M data structures to display and modify task-based information. It can also display and modify kernel resources. PassKey/MQX can be used with serial interfaces, network interfaces, emulators, and board-level monitors.

Using PassKey/MQX's flexible, multi-window interface, you can access and display target registers, monitored variables, and stack levels. The standard graphical interface (GUI) with drag-and-drop, point-and-click, pull-down menus, multiple viewing windows, dialog boxes, push buttons, and so on—in addition to a programmable toolbox that lets you define macros and connect them to your own push buttons—gives you point-and-click access to powerful features such as C source tracing, C expression evaluation, stack tracing, breakpoints (with action lists), session record and playback, and assertions. For more complex requests, such as establishing an action list, a dialog box is available.

- **Single stepping** at the source or machine level, in or over procedure calls, lets you watch your program executing in "stop motion" so that you can observe variables, examine parameter passing, stacks, and structures. Also, you can continue to step until a specified condition is met.
- **Macros** let you replay with a single mouse click definitions that may consist of any combination of PassKey/MQX commands, C expressions, functions, or conditionals. You can use them in command lists of breakpoints or assertions, or have macros call other macros. Best of all, you can program the buttons of the toolbox to execute them.

  Macros can accept multiple parameters. You can even save and load macros from files. The benefits of macros are almost unlimited. They can be used to open playback files, to rename or redefine PassKey/MQX commands, to request evaluation of a complex
expression, to minimize typing for frequently used command sequences, and so on.

- **Assertions**, which are user-specified command lists that execute after every line of source code. You can use them to set up sophisticated error checking mechanisms that will uncover even the most elusive bug.
- **Simulated input/output** (I/O) so that you can debug programs before your actual input and output devices are available. You may enter input from the keyboard or a file, and write output to the screen or a file, and you can use your own I/O routines. You may view the data in a variety of formats, such as hexadecimal and character. You can even have eight active streams at a time.
- **Configurable windows** can show, for example:
  - Most-recently-executed C statements and machine instructions
  - **C expression evaluations** (including PassKey/MQX commands and functions in your source code).
  - **stack tracing** for calling sequence and the values of the parameters passed at each level.
  - **breakpoints** to simulate patching of defective code, or patching in new code, without recompiling. This lets you halt the program in critical places to observe values or execute a list of commands. **Code and data-watch breakpoints** can be set anywhere without limit in the source code.
  - a **continually updated display** of variables and expressions
- **a powerful record and playback facility** for all or part of your debugging session, including commands and, optionally, their output. You can play back files of PassKey/MQX commands, either all at once or selectively, one command at a time. You can edit playback files with any standard text editor to set up standardized debugging tests and save the results for later study.
- Debug **assembly language** programs or assembly language modules linked with your C source.
- **Communicate directly** with the target execution environment (an in-circuit emulator, ROM monitor, or development board) from the PassKey/MQX command window. This gives you access to the
complete range of low-level hardware features offered by your target system.

In addition to the core functionality described above, PassKey/MQX provides easy point and click access to MQX data structures. This gives users the ability to observe system activity at a task level, rather than a source or assembly line level. It is much more efficient to debug multitasking systems initially at a task level, then when necessary, zero in on the details of a particular task by viewing task behavior at a source or machine level. The MQX windows were designed with this methodology in mind.

**Task Summary** provides a snapshot of the status of all MQX tasks created. You can instantly see which task is the current active task and the message queue and error status of all other tasks. From this overview you can “zoom” in on details of a particular task of interest. Via point and click operations you can quickly view the current context of any task including: stack, registers, queue status, error status, and messages. You can even scope to an inactive task’s context and thereby view the task’s code, view it’s local variables or set local breakpoints.

You can also get a higher level view of your application by viewing **Kernel Data** structure elements. Again, with simple point and click operations, you may observe **Ready Queue** status, **Memory Block**, and **Buffer Pool** usage.

PassKey/MQX provides an easy to use and easy to read interface which facilitates observing the behavior of intertask communication and synchronization mechanisms such as **semaphores**, **events**, and **mutexes**.

Users of the **MQX Clock Server** component can observe the status of task level alarms set by their application as well as status of all system time-out queues.

In addition to observing system behavior, users can also monitor system performance if they have included the **MQX Logging component** in their application. PassKey/MQX will let you observe the activity of the Logging component and provide the ability to save logged data to a file for future processing and analysis.
Target Communication

PassKey/MQX can be invoked via the IDE or a host command shell. Communication to the target can take one of a variety of forms. The communications mechanisms presented here can be used with your target hardware or any off-the-shelf board you intend on using as a development vehicle.

BDM

Developers of products implementing a Motorola 683xx family microprocessor can take advantage of the CPU32's Background Debug Mode (BDM). Motorola has designed this family of MPUs with a mechanism for fast communication directly with the MPU core via a special set of pins. Intermetrics has designed a cable and Windows driver to be used with PassKey/MQX. As long as your target provides a compatible connector, you can achieve almost clock speed downloads.

In addition, to fast download speeds, another advantage of BDM is that it requires no target resources. Since PassKey/MQX is communicating directly with the MPU's internal microcode, there is no requirement for a ROM monitor or any memory resources short of what is required by your application.

SmartMON

If you are developing with some other Motorola family of microprocessors, then BDM is not option. However, there are several other options open to you for PassKey/MQX communication with your target. One of these is by putting Intermetrics ROM monitor, SmartMON, on your target. A ROM monitor is supplied for every target the Intermetrics' C Compiler supports. You have two communication options if you choose to use our debug monitor - RS232 or Ethernet. Although neither media will be as fast as BDM, there are several advantages to using SmartMON on your target.

Besides providing a debug environment for PassKey/MQX, SmartMON is a full featured stand alone debugger. You may choose, as many customers do, to incorporate SmartMON into your final product to provide some low level debug capability. If you use the serial version of SmartMON, you can field debug your product simply by connecting a terminal to an available
serial port. Of course should you choose to use the Ethernet version, the possibility opens up for remote testing and debug. Besides the communications port, no additional hardware or software is required for this low level debug capability.

Some of the debug facilities which SmartMON provides include:

- S/W Breakpoints
- Single Stepping
- Display/Change Registers
- Display/Change Memory
- Disassembly
- Trace

PassKey/MQX provides transparent access to the monitor commands, when you feel it is necessary for low level debug.

**MPD**

MPD is a multitasking, multiprocessor debugger that combines the features found in conventional debug monitors with new features that support the debugging of concurrently operating tasks on multiple processors. It is essentially a version of SmartMON that operates as a task under MQX and communicates with the host via a TCP/IP protocol stack.

**ICE**

Intermetrics' expertise is in developing and marketing high quality software products. We are not a hardware manufacturer. We recognize there are several reputable manufacturers of In-Circuit Emulators (ICE) and rather than compete with these companies we choose to work together with them so that we can include each other's products in our respective toolsets. In this way, you, the customer, get the best of both worlds.

At present, some or all of our tools work with ICEs from many manufacturers (such as HP, EST, Nohau, Softaid, Pentica, HMI, Microtek, AMC). As a compiler company we have long standing relationships with these companies and are constantly upgrading our level of support and compatibility. Call us for details if you have interest in a particular emulator.
The Integration & Test Phase

For developers of proprietary hardware, the integration of software and hardware is an evolutionary process that occurs throughout the debug phase. Since the Precise Solution supports most popular In-Circuit Emulators, no additional development hardware or software is needed as you migrate target resources from the emulated environment to the “real word.”

Porting MQX

Chances are you will do most of your code development on “borrowed” hardware. Unless your product uses commercially available hardware (e.g., VME boards), you probably will accomplish a large part of your code development using an off-the-shelf board or a combination of an In-Circuit Emulator and prototype hardware. If you are using commercial hardware, chances are we have ported MQX to your target. If, however, you are developing custom hardware, the task of porting MQX is one you must address. Since most of MQX is written in ANSI C, porting the kernel is a straight-forward process. We supply with each kernel a board support package that you can use as a template for your port. On the other hand, if it would be more cost- or time-effective for you, we can be contracted to supply the board support package for your proprietary hardware. Contact Intermetrics for a current list of supported processors and supported boards.
Performance Analysis

Once you have fully integrated your hardware and software, you will probably want to test the product against your initial performance specification. Since MQX provides a logging component, you now have the ability to track the operation of your multitasking environment and time stamp each of these operations. All of these performance measurements can be saved to a file that can be analyzed via a standard spreadsheet or special-purpose analysis package. Once you are satisfied with your system’s performance, you can remove the logging component or leave it configured for production performance test.
The Production Phase

Often, many of the tools that you use for product development can be passed along to the production test department. This allows the tools to follow the life cycle of the product and become easier to track. It also allows cost savings. The toolset that makes up the Precise Solution offers such benefits.

For low volume testing, In-Circuit Emulators via macro facilities and playback files can provide automated test capabilities. Most ICE manufacturers offer automated test programs or capabilities expressly for this purpose.

As we mentioned earlier, you can design some of the components of the Precise Solution into your product, thereby making production test and post-production maintenance easier. Using these components in this manner can provide an additional project cost savings by negating the need for additional test hardware.

Some Precise Solution components that are useful in this way are:

- SmartMON, an on-board debug monitor with serial I/O
- MPD, a task version of SmartMON designed for use with Ethernet communications
- The kernel logging component, for tracking MQX performance. Leave this component in your production model for task level performance tracing in the field.
Other Considerations

To make your purchasing decision easy, we supply all the features and benefits shown in the extensive checklist below. If you are comparing the Precise/MQX Solution to another company's proposed answer to your real-time programming needs, sharpen your pencil and check the boxes for the features and benefits that they provide. If you are looking at more than one other company, you might want to photocopy this checklist—otherwise you'll need to erase your checkmarks and start over. (But that shouldn't really be a problem, because we don't think you'll check many of these boxes to start with.)

- **Kernel for PCs and UNIX**: our kernel is especially adapted for execution on PCs and UNIX workstations.
- **A compatible software debugger** with the capability to support the debugging of multi-tasking programs (such as our Precise/MPD).
- **Industry-standard, conventional source-level cross debuggers** as provided with your C compiler. (Since the complete source code of MQX is provided, you can recompile with any ANSI C compiler, providing full symbolic debug information to debuggers such as PassKey/MQX.)
- **A concurrent, source-level symbolic debug tool** that includes enhancements specifically designed to aid in the debugging of MQX applications (such as our PassKey/MQX).
- **Reasonable per-project cost.** You can use it on every project, even the low-budget ones. Currently $8500 per package: kernel,
compiler, integrated development environment tools, 3 months of technical support.

- **Royalty free.** No inventory to track
- **Break-the-seal licensing.** No need to get legal involvement. This results in much lower over-all project costs.
- **Source code—fully commented, too!** You have no need to escrow your source or pay escrow costs for long-life projects. You can have complete control, knowledge, and confidence in all the code in your application—just the code that you write yourself. You can solve problems even if they show up at 2:00 a.m. New Year's Day.
- **ANSI C.** Your investment is protected. You can change processors and move the kernel with ease. We can do most ports for you within four weeks, or you can do them yourself. You have all the benefits of having your own real-time kernel without having to go through all the problems of writing it, making it work, documenting it work, making debug tools for it, supporting it, fixing it, maintaining it.
- **Scaleable libraries.** #include only what you need. Modules increase the size from a minimum 12K to a maximum 25K. You can also install your own components
- **Seamless multi-processor migration,** thanks to the architecture of our multi-processor message passing that relies on transparent IPC communications and our embedded I/O components. Your investment is protected if you need more power. We have the best multi-processing support. You don't even need to stay within one processor family.
- **Communications media independent multiprogramming.** You can drop in whichever of our IPC standard, off-the-shelf components, or you can create your own IPC driver.
- **Kernel performance analysis** based on MQX's kernel logging component. You can take the data and analyze it using any technique that you wish. You're not locked into just a fancy display.
- **Windows and UNIX-based IDE** using Codewright and Crisp in Windows and Crisp in UNIX. However, we don't force you to use our
development tools. You're not locked into our solution if you already have tools that you prefer, or if you find some that are (temporarily) better.

- **Windows-based, task-aware debugging.** You can debug any task in your application. If the task is blocked, you can still get information about its data structures.

- **We make all our own I/O components.** We're not aligned with or dependent on third parties, which often is the case other royalty-free, real-time solution vendors who don't have the size or bandwidth to create their own I/O components. We also supply source code.

- **Proven—and pedigreed—technology.** Precise/MQX is based on more than 12 years of proven technology with the Harmony RTOS, which itself is based on Thoth, a portable, minicomputer, real-time OS that was developed in the late 1970s, and upon which many others have relied, e.g., the Waterloo port, the Stanford V kernel, the QNX systems of Dan Dodge and Gordon Bell, and Oryx.

- **Priority inversion protection.** Priority inheritance semaphores solve for real-time kernels the problem that inheritance inversion has not previously been deterministic. This happens when a low-priority task with a semaphore has been pre-empted by a mid-priority task, which itself becomes pre-empted by a higher one. Priority inheritance artificially raises the low-priority, blocked task's priority to meet that of the highest pre-empting task. With this new priority level, the original low-priority task can now run to completion.

- **POSIX compliance** in the scheduler, semaphore, mutex, and event packages. (MQX’s POSIX-compliant mutex uses a binary semaphore rather than a counting semaphore.)

- **Deadlock prevention.** MQX’s watchdog function lets you set an alarm that will wake your task if your function does not run to completion within the time that you set.
Other Considerations

- Prevention and warning of memory leaks. For instance, our alloc and free calls have debug switches to set counters that record their usage.
- Cleaning up of variable size memory on task deletion.
- Debugger can detect memory leaks and corruption.
- Debugger and kernel detect stack overflow.
- Graceful task destruction and resource reclamation. If a task is killed, the system automatically cleans up its resources as well as those of any of its children.
- Solution to memory fragmentation problems. Most real-time kernels, as does MQX, use nearest-neighbor coalescing when they return a memory block to the memory pool. (This means they look to the left and right of where the block will go and coalesce that block with its neighbor—if the neighbor is free.) This is a deterministic algorithm. MQX also uses a non-deterministic but fast insertion sort compiler option to keep the free blocks linked according to their sizes.
- Continual testing of memory. MQX's _testmem function runs as a background process, continually testing memory headers and blocks for overwrites and corruption.
- Fully reentrant kernel allows all non-blocking calls to be made by interrupt service routines, resulting in really fast interrupt activity decoupled from task activity.
For More Information . .

.