MCX11

MicroController eXecutive
for the
Motorola MC68HC11

Version 1.5

by

A.T. Barrett & Associates
11501 Chimney Rock
Houston, Texas 77035
(713)728-9688

April 1990

Support Contact:
Mike Wood
M/S OE319
Motorola, Inc.
6501 William Cannon Drive West
Austin, Texas 78735
(512)891-2717

Contents

SECTION 1

INTRODUCTION

1.1 BACKGROUND AND FEATURES

MCX11, the MicroController eXecutive for the MC68HC11, is an efficient software framework for embedded real-time applications using the Motorola MC68HC11 microcontroller. Motorola, Inc. makes MCX11 available free of charge via the Freeware Bulletin Board System. MCX11 is written in the AS11 assembly language which is also available on the Motorola Freeware BBS (512/891-3733).

MCX11 provides many features which are designed to support real-time applications. These features include:

- Multitasking for up to 126 tasks
- Pre-emptive task scheduling by priority
- Intertask communication and synchronization via semaphores, messages, and queues.
- Support for timed operations
- Fast context switch
- Very small RAM and ROM requirements
- Fifteen Executive Service Request functions

1.2 MCX11 AS A SOFTWARE COMPONENT

MCX11 is furnished in AS11 source form and is downloadable via a modem connection to the Freeware BBS. In its distribution form, it is not executable. It may be assembled separately or with the application programs and any device drivers peculiar to the application. MCX11 may be treated as any other software library. The user need not know how MCX11 performs its functions internally. Rather, the user need only know what functions of MCX11 to use to achieve a desired result. Thus, MCX11 becomes much like a large scale integrated circuit component in the hardware.

Knowledge of what inputs produce what outputs is all that is needed to use the "chip" successfully. Unlike a chip, however, MCX11 users are provided with the source code to the kernel so that they can exercise complete control over their applications. With the source code, users may even wish to extend the functionality of MCX11 by the addition of new services.

This manual is not meant as a tutorial on real-time kernels in general. It is intended to explain the "inputs" and "outputs" of MCX11 as a software component of the user's application. In an effort to assist the user in the successful employment of MCX11, this manual will cover the following subjects:

- basic concepts of MCX11 design
- interfaces to the MCX11 services
- system generation procedure
- assembly procedure for MCX11 and application source files
- procedure for merging MCX11 with the application object files

1.3 TARGET ENVIRONMENT

MCX11 is designed to operate in an embedded processor, that is, one whose functions are fixed within the application. No assumptions are made about the configuration of the target system. It is the responsibility of the user to define the target environment and to insure that all necessary devices have program support. No distinction is made between Single-Chip Operating mode or Expanded Multiplexed Operating mode in the design of MCX11. If there are special requirements in the application for functions related to a certain operating mode, it will be the responsibility of the user to add them.

1.4 DEVELOPMENT ENVIRONMENT

MCX11 development requires a computer compatible with the IBM PC and having at least 256K bytes of RAM, 1 floppy disk, a keyboard, and a video monitor, an RS-232 compatible serial I/O port, a Centronics compatible parallel printer port, and a printer (preferably one which is Epson compatible). A hard disk is very desirable for development work. This configuration is needed to run the AS11 cross assembler and serve as a debug terminal to the target assuming MC68HC11EVM Evaluation Module is being used. Consult the EVM User Manual for specifics on connection tp PC. Recommended software for the development environment besides MS-DOS includes an editor, and the Kermit communications program (public domain).

1.5 MANUAL FORMAT

Section 1 of this manual is simply an overview of MCX11.

Section 2 gives the functional overview of MCX11.

Section 3 details the organization and content of the various MCX11 control and data structures.

Section 4 gives a description of the Executive Service Requests (ESRs) provided by MCX11, subdivided into their classes.

Section 5 gives an alphabetical presentation of all MCX11 Executive Service Requests.

Section 6 deals with the configuration and generation of MCX11 systems.

Section 7 discusses device drivers and interuupt service routines.

Section 8 presents technical information on advanced topics related to the use of MCX11.

Appendices round out the specifics of necessary user knowledge.

SECTION 2

MCX11 FUNCTIONAL OVERVIEW

2.1 WHAT IS MCX11

A real-time executive provides a software framework within which different processes can operate and gain access to various system resources. Real-time systems usually consist of several processes, or tasks, which need to have control of the system resources at varying times due to the occurrence of external events. These tasks are at various times competing for system resources such as memory, execution time, or peripheral devices. They range from being compute bound to I/O bound. Tasks which are I/O or compute bound cannot be allowed to monopolize a system resource if a more important function requires the same resource. There must be a way of interrupting the operation of the task of lesser importance and granting the needed resource to the more important task.

A multitasking real-time executive promotes an orderly transfer of control from one task to another such that efficient usage of the computer's resources is achieved. Orderly transfers require that the executive keep track of the needed resources and the execution state of each task so that they can be granted to each task in a timely manner. The key word there is timely. A real-time system which does not perform a required operation at the correct time has failed. That failure can have consequences which range from the benign to the catastrophic. Response time to a need for executive services and the execution time of such services must be sufficiently fast enough so that no need goes undetected.

One way to achieve timeliness is the assignment of a priority to each task. The priority of a task is then used to determine its place within the sequence of execution of all tasks. Tasks of low priority may have their execution pre-empted by a task of higher priority so that the latter can perform some time critical function.

An event can be any stimulus which requires a reaction from the executive or a task. Examples of an event would include a timer interrupt, an alarm condition, or a keyboard input. Events may originate externally to the processor or internally from within the software. An executive which responds to these events as the stimuli for allocating resources is said to be event driven. If response time to any event occurs within a period of time which can be accurately defined and guaranteed, the executive can be said to be deterministic.

By these definitions, MCX11 is a deterministic, event driven, multitasking, real-time executive.

2.2 TASKS

In MCX11, a task is a program module which exists to perform a defined function or set of functions. A task is independent of other tasks but may establish relationships with other tasks. These relationships may exist in the form of data structures, input/output, or other constructs. A task executes when the MCX11 task dispatcher determines that the resources required by the task are available. Once it begins running, the task has control of all of the system's resources. But as there are other tasks in the system, a running task cannot be allowed to control all of the resources all of the time. Thus, MCX11 employs the concept of multitasking.

2.3 MULTITASKING

Multitasking appears to give the computer the apparent ability to be performing multiple operations concurrently. Obviously, the computer cannot be doing two or more things at once as it is a sequential machine. However, with the functions of the system segregated into different tasks, the effect of concurrency can be achieved. In multitasking, each task once given operating control either runs to

completion, or to a point where it must wait for an event to occur, for a needed resource to become available, or until it is interrupted. Efficient use of the computer can be obtained by using the time a task might otherwise wait for an event to occur to run another task.

This switching from one task to another forms the basis of multitasking. The result is the appearance of several tasks being executed simultaneously.

2.4 PRIORITY AND SCHEDULING

When several tasks can be competing for the resource of execution time, the problem is to determine how to grant it so that each gets access to the system in time to perform its function. The solution is to assign a priority to each task indicative of its relative importance to other tasks in the system. MCX11 uses a fixed priority scheme in which up to 126 tasks may be defined. Tasks which have a need to respond rapidly to events are assigned high priorities. Those which perform functions that are not time critical are assigned lower priorities.

It is the priority of each task that determines where it is to run in the hierarchy of tasks. When a task may run depends on what is happening to the tasks of higher priority. Tasks are granted execution time in a strict descending order of priority. While executing, a task may be interrupted by an event which causes a task of higher priority to be runnable. The lower priority task is placed into a temporary state of suspension and execution control is granted to the higher priority task. Eventually, control is returned to the interrupted task and it is resumed at the point of its interruption. Thus, when any task is given execution control, no higher priority task can be in a runnable state. This is a most important point to remember. When all tasks are in an unrunnable state, control is granted to the null task.

The null task is a do-nothing task which allows the system to run in an idle mode while waiting for an event which will resume or start a higher priority task. The null task is always the lowest priority task in the system and is always runnable. It is an integral part of the kernel.

2.5 INTERTASK COMMUNICATION AND SYNCHRONIZATION

MCX11 provides an environment whereby two or more tasks can communicate with one another. The three major ways in which this is done are through the mechanisms of semaphore signalling, message transmission, and queues. A semaphore is actually a flag which contains information about the state of the associated event. Any event which is used for task synchronization will be associated with a particular semaphore. The occurrence of a specific event can be signalled by manipulating the semaphore associated with that event.

Message transmission involves the logical transfer of data packets from one task to another. These data packets are called "messages". Messages are sent from one task and placed in the "mailbox" of the receiving task in the order of the priorities of the senders. Messages may be of any format recognizable by the sender and receiver and data may be passed in either direction. That is, it is possible for two tasks to alternate the roles of sender and receiver.

A third technique whereby two tasks can communicate and synchronize is via a first-in-first-out (FIFO) queueing mechanism. The queueing techniques used by MCX11 involve the physical transfer (copying) of data packets from one task to another. Task synchronization due to queueing operations is automatically performed by MCX11.

2.6 TIME BASED FUNCTIONS

An MCX11 system is configured with an interval timer using the Real Time Interrupt (RTI) clock as a peripheral device. The timer permits task control on a timed basis. A generalized scheme using

one-shot and cyclic timers in conjunction with semaphores is provided. MCX11 efficiently manages multiple timers using an ordered linked list of pending timer events. A timer for an event is inserted into the linked list in accordance with its duration. A differential technique is employed so that the timer with the shortest time to expiration is at the head of the list. Timed events may even be co-terminous. Directives for scheduling and cancelling timed events are an integral part of the executive.

2.7 INTERRUPT SERVICE

Support for a generalized interrupt service scheme is provided within the MCX11 kernel. The Interrupt Service Routine (ISR) code, however, is provided by the User. The rules for writing interrupt service routines that function with MCX11 are quite simple and require only minor housekeeping chores. MCX11 also provides the common Interrupt Service Exit function. This logic is used by all interrupts to determine if a context switch is in order as the result of the interrupt. If no task switch is required, the interrupted task is resumed at the point of the interrupt. If, however, a task switch is in order, the interrupted task's context is saved, the new task's context is loaded and control is given to it.

SECTION 3

MCX11 CONTROL & DATA STRUCTURES

3.0 INTRODUCTION

In order to understand how MCX11 works and how to build real-time application systems around it, it is necessary to understand how its various control and data structures work. This section describes these structures and their interrelationships. The descriptions will include:

- tasks
- semaphores
- messages and mailboxes
- queues
- timers

The MCX11 executive services which deal with these classes of control and data structures will be presented in a subsequent section.

3.1 TASKS

In a real-time application using MCX11, the functions of the system are assigned to various tasks. MCX11 supports up to 126 tasks in a single system. The nature of each task is, of course, application dependent and left to the imagination of the system designer. However, there are attributes which all tasks share. These include:

- a task identifier
- a priority
- a task state table
- a stack
- a mailbox
- an entry point

3.1.1 Task Identifier & Priority

Each task is identified by a numerical identifier which is a number from 1 to 126. The task identifier, or number, serves as a reference during executive operations. Task 1 is executed before task 2 which is executed prior to task 3 and so on. Thus, the task number is inversely related to the task's execution priority. The lower the task number, the higher the priority. MCX11 uses this fixed task numbering scheme because it is low in overhead and quite adequate for most applications. The task number permits a direct access to the task's state table without the need for linked lists or other commonly used technique.

3.1.2 Task State Table

The state table is commonly referred to as a Task Control Block (TCB). A TCB in MCX11 is contained in two separate sections in order to minimize the use of system RAM. Only those data about the execution state of the task are kept in RAM. Static information about the task is kept in ROM. Both sets of this TCB information are kept in arrays which allow direct access to the data based on the task number. This

makes for very quick access without the commonly wasted time of searching a linked list.

The RAM portion of the TCB contains only volatile data about the task and includes the following:

- the execution state

- the stack pointer

- the first link in the task's mailbox

Whenever the MCX11 dispatcher is looking for a task to execute, it examines the execution state variable to see if the task is runnable or not. The execution state is contained in a single byte and a value of $00 indicates that the task is runnable. Any other content in the task state and the task is blocked from running.

The stack pointer contains the address of the current top-of-stack.

The first link in the task's mailbox contains the address of the highest priority message waiting to be received by the task. If no message is waiting, the pointer contains zero.

The ROM part of the TCB contains:

- the initial execution state

- the entry point

- the base address of the stack

- the address of the RAM part of the TCB

The initial execution state is specified by the system designer at the time of system configuration. This becomes the initial content of the execution state. By specifying a value of zero, the task will be executed immediately after initialization in accordance with its priority. Specification of a non-zero value will prevent the task from automatically starting execution after initialization.

The entry point is the address where the task is to begin execution. The base address of the stack identifies that location in RAM where the task's stack begins.

The address of the RAM part of the TCB table is included for completeness and is sometimes used to save time in setting up the address of the RAM TCB area.

3.1.3 Stack

Each task must have a stack on which the context of the task is saved during those periods when the task is not actually executing. The base address of the stack is stored in the ROM portion of the TCB. The stack extends from the base address downward in memory, i.e. towards address $0000, for a number of bytes specified by the given stack size for the task. The size of each task's stack is dependent on many things such as the maximum depth of nested subroutines calls, the maximum amount of working space needed for temporary variables, and the size of any stack frames used by the task. At minimum, the size of the stack must allow for the storage of a complete context which requires nine (9) bytes. If XIRQ is to be used in the design, an extra nine (9) bytes must be allocated to each task's stack due to the possibility of stacking an IRQ followed by an XIRQ. This is possible because the XIRQ cannot be masked and can interrupt an IRQ interrupt service routine at any point.

In addition to the stacks needed by the tasks, there is also the need for a stack for MCX11. This system stack must have sufficient space to handle the maximum number of interrupts possible at any given time, less one interrupt. Each interrupt requires nine bytes. MCX11 functions have a worst case stack

requirement of seven (7) bytes not including those bytes required for the system stack.

The sum of all of the stack requirements should not be allowed to exceed available RAM or there will be trouble.

3.2 SEMAPHORES

Semaphores (event flags) are used to synchronize two tasks to the occurrence of an asynchronous event. One task may need to wait for the other to reach a certain point before it can continue. Input/output operations are an example of this type of synchronization. For instance, when an input from an external device is desired by a task, it must wait for the input event to occur. When the input operation is completed, the device driver signals that the event has occurred which indicates the data is available. The signalling process causes the waiting task to resume, presumably to process the input data. The synchronization of the task with the event is done with the use of a semaphore.

There are several forms that a semaphore may take in the design of a real-time kernel. MCX11 uses a basic semaphore construct that is known to handle most events and is low in overhead. A semaphore is referenced by a numerical identifier limited to a value from 1-255. A semaphore identifier of zero has a special meaning. There are three different classes of semaphores used by MCX11 as follows:

- Named semaphores

- Task semaphores

- Queue semaphores

Named semaphores are those normally referenced by the user in the application code. Task and queue semaphores are used by MCX11 internally for the purpose of orthogonality with the semaphore service functions. Task semaphores are always referenced by the identifier of zero (0) which is translated by MCX11 into the proper number for the task semaphore. Queue semaphores should never be referenced by the user lest undesirable results should occur.

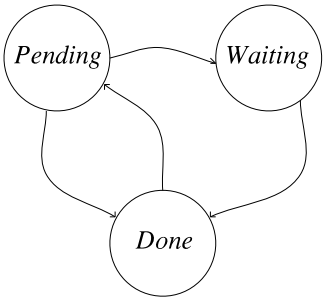
An MCX11 semaphore is a single byte (8 bits) which contains a value representing one of the three possible states in which it can exist. These states are:

- PENDING

- WAITING

- DONE

A PENDING state indicates that the event associated with the semaphore has not yet occurred and is therefore pending. The WAITING state shows that not only has the event not yet occurred but a task is waiting for it to happen. The DONE state tells that the event has occurred. MCX11 semaphores have a very strict state transition protocol which is automatically managed by MCX11. The permissible state changes are shown in the figure below:



If a task attempts to wait on a semaphore which is in the DONE state, the wait does not occur and the task is immediately resumed (since the desired event has already happened) and the semaphore state is changed back to PENDING. If a task attempts to wait on a semaphore in the PENDING state, the state of the semaphore is changed to WAITING and the task's execution is suspended until the event occurs. An attempt to wait on a semaphore which is already in the WAITING state can cause unpredictable results and should not be attempted. It should be considered a design error.

If an attempt is made to signal a semaphore which is in a PENDING state, the semaphore is placed into the DONE state. As this action does not concern any task, no further action is taken. However, if a signal is sent to a semaphore in the WAITING state, the semaphore transitions to a DONE state and the task waiting on the event is made runnable again. The semaphore is then automatically set back to a PENDING state by MCX11.

3.3 MESSAGES AND MAILBOXES

Messages are one of the means by which data is transmitted from a sender to a receiver task. Every task running under MCX11 is capable of being both a message sender and receiver. Each task is assigned a "mailbox" into which are placed all messages sent to it. When a sender task sends a message, it is put into the mailbox of the receiver task. If two or more tasks are sending messages to a single receiver, their messages are put into the mailbox in strict sequence of their priorities, highest priority first.

The content of the message is not actually moved from the sender to the receiver when it is sent. Instead, messages are linked into a threaded list according to the priorities of their senders. When the receiver requests receipt of the next message, its address is returned. This technique guarantees the receiver that the highest priority message is received first. It is possible, however, to temporarily suspend this order by requesting only those messages from a particular sender. This is useful when it is desirable not to mix messages on a shared resource, for example, a printer.

The mailbox of a task is a threaded (linked) list of all messages sent to but not yet received by that task. The mailbox consists of a pointer (link) to the message sent by the highest priority sender. That message is linked to the next highest priority message and it in turn is linked to the third highest priority message, and so on until the end of the thread. The last message in a task's mailbox will contain a NULL (value of zero) link.

Messages may be sent with or without waiting for a response from the receiver. When a task wants to send a message and wait until the receiver has processed the message, it will include a semaphore number as one of the arguments in the MCX11 send message directive. That semaphore is then associated with the event of completed processing of the message. Once the message is linked into the mailbox of the receiver task, the sender is put into a WAIT state on the message processing complete event.

The receiver removes the message from the mailbox and processes it according to the content of the message body. When the receiver no longer needs the data in the body of the message, it signals the semaphore associated with the completion of message processing making the sender task runnable again and, thereby, allowing it to continue its operation.

It should be remembered that the body of the message can also be used by the receiver to send a response back to the sender. This is a very efficient way of passing data bi-directionally between two tasks with little overhead.

If the sending task does not wish to wait on the action of the receiver or if there is no response required, a message can be sent without waiting for completion of processing. This makes it possible for a sender to send multiple messages to a receiver, or, simply do something else while the receiver processes the message. However, even though a message is sent without waiting for the response, it is still possible at

a later time for the sender to wait for the completion of processing event.

If the receiver has completed the use of the message by the time the sender waits for that event, the sending task continues operation without interruption. If the receiver has not yet completed processing of the message, the sender must wait for the event to occur. When it does occur, the sender's operation is resumed.

A message is a four-element structure set up in a contiguous area of RAM. The first three elements of the message are used by MCX11 to control the transmission. The user need only set up the content of the message body. The format of a MCX11 message is as follows:

- Mailbox link (supplied by MCX11)

- Task number of sender (supplied by MCX11)

- Message semaphore number (optional)

- Message body (supplied by User)

The first element in the message provides the link to the next message, if any, in the mailbox.

The second element is the task number of the sender. This field is included so that the receiver task can selectively receive only those messages from a specified task.

The third element of the message contains an optional message semaphore number. Typically, when a sending task wants to send a message and wait until the receiver has processed the message, it will include a non-zero semaphore number as one of the calling arguments for the MCX11 directive. The sending task will then suspend execution until the receiver task completes the processing of the message content. When the receiver no longer needs the data in the body of the message, it signals the message semaphore to notify the sender that it may continue processing.

The last element, the body, of the message may containwhatever data is required by the receiver task and in whatever format. It may be fixed or variable in length as required.

3.4 QUEUES

MCX11 supports first-in-first-out (FIFO) queues having single or multiple bytes per entry. The queues can support multiple producer/single consumer service but are usually employed in only a single producer/single consumer mode. Queues are referenced via a numerical identifier which is unique to each queue. Queue numbers are assigned in the system generation procedure. MCX11 queues differ from messages in that the actual entry data rather than its address is entered into or removed from the queue. Another difference is that the queue entries represent the chronological order of processing. Also, the priorities of the sender or receiver are not considered as with messages. Queues are usually used to handle data such as character stream input/output or other data buffering. Queues require more overhead as they move the data from the source to the queue or vice versa but they also provide automatic synchronization between the queue's producer and its consumer. It is recommended that they not be used where high throughput rates are required.

A queue consists of two parts: the queue header and the queue body. The queue header contains that information needed by the MCX11 internal routines to enable them to move data into and out of the queues properly. The organization of the queue header consists of the following elements:

- Width, queue entry size (in bytes)

- Depth, maximum length (in entries)

- Current size of the queue (in entries)

- Put index (in bytes; MCX11 internal use)

- Address of queue body

- Queue not empty semaphore number

- Active semaphore number

The first three elements are defined during the system generation process. The Current Size and the Put Index fields are initialized to zero (empty). Both of these fields are maintained internally by the MCX11 queue handling routines. They should never be manipulated by a user task.

Semaphores are used to signal internal events such as queue-not-empty or queue-not-full. The semaphores for these events are allocated as a consecutive pair during the system initialization process. The queue-not-full semaphore number is equal to the number of the queue-not-empty semaphore plus one. A queue can have a semaphore active only for those two mutually exclusive events. Consequently only the queue-not-empty semaphore number requires definition in the queue header.

The active semaphore number field in the queue header is used to contain the number of the semaphore associated with one of the two possible events. If the queue is empty and a task attempts to remove an entry, the active semaphore field in the queue header is set to contain the number of the queue-not-empty semaphore. Conversely, if a task attempts to put an entry into a queue which is already full, the active semaphore field is set to contain the number of the queue-not-full semaphore.

By use of these semaphores, it is possible to send data between tasks asynchronously but without the worry of over-running the queue.

3.5 TIMERS

In the descriptions of time based functions to follow in the succeeding sections, it will be important to understand the conventions used by MCX11. All timers in MCX11 are 16-bits long. The two time units used internally by MCX11 are ticks and tocks. A tick gives the amount of time between clock generated system interrupts, or equivalently, the period between clock interrupt service requests. A tock is an integral number of ticks and is the time unit used by all of the time dependent functions in MCX11. The tick frequency of the RTI clock is set during the system initialization process but was determined, as was the number of ticks per tock, by the system designer during MCX11 system configuration.

As an example, suppose the hardware clock is configured to cause interrupts every 25 milliseconds and that 0.25 seconds (250 milliseconds) is the time most convenient interval for timing events in the application. In order to determine the number of ticks per tock, simply divide the desired duration of the tock (250 ms) by the duration of a tick (25 ms). Thus, in this case, 1 tock = 10 ticks.

Timer values are defined as the number of clock tocks required to form the needed amount of time. If a timer value is positive, it is used only once (one-shot timer). However, if the value is negative, it is used as a cyclic timer having a period equal to the 2's complement value. For example, if a clock tock equals 250 milliseconds, a timer value of 20 would produce a once-only period of 5 seconds. If the timer were -20, the period is still 5 seconds but the timer is cyclic.

All timed event operations and data structures are handled by MCX11. A task should not attempt to manipulate any of the timer related control or data structures.

SECTION 4

EXECUTIVE SERVICE REQUESTS (ESRS)

4.1 PURPOSE

Executive Service Requests (ESRs) are the functions that a real-time kernel performs and serve to give it its flavor. This section will describe the classes of the MCX11 directives. The directives are divided into five classes as follows:

- Semaphores
- Messages
- Queues
- Task control
- Time based

A brief functional description of each ESR within the class is given along with their respective calling arguments. The calling arguments needed for each ESR appear below in parentheses. Where two or more arguments are required, they are separated by commas. These are not the actual calling sequences but are for purposes of explanation only. Actual calling sequences are given in Section 5.

In the paragraphs below, the ESR function name is enclosed within two periods. This will be the convention used for MCX11 to denote an ESR. The following abbreviations are used below for the arguments:

- SEMA Semaphore number
- TASK Task number
- QUEUE Queue number
- TOCK Timer value
- PNTR Address pointer or data value
- MSG Message address

4.2 SEMAPHORE ESRS

A complete set of directives for managing semaphores is provided by MCX11.

.wait. (SEMA) wait for a specific sema to be signalled

.signal. (SEMA) signal semaphore

.pend. (SEMA) reset semaphore state to PENDing

4.3 MESSAGE ESRS

The message directives provide a means of transferring large amounts of data between tasks with minimal overhead since only pointers (addresses) are passed. Message receipt acknowledgement is also provided for task synchronization. The format of a MCX11 message and function prototypes are noted.

.send. (MSG,TASK) send message to a task

.sendw. (MSG,TASK,SEMA) send & wait acknowledgement

.receive. (TASK) receive message

4.4 QUEUE ESRS

Queue directives provide a means of passing multiple byte packets of information between tasks with automatic task synchronization of queue full and empty conditions.

.dequeue. (QUEUE,PNTR) remove entry from queue

.enqueue. (QUEUE,PNTR) insert entry into queue

4.5 TASK CONTROL ESRS

The task control directives provided by MCX11 allow for low level complete control of tasks and their respective interactions.

.resume. (TASK) resume specific task

.suspend. (TASK) suspend specific task

.terminate. (TASK) terminate specific task

.execute. (TASK) execute task from starting point

4.6 TIME BASED ESRS

The time based directives provide for the synchronization of tasks with timed events. In addition, a generalized time based semaphore scheme for more advanced time based requirements is provided.

.delay. (TASK,SEMA,TOCK) delay task

.timer. (TASK,SEMA,TOCK) schedule timer for task/sema

.purge. (TASK,SEMA,PNTR) purge timer for task/sema

SECTION 5

ALPHABETICAL LISTING OF ESRs

5.0 INTRODUCTION

In the pages to follow, each ESR will be shown in alphabetical order. Each ESR will be given in a standard format:

Name - brief functional description

CLASS
One of the five ESR classes of which it is a member.

SYNOPSIS
The prototype declaration including argument typing. The decimal value of the function code is enclosed in brackets [xx].

DESCRIPTION
A description of what the ESR does, data types used, etc.

CALLING SEQUENCE
The formal AS11 calling sequence for the ESR. The following notations will be used:
ACCA = Accumulator A, 8 bits
ACCB = Accumulator B, 8 bits
IX = Index register X, 16 bits

RETURN VALUE
A description of the return values from the ESR.

EXAMPLE
One or more typical ESR uses. The examples assume the syntax of AS11 cross assembler.

SEE ALSO
List of related ESRs that could be examined in conjunction with the current ESR.

5.1 .delay. - DELAY A TASK FOR A PERIOD OF TIME

CLASS
Timer

SYNOPSIS
.delay. (TASK,SEMA,TOCK) [13]

DESCRIPTION

The delay directive places the specified task into a Wait state for the period defined by the TOCK value. TASK is a task number in the range of 1-126. Or, TASK may be 0 if the reference is to SELF. Thus, a task need not know its own task number to schedule a delay for itself. SEMA is the semaphore number associated with the expiration of the timer. It may be a named semaphore if its number > 0. Or, it may be the task semaphore of the calling task if it is specified as 0. TOCK is a 16-bit timer. If TOCK is > 0, the timer is a one-shot. If TOCK < 0, the timer is cyclic. Caution should be exercised when scheduling delays for other tasks.

CALLING SEQUENCE
ACCA = Task number (0 if SELF)
ACCB = Semaphore number (0 if Task's semaphore)
IX = Number of clock tocks to delay
swi
FCB .delay.
return

RETURN VALUE
None. Registers unchanged.

EXAMPLE

Assume that the RTI is used as the time base using a 2**23 crystal and the clock TICK is defined to be 15.625 msec. Further assume that there are 8 TICKS per TOCK so that a TOCK is 125 msec. If a 2.5 sec delay is desired for the calling task using a named semaphore, TIMEOUT, the request would appear as:

```
TOCK        equ     8
TOCKTIME    equ     1000/8      TOCKTIME = 125 msec
TIMEOUT     equ     3           Named semaphore #3
.
:
clra                            Task is self (ACCA=0)
ldab        #TIMEOUT            Semaphore = TIMEOUT
ldx         #2500/TOCKTIME      Timer = 20 TOCKS
swi
FCB         .delay.
. Point of continuation
```

The requesting task would be put into a WAIT state on semaphore TIMEOUT while the 2.5 sec clock runs down. When it expires, TIMEOUT would be signalled and the task would continue at the instruction following the FCB declaration.

SEE ALSO
.timer., .purge.

5.2 .dequeue. - GET AN ENTRY FROM A FIFO QUEUE

CLASS
Queue

SYNOPSIS
.dequeue. (QUEUE,PNTR) [7]

DESCRIPTION

Dequeue is used to get an entry from a FIFO queue. The number of bytes in the entry is determined by the Width entry in the Queue Header. If the queue is empty, the calling task is placed into an EVENT WAIT state on the queue-not-empty semaphore. If the queue is not empty, the oldest entry in the queue is removed and returned to the calling task. If the size of the queue entry is greater than two bytes, the returned data is in the RAM area beginning at the address specified by PNTR. If the queue entry is one or two bytes, PNTR is ignored and the entry is returned in one or two registers.

CALLING SEQUENCE
ACCA= Queue number
IX = Destination address of queue entry if size > 2
swi
FCB .dequeue.
return

RETURN VALUE
If queue entry size = 1, ACCA= queue entry
If queue entry size = 2, ACCA = byte 1 of queue entry,
and ACCB = byte 2 of queue entry
Other registers unchanged

EXAMPLE
Example #1:

There exists a FIFO queue, DBLQ, containing two character entries. The queue currently contains four entries as follows:

ab
DE
Mc
xY

A task is programmed to remove the oldest entry in the queue and return it for subsequent processing. The entry is dequeued by:

```
DBLQ        equ         4           Queue #4
.
:
ldaa        #DBLQ                   ACCA = queue # 4
swi
FCB         .dequeue.
. Point of continuation
```

At the instruction following the FCB declaration, ACCA and ACCB will contain the two characters, ab. The queue will now contain the three entries:

DE

Mc
xY

Example #2:

Another queue exists, QUADQ, which contains four byte entries and is currently empty. The request to get an entry from the queue would appear as:

```
QUADQ equ 5 Queue #5
.
:
ldaa        #QUADQ          ACCA = queue # 5
ldx         #destn          IX = destination address
swi
FCB         .dequeue.
. Point of continuation
:
destn       ZMB     4       Where the entry is placed
:
```

Since there are no entries available to dequeue, MCX11 puts the task into a WAIT state on the NOT_EMPTY semaphore associated with queue #5. The task will remain in that state until another task puts a 4-byte entry into QUADQ. When that happens, the waiting task is put into the RUN state. The entry is removed and is stored at the specified destination address, destn, and the task continues at the instruction following the FCB declaration.

SEE ALSO
.enqueue.

5.3 .enqueue. - INSERT DATA INTO FIFO QUEUE

CLASS
Queue

SYNOPSIS
.enqueue. (QUEUE,PNTR) [8]

DESCRIPTION
Enqueue inserts an entry into a FIFO queue. If the queue is full, the calling task is set to a WAIT state on the queue NOT FULL condition. When the queue is not full, the entry is inserted into the queue. If the width of the queue entry is greater than 2 bytes, the data to be put into the queue is found at the address specified by the contents of register IX. If the width is 1 byte, the actual data to be enqueued is found in bits 8-15 of IX. If the width is equal to 2 bytes, then byte 1 of the entry is found in bits 8-15 of IX, and byte 2 is in bits 0-7 of IX.

CALLING SEQUENCE
ACCA= Queue number
IX = Source address of entry to be enqueued if size > 2
Or, IX8-15 = byte to be enqueued if size = 1
Or, IX8-15 = byte 1 and IX0-7 = byte 2 if size = 2
swi
FCB .enqueue.
return

RETURN VALUE
None. Registers unchanged.

EXAMPLE
Example #1:
Using the 2-byte queue, DBLQ, defined above, a task wants to put an entry into it. Assume that the queue contents are the three entries:

DE
Mc
xY

The queue entry size is two bytes which permits the entry to be passed to the queue routine in ACCA and ACCB. The request to put a two-byte entry, LW, into DBLQ is:

```
ldaa    #DBLQ       ACCA = queue # 4
ldx     #"LW"       IX = "LW"
swi
FCB     .enqueue.
. Point of continuation
```

Since the queue has content, no task is waiting for an entry to dequeue; therefore, the entry is put into the queue following the "xY" entry and the task continues at the instruction following the FCB. The new content of DBLQ is:

DE
Mc
xY
LW

Example #2:
Using the queue, QUADQ, previously defined but now containing three entries:
abcd
EFGH
iJkL
A request to put a new entry into the queue, MnOp, is given by:

```
ldaa        #QUADQ          ACCA = queue # 5
ldx         #srcadr         IX = source address
swi
FCB         .enqueue.
. Point of continuation
:
srcadr      FCC     "MnOp"
:
```

Since the entry size is four bytes, IX must contain the address of the entry to be enqueued. After continuing at the instruction following the FCB declaration, QUADQ contains:
abcd EFGH IjKl MnOp

SEE ALSO
.dequeue.

5.4 .execute. - EXECUTE A TASK

CLASS
Task control

SYNOPSIS
.execute. (TASK) [12]

DESCRIPTION
The execute task directive is used to start or restart a task from its beginning. The program counter (PC) and stack pointer (SP) are initialized to their starting values and the specified task is made runnable. If the new task is of higher priority than the current task, a context switch is performed and the new task runs, else control is returned to the caller. The task being started cannot be SELF (task number argument = 0).

CALLING SEQUENCE
ACCA= Task number
swi
FCB .execute.
return

RETURN VALUE
None. Registers unchanged.

EXAMPLE
The current task wishes to cause task number 6 to begin execution at its starting point. Task number 6 is currently IDLE. The request appears as:

```
TASK6       equ       6       Task #6
.
:
ldaa        #TASK6              ACCA = task # 6
swi
FCB         .execute.
. Point of continuation
```

If TASK6 is of higher priority, then the current task is interrupted, but still runnable, and TASK6 is made the current task. The task which scheduled TASK6 is not given control again until all other higher priority tasks, including TASK6, are not runnable. When it does continue, it will do so following the FCB declaration.

If TASK6 is of lower priority that the current task, no context switch occurs and the current task continues at the instruction following the FCB declaration.

SEE ALSO
.terminate.

5.5 .pend. - FORCE A SEMAPHORE TO A PENDING STATE

CLASS
Semaphore

SYNOPSIS
.pend. (SEMA) [3]

DESCRIPTION
To understand the use of the pend directive, recall that the WAIT directive will return control to the calling task immediately if the task requests to wait on a semaphore found to already be in a DONE state. In this sense, semaphores keep a history of their associated events, even if no task is waiting at the time of the signal. If a task needs to go unconditionally into a Event Wait state on a semaphore, a call should first be made to PEND followed closely by a WAIT directive.

CALLING SEQUENCE
ACCB = Semaphore number
swi
FCB .pend.
return

RETURN VALUE
None. Registers unchanged.

EXAMPLE
The current task is going to make use of a semaphore and wants to be certain of its state prior to use. It forces the semaphore into a PENDing condition by the following request:

```
NEWSEMA     equ         14          Semaphore #14
.
:
ldab        #NEWSEMA                ACCB = semaphore # 14
swi
FCB         .pend.
. Point of continuation
```

SEE ALSO
.wait.

5.6 .purge. - PURGE A TASK'S TIMERS

CLASS
Timer

SYNOPSIS
.purge. (TASK,SEMA,PNTR) [15]

DESCRIPTION

The purge timer directive is used to remove all active timers for the specified task or semaphore number. A specification of a zero semaphore number will purge all semaphore timers for the given task. When the purge is completed, the current task is resumed.

CALLING SEQUENCE
ACCA = Task number (0 if SELF)
ACCB = Semaphore number (0 if task semaphore)
IX0-7 = Switch: 0 = purge all timers for given task
>0 = purge time for given task & sema
swi
FCB .purge.
return

RETURN VALUE
None. Registers unchanged.

EXAMPLE

The most common use of this ESR is for the current task to get rid of one or all of its active timers. Since timers are associated with semaphores, the combination of task number and semaphore number is sufficient to identify a timer uniquely. The request to purge the current task's timer associated with the task semaphore appears as:

```
clra                        ACCA = 0 means SELF
clrb                        ACCB = 0 means task sema
ldx         #1              Only purge one timer
swi
FCB         .purge.
. Point of continuation
```

If all timers for a task are to be purged, the request appears as:

```
clra                        ACCA = 0 means SELF
ldx         #0              Purge all timers
swi
FCB         .purge.
. Point of continuation
```

SEE ALSO
.timer., .terminate.

5.7 .receive. - RECEIVE A MESSAGE

CLASS
Message

SYNOPSIS
.receive. (TASK) [6]

DESCRIPTION
The receive message directive is used to fetch messages from a calling task's mailbox. Because the messages are placed in the mailbox in the order of the sender's priority, they are processed in the same sequence. If the receiving task specifies a non-zero task number, the first message from that task will be returned. With a zero task number, the first message in the mailbox is returned. If there are no messages in the mailbox, the task's context is saved, and the task is put into a Message WAIT state. When the next message is sent to the task it will be resumed and the attempt to receive a message will be reissued.

CALLING SEQUENCE
ACCA = Task number if messages from specific task are wanted. Or, ACCA = 0 if highest priority message is wanted.

```
swi
FCB.receive.
return
```

RETURN VALUE
Register IX contains the address of the received message. Other registers unchanged.

EXAMPLE
Example #1:
The current task is a server which operates only when another task sends it a message instructing it to perform some function. The server task receives messages by making a request such as:

```
clra                                    0 = next message
swi
FCB             .receive.
. Point of continuation
```

The server task will not continue unless there is a message waiting in its mailbox. It will be put into a WAIT state until such time as another task sends it a message. No waiting occurs if a message is already in the mailbox at the time of the request. When the message is available, its address is returned in the IX register.

Example #2:

If the server wants to receive only those messages from a specific task, and the task's number is to be found in a variable named 'attach', the request is:

```
ldaa        attach          variable contains task #
swi
FCB         .receive.
. Point of continuation
:
attach      ZMB     1
```

The server task will now only listen for messages from the task whose number is in the variable 'attach'. All other messages will remain in the mailbox of the server. The server will continue only when it has received a message from the specific task.

SEE ALSO

.send., .sendw.

5.8 .resume. - RESUME A TASK

CLASS
Task control

SYNOPSIS
.resume. (TASK) [9]

DESCRIPTION
The resume directive clears the suspended state of a task caused by a suspend directive. If the resumed task is made runnable and is higher priority, a task switch is performed. Otherwise, control is returned to the calling task.

CALLING SEQUENCE
ACCA = Task number
swi
FCB .resume.
return

RETURN VALUE
None. Registers unchanged.

EXAMPLE
A task which has been previously suspended may be resumed from the point of suspension only by a .resume. request from another task. The request appears as:

```
TASK6           equ         6           Task #6
.
:
ldaa            #TASK6                  ACCA = task # 6
swi
FCB             .resume.
. Point of continuation
```

Upon resuming the given task, a context switch may occur if the resumed task is of higher priority than the task making the request. If not, no context switch occurs. The requesting task continues at the instruction following the FCB.

SEE ALSO
.suspend.

5.9 .send. - SEND A MESSAGE TO A TASK

CLASS
Message

SYNOPSIS
.send. (MSG,TASK) [4]

DESCRIPTION
The send message directive inserts a message into the target task's mailbox in the order of the sending task's priority. If the receiving task is waiting to receive a message and is of higher priority than the sender, a task switch is performed. If the receiving task is of lower priority and waiting on a message, then the receiving task is marked runnable but control is returned on a message, the message is simply inserted to his mailbox for later receipt.

CALLING SEQUENCE
ACCA = Task number of the receiver
ACCB = Semaphore number (task semaphore number = 0)
IX = Message address
swi
FCB .send.
return

RETURN VALUE
None. ACCB will contain the number of the task semaphore if ACCB was = 0 at time of swi. All other registers unchanged.

EXAMPLE
A task wishing to send a message to a receiver task can do so by the following code:

```
RCVRTSK     equ     12      Receiving task = task #12
MSGSEMA     equ     7       Use sema #7 for Message
.
:
ldaa        #RCVRTSK        ACCA = receiver task #
ldab        #MSGSEMA        ACCB = Message semaphore
ldx         #msgadr         IX = Address of message
swi
FCB .send.
. Point of continuation
:
msgadr      ZMB     4       Message overhead
FCC         "THIS IS THE MESSAGE BODY"
```

The message located at 'msgadr' will be put into the mailbox of RCVRTSK. Processing in the sending task will resume at the instruction following the FCB declaration.

SEE ALSO
.receive., .sendw.

5.10 .sendw. - SEND A MESSAGE AND WAIT

CLASS
Message

SYNOPSIS
.sendw. (MSG,TASK,SEMA) [5]

DESCRIPTION
The send message and wait directive is similar to the send message directive described above, except that the sending task is always put into a Event Wait state on the message semaphore. This allows the receiving task to signal the sender when the message processing is complete and/or that there is data returned in the message area.

CALLING SEQUENCE
ACCA = Task number of the receiver
ACCB = Semaphore number (task semaphore number = 0)
IX = Message address
swi
FCB .sendw.
return

RETURN VALUE
None. ACCB will contain the number of the task semaphore if ACCB was = 0 at time of swi. All other registers unchanged.

EXAMPLE
The example is the same as for .send. except that the ESR request will be .sendw. Another difference is the WAIT state into which the sending task is placed until the receiving task signals the message semaphore. Until that signal is received, the sending task proceeds no further. When the receiver signals the message semaphore, the sending task will continue at the instruction following the FCB.

SEE ALSO
.send., .receive.

5.11 .signal. - SIGNAL A SEMAPHORE

CLASS
Semaphore

SYNOPSIS
.signal. (SEMA) [2]

DESCRIPTION

The signal semaphore directive sets the state of a specified semaphore to DONE. If the semaphore is currently in a WAIT state, the Event Wait state of the waiting task is removed, and the semaphore is set PENDing. If the waiting task becomes runnable and is of higher priority than the signalling task, the context of the current task is saved, and a context switch is performed.

If the state of the semaphore was either PENDing or DONE, the semaphore is placed in the DONE state and the current task is resumed following the signal ESR.

CALLING SEQUENCE
ACCB = Semaphore number
swi
FCB .signal.
return

RETURN VALUE
None. Registers unchanged.

EXAMPLE

Assume that the current task wishes to signal the occurrence of a particular event. The event is associated with a named semaphore, EVNTSEMA. The task signals the semaphore by the following procedure:

```
EVNTSEMA    equ         15          Named semaphore #15
.
:
ldab        #EVNTSEMA               ACCB = semaphore # 15
swi
FCB         .signal.
. Point of continuation
```

Depending on the state of semaphore EVNTSEMA, the signal may or may not cause a context switch. The signalling task will continue processing at the instruction following the FCB declaration.

SEE ALSO
.pend., .wait.

5.12 .suspend. - SUSPEND A TASK

CLASS
Task control

SYNOPSIS
.suspend. (TASK) [10]

DESCRIPTION
The suspend directive causes the specified task to be placed into a suspended state. The suspended state will remain in force until it is removed by a resume, execute, or reset directive. A task may suspend itself. Task 0 indicates the SELF task.

CALLING SEQUENCE
ACCA = Task number (this may be SELF [0])
swi
FCB .suspend.
return

RETURN VALUE
None. Registers unchanged.

EXAMPLE
The current task wants to suspend itself until some indeterminate point in the future. To do so, it makes the following request:

```
clra                                    ACCA = 0 means SELF
swi
FCB             .suspend.
. Point of continuation
```

The task remains in a suspended state until resumed by a .resume. ESR issued from another task. At that time, the resumed task will continue at the instruction following the FCB declaration.

SEE ALSO
.resume.

5.13 .terminate. - TERMINATE A TASK

CLASS
Task control

SYNOPSIS
.terminate. (TASK) [11]

DESCRIPTION
The terminate directive is used to terminate a task's operation. During termination, the task's state is cleared and the task state is set to Terminated. The timer queue is searched and all active timers for the task are purged. Also, all messages pending for the task are removed. A task number of 0 indicates self termination. In all cases, the highest priority task will execute next.

CALLING SEQUENCE
ACCA = Task number (may be = 0 if terminating SELF)
swi
.terminate.
return

RETURN VALUE
None. Registers unchanged. Returns only if not terminating SELF.

EXAMPLE
The current task commits suicide by terminating itself with the following procedure:

```
clra                                    ACCA = 0 means SELF
swi
FCB             .terminate.
```

There is no point of continuation following a suicide. If another task is terminated by the current task, the current task will continue processing at the instruction following the FCB declaration.

SEE ALSO
.purge.

5.14 .timer. - START A TIMER

CLASS
Timer

SYNOPSIS
.timer. (TASK,SEMA,TOCK) [14]

DESCRIPTION
The timer directive inserts an entry of the specified time duration into the timer queue. The timer can be cyclic or one-shot. Cyclic timers are indicated by a negative duration while one-shots are of positive duration. Further options allow the timer to be associated either with a specified task number (0 = SELF) or a semaphore number. After the timer is inserted, the current task is resumed. See the Advanced Topic section for task versus semaphore-based timers.

CALLING SEQUENCE
ACCA = Task number (0 if SELF)
ACCB = Semaphore number (0 if task's semaphore)
IX = Number of clock tocks in timer. If > 0, timer is a
one-shot. If < 0, timer is cyclic.
swi
FCB .timer.
return

RETURN VALUE
None. ACCA will contain the current task number if ACCA was = 0 at swi. Other registers unchanged.

EXAMPLE
Example #1:
Assume that the current task wants to set up a one-shot timer of 5 seconds. Also assume that the RTI is used as the time base using a 2**23 crystal and the clock TICK is defined to be 15.625 msec. Furthermore, there are 8 TICKS per TOCK so that a TOCK is 125 msec. The setup of a 5 second timer associated with a named semaphore, TIMEOUT, would appear as:

```
TOCK        equ     8
TOCKTIME    equ     1000/8      TOCKTIME = 125 msec
TIMEOUT     equ     3           Named semaphore #3
.
:
clra                            Task is self (ACCA=0)
ldab        #TIMEOUT            Semaphore = TIMEOUT
ldx         #5000/TOCKTIME      Timer = 40 TOCKS
swi
FCB         .timer.
. Point of continuation
```

The current task would continue processing immediately after the ESR is finished because the .timer. ESR does not cause the requestor to be put into a WAIT state nor is there a context switch possible. The task may attempt to wait on the timer semaphore at some later time.

SEE ALSO .purge.

5.15 .wait. - WAIT ON SEMAPHORE

CLASS
Semaphore

SYNOPSIS
.wait. (SEMA) [1]

DESCRIPTION

The wait directive is a fundamental function in MCX11. It is used to make a task wait for a specified event to occur. The event must be associated with the given semaphore. If the semaphore is found to be in a PENDing state, the task is placed into an Event Wait state and the semaphore is changed to WAITing, and the context of the current task is saved. If the semaphore is in a DONE state, no wait occurs and the task resumes immediately. Upon task resumption, the wait directive returns the semaphore number of the event which occurred.

CALLING SEQUENCE
ACCB = Semaphore number
swi
FCB .wait.
return
RETURN VALUE
None. Registers unchanged.

EXAMPLE

The current task wants to wait on a timer semaphore set up by a previous .timer. request. The semaphore is a named semaphore, TIMEOUT. The procedure would appear as follows:

```
TIMEOUT     equ     3           Semaphore #3
.
:
ldab        #TIMEOUT  ACCB = semaphore # 3
swi
FCB         .wait.
. Point of continuation
```

The current task is put into a WAIT state on semaphore TIMEOUT as a result of this request. It will remain in this state until another task, in this case, the clock driver, signals TIMEOUT that the timer period has elapsed. The task would then continue at the instruction following the FCB.

SEE ALSO
.pend., .signal.

SECTION 6

SYSTEM CONFIGURATION

6.1 SYSTEM CONFIGURATION CONCEPTS

Another configuration concept central to the operation of MCX11 is that of pre-definition. Most real time systems used in embedded operations have a fairly static configuration. That is to say, the equipment configuration is not subject to frequent changes nor is the basic system functionality. Consequently, the system designer can define the operating environment so that the operating system dynamics are avoided to the greatest degree possible. In other words, the number of tasks concurrently executing may change but the relative priorities of tasks remain fixed. Thus task priorities can be pre-assigned. The amount of free memory can be computed and established, number of queues and their sizes, and the number of timers and semaphores are all defined prior to runtime. The result is that MCX11 works in an established environment which yields the effect of less system overhead, smaller code, and faster execution; all three being desirable in a real time system.

In order to define a MCX11 system configuration it is necessary to configure those data and control structures with the application specific information which MCX11 requires. The configuration is done by creating AS11 source code files which define the necessary information needed to assemble with MCX11 source code to define a particular configuration. The following paragraphs describe the techniques.

6.2 MCX11 SYSTEM TABLE MEMORY REQUIREMENTS

MCX11 system tables must necessarily reside in an area of RAM. This section is included to assist the system designer in determining the amount of ROM and RAM required for MCX11 tables. The amount of RAM and ROM may be computed from the equations below given the following system configuration information:

NTASKS The number of tasks

NNAMSEM The number of named semaphores

NQUEUES The number of queues

NTIMERS The number of timers

The symbols above are used by MCX11 and should be considered reserved symbols. These numbers are obviously exclusive of any RAM and ROM requirements of the application tasks.

6.2.1 Tasks

A Task Control Block occupies 5 bytes of RAM and 7 bytes of ROM.
TASK bytes (RAM) = NTASKS * 5 + 5
TASK bytes (ROM) = NTASKS * 7

5.2.2 Semaphores

Each semaphore requires 1 byte of RAM. Total system area required for semaphore control is the number of named semaphores + the number of tasks + 2 times the number of queues.
SEMA bytes (RAM) = NNAMSEM + NTASKS + 2 * NQUEUES

5.2.3 Queues

Each queue consists of a queue header and the queue body. Each queue header requires 3 bytes of RAM and 6 bytes of ROM. Each defined queue body requires an area of RAM defined by the product of

its Width (W) and its Depth (D).

QUEUE bytes (RAM) = NQUEUES * 3 + Sum(Wi * Di),

where Wi is the width and Di is the depth of the ith queue, and i ranges from 1 - NQUEUES.

QUEUE bytes (ROM) = NQUEUES * 6

5.2.4 Clock

Timers require 8 bytes of RAM each. The total area required for timer control and storage is the product of 8 and the number of defined timers.

TIMER bytes (RAM) = NTIMERS * 8

SECTION 7

DEVICE DRIVER & INTERRUPTS

7.1 DRIVER CONCEPTS

Device drivers are program modules which provide an organized software interface between a physical device and the application programs using the device. The intent is to mask the specifics of the device's hardware peculiarities from the application software. By doing so, the application code need only conform to the protocol which the device driver expects in order to perform a function. Each device driver may have a protocol unique to its function, i.e., a disk driver has different functional requirements from an analog-to-digital converter.

In order to accommodate the wide range of devices which are found in real-time systems, the device drivers in MCX11 are structured as tasks. As a task, a device driver has the most flexible environment with complete access to system resources via the Executive Service Requests. Actually, a device driver usually consists of two parts: the driver task, and the interrupt service routine (ISR). The latter is used when the device can cause an interrupt to signal the completion of a function or the availability of the device.

An ISR exists to handle an exception in the normal flow of processing. The general philosophy of MCX11 drivers is to minimize the time spent in the interrupt service routine and let the task portion handle the real work of the driver. This design concept places no restrictions on the designer for either the ISR or the task functionality. The structure of the ISR is explained more completely in the following section.

7.2 INTERRUPT SERVICE

The classic interrupt processing philosophy is to acknowledge the interrupt and save the current processor context. Then a branch is made to a software routine where the code services the specific interrupt and performs any required operations. When the interrupt service routine is complete, control of the main program is resumed at the point of interruption. MCX11 uses a modified version of that sequence.

The MC68HC11 microcontroller always pushes the current processor context of nine (9) bytes on the active stack when an interrupt occurs. That having been done, it acknowledges the interrupt and vectors automatically to the specific ISR written for that event. The user must initialize each two-byte interrupt vector location specific to his application as part of the initialization code.

MCX11 presets the vector locations for the following interrupts. All others not specifically reserved by the MC68HC11 are available to the user.

Event	Address
Clock (RTI)	$FFF0-$FFF1
MCX11 ESR request	$FFF6-$FFF7
Power On/Reset	$FFFE-$FFFF

MCX11 uses a modified version of the classical ISR approach. The sequence is the same until the return to the interrupted program is made at the end of the ISR. In MCX11, an interrupt may be the event for which a task is waiting. If so, a notification of the event's occurrence must be given to the task. But the possibility exists that the waiting task has a higher priority than the interrupted task. Notification of the event should remove the wait state from the higher priority task thus making it runnable, assuming it is not otherwise blocked.

At that point, a decision has to be made as to how to continue. Whichever task is of higher priority, the one interrupted or the one which was waiting, it will be the one to resume processing. If it is the interrupted task, it resumes at the point of interruption as in the classic sequence. If the latter task is of higher priority, a context switch is performed in which it is made the current task. Its processor context is restored from its stack and the task resumes at the location following the ESR which caused it to wait. The interrupted task remains runnable and will continue at the point of its interruption at some later time when it becomes the highest priority runnable task.

MCX11 provides a common exit routine to which all ISRs should branch if there is the possibility of a context switch as the result of the interrupt. The function's primary responsibility is to signal a specified semaphore of the occurrence of an event and to dispatch control to the highest priority task which is runnable after the interrupt has been serviced.

7.2.1 Interrupt Processing Variables

In order for MCX11 interrupt servicing to operate properly, it is necessary to observe a few rules of the road. There are some system variables which are used during interrupt processing in the ISR and in the common ISR exit routine. These variables must be used only in the ISR and should not be used elsewhere. A listing of the variables and a brief description of them follows. Their addresses are also given as a displacement from the base address of the MCX11 system variables, MCXVAR. MCXVAR may be located at any RAM address but must be predefined and assembled with the MCX11 kernel code.

intlvl (address MCXVAR+11)

This byte variable is a counter of the number of interrupts stacked but not yet processed completely. When an interrupt occurs in a task, intlvl will always contain a value of zero. The ISR code must increment it by 1 to indicate that an ISR is in process. If a task calls the MCX11 kernel, intlvl is incremented to a value of 1. If an interrupt were to occur while control is in the MCX11 kernel, intlvl would be incremented to 2 by the ISR, etc.

MCX11 allows the existence of interruptible ISRs; therefore, intlvl can assume any value up to the number of possible interrupts in the system. As interrupts are processed by the common ISR exit logic, intlvl is decremented. A resumption of processing at the task level can only occur when the decremented value of intlvl becomes zero. Otherwise, the non-zero value indicates the existence of an interrupt which has been incompletely serviced. In that case, the common ISR exit logic returns control to the interrupted ISR and resumes processing at that point. Thus, nested interrupts and re-entrant ISRs are handled cleanly, quickly, and automatically by MCX11.

curtsk (address MCXVAR+5)

This byte variable contains the number of the current task. It is to be considered a READ-ONLY variable. The user should NEVER write to this variable. It is needed during ISR processing to determine if the MCX11 Dispatcher was interrupted.

curtcb (address MCXVAR+6)

This 2-byte variable contains the address of the current task's Task Control Block (TCB). The TCB contains an entry for the task's stack pointer. MCX11 interrupt processing requires that the ISR which increments intlvl to a value of 1 must also save the current task's stack pointer in its TCB. The variable, curtcb, is READ-ONLY.

7.2.2 Interrupt Processing Code

All ISRs which have the capability of causing a context switch should incorporate the following routine as in-line code in the ISR. If the ISR is to be allowed to enable interrupts, this code should precede the CLI instruction. As a rule it should occur early in the ISR but there are exceptions. (Refer to the MCX11 clock driver's ISR, whose entry address has the symbol rtiisr, for a good example of an exception to the rule.)

Note that the label, notlvl0, used in the code segment below will need to be changed for each ISR. The symbols ACTSP and SYSTACK are not variables and should not be changed. ACTSP is equated to the value of 1 and represents the displacement into the TCB where the task's stack pointer is stored. SYSTACK is the location of the base of the System Stack area and is defined in the System Configuration source code file, SYSTEM.AS.

```
                tst     intlvl      Test for task level interrupt
                bne     notlvl0     Branch if not task level int
                tst     curtsk      See if Dispatcher interrupted
                beq n   otlvl0      Branch if so
                ldx     curtcb      Get task's TCB address
                sts     ACTSP,x     Save task's stack pointer
                lds     #SYSTACKChange to System Stack
notlvl0         inc     intlvl      Increment the interrupt level
```

After this point, the CLI instruction may be inserted if desired.

The interrupt service code follows. An important factor in interrupt processing throughput is the length of time interrupts are disabled during the ISR. In all ISRs, interrupts must be disabled long enough to perform the code segment above and only then should interrupts be re-enabled. Ideally, interrupt routines should be performed with interrupts enabled to allow other interrupts to be serviced.

The exit from the ISR routine is via a branch to the MCX11 common Interrupt Service Routine Exit routine located at the address having the symbol, isrrtn. This routine allows for optionally signalling a semaphore during the exit process. The ISR may wish to signal a semaphore that the event associated with the interrupt has occurred. To do so, the ISR simply puts the number of the semaphore to be signalled into the B-Accumulator and the executes a branch to isrrtn. From there on, the further processing of the interrupt is performed automatically by MCX11. As an example, assume the ISR wants to signal a semaphore and that the label used for the semaphore is EVNTSEMA. Then the concluding code segment in the ISR is

```
        ldab    #EVNTSEMA       Load ACCB with semaphore #
        jmp     isrrtn          Jump to exit logic.
```

If no semaphore is to be signalled during the exit routine, the ISR should make sure that the B Accumulator is cleared before branching to isrrtn.

SECTION 8

ADVANCED TOPICS

8.1 OTHER TOPICS

Other topics of interest will be added to this section asthey are researched and documented.

APPENDIX A

MCX11 DISTRIBUTED SOURCE FILES

MCX11 is distributed as a collection of files. Those files and their purposes are briefly described below.

CLKDRVR.AS

The AS11 source code of the RTI clock driver and RTI Interrupt Service Routine.

ESREQU.AS

These are just the equates for MCX11 ESRs included for completeness.

MCX.AS

This is the AS11 source code of the MCX11 kernel. Unless you are expanding the functions of the kernel or fixing a reported bug, this file probably won't change too much.

SYSTEM.AS

This AS11 source code file contains the application specific information and should be studied very closely. It also contains initialization code which may be necessary to change to meet the needs of a given application. The file is pre-configured with the specifications needed for the demonstration programs in file TEST.AS.

TEST.AS

This is an AS11 source code file containing 4 tasks which demonstrate the use of all of the MCX11 ESRs. It doesn't do anything practical except to give the user a limited look at how to accomplish certain operations.

VECTORS.AS

AS11 source code containing all of the vectors used by MCX11. This file may need to be edited for a given application.

ASSEMBLE.BAT

A MS-DOS Batch file used to assemble MCX11 and the TEST programs. It also produces a listing file.

MANUAL.DOC

The MCX11 User's Manual is contained in this file. It can be printed out directly to any ASCII compatible printer.

README.DOC

Some notes about getting started with MCX11.