
# COMPLETE EMBEDDED SOLUTIONS

CMX SYSTEMS
680 Worcester Road • Framingham, MA 01702 USA
Phone: (508) 872-7675 • Fax: (508) 620-6828
Email: cmx@cmx.com • WWW: http://www.cmx.com

Dear Software Developer: Thank you for your purchase.

You **MUST** Fill out this registration form to receive support, upgrades, etc. Please mail it to the address above or FAX it. Thank you. Also please make a copy for your records.

CMX REGISTRATION CARD

Your Name: _____

Your Company: _____

Address: _____

Address: _____

City: _____ State: _____ Zip: _____

Country: _____

Phone: _____

Fax: _____

Email: _____

Software Package: _____ Version:_____
Located on Diskette

Serial Number: _____
Located on Licensing Agreement

I hereby agree to the terms and conditions of the license agreement for this software.

Signed: _____

Unsigned registrations will not be accepted.

680 Worcester Road • Framingham, MA 01702 USA
Phone: (508) 872-7675 • Fax: (508) 620-6828
Email: cmx@cmx.com • WWW: http://www.cmx.com

## CMX SYSTEMS, INC., SOFTWARE LICENSE

The enclosed software and documentation are the exclusive property of **CMX Systems, Inc.** (CMX) and **CMX Company**, a division of **CMX Systems, Inc.**, protected under the copyrights laws of the United States of America and under international treaty provisions. LICENSOR is an authorized licensor of CMX software products (herein referred to as the "SOFTWARE"). LICENSOR agrees to grant LICENSEE a license to use the SOFTWARE and LICENSEE agrees to pay for this license in accordance to the terms specified. By using the SOFTWARE, the LICENSEE has agreed to the terms set forth in this Agreement whether or not this Agreement is read.

**GRANT OF LICENSE**: LICENSOR grants to licensee a nontransferable license to use the SOFTWARE. LICENSEE acknowledges that by virtue of this Agreement, LICENSEE acquires only the right to use the SOFTWARE and does not acquire any right of ownership in the SOFTWARE. LICENSOR offers a PER USER PER SITE license or a PER PRODUCT PER SITE license based upon the SOFTWARE purchase terms and as defined below:

**PER USER PER SITE LICENSE**: CMX authorizes the purchaser to make backup copies of the SOFTWARE for archival purposes only. CMX authorizes the purchaser to have only a single user use the SOFTWARE as a "single seat license" on a royalty free basis. For each "single seat license", only one (1) person may use this software at a specified site and CMX will only support that single person. The SOFTWARE or its accompanying documentation must not be copied or distributed to others in any way. This license is effective until terminated. You may terminate it by destroying the SOFTWARE and documentation and all copies thereof. The license will also terminate if you fail to comply with any terms or condition of this agreement. You agree upon such termination to destroy all copies of the SOFTWARE and documentation.

**PER PRODUCT PER SITE LICENSE**: The CMX software is licensed on a SINGLE LICENSEE DEVELOPED PRODUCT BASIS for ONE CPU and ONE DEVELOPMENT FACILITY on a royalty free basis. If the LICENSEE develops and sells additional LICENSEE DEVELOPED PRODUCT(s), then the LICENSEE must pay CMX the full purchase price of the SOFTWARE, for EACH additional LICENSEE DEVELOPED PRODUCT. CMX authorizes the purchaser to have as many users at a single facility use this software, but only one user may be the point of contact for technical support and CMX will only support that single person. If additional technical support is required by more than one person, then the LICENSEE may purchase SUPPORT licenses for those additional people. The SOFTWARE or its accompanying documentation may be copied and distributed to others only within the same facility.

This license is effective until terminated. You may terminate it by destroying the SOFTWARE and documentation and all copies thereof. The license will also terminate if you fail to comply with any terms or condition of this agreement. You agree upon such termination to destroy all copies of the SOFTWARE and documentation.

**TERMS OF SOFTWARE**: LICENSEE agrees not to distribute the SOFTWARE source code to any third party in any manner. LICENSEE'S use of the SOFTWARE shall be limited to integrating the SOFTWARE as an integral component within the LICENSEE'S own product (PER PRODUCT PER SITE) or products (PER USER PER SITE), depending upon the type of license acquired from LICENSOR. LICENSEE shall have the right to distribute the SOFTWARE as an integral component of the LICENSEE'S own products as long as the SOFTWARE is in absolute machine readable format (e.g., HEX file). The LICENSEE can NOT sell a product that allows the user(s) of the licensee product(s) to be able to indirectly call the CMX functions (e.g., The licensee product contains an API [Application Program Interface] that allows the user(s) to indirectly call and use the CMX functions), without the user(s) of the licensee's product(s) also purchasing a license to use the SOFTWARE.

Over, please →

**LIMITATION OF LICENSOR'S LIABILITY**: LICENSOR shall not be liable for any damages, including but not limited to, interruption of business, loss of profit, incidental, consequential or any other claims either by LICENSEE or any other party. LICENSOR shall not be liable for any damages incurred by LICENSEE or any other person as a result of LICENSEE'S use or misuse of the SOFTWARE, even if LICENSOR had been advised of the possibility of such damage.

**SEVERABILITY**: If any provision of this Agreement shall be held illegal, unenforceable or in conflict with any law governing this Agreement, the validity of remaining portions shall not be effected thereby.

**NON-WAIVER**: Failure of LICENSOR at any time to require performance of this Agreement shall not limit LICENSOR'S right to enforce the provision, nor shall any waiver by LICENSOR of any breach of provision constitute a waiver of or prejudice LICENSOR'S right otherwise to demand strict performance or the provision or any other provision.

**U.S. GOVERMENT RESTRICTED RIGHTS**: This SOFTWARE and documentation are restricted computer software and are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions of the sort set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013.

**LIFE SUPPORT APPLICATIONS**: CMX SOFTWARE is not designed for use in life support appliances, devices or systems where malfunction of the SOFTWARE can reasonably be expected to result in a personal injury. CMX customers using or selling CMX SOFTWARE for use in such applications do so at their own risk and agree to fully indemnify CMX for any damages resulting from such improper use or sale.

**WARRANTIES**: LICENSOR represents and warrants the following: That LICENSOR has the right to grant LICENSEE a license to use the SOFTWARE and to enter into this Agreement. That the physical media, on which the SOFTWARE is shipped, is free from defects and that if a defect is found, a replacement copy will be provided. This limited warranty gives you specific legal rights. You may have others which vary from state to state.

LICENSEE EXPRESSLY AGREES AND ACKNOWLEDGES THAT THE FOREGOING WARRANTIES ARE IN LIEU OF ALL OTHER WARRANTIES EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO AN IMPLIED WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

**GOVERNING LAW**: This Agreement shall be governed by the laws of the Commonwealth of Massachusetts.

Some states do not allow limitations on the duration of an implied warranty, so the above limitations may not apply to you. This limited warranty gives you specific legal rights. You may have others, which vary from state to state. Because some states do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitations may not apply to you.

680 Worcester Road • Framingham, MA 01702 USA
Phone: (508) 872-7675 • Fax: (508) 620-6828
Email: cmx@cmx.com • WWW: http://www.cmx.com

## CMX INSTALLATION INSTRUCTIONS

The enclosed diskette contains the CMX Real-Time Multi-Tasking Operating System. We suggest making a copy of this diskette for archival purposes only.

The root directory contains all the CMX files, that you will need to work with.

The CMXMOD directory contains all the CMX functions, make file and support files in order to create a different memory model library or if the user is using a different released version of the C compiler, assembler, linker, librarian then what CMX used.

The user should copy all the ROOT directory files to the directory that contains the target processor C compiler and tools.

The user should then make a sub directory named CMXMOD or user selected name. Then copy all the files from the diskette CMXMOD directory into this directory.

The user should read the CMXREAD.DOC file that explains what versions that CMX used for the particular C vendor's compiler, assembler, linker, and librarian. Also it may include additional information particular to the processor and C vendor that can not be found in the manual.

Please thoroughly READ the manual and look at both the C source code and assembly source code to get a better idea how CMX works. Also work with the sample(s) supplied before jumping right in and coding your first application program with CMX.

Remember that there will be a learning curve, and some time will be needed to get a "good feeling" as to how everything comes together. PLEASE be PATIENT.

## CMX-RTX AND PCPROTO-RTX USERS

Please disregard any references to CMXTracker and CMXTracker files, if you have not purchased it. The documentation for CMXTracker is provided to make it easier for you if you decide to purchase it at a later date.

Please contact us with any questions that you may have with regards to this matter.

680 Worcester Road • Framingham, MA 01702 USA
Phone: (508) 872-7675 • Fax: (508) 620-6828
Email: cmx@cmx.com • WWW: http://www.cmx.com

## SOFTWARE PROBLEM REPORTING

If the user feels there is a problem with the CMX software, the following steps should be performed.

Please be some what ensured that it is a problem with the CMX Real-Time Multi-Tasking operating System code supplied and NOT: application coding errors, poor design of the application code and how it is integrated, misuse of the CMX RTOS, etc. Try a different approach or try to exercise just the particular problem in a minimal test if possible.

Email, call or FAX the following.

PRODUCT DETAILS: The company name, name of person reporting problem, the name and version of the manufacturer's C compiler being used, the version number and serial number of the CMX RTOS.

PROBLEM SEVERITY: how serious is the problem.

PROBLEM DESCRIPTION: a concise and informative description of the problem, the results that occurred, and any solution if known.

CMX will try to resolve any problems within a reasonable amount of time. If needed, CMX may request the user to send or fax the portion of the code that is not working correctly.

DISCLAIMER: While CMX Systems, Inc. will investigate the problem as soon as possible and make every attempt to come up with a solution, we do not guarantee to provide a solution.

NOTICE: CMX is not in the business of writing other companies application code around the CMX Real-Time Multi-Tasking Operating System, so DO NOT call expecting this. Also if the user manipulates the CMX code for any reason, then CMX will not be responsible for how the code executes.


# USER MANUAL



CMX Systems, Inc.
680 Worcester Road
Framingham, MA 01702 U.S.A.
Phone: (508) 872-7675
FAX: (508) 620-6828
Email: cmx@cmx.com
WWW: http://www.cmx.com

## IMPORTANT LEGAL NOTICE

Please note that there are specific references made in this user's manual, in our marketing literature, and in the actual software files to CMX Systems, Inc., and CMX Company. CMX Company is a division of CMX Systems, Inc. and the two company names should be regarded as the same legal entity for the purposes of copyright, trademark, licensing and any other legal issues.

## COPYRIGHT NOTICE



This documentation is confidential. CMX may, if they so choose, to put some portions of this documentation within the public domain and if so, then those respective portions will not be consider confidential.

## DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment on any part of CMX Systems, Inc. While the information contained herein is assumed to be accurate, CMX Systems, Inc. assumes no responsibility for any errors or omissions.

In no event shall CMX Systems, Inc., its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

## TRADEMARKS

CMX is a trademark of CMX Systems, Inc.

All other product names are trademarks or registered trademarks of their respective owners.

# Table of Contents

# Table of Contents

## THE CMX MULTI-TASKING EXECUTIVE

Welcome to the world of multi-tasking. CMX provides the necessary function calls and operating system to write efficient C and/or assembly code to create a well designed, multi-tasking application.

The ability of a single processor to run many tasks by swapping different tasks in and out creates the feeling that many tasks are operating simultaneously. This is what multi-tasking is all about. CMX is a real-time multi-tasking operating system giving you function calls and an operating system kernel that will:

allow control of tasks
send and receive messages
handle events
control resources
control semaphores
regulate timing in a variety of ways
provide memory management
handle interrupts
swap tasks.

## THE CMX SCHEDULER

The CMX operating system provides real-time processes the ability for task switching according to the input stimulus received. The heart of this operating system is the scheduler. The scheduler is based on true preemption. This means that tasks and interrupts can cause an immediate task switch, if a higher priority task becomes able to run as the result of a CMX function. Cooperative scheduling is also possible, a task can let the next task (same or lower priority) run if desired. True time slicing is also available. The time slicing mechanism allows a higher priority task to preempt the current running task. The sliced task regains control, when the higher priority task is done, until its allocated time slice expires.

The scheduler keeps track of variables associated with tasks. Task switches will be performed by the scheduler depending upon the state of these variables. A task or interrupt may cause a preemption, which informs the scheduler that a higher priority task needs servicing. Possibly the interrupt that determines the "system tick", the basis for all time related activity, occurred. The CMX tick function will determine whether there are any time related activities that need attention and will tell the scheduler to call the CMX timer task, which will handle the timing chores.

If a preemption occurs, the task that was running has its context saved. The scheduler decides which of the tasks to run next. The scheduler will load all the proper information for this task to operate. If the task had been suspended, for whatever reason, by a function call or a higher priority task, then all the saved task variables are reloaded, all registers are restored to their respective values for this task and the task resumes where it left off as if it was never suspended.

For more information, please study the CMX Scheduler Chapter which explains how the scheduler works in detail. That chapter explains the different flags the scheduler will act upon and how the scheduler interfaces to the interrupt driven *K_OS_Tick_Update* function, tasks, CMX functions and other interrupts.

## WHEN A TASK IS INTERRUPTED

A task may suspend itself by a variety of CMX function calls forcing the scheduler to reschedule immediately without regard to the specified system tick time interval. Also some CMX function calls that take a task out of the suspended state, or that start a task that was idle, will force immediate rescheduling, if the suspended or new task has a higher priority than the running task.

CMX gives you the ability to handle single or multiple (nested) interrupts. The interrupts may call many CMX functions and CMX provides the necessary interrupt functions that save and restore the contexts of a task, or interrupt, when nesting occurs.

When a task's (or interrupt's) context is saved, you can be assured that all parameters dealing with this task or interrupt -- all the CPU registers, local variables and parameters passing variables -- will be restored properly, as if the task or interrupt was never suspended.

## TASK STATES

There are several possible states a task can be in, though a task can be in only one state at a time. These states are: IDLE, READY, RUN, WAIT, and ready to RESUME.

### IDLE state

A task that has been created with the *K_Task_Create* function, but not started by the *K_Task_Start* function is in the IDLE state. A task that has completed its code and called the *K_Task_End* function is placed into the IDLE state if there are no outstanding triggers in its control block. A task that is in its IDLE state will not run.

### READY state

The READY state informs the scheduler that the task is ready to run, but NOT running. This allows the scheduler to determine what task to run when a scheduling takes place according to the task’s priority in relation to the other tasks’ priorities.

### RUN state

The task that is executing is in the RUN state and owns the CPU time. Only one task may be in the RUN state at any one time.

### WAIT (suspended) state

A task that suspended itself by a CMX function call is in the WAIT (suspended) state. There are many function calls that will suspend a task. The WAIT state consists of a task that is waiting on one or more of the following: time, events, flags, messages, on a reply, etc.

### RESUME state

The RESUME state is treated the same as the READY state. The only difference is that it informs the scheduler that the task had been started, yet not finished, with its code. This means that a higher priority task has preempted and forced the original task that was running to become ready to RESUME, or that the task had suspended itself by a function call and now has removed itself out of the WAIT (suspended) state into the ready to RESUME state.

## SETTING UP TASKS

Tasks should be coded to perform specific duties. It is up to you to properly create the tasks to do specific jobs in an orderly fashion. Structuring the individual tasks' responsibilities in relation to each other, and providing the proper interrupt handling with respect to other interrupts and tasks is probably the most challenging problem in writing real-time multi-tasking code.

When beginning a new project and associated application code, you must tell CMX specific information. This is described briefly here and later in greater detail. Initially, until the project becomes more defined and near completion, it is recommended that the values selected are increased over their designed values, so you do not have to change these parameters constantly.

Some of the information CMX needs to know are:

the number of tasks
the number of cyclic timers
the number of mailboxes
the total number of messages
the number of resources
the stack size for all the tasks
the interrupt stack size
the number of semaphores
the number of queues.

CMX allocates all needed memory before you can enter the operating system. This saves CPU time when a CMX function is used that involves memory. CMX feels it is the best way to achieve the fastest possible execution of code. What would the operating system do if it allocated memory dynamically and found there was no more memory available? The task would then have to decide how to handle this and always test whether the CMX function returned indicating that no more memory is available.

For those of you writing part of your application code in "assembly", parameter passing is done just as the equivalent C code would do. You would compile a function that calls a CMX function and check the compiler-generated assembly code to call that particular CMX function.

CMX highly recommends you read the complete manual and any additional CMXREAD.DOC files that may be supplied before trying to interface with CMX. You should realize there will be a "learning curve" as with any new software. The more you work with it, the better you will understand the CMX software and how to incorporate it into your application code.

## CMX RETURN STATUS BYTE VALUES

| Symbol | Hex | Value | Explanation |
|---|---|---|---|
| K_OK | 00 | Good | CMX call was successful |
| K_TIMEOUT | 01 | Warning or Error | Time out occurred |
| K_NOT_WAITING | 02 | Error | Task not waiting for wake request |
| K_RESOURCE_OWNED | 05 | Error | Resource is already "owned" |
| K_RESOURCE_NOT_OWNED | 06 | Error | Resource not owned by calling task |
| K_QUE_FULL | 0A | Warning | Queue now full, slot was added |
| K_QUE_EMPTY | 0B | Warning | Queue now empty, slot was removed |
| K_SEMAPHORE_NONE | 0C | Error | Semaphore is not available |
| K_ERROR | FF | Error | General error, CMX call unsuccessful |

Some functions, such as *K_Mesg_Get*, may return a NULL pointer if there is no message available. This indicates a possible warning/error to the caller, for NO message was retrieved. In some cases a return value of zero indicates the CMX function is telling the caller the item it wanted was not there or a time out occurred.

## CMX DATA TYPES

CMX has declared the following data type names within the cxdefine.h header file. There are a few listed here. These may possibly change depending on the processor and C vendor that you are working with. Look at the cxdefine.h file, to see the ones pertinent to your processor and C vendor.

```
typedef unsigned char byte
typedef unsigned short word16
typedef unsigned short bit_word16
or
typedef unsigned int bit_word16
typedef signed short sign_word16
typedef unsigned long word32
```

## LAYOUT OF THE FUNCTIONS

Within each function, we will describe in detail the purpose of that particular function. Explained will be the parameter declarations, if any, the functions's return type, if any and any other useful information. Also within each function section, will be the following:

**Called**

Before entering RTOS, Tasks, Interrupts.

[Before entering RTOS] means the call may be used prior to entering the CMX operating system. [Tasks] mean a task may use this CMX call. [Interrupts] mean interrupts may use this CMX call INDIRECTLY.

☞ *The K_OS_Init function must be called before any CMX function call may be used.*

The user will then see what header files are needed that identify the function prototypes. Also the parameters needed will be shown in an example style, with a brief commented description, followed by the function prototype, with the respective example parameters used within the function prototype. An example is below.

#include <cxfuncs.h> /* has function prototype */

Many of the parameters passed are of constant value, so we have used #defines to identify them, with 3 question marks (i.e. ???), indicating the value would need to be selected by the user. Also we have tried to use descriptive words to indicate the meaning of the particular parameter. You may choose any text you like for the name of a parameter, but we recommend that it be meaningful.

#define PRIORITY ???

unsigned char TASK_SLOT; /* should be global */

void TASK(void); /* prototype of task function */

#define STACK_SIZE ???

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Create*(PRIORITY,&TASK_SLOT,TASK,STACK_SIZE);

**Passed**

This will identify each of the parameters that this function will be passed and what they indicate.

PRIORITY is the priority for this task. The lower the number the higher the priority.

&TASK_SLOT is the address where CMX will put task slot number. Must be used for all references to this task.

TASK is the address where the task resides in code. When task begins execution, this is where CMX will vector to.

STACK_SIZE is the number of bytes set aside for this task stack area. You must make sure the stack size is large enough for all the levels of nesting, and the depth of one interrupt.

**Returned**

This will identify the return-type of this function and what it indicates.

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: no free task control block available.

If STATUS equals K_OK, then TASK_SLOT contains the task identification number assigned by CMX. This identification number must be used for all CMX function calls that deal with this task.

***Example***

We will then provide one or more examples of using the function and a comment on how we are using it.

The following is an example for creating a task to the RTOS.

void task1(void); /* function prototype, show that task1 does not receive nor return parameters */

unsigned char task1_slot; /* create storage for CMX to return task1 slot number */

```
void main(void)
{
  unsigned char status; /* create a local status byte */
  status = K_Task_Create(5,&task1_slot,task1,128);

  /* call CMX function K_Task_Create with task1 having a priority
  of 5, the address for storage of task1 slot number, the address
  of task1, and finally a stack size of 128 bytes */

  if (status != K_OK) /* check status, make sure good function
  call */
    {
    error_handler(); /* go to error handler */
```

```
    }
}
```

There may or may not be an additional comment field as shown below. The comment field is to reinforce what we are trying to do with the function.

**Comments**

CMX returns the status of the *K_Task_Create* function call indicating whether the call was successful or not. If the status is good then the slot number returned must be used from now on for any CMX function calls dealing with this task. Usually this slot number will be stored in external RAM. When a task is created, CMX puts the task into the IDLE state. This means that the task is loaded but will never RUN until started.

## TASK MANAGER FUNCTIONS

The task manager is part of the CMX library and provides the necessary functions for controlling your tasks. The task manager functions are listed below along with their reference pages.



### *The K_Task_Create function*

The CMX *K_Task_Create* function is used to create a task. You can create a task before entering the CMX operating system or dynamically while running under the CMX operating system. For maximum speed and because you may know ahead of time what tasks will be needed, it is highly recommended that the tasks be created before entering RTOS. The creation tells CMX where the task's execution code resides in ROM, the stack size for this task (each task may have a different size stack), the priority of this task, and the address of the slot number CMX assigns for this task.

The parameters you need to send are the following.

The priority for this task. Priority numbers may range from zero to 254. The priority tells CMX the order in which to run tasks when they become READY. The lower the number, the higher the priority. At rescheduling the highest priority task (lowest priority number) that is READY to run becomes the RUNNING task. If tasks have the same priority then it is determined by the order of creation, the first task created with the same priority as another task created later is given the first option to run, then the later one. The priority also is used by the CMX time slicing mechanism. Tasks will be time sliced, starting with tasks with the same or lower priority as the current task, if time slicing is enabled by calling the *K_OS_Slice_On* function. See the chapter about time slicing for more detailed information on time sliced tasks and how they work.

Another parameter is the address of an unsigned character to put the slot number CMX will assign to this task. The task slot number is used for ALL CMX function calls that require the task number. It is up to the user to make sure that they do not destroy or corrupt this slot number. If the task is removed, then the slot number is no longer valid. If another task is created after a task is removed, then the newly created task may have the "old" slot number, which the previously removed task had.

The next parameter is the address where the task's code will begin execution. This address is where CMX will start the task's code when the task switches from the READY state to the RUNNING state.

The last parameter supplied is the size of the stack for this task. Since insufficient stack size is one of the most common causes of system crashes and corrupt memory, it is recommended you double the estimated size. As you become more knowledgeable and actually test your code, then the size may be reduced. See the chapter on stacks for more information on how to calculate the size of the stack for a particular task.

This is an example of the *K_Task_Create* function:

**Called**

Before entering RTOS and tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Task_Create(byte, byte *, K_FP, word16); /* this is the function prototype */

#define PRIORITY ???

unsigned char TASK_SLOT; /* should be global */

void TASK(void); /* prototype of task function */

#define STACK_SIZE ???

unsigned char STATUS; /* should be local */

STATUS = K_Task_Create(PRIORITY,&TASK_SLOT,TASK,STACK_SIZE);
```

**Passed**

PRIORITY is the priority for this task. The lower the number the higher the priority.

&TASK_SLOT is the address where CMX will put task slot number. Must be used for all references to this task.

TASK is the address where the task resides in code. When task begins execution, this is where CMX will vector to.

STACK_SIZE is the number of bytes set aside for this task stack area. You must make sure the stack size is large enough for all the levels of nesting, and the depth of one interrupt.

void task1(void); /* function prototype, show that task1 does not receive nor return parameters */

unsigned char task1_slot; /* create storage for CMX to return task1 slot number */

```
void main(void)
{
  unsigned char status; /* create a local status byte */

  status = K_Task_Create(5,&task1_slot,task1,128);

  /* call CMX function K_Task_Create with task1 having a priority
  of 5, the address for storage of task1 slot number, the address
  of task1, and finally a stack size of 128 bytes */

  if (status != K_OK) /* check status, make sure good function
  call */
    {
    error_handler(); /* go to error handler */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: no free task control block available.

If STATUS equals K_OK, then TASK_SLOT contains the task identification number assigned by CMX. This identification number must be used for all CMX function calls that deal with this task.

CMX returns the status of the *K_Task_Create* function call indicating whether the call was successful or not. If the status is good then the slot number returned must be used from now on for any CMX function calls dealing with this task. Usually this slot number will be stored in external RAM. When a task is created, CMX puts the task into the IDLE state. This means that the task is loaded but will never RUN until started.

### *The K_Task_Create_Stack function*

The CMX *K_Task_Create_Stack* function is used to create a task. The main difference between this create task function and the *K_Task_Create* function, is that you pass the address of the task's stack address to the *K_Task_Create_Stack* function versus the size of the how much stack space is required when you use *K_Task_Create* function. When you use the *K_Task_Create* function and 'kill' a task, the stack space does not get reclaimed. Thus if you created and killed a fair number of tasks, then you would most likely run out of stack space. The new *K_Task_Create_Stack* function avoids this, for now you can use memory allocation functions to gain and then free stack space needed for many tasks. This function is very useful for embedded systems that may run a TCP/IP stack, where multiple clients may need servicing. This way you can allocate stack space when needed when creating a new task and when that task is no longer needed, you can then kill it and reclaim that stack space. You can create a task before entering the CMX operating system or dynamically while running under the CMX operating system. The creation tells CMX where the task's execution code resides in ROM, the stack starting address, the priority of this task, and the address of the slot number CMX assigns for this task.

The parameters you need to send are the following.

The priority for this task. Priority numbers may range from zero to 254. The priority tells CMX the order in which to run tasks when they become READY. The lower the number, the higher the priority. At rescheduling the highest priority task (lowest priority number) that is READY to run becomes the RUNNING task. If tasks have the same priority then it is determined by the order of creation, the first task created with the same priority as another task created later is given the first option to run, then the later one. The priority also is used by the CMX time slicing mechanism. Tasks will be time sliced, starting with tasks with the same or lower priority as the current task, if time slicing is enabled by calling the *K_OS_Slice_On* function. See the chapter about time slicing for more detailed information on time sliced tasks and how they work.

Another parameter is the address of an unsigned character to put the slot number CMX will assign to this task. The task slot number is used for ALL CMX function calls that require the task number. It is up to the user to make sure that they do not destroy or corrupt this slot number. If the task is removed, then the slot number is no longer valid. If another task is created after a task is removed, then the newly created task may have the "old" slot number, which the previously removed task had.

The next parameter is the address where the task's code will begin execution. This address is where CMX will start the task's code when the task switches from the READY state to the RUNNING state.

The last parameter supplied is the stack address for this task. Note that it is up to you to pass the address of memory, that will be large enough to be used by this task and to handle the number of nested function calls, locals, saving of registers, etc. Also on most processors the stack walks downward, so the user must ensure that they pass the stack address pointing to the 'top' of the stack space that they have freed up and not the bottom. Of course if you are using a processor where the stack grows upward, then you would pass the base address of the stack space. Since insufficient stack size is one of the most common causes of system crashes and corrupt memory, it is recommended you double the estimated size. As you become more knowledgeable and actually test your code, then the size may be reduced. See the chapter on stacks for more information on how to calculate the size of the stack for a particular task.

This is an example of the *K_Task_Create_Stack* function:

**Called**

Before entering RTOS and tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Task_Create_Stack(byte, byte *, K_FP, word16); /* this is the function
prototype */

#define PRIORITY ???

unsigned char TASK_SLOT; /* should be global */

void TASK(void); /* prototype of task function */

unsigned ??? STACK_ADDRESS; /* The beginning stack address for this task, must
be aligned to match the stack pointer alignment criteria. */

unsigned char STATUS; /* should be local */

STATUS =
K_Task_Create_Stack(PRIORITY,&TASK_SLOT,TASK,&STACK_ADDRESS);
```

**Passed**

PRIORITY is the priority for this task. The lower the number the higher the priority.

&TASK_SLOT is the address where CMX will put task slot number. Must be used for all references to this task.

TASK is the address where the task resides in code. When task begins execution, this is where CMX will vector to.

&STACK_ADDRESS is the task stack. You must make sure the stack memory size that you have allocated is large enough for all the levels of nesting, and the depth of one interrupt.

void task1(void); /* function prototype, show that task1 does not receive nor return parameters */

unsigned char task1_slot; /* create storage for CMX to return task1 slot number */

There are many ways to create a stack for a task, we will show you a few ways. Please ensure that you pass the top of the memory allocated to the task stack, if the stack pointer grows downward.

```
struct {
    unsigned int task1_stk[1000];
    unsigned int dummy;
} task1_stack;

void main(void)
{
  unsigned char status; /* create a local status byte */

  status =
  K_Task_Create_Stack(5,&task1_slot,task1,task1_stack.dummy);

  /* call CMX function K_Task_Create_Stack with task1 having a
  priority of 5, the address for storage of task1 slot number,
  the address of task1, and finally a stack size of 128 bytes */
  if (status != K_OK) /* check status, make sure good function
  call */
    {
    error_handler(); /* go to error handler */
    }
}
```

Another way.

```
void *alloc;

void main(void)
{
  unsigned char status; /* create a local status byte */

  if ((alloc = malloc(1000)) != NULL)
    {
```

```
    status = K_Task_Create_Stack(5,&task1_slot,task1,((alloc) +
    998));
    /* call CMX function K_Task_Create_Stack with task1 having a
    priority of 5, the address for storage of task1 slot number,
    the address of task1, and finally a stack size of 128 bytes */

    if (status != K_OK) /* check status, make sure good function
    call */
      {
      error_handler(); /* go to error handler */
      }
    }
  else
    {
    Handle memory allocation error
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: no free task control block available.

If STATUS equals K_OK, then TASK_SLOT contains the task identification number assigned by CMX. This identification number must be used for all CMX function calls that deal with this task.

CMX returns the status of the *K_Task_Create_Stack* function call indicating whether the call was successful or not. If the status is good then the slot number returned must be used from now on for any CMX function calls dealing with this task. Usually this slot number will be stored in external RAM. When a task is created, CMX puts the task into the IDLE state. This means that the task is loaded but will never RUN until started.

### *The K_Task_Start function*

Since the *K_Task_Create* function puts a task into the IDLE state when created, this function allows a task to be started. This function may be called anytime. The *K_Task_Start* function really puts the task into the READY state allowing it to become the RUNNING task when it is the highest priority task ready to run. If the task was IDLE, then the task becomes READY, however not necessarily the RUNNING task. Once a task is out of the IDLE state any additional *K_Task_Start* calls to this task are queued up to a maximum of 255.

When the task normally ends its code and puts itself back into the IDLE state, if there are any outstanding start requests then the task will automatically put itself back into the READY state. The task's slot number is passed to this function indicating which task to start. This function can be called before entering the CMX operating system, while in the operating system, and by interrupts. The maximum number of *K_Task_Start* calls that any one task will queue up is 255.

This is an example of the *K_Task_Start* function:

**Called**

Before entering RTOS, tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

#include <cxfuncs.h> /* has function prototype */
byte *K_Task_Start*(byte); /* this is the function prototype */

unsigned char TASK_SLOT; /* global, declared earlier */

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Start*(TASK_SLOT);

**Passed**

TASK_SLOT is the name where the particular task's slot number (I.D. number) resides.

unsigned char task1_slot; /* defined earlier, contains task 1 slot number */

```
void task2(void)
{
  unsigned char status;
  status = K_Task_Start(task1_slot); /* start task 1, put into
  READY state if this is the first trigger */
  if (status != K_OK) /* check status, make sure good function
  call */
    {
    error_handler(); /* go to error handler */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist.

If STATUS equals K_OK, then the task is put into the READY state. If it is in the READY state already, then the trigger will be put into the task's trigger queue.

Again the status is passed back indicating whether the CMX *K_Task_Start* call was successful or not. If the task being started has a higher priority (lower priority number) than the task that is currently RUNNING, then an immediate task switch will occur, bypassing the normal rescheduling caused by the system tick.

### *The K_Task_Priority function*

The *K_Task_Priority* function changes a task's priority. This function may be called any time. You will send two parameters: the slot number of the task you want to change and the new priority for this task. The task must have been created or an error will be returned.

This is a useful function if the task has a low priority in contrast to other tasks, yet when it becomes the RUNNING task, its priority must become higher. Also you may dynamically change priorities of tasks according to conditions created by task processes and outside variables processed. In a well-designed system, this function will be called very little or not at all.

If the new priority is the greater than the priority of the current task and time slicing is enabled, then the task will become a time sliced task and time slicing will continue starting with the new priority. See the Time Slice Chapter for more details on time slicing and how it is incorporated and used by the CMX operating system.

This is an example of the *K_Task_Priority* function:

**Called**

Before entering RTOS and tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Task_Priority(byte,byte); /* this is the function prototype */

unsigned char TASK_SLOT; /* global, declared earlier */

#define NEW_PRIORITY ???

unsigned char STATUS; /* should be local */

STATUS = K_Task_Priority(TASK_SLOT,NEW_PRIORITY);
```

**Passed**

TASK_SLOT is the name where the particular task's slot number resides.

NEW_PRIORITY is the new priority for the task. The lower the number, the higher the priority The value may range from zero to 254.

unsigned char task1_slot; /* defined earlier, contains task 1 slot number */

```
void task2(void)
{
  unsigned char status;

  status = K_Task_Priority(task1_slot,3); /* change task1
  priority from 5 to 3 */

  if (status != K_OK) /* check status, make sure good function
  call */
    {
    error_handler(); /* go to error handler */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task I.D. number does not exist.

If STATUS equals K_OK, then the task has the new priority for its priority. Note that the new priority will become effective immediately but will not cause a rescheduling.

### *The K_Task_Wait function*

The *K_Task_Wait* function enables a task to suspend itself for a specified amount of time or indefinitely. This function allows a task to become synchronized with another task, interrupts, or the system tick. Letting the task suspend itself for a specified period, knowing that the task will RESUME when the time expires, can be very useful. This function also allows a task to wait with or without a time period for an interrupt to wake it, notifying the task that an event has happened. (The event manager handles multiple events per task. This is described fully later). Only tasks may call this function.

The amount of time, which is a multiple of system ticks, is passed to this function. This value is the number of system ticks you want the task suspended for. The amount of time may be from zero to 65535 (zero to FFFF hex). A time period of zero will result in the task waiting indefinitely until the *K_Task_Wake* or *K_Task_Wake_Force* function is used to wake this task.

The task will automatically suspend itself for a specified time period (if non zero) and then become READY to RESUME execution at the end of this time. The *K_Task_Wake* or *K_Task_Wake_Force* function may be called to wake this task earlier than the time specified. This will be reported by a returned status byte. If the *K_Task_Wake* function was used, the task that had been waiting will be returned a K_OK status, since the 'time-out' did not occur. If the *K_Task_Wake_Force* function was used, the task that had been waiting will be returned a K_ERROR status, since the task was forcefully woken. Remember that other tasks and interrupts may use the *K_Task_Wake* or *K_Task_Wake_Force* function to wake this task.

The accuracy of this call is a derivative of the system tick specified. For example, say you have created a 20 millisecond system tick. If the task requests 10 system ticks, and calls *K_Task_Wait*, the following will happen. The time period is decremented at every system tick. When the time becomes zero, then the task is automatically put into the READY to RESUME execution state. You can see, depending on when the task calls *K_Task_Wait* in relation to the system tick, the task will wait anywhere from 180 milliseconds to 200 milliseconds.

This is an example of the *K_Task_Wait* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Task_Wait*(word16); /* this is the function prototype /*

#define TIME_CNT ???

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Wait*(TIME_CNT);

**Passed**

TIME_CNT is number of system ticks that this task will suspend itself. If the value is zero then the task will be suspended indefinitely until the *K_Task_Wake* function is used. If the value is non-zero, then the task will be suspended for that number of system ticks. The *K_Task_Wake* function may be used prior to the time period expiring, to wake this function and put it back into the READY state. The maximum value that TIME_CNT may be is 65535.

```
void task2(void)
{
  unsigned char status;

  status = K_Task_Wait(100); /* set task2 to wait for 100 system
  ticks */
```

```
  if (status != K_OK) /* check status, make sure task was woken
  up before time elapsing */
    {
    /* maybe take corrective action if the time period expired,
    unless the task wanted to be synchronous with the system tick
    */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist.

K_TIMEOUT = Warning: the time specified has elapsed or the *K_Task_Wake_Force* function was used to wake this task.

If STATUS equals K_OK, then the task has RESUMED execution because the *K_Task_Wake* function was used to wake this task. If STATUS equals K_TIMEOUT, then the time period specified has expired or *K_Task_Wake_Force* and this is why the task was awakened.

> ✗ **WARNING: This function does not test to see if the caller is a task or not, so make sure that only tasks call this function.**

The status returned by this function will indicate that either the time period specified (if non zero) has elapsed, or that the task was awakened before the time period specified by the *K_Task_Wake* or *K_Task_Wake_Force* functions.

### *The K_Task_Wake function*

The *K_Task_Wake* function wakes a task that had put itself into the suspended state. Tasks and interrupts may call this function. The caller sends the task's slot number to the *K_Task_Wake* function. This function then takes the task and places it into the READY to RESUME execution state.

Note that the awakened task might not immediately become the RUNNING task because of its priority. If the task's priority is higher than the current RUNNING task, then an immediate task switch will occur regardless of the system tick.

This is an example of the *K_Task_Wake* function:

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

#include <cxfuncs.h> /* has function prototype */
byte *K_Task_Wake*(byte); /* this is the function prototype */

unsigned char TASK_SLOT; /* global, declared earlier */

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Wake*(TASK_SLOT);

**Passed**

TASK_SLOT is the name where the particular task's slot number resides.

unsigned char task1_slot; /* defined earlier, contains task 1 slot number */

```
void task2(void)
{
  unsigned char status;

  status = K_Task_Wake(task1_slot); /* wake task 1 up */

  if (status != K_OK) /* check status, make sure task was waiting
  */
    {
    /* maybe take corrective action if task 1 wasn't waiting */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist.

K_NOT_WAITING = Error: the task specified was not waiting.

If STATUS equals K_OK, then the task specified has been "awakened" and put into the READY state. If STATUS equals K_NOT_WAITING, then the task specified was not waiting for this function call.

### *The K_Task_Wake_Force function*

The *K_Task_Wake_Force* function closely resembles the *K_Task_Wake* function. It is used to forcefully wake a task that is suspended in the wait state. If you want a task to terminate its wait earlier than indicated by the function call, then you can use this function. The *K_Task_Wake_Force* function can be called by tasks and interrupts and should never be used except in certain circumstances and emergencies.

The difference between the *K_Task_Wake* and *K_Task_Wake_Force* functions is that the *K_Task_Wake* function will only wake a task if it is waiting on time or indefinitely. The *K_Task_Wake_Force* will wake a task regardless of what it is waiting for.

A task that had called the *K_Task_Wait*, *K_Event_Wait*, *K_Mesg_Wait*, *K_Resource_Wait*, or *K_Mesg_Send_Wait* functions will be put into the wait state if the entity it is 'waiting' for, is not present. This means the task is suspended and will not run again until the required action for the particular function call takes place or a time out occurs.

This is an example of the *K_Task_Wake_Force* function:

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

#include <cxfuncs.h> /* has function prototype */
byte *K_Task_Wake_Force*(byte); /* this is the function prototype */

unsigned char TASK_SLOT; /* global, declared earlier */

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Wake_Force*(TASK_SLOT);

**Passed**

TASK_SLOT is the name where the particular task's slot number resides.

unsigned char task1_slot; /* defined earlier, contains task 1 slot number */

```
void task2(void)
{
  unsigned char status;

  status = K_Task_Wake_Force(task1_slot); /* forcefully wake
  task 1 up */
```

```
  if (status != K_OK) /* check status, make sure task was waiting
  */
    {
    /* maybe take corrective action if task 1 wasn't waiting */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist.

K_NOT_WAITING = Error: the task specified was not waiting.

If STATUS equals K_OK, then the task specified has been "awakened" and put into the READY state. If STATUS equals K_NOT_WAITING, then the task specified was not waiting for this function call.

> ✗ **WARNING: This function should not be used except in emergencies. A well designed system will very rarely need to use this call, if at all.**

***The K_Task_Lock function***

The *K_Task_Lock* function is very powerful and great care must be used if this function is called. This function raises the privilege flag for the task that calls it. When the privilege flag is raised, the task owns all the CPU time; even if a higher priority task can run, it will not.

The task that owns the privilege flag should never suspend itself for any reason, because the system would not allow any other task to run. The task that called *K_Task_Lock* is the only task that may lower the privilege flag. If the privilege flag is raised, interrupts will still be processed and the system tick will still occur, but the CMX timer task (which decrements the task's timers and executes timed procedures) will be delayed until the task calls the *K_Task_Unlock* function which lowers the privilege flag.

Be aware that all tasks' time-outs will not be decremented and tested until the timer task runs. The interrupts are still processed normally, but the interrupt pipe contents, if any, will not be executed. The big difference is that when the CMX scheduler would normally start another task of higher priority that is READY to run, this will not happen. This function should be used very sparingly or not at all. This function may be called only by tasks.

This is an example of the *K_Task_Lock* function:

**Called**

Tasks only.

#include <cxfuncs.h> /* has function prototype */
void *K_Task_Lock*(void); /* this is the function prototype */

*K_Task_Lock*();

**Passed**

Nothing is passed. Only tasks can call this function.

```
void task2(void)
{
  K_Task_Lock(); /* task 2 will raise the privilege flag
  therefore owning all the CPU time, no scheduling will take
  place */

  /* perform code, don't suspend task */

  K_Task_Unlock(); /* lower the privilege flag*/
}
```

**Returned**

No status is returned.

☞ *The CMX timer task (which executes cyclic timers and handles the tasks timers) will not execute when the privilege flag is raised.*

> ✗ **WARNING: This function does not test to see if the caller is a task or not.**

***The K_Task_Unlock function***

This function lowers the privilege flag. Only the task that called *K_Task_Lock*, which raised the privilege flag, may call this function. Once the privilege flag is lowered, the CMX scheduler acts normally again.

This is an example of the *K_Task_Unlock* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_Task_Unlock*(void); /* this is the function prototype */

*K_Task_Unlock*();

**Passed**

Nothing is passed. Only tasks can call this function.

```
void task2(void)
{
  K_Task_Lock(); /* raise the privilege flag*/

  /* application code here */

  K_Task_Unlock(); /* task 2 will lower the privilege flag
  therefore allowing normal scheduling to take place */
}
```

**Returned**

No status is returned.

> ✗ **WARNING: This function does not test to see if the caller is a task or not, so you must ensure that only tasks call this function.**

### *The K_Task_Coop_Sched function*

The *K_Task_Coop_Sched* function does a cooperative rescheduling. Normally CMX reschedules when a higher priority task is READY to run. If a task calls this function then CMX schedules the next task that is READY to run despite its priority.

This allows a task to let another task of the same or lower priority become the RUNNING task (if the task is READY). The task that calls this function will immediately be placed into the READY to RESUME state. In most cases, this function would not be used because the operating system is based on preemption (meaning the highest priority task that can run is the RUNNING task). The task could call the *K_Task_Wait* function specifying a time out period of one, which would let another task with the same priority as the calling task become the RUNNING task. Only tasks should call this function.

This is an example of the *K_Task_Coop_Sched* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_Task_Coop_Sched*(void); /* this is the function prototype */

*K_Task_Coop_Sched*();

**Passed**

Nothing is passed. Only tasks can call this function.

```
void task2(void)
{
  /* application code here */
  K_Task_Coop_Sched(); /* allow a rescheduling now, do not wait
  for normal preemption to perform a normal scheduling, NOTE:
  task context is saved completely and will RESUME when task is
  highest priority READY to run */
}
```

**Returned**

No status is returned.

### *The K_Task_End function*

This function allows a task to terminate itself either prematurely or at the end of its code. The *K_Task_End* function MUST be called by all tasks that normally would hit their "end brace". If a task calls this function before its end brace, then all variables, pushes and calls on the stack will be forgotten. This function is also useful for exiting out of a serious or nonrecoverable error. The task could first pass a message to another task with the message saying what task and what type of error and then call *K_Task_End*.

When the *K_Task_End* function is called, the task is automatically terminated, resetting its stack pointer and code pointer to the task's beginning. The task is still able to execute whenever it is started again or if there were additional starts (see *K_Task_Start* function) in the start queue. Only tasks should call this function. If there are additional starts in the task's start queue, the task will automatically be placed into the READY state, and may again immediately be the RUNNING task, if it's the highest priority task READY to run.

☞ *If, because of an indefinite while statement, the task never hits its end brace, then the task does not need the K_Task_End function call at the task's end brace.*

If a task about to call the *K_Task_End* function retrieved a message from a mailbox and the sending task used the *K_Mesg_Send_Wait* function (indicating that it is waiting for the *K_Mesg_Ack_Sender* call) you must remember to call the *K_Mesg_Ack_Sender* function (which wakes the task that sent the message), before calling *K_Task_End*.

This is an example of the *K_Task_End* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_Task_End*(void); /* this is the function prototype */

*K_Task_End*();

**Passed**

Nothing is passed. Only tasks can call this function.

```
void task2(void)
{
  ..../* application code here */

  K_Task_End(); /* MUST be the last C statement before the task's
  end brace */
}
```

OR

```
void task2(void)
{
  ..../* application code here */

  if (???) /* serious error or non recoverable */
    {
    K_Task_End(); /* End task 2, because of serious or non-
    recoverable error */
    }
  ..../* more code here */
  K_Task_End(); /* normal exit function here */
}
```

Or if a task is never going to hit its end brace, such as below.

```
void task2(void)
{
  while(1)
    {
    ..../* application code here */
    }
  /* K_Task_End function NOT NEEDED here */
}
```

**Returned**

No status is returned.

☞ *An immediate rescheduling (with possibly this task executing from its beginning brace) will occur when this function is called. If the task has additional trigger (K_Task_Start) requests, then the task will be put into the READY state, otherwise the task will become IDLE. Remember, all local variables will be lost. Other tasks may be waiting for this task to use a CMX function call to wake it. A task that owns a resource should never call this function before releasing the resource.*

> ✗ **WARNING: ALL tasks must call this function prior to executing their right-end brace. If the task will never, for whatever reason, execute its end brace, then this function does not have to be called. It is highly recommended that all tasks have this function.**

***The K_Task_Delete function***

This function gives you the ability to remove a task permanently from the task control block queue. The calling task will send the slot number of the task to be removed (it may send its own slot number if desired) to the *K_Task_Delete* function.

If the task requested to be removed is waiting, then no removal will take place and an error status will be returned. If the task is not waiting, then the task will be completely removed. This task will never run again and any further request to this task will result in an error status being returned to caller.

If the task calling the *K_Task_Delete* function is trying to remove itself, then it will be removed and an immediate task switch will occur. Please note: all tasks waiting on the removed task will not be notified when the task is removed and may wait forever. For example: if task 1 has sent a message to task 2 using the *K_Mesg_Send_Wait* function and task 2 is removed by task 3 before task 2 calls the *K_Mesg_Ack_Sender* function (waking up task 1), then task 1 will wait forever. To avoid this, you could use the *K_Task_Wake_Force* function to forcefully wake task 1.

This is an example of the *K_Task_Delete* function:

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Task_Delete(byte); /* this is the function prototype */

unsigned char TASK_SLOT; /* global, declared earlier */

unsigned char STATUS; /* should be local */

STATUS = K_Task_Delete(TASK_SLOT);
```

**Passed**

TASK_SLOT is the name where the particular task's slot number resides.

unsigned char task2_slot; /* defined earlier, contains task 2 slot number */

```
void task2(void)
{
  /* application code here */
  K_Task_Delete(task2_slot); /* remove task 2, any further
  reference to task 2 will be in error */
}
```

**Returned**

STATUS returned is one of the following:

☞ *Only if the task is not removing itself*

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist or the task was in the WAIT state.

If STATUS equals K_OK, then the task has been removed successfully. If the task is removing itself, then an immediate task switch will occur.

> ✗ **WARNING: Make sure the task about to be removed does not own a resource. This function does not test to see if a task is calling it so you must also ensure only tasks call this function.**

***The K_Task_Name function***

This function gives you the ability to name a task, thus helping the user to know the name of the task, when working with CMX add on modules, such as CMXBug and/or CMXTracker. The function is called with the slot number of the task to be named and a pointer to the task's name that the user wants to name it. Note that the tasks name can be as long as the user would like, but only the first 12 characters of the task's name will be displayed by CMXBug and CMXTracker. This is an example of the *K_Task_Name* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Task_Name*(byte, char *); /* this is the function prototype */

unsigned char TASK_SLOT; /* global, declared earlier */

char *TASK_NAME; /* The address of the user defined task name */

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Delete*(TASK_SLOT,TASK_NAME);

**Passed**

TASK_SLOT is the name where the particular task's slot number resides.

TASK_NAME is the address of where the user defined task name resides in memory. Note that the name may be any number of characters, but only the first 12 characters of the task's name will be shown by CMXBug and CMXTracker.

unsigned char task2_slot; /* defined earlier, contains task 2 slot number */

```
void task2(void)
{
  /* application code here */
  K_Task_Name(task2_slot,"TASK2"); /* Name task 2, utilized by
  CMXBug and CMXTracker when displaying tasks */
}
```

**Returned**

STATUS returned is one of the following:

☞ *Only if the task is not removing itself*

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist or the task was in the WAIT state.

If STATUS equals K_OK, then the task has been successfully named.

## EVENT MANAGER FUNCTIONS

The event manager is part of the CMX library and provides the necessary functions for controlling events. The event manager functions are listed below along with their page references.

*K_Event_Wait (Page 30)*
*K_Event_Signal (Page 34)*
*K_Event_Reset (Page 39)*

The CMX event functions are very sophisticated and powerful, yet easy to use. Each task has 16 events. Each event is considered a bit, with the bit capable of being either set or cleared. An event, or bit, that is set, indicates the event occurred. Tasks can wait on any combination of events. When any event the task is waiting on occurs, the task is awakened and notified which event occurred. Also, a task can specify a time-out period to wait for, or to wait indefinitely.

The task can specify the event it is waiting for be automatically cleared prior to waiting for that event (which means the task will become suspended), automatically cleared after the event occurred, and the task is awakened, or not cleared at all. Tasks also can call a function that will reset an event at any time.

The signaling of an event is very flexible and can be done by tasks, interrupts, cyclic timers and mailboxes. You can specify to set an event in seven different ways:

a specific task
the highest priority task
the highest priority task that is waiting for this event
all tasks
all tasks waiting on this event
all tasks with the same priority
all tasks with the same priority waiting on this event.

### *The K_Event_Wait function*

This function allows a task to wait for specific events with a specified time out if so desired. This function is very flexible and powerful because it allows a task to wait for one or more events to happen. It also allows the task to specify a certain amount of time it is willing to wait, so it can take corrective action if the events do not become set. Also, the clear mode command offers the task the ability to be truly synchronized or not with respect to the entities that may set an event such as other tasks, mailboxes, cyclic timers and interrupts.

The task supplies the *K_Event_Wait* function with three parameters:

1.) the event(s) to wait on

2.) the command whether to automatically clear the event(s) the task wants to wait on prior to testing, after the event(s) happen, both prior and after or not at all

3.) A time period indicating whether to wait for a certain amount of time or indefinitely for a match. A time period of zero means that the task will wait indefinitely for the event(s) to occur.

The specified events are tested when this function is called. If one or more of the events specified are set when the *K_Event_Wait* function is called, the task is returned to immediately identify which of the events are a match (set). If the events are not present when the task calls this function, then the task will wait for the events. The task will automatically resume (wake up) when at least one of the specified events is set or the specified time period expires. Also, the *K_Task_Wake_Force* function could be used to forcefully wake this task, without the events being set or the time period expiring.

**The clear mode command**

The clear mode command has four different possibilities.

**A clear mode value of 0 (zero)** indicates not to clear the task's event flags at all when the *K_Event_Wait* is called and used. This means the task's events will stay in their present state.

**A clear mode value of 1** indicates the *K_Event_Wait* function will automatically clear the task's events it is waiting on. This means if the task is about to wait for events 0 and 2, then those events will be cleared prior to testing to see if there is a match. This will result in the task becoming suspended for there will be no match.

**A clear mode value of 2** will have the *K_Event_Wait* function automatically clear the events the task is waiting for after there is a match. The task will still be notified what events became set that the task was waiting for and those event states will be cleared. Of course if the specified time out period expires or the task is awakened by the *K_Task_Wake_Force* function, then the event states will be left as they were, in the clear state.

**A clear mode value of 3** instructs the *K_Event_Wait* function to perform clear mode values 1 and 2. This means the events the task is about to wait on will be forced to the clear state and when the task resumes, because at least one of the events it was waiting on happened, then those events will be cleared. Clear mode values 1 and 2 are explained in the above two paragraphs.

Remember, the *K_Event_Wait* function works and manipulates only the events the calling task is requesting. This means all of the task's other event states, will be left in their state. Also remember, each task has 16 events and the *K_Event_Wait* function will not manipulate the other task's event states.

Only tasks may call this function. When called, the task will be suspended until the required match takes place or the time out period expires (if time period was non zero). When an event becomes set that the task is waiting on, the task will automatically be put back into the RESUME state, again ready to RESUME running. The value returned will indicate what events were set that the task was waiting for. Keep in mind that possibly more than one event may be set and identified to the calling task when the task resumes. This is because the task may not immediately become the RUNNING task because of its priority.

If the time period expires, the task will be placed into the RESUME state and its value returned to zero indicating a time out occurred. When a time out occurs, it allows the task to possibly take corrective action because the events did not take place within the amount of specified time.

If a task calls *K_Event_Wait* with a clear mode of 0 (zero) or 2, and the event flags are already set to the specified criteria, then the task will not be suspended. If you want the task to wait for only one event, then use the *K_Task_Wait* and the *K_Task_Wake* functions. These functions are a little faster in execution time because the *K_Task_Wake* function deals with one specific task where as the *K_Event_Signal* function could possibly deal with several tasks.

This is an example of the *K_Event_Wait* function:

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
word16 K_Event_Wait(word16,word16,byte); /* this is the function prototype */

#define MATCH ???

#define TIME_CNT ???

#define MODE ???

unsigned short EVENTS; /* should be local */

EVENTS = K_Event_Wait(MATCH,TIME_CNT,MODE);
```

**Passed**

MATCH is a 16 bit wide parameter indicating the specific events that this task would like to have set. More than 1 event bit may be specified.

TIME_CNT is the number of system ticks to wait for a match. If the value is 0 then the task will wait indefinitely for event match. The maximum value is 65535.

MODE is the mode in which this function will clear event bits, when they are set or become set. The values are below.

0 = do not clear the event bits.

1 = clear the event bit(s) according to the ones set within the MATCH parameter at BEGINNING of function.

2 = clear the event bit(s) according to the ones set within the MATCH parameter at END of function.

3 = do both modes 1 and 2.

#define TSK2_EVENT1 0x01

#define TSK2_EVENT3 0x04

#define TSK2_EVT1AND3 TSK2_EVENT1 | TSK2_EVENT3

```
void task2(void)
{
  unsigned short events;

  /* application code here */
  /* NOTE: could use any 1 of 3 ways to identify the events */
  events = K_Event_Wait(TSK2_EVENT1 | TSK2_EVENT3,100,2);
  events = K_Event_Wait(TSK2_EVT1AND3,100,2);
  events = K_Event_Wait(0x05,100,2);
  /* task 2 will wait for the events 0 and/or 2 to be set.  Task
  2 will wait for 100 system ticks for this match to happen. Also
  task 2 is requesting that the clear mode command be of value
  2, which indicates that the events 0 and 2 will be
  automatically cleared when a match happens and the task RESUMES
  execution.  If the time period expires, then the value returned
  to task 2 will be 0, indicating that the time period expired,
  prior to the events 0 or 2 being set. */
  if (events == 0) /* test to see if error */
    {
    /* events 0 or 2 did NOT become set, take corrective action,
    within the specified time period or the K_Task_Wake_Force
    function was used on this task. */
    }
  else
    {
    if (events & TSK2_EVENT1)
      {
      /* application code here for event 0 being set */
      }
    if (events & TSK2_EVENT3)
      {
      /* application code here for event 2 being set */
      }
    }
}

  NOTE: you could set up a switch statement by doing the
  following */
  MASK = 0x0001;
```

```
  for (ctr = 0; ctr < 16; ctr++)
    {
    switch (events & MASK)
      {
      case 0x0001:
      .....
      break;
      case 0x0002:
      .....
      break;
      case 0x0004:
      .....
      break;
      case 0x0008:
      .....
      break;
      etc....

      case 0x8000:
      .....
      break;
      default:
      break;
      }
    MASK = MASK << 1;
    }
}
```

**Returned**

EVENTS will either contain a zero indicating the time period specified expired before any of the events the task is waiting on became set, or the specific events that were set only according to the MATCH parameter.

☞ *Remember only events that are selected by the MATCH parameter are worked within this function. The MODE parameter allows powerful synchronization as to when the task's events are cleared.*

***The K_Event_Signal function***

The *K_Event_Signal* function sets a specific event. This can be done in a variety of modes. This function may be called by tasks, cyclic timers, mailboxes or interrupts. The caller will select which event, the mode of event set it wants, and the task slot number or priority, depending upon the mode selected. Some cyclic timer and mailbox functions use *K_Event_Signal* to signal that time expired or mail has arrived at the mailbox, respectively.

Only a single event should be set each time by the *K_Event_Signal* function. Technically you could call the *K_Event_Signal* function with more than one event to be set. CMX recommends that you do not however, because there is rarely any reason to set more than one event.

### Mode values in the *K_Event_Signal* function

The caller will specify either the task slot number, the priority or an unused value it wants the *K_Event_Signal* function to work with, depending upon the mode command.

The caller will also specify the mode of action the *K_Event_Signal* function will perform. There are seven different ways the function will work. The mode will specify one of the following:

| Mode # | Acts On | Task slot # or priority |
|---|---|---|
| 0 (zero) | The Specified task | Task slot # |
| 1 | Highest priority task | Not used |
| 2 | Highest priority task waiting on the specified event | Not used |
| 3 | All tasks | Not used |
| 4 | All tasks waiting on the specified event | Not used |
| 5 | All tasks with the specified priority | Priority |
| 6 | All tasks with the specified priority waiting on the specified event | Priority |

**A mode value of 0 (zero)** sets the specified event of the specified task by passing the task's slot number. The specified task does not have to be waiting on that event. This is the fastest, for the *K_Event_Signal* function does not have to test many variables or loop. This mode can be used very effectively by interrupts, cyclic timers, and mailboxes. Mailboxes, however, can only use this mode to notify a particular task that there are messages (mail) in the mailbox (see Message Management chapter for more details).

**A mode value of 1** sets the highest priority task event. In this mode, the *K_Event_Signal* function automatically finds the highest priority task created and sets the specified event of that task. If two or more tasks have the highest priority, then the first one created by the *K_Task_Create* function will have its event set. (Note that the parameter that normally specifies either the task slot number or priority is not used and may be any value.)

**A mode value of 2** indicates that the *K_Event_Signal* function will find the highest priority task that is WAITING for this event to be set. This will make the function start at the highest priority task, test to see if the task is waiting for this event (because the task used the *K_Event_Wait* function) and if not will continuously go to the next highest priority task and perform the test again. When either the task is found (because it is waiting on the event) or ALL tasks have been tested, then the function will quit. (Again note that the parameter that specifies the task slot number or priority may contain any value, for it is not used in this case.)

**A mode value of 3** tells the *K_Event_Signal* function to set the specified event in ALL tasks. The function will loop through all tasks setting the specified event, regardless of whether the task is waiting for this event to be set or not. Remember that this will take a fair amount of time depending upon the number of tasks created. (The parameter that represents the task slot number or priority is not used and may be any value.)

**A mode value of 4** sets the event of all tasks WAITING on this event. The *K_Event_Signal* function will loop through testing all tasks. If the task is waiting for this event, then the task specified event will be set. When the function has tested all tasks, then it will have finished. (Again the parameter that specifies the task's slot number or priority is not used in this case and may be any value.)

**A mode value of 5** causes the *K_Event_Signal* function to set the specified event of all tasks that have the same priority as the one specified. The task slot number or priority parameter must contain the priority of the tasks the caller would like to have their event set. The tasks do not have to be waiting for this event to be set. The function will loop through, testing to see if each task has the same priority as the one specified and if so, will set that task's specified event. Note that if none of the tasks have the same priority as the specified priority, then no task will have their event set.

**A mode value of 6** indicates that any task with the same specified priority and WAITING on a specified event, will have its event set. The *K_Event_Signal* function will test each task and if it meets the criteria, the task will be put into the READY to RESUME state.

When the *K_Event_Signal* function is called, if a task is waiting on the specified event, and the mode selects this task, then the following will occur. The task that is waiting on the event, will have its event set. This will then place the task into the READY to RESUME state, indicating that the task is able to resume its code where it left off. Also, if the task is just awakened because the event it was waiting on occurred and has a higher priority than the current RUNNING task, then rescheduling will be done after the *K_Event_Signal* function finishes, creating a task switch. If a task is not waiting for the specified event and has its event set, its state will stay the same.

Remember the tasks, cyclic timers, mailboxes or interrupts that calls the *K_Event_Signal* function, do not have to know whether any task is waiting for an event match, or that one or more tasks are waiting on this event. The *K_Task_Wait* and *K_Task_Wake* functions can be used if only single "event" synchronization is needed. These are slightly faster.

This is an example of the *K_Event_Signal* function:

**Called**

Tasks, interrupts, cyclic timers and mailboxes

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Event_Signal(byte, byte, word16); /* this is the function prototype */

#define MODE ???

#define EVENT_TO_SET ???

unsigned char TASK_PRI;

unsigned char STATUS; /* should be local */

STATUS = K_Event_Signal(MODE,TASK_PRI,EVENT_TO_SET);
```

**Passed**

MODE is the mode in which this function will determine which tasks to work with. The values are as below.

| Mode | Tasks to Work With |
|---|---|
| 0 | the specified task |
| 1 | highest priority task, excluding CMX timer task |
| 2 | highest priority task, that is WAITING for the specified event. |
| 3 | all tasks, all tasks created will have this event bit set |
| 4 | all tasks WAITING on the specified event |
| 5 | all tasks with the specified priority |
| 6 | all WAITING tasks with the specified priority and specified event |

☞ *The following parameter is not always used depending on the MODE.*

TASK_PRI is either the task slot number or the priority in which this function will work with according to the MODE selected.

EVENT_TO_SET is an unsigned 16 bit wide variable or constant indicating the desired event bit to set.

#define TSK2_EVENT1 0x01

#define Mode_0 0x00

unsigned char task2_slot

```
void task1(void)
{
  unsigned char status;

  /* application code here */
  status = K_Event_Signal(Mode_0,task2_slot,TSK2_EVENT1);
  /* task 1 will now set task 2's event (really bit 0), does not
  care if task 2 is waiting for event or not, if task2 is waiting
  on this event, then task 2 will automatically resume. */
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task does not exist if MODE = 0, or the MODE is out of range.

If STATUS equals K_OK, then the function performed as the MODE indicated it should.

Remember that cyclic timers and mailboxes can be coded to automatically call this function. Also interrupts may call this function.

The above two functions, *K_Event_Wait* and *K_Event_Signal*, allow a task or interrupt to signal other tasks and interrupts without the calling task or interrupt knowing that a task is waiting for its signal. For example, if task 1 is waiting for a pneumatic valve to close to continue its job responsibilities, and task 2 is waiting for task 1 to receive this signal before task 2 can go any further with its code, then task 2 would call the *K_Task_Wait* function, identifying that it will wait indefinitely, until a task or interrupt wakes it up.

Task 1 would then call the *K_Event_Wait* function indicating a match of the event flag, meaning the pneumatic valve was closed. This is sensed by an external sensor feeding its output to an interrupt. When the sensor indicated the valve had closed and generated an interrupt, the interrupt code could call *K_Event_Signal* with the proper parameters to indicate the event had happen and to set an event in task 1.

At the time of the *K_Event_Signal* call, task 1 would automatically leave the suspended state and become READY again. Task 1 then could use the *K_Task_Wake* function to notify task 2 that it had received the signal and the proper global variables were set up for task 2 to continue processing. Possibly, if the interrupt did not happen and task 1 had used the time out period of non 0, then at the end of the time period task 1 would become READY. When the task started RUNNING again it could look at the return value and if a time out had occurred by the return value being 0 (zero), it could take corrective action, such as sounding an alarm.

### *The K_Event_Reset function*

The *K_Event_Reset* function allows a task to clear one or more specific events of a task. This means that a task can clear another task's specific event or its own. This gives you the ability to clear events, if not done so with the clear mode command within the *K_Event_Wait* function. Note, however, that tasks do not wait on events to become clear, so the respective task will not know that one of its event states was cleared unless the task is clearing its own.

This is an example of the *K_Event_Reset* function:

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Event_Reset(byte tskid,word16 event); /* this is the function prototype */

unsigned char TASK_SLOT; /* should be global, slot number of task */

unsigned short EVENTS_TO_CLEAR; /* the event bits to clear, or could be a #define

#define EVENTS_TO_CLEAR ??? instead of variable. */

unsigned char STATUS; /* should be local */

STATUS = K_Event_Reset(TASK_SLOT,EVENTS_TO_CLEAR);
```

**Passed**

TASK_SLOT is the slot number of the task for which the event bit(s) will be cleared.

EVENTS_TO_CLEAR is a unsigned 16 bit wide variable or constant indicating the desired event bits to clear within this task.

#define TSK1_EVENT1 0x0001

unsigned char task1_slot;

```
void task1(void)
{
  unsigned char status;

  status = K_Event_Reset(task1_slot,TSK1_EVENT1);
  /* task 1 is requesting to clear its own event bit 0. */

  .../* more task 1 code */
  K_Task_End(); /* notify CMX that the task is done */
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task does not exist.

If STATUS equals K_OK, then the event bits were cleared according to the EVENTS_TO_CLEAR parameter passed.

☞ *This function clears just the event bits within the task that is being referenced. It does not change any of the other task event bits.*

## QUEUE MANAGER FUNCTIONS

The queue manager is part of the CMX library and provides the necessary functions for controlling queues. The queue manager functions are listed below along with their page references.

### *The K_Que_Create function*

This function is use to create a circular queue. You supply the number of slots or records for this particular queue. A maximum of 32767 slots per queue is allowed.

Another parameter is the number of bytes to allocate to each slot. You decide on the size of each slot within this queue. The maximum size of each slot must be no more than 255 bytes, and the size selected is the size for all the slots within this particular queue. The *K_Que_Create* function may be called before entering the CMX operating system or from within.

You supply the beginning address of external RAM where the queue will reside.

☞ *On CPUs that require pointers or integers to reside on an even or odd address, the queue's address supplied to the K_Que_Create function may have to reside on that particular address boundary. The number of bytes per slot may also be required to be a multiple of the alignment size.*

The memory needed for a particular queue calling the *K_Que_Create* function is the number of slots times the number of bytes per slot. CMX does not test to see if you have properly allocated the correct number of bytes for this queue. If the queue size is smaller than the proper calculation of slot size times the number of slots, then unknown results are possible.

Memory contention is not checked in this function. The system assumes all the memory needed for this queue is free at all times. This function allows tasks to add and remove slots to and from the queue at any time.

The queue number, which all queue functions will use, must also be supplied. This is a number that identifies which queue the function is going to work with, so it may properly manipulate the CMX queue's structure. This number ranges from zero to one less than the maximum number of queues told to CMX by your configuration file.

Understand that the queue is maintained with each slot acting as an array of unsigned characters, allowing the storage for characters, integers, pointers, longs, etc.

If you want to pass a large number of bytes to a queue, it is recommended you pass the pointers of the source bytes to the queue. This requires the size of the slots be only the size of a pointer. This will make the queue very fast in execution speed, but then the storage location of the source bytes can not be used until you have removed that pointer and used those bytes accordingly.

The slot size may be 20 bytes in length, but not all 20 bytes need be filled each time you add to this queue. For example, using variable length records, the maximum record being 19 bytes, you could set up a structure where the first byte would be the number of bytes for this record (slot) and the remaining bytes be the record bytes. Then you could pass the address of this structure to add to the queue functions.

CMX would copy all 20 bytes, regardless if some of the last few bytes may be immaterial or belong to another data item. The task receiving the slots from the queue would test the count byte and work with just the bytes identified. Again make note, CMX suggests using pointers if at all possible, instead of a large slot size. But there are times when the data should be truly copied and passed.

This is an example of the *K_Que_Create* function:

**Called**

Before entering RTOS and tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Que_Create(sign_word16,byte,byte *,byte); /* this is the function prototype */

#define NUM_SLOTS ???

#define SIZE_SLOT ???

#define QUE_NUM ???

unsigned char STATUS;

unsigned char QUE_NAME[NUM_SLOTS * SIZE_SLOT];

/* should be global */

STATUS = K_Que_Create(NUM_SLOTS,SIZE_SLOT, QUE_NAME,QUE_NUM);
```

**Passed**

NUM_SLOTS is the number of slots that this particular queue will have and the maximum is 32767.

SIZE_SLOT is the number of bytes that each slot will hold (size of each slot) within this queue.

QUE_NAME is the beginning address where this queue will reside in memory.

QUE_NUM is the queue identification number that all queue functions will use in determining the queue's memory location.

```
#define QUE1_SLOTS 50 /* this queue will have 50 slots */

#define QUE1_SIZE 10 /* each slot will be 10 bytes in length */

#define QUE1 1 /* identifier, identifies queue number 1, needed for functions, to
identify to CMX with queue to work with. */
```

unsigned char queue1[QUE1_SLOTS * QUE1_SIZE]; /* allocate storage for queue */

```
void task2(void)
{
  unsigned char status;

  /* application code here */
  status = K_Que_Create(QUE1_SLOTS,QUE1_SIZE,queue1,QUE1);

  /* create a circular queue with 50 slots (records) within this
  queue and each slot capable of storing 10 bytes worth of
  information.  Also pass down the queue beginning address. */
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful, queue created.

K_ERROR = Error: queue number out of range.

☞ *Remember that it is up to you to ensure enough memory for this queue exists. No memory contention is tested for.*

***The K_Que_Reset function***

The *K_Que_Reset* function is used to reset a queue to the empty state. This function when called will free all used slots, permanently deleting all slots that contained information. This function may be called by a task.

The only parameter the *K_Que_Reset* function needs is the queue identification number. This number ranges from zero to one less than the maximum number of queues that CMX was told.

This is an example of the *K_Que_Reset* function:

**Called**

Before entering RTOS and by tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Que_Reset*(byte); /* this is the function prototype */

#define QUE_NUM ???

unsigned char STATUS; /* should be local */

STATUS = *K_Que_Reset*(QUE_NUM);

**Passed**

QUE_NUM is the queue number, which identifies a particular queue, that was created with the *K_Que_Create* function. This can be from zero to one less than the maximum configured.

#define QUE1 1 /* queue 1 identifier */

```
void task2(void)
{
  unsigned char status;
  /* application code here */
  status = K_Que_Reset(QUE1); /* reset queue1, all slots are now
  free */
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: queue was empty, no slot available.

***The K_Que_Add_Top function***

This function is used to fill the top free slot of a particular queue. The caller sends the *K_Que_Add_Top* function the queue number in which they want to add. Also passed is the source's beginning address byte resides regardless of what will be copied, whether it is a byte, integer, pointer, long, etc.

Understand that if you create a queue with a slot size of two bytes, and pass the address of the long, then just the first two bytes of the long will be copied. In the same respect, if you pass the address of just a byte, then that byte and the next contiguous byte will be copied.

The *K_Que_Add_Top* function takes the next top free slot, if one is available, in this circular queue and copies the contents from the supplied address into the queue slot (according to the slot size for that particular queue as supplied by the *K_Que_Create* function). This frees your address bytes for other uses.

The *K_Que_Add_Top* function tests the validity of the queue number and also tests to see if the queue slot size is greater than zero. If the C compiler zeros out all non-initalized RAM, this will ensure the queue has been created. The slot used by this function will always be from the top of the circular queue. Note that the queue will wrap. Tasks may call this function.

This is an example of the *K_Que_Add_Top* function:

**Called**

Before entering RTOS and tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Que_Add_Top*(byte,void *); /* this is the function prototype */

#define QUE_NUM ???

unsigned char *SOURCE_POINTER; /* could be local or global depending upon application and user. */

unsigned char STATUS; /* should be local */

STATUS = *K_Que_Add_Top*(QUE_NUM,SOURCE_POINTER);

**Passed**

QUE_NUM is the queue number, which identifies a particular queue, that was created with the *K_Que_Create* function. This can be from zero to one less than the maximum configured.

SOURCE_POINTER is a pointer that should contain the address of the source bytes.

#define QUE1 1 /* queue 1 identifier */

```
void task2(void)
{
  unsigned char status;

  status = K_Que_Add_Top(QUE1,"hello world"); /* add this
  message to top of queue */
}
```

**Returned**

The status of this operation will return one these status codes:

K_OK: good operation, function was successful

K_QUE_FULL: the operation was good and now the queue is full

K_ERROR: an error indicating the queue was full or that the queue number was out of range or not created.

☞ *Remember the pointer may point to anything. This may be done by casting, so actually longs could be passed, other pointers, etc. Remember that it is up to you to ensure that the queue number to this function is the queue in which they want to add this to, and that the queue had been created.*

***The K_Que_Add_Bottom function***

This function is used to fill the bottom free slot of a particular queue. The caller sends to the *K_Que_Add_Top* function the queue number in which they want to add. Also passed is the address where the source's beginning byte resides regardless of what will be copied, whether it is a byte, integer, pointer, long, etc.

Understand that if you create a queue with a slot size of two bytes and pass the address of the long, then just the first two bytes of the long will be copied. In the same respect, if you pass the address of just a byte then that byte and the next contiguous byte will be copied.

The *K_Que_Add_Bottom* function takes the next bottom free slot, if one is available, in this circular queue and copies the contents from the supplied address into the queue slot (according to the slot size for that particular queue as supplied by the *K_Que_Create* function). This frees your address bytes for other uses.

The *K_Que_Add_Bottom* function tests the validity of the queue number. The slot used by this function will always be from the bottom of the circular queue. Tasks may call this function.

This is an example of the *K_Que_Add_Bottom* function:

**Called**

Before entering RTOS, and by tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Que_Add_Bottom(byte,void *); /* this is the function prototype */

#define QUE_NUM ???

unsigned char *SOURCE_POINTER; /* could be local or global depending upon
application and user. */

unsigned char STATUS; /* should be local */

STATUS = K_Que_Add_Bottom(QUE_NUM,SOURCE_POINTER);
```

**Passed**

QUE_NUM is the queue number, which identifies a particular queue created with the *K_Que_Create* function. Can be from zero to one less than the maximum configured.

SOURCE_POINTER is a pointer that should contain the address of the source bytes.

#define QUE1 1 /* queue 1 identifier */

```
void task2(void)
{
  unsigned char status;

  status = K_Que_Add_Bottom(QUE1,"12345\n"); /* add this message
  to bottom of queue */
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: queue was full, no slot available.

K_QUE_FULL = Warning: queue is now full, slot was filled.

If the STATUS equals K_OK, the source contents were copied into the queue slot. This is also true if the STATUS equaled K_QUE_FULL, indicating that the queue is now full.

### *The K_Que_Get_Top function*

This function allows a task to remove the contents of a slot from a particular queue. The slot that the contents will be copied from and returned to the caller is from the very last top slot filled or used by the *K_Que_Add_Top* function.

The caller accesses the *K_Que_Get_Top* function by passing parameters indicating the queue number and the address where the slot's contents will be copied to. This function then copies the contents of the last top slot used to the specified address, according to the slot size of that queue, and releases the slot so it may be used again.

No memory testing is done to insure the address supplied is truly a queue address.

This is an example of the *K_Que_Get_Top* function:

**Called**

Before entering RTOS and tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Que_Get_Top*(byte,void *); /* this is the function prototype */

#define QUE_NUM ???

unsigned char *DEST_POINTER; /* could be local or global depending upon application and user. */

unsigned char STATUS; /* should be local */

STATUS = *K_Que_Get_Top*(QUE_NUM,DEST_POINTER);

**Passed**

QUE_NUM is the queue number, which identifies a particular queue, that was created with the *K_Que_Create* function. This can be from zero to one less than the maximum configured.

DEST_POINTER is a pointer that should contain the address where the slot bytes will be copied to in memory.

#define QUE1 1

```
void task2(void)
{
  unsigned char status;
  unsigned char dest[20]; /* create area that will obtain
  contents from queue */

  status = K_Que_Get_Top(QUE1,dest); /* remove the contents from
  last top slot filled (used) */

  ..../* process contents of dest */
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: queue was empty, no slot available.

K_QUE_EMPTY = Warning: queue is now empty, the slot contents were transferred.

If the STATUS equals K_OK, the slot contents were copied to the destination's address. This is also true if the STATUS equaled K_QUE_EMPTY, indicating that now the queue is empty.

☞ *Remember the slots may contain anything, bytes, integers, pointers, etc. Once the contents from a slot are removed, you may cast those bytes into what you would like. It is up to you to ensure that the queue number of this function is the same queue you added this to, and that it has been created.*

***The K_Que_Get_Bottom function***

The *K_Que_Get_Bottom* function acts exactly as the above *K_Que_Get_Top* function with one exception. The slot the contents are copied from and returned to the caller is the last one added to the bottom of the queue instead of the last one added to the top of the queue as in the *K_Que_Get_Top* function. This function allows a task to remove the contents of a slot from a particular queue.

The caller accesses the *K_Que_Get_Bottom* function by passing the parameters indicating the queue number and the address where the slot's contents will be copied to. This function then copies the contents of the last bottom slot used to the specified address according to the slot size of that queue, and release the slot so it may be used again.

☞ *No memory testing is done to insure the address supplied is truly a queue address.*

The queue functions allow tasks to pass parameters to other tasks. You can use a queue in many ways such as FIFO (first in, first out), LIFO (last in, first out), or a combination. The contents of a slot from a queue may be a variety of things: integers, pointers, commands to the command task, error codes to the error task, the address location where other parameters are held that need processing by a task, and so forth. You may mix and match the *K_Que_Get_Top* and *K_Que_Get_Bottom* functions together when dealing with a common queue.

Remember, when a slot is removed from either the top or bottom of a queue the slot is considered empty and the queue marks it as free. No other remove functions will have access to this slot until the slot becomes used again.

The larger the slot size for a particular queue, the longer it will take you to copy the contents from the source to the slot, or slot to destination. It is recommended that you keep the slot size low and use pointers for large data blocks, then just pass and remove the pointers to the queue.

This is an example of the *K_Que_Get_Bottom* function:

**Called**

Before entering RTOS and tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Que_Get_Bottom(byte,void *); /* this is the function prototype */

#define QUE_NUM ???
```

unsigned char *DEST_POINTER; /* could be local or global depending upon application and user. */

unsigned char STATUS; /* should be local */

STATUS = *K_Que_Get_Bottom*(QUE_NUM,DEST_POINTER);

**Passed**

QUE_NUM is the queue number, which identifies a particular queue was created with the *K_Que_Create* function. Can be from zero to one less than the maximum configured.

DEST_POINTER is a pointer that contains the address where the slot bytes will be copied to in memory.

#define QUE1 1

```
void task2(void)
{
  unsigned char status;
  unsigned char dest[20]; /* create area that will obtain
  contents from queue */

  status = K_Que_Get_Bottom(QUE1,dest); /* remove the contents
  from last bottom slot filled (used) */

  ..../* process contents of dest */
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: queue was empty, no slot available.

K_QUE_EMPTY = Warning: queue is now empty, the slot contents were transferred.

If the STATUS equals K_OK, the slot contents were copied to the destination's address. This is also true if the STATUS equaled K_QUE_EMPTY, indicating that now the queue is empty.

☞ *Remember the slots may contain anything, bytes, integers, pointers, etc. Once the contents from a slot are removed, you may cast those bytes into what you would like. It is up to you to ensure the queue number of this function is the same queue you added this to, and that it has been created.*

## MEMORY MANAGER FUNCTIONS

The memory manager is part of the CMX library and allows you to create fixed block size memory pools, get a free block from a memory pool and release that block back to the memory pool. The functions are listed below along with their page references.



### *The K_Mem_FB_Create function*

This function creates a memory pool. No memory contention is checked for by this function. It is assumed the pool's memory is free to be used and will not be used by code except through the memory function supplied.

When you call the *K_Mem_FB_Create* function, you supply the following parameters. First is the starting address for this memory pool. Next is the size in bytes you want each block size to be within this memory pool. The last parameter is the number of blocks that will reside within the memory pool.

If you are dealing with a CPU that needs integers and pointers aligned on either even or odd memory address, the memory block address should also be aligned on that address. Also, the size of the blocks must be an even number. This is because pointers are maintained within each memory pool as to where the next free block resides.

When a memory pool is created, an additional number of bytes are needed and that number is determined by the size of a character pointer. For example, you identify the size of each block at 50, the number of blocks at 10 and the size of a character pointer is two bytes. Then 502 bytes are needed for this memory pool. Declare an array of unsigned characters with the size of the array dictated as above so the array's starting address will be the address of the memory pool. You may create as many memory pools as you like, provided the memory is not used for anything else.

You must align the memory pool to the CPU's requirements, so that the list starts at the proper memory alignment boundary for integers. This is because the address passed to the pointers when you call the *K_Mem_FB_Get* function must follow the CPU's memory requirements. Also, the block size must follow the CPU's memory alignment requirements as well.

For example, Intel's 80196 processor dictates that integers reside on EVEN address, indicating that the memory begins on an EVEN address and that the block size be a multiple of two. Some other processors may require the block size to be a multiple of 4.

It is assumed the memory buffer pool can be no larger than 64K bytes. You are free to set up a structure to ensure the memory pool partition begins on the proper CPU boundary.

The parameters supplied are assumed to be correct and are not tested in any way. Once a pool is created you may retrieve and release blocks from that memory pool.

This is an example of the *K_Mem_FB_Create* function:

**Called**

Before entering RTOS, tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_Mem_FB_Create*(void *,word16,word16); /* this is the function prototype */

#define BLK_SIZE ??? /* size of each memory block */

#define NUM_BLOCKS ??? /* number of memory blocks */

struct {

    unsigned char *dummy_ptr; /* will allocated space for CMX */
    unsigned char pool_bytes[BLK_SIZE * NUM_BLOCKS]; /* allocate enough
    memory for this memory pool. */
} MEM_POOL1; /* this is memory pool 1 */

void *K_Mem_FB_Create*(&MEM_POOL1,BLK_SIZE,NUM_BLOCKS);

**Passed**

&MEM_POOL1 is the beginning address where this memory pool will reside in memory.

BLK_SIZE is the size in bytes, that each block within this memory pool will have. Maximum of 255 bytes

NUM_BLOCKS is the number of fixed blocks within this memory pool. Maximum is 65535.

#define BSIZE 10 /* should be a multiple of the alignment size if CPU cares about pointer alignment */

#define NBLOCKS 20 /* The number of blocks within memory pool */

struct {
unsigned char *head_ptr;
unsigned char body[BSIZE * NBLOCKS];
} MEM_POOL1;

```
void task2(void)
{
```

```
  K_Mem_FB_Create(MEM_POOL1,BSIZE,NBLOCKS); /* create a fixed
  block memory pool with 20 fixed blocks of size 10 bytes each */
}
```

**Returned**

No status is returned.

☞ *Remember you must ensure enough memory for this memory pool to exist, and no memory contention is tested for. Also, if the processor must have pointers residing on specific boundaries, like an even address, then the size of a block must be even. This is because CMX places pointers within the unused memory blocks, for internal use.*

***The K_Mem_FB_Get function***

The *K_Mem_FB_Get* function gets a fixed block of free memory from a memory pool, if one is free. Two parameters are passed to this function. The beginning address of the memory pool you would like to obtain a block from is the first parameter. It is up to you to send the same beginning address of the particular memory pool as was sent to the *K_Mem_FB_Create* function. The second parameter is the address of the pointer that will receive the address of the block.

The *K_Mem_FB_Get* function determines if a block is free and available from the specified memory pool and if so, returns the beginning address for that block to the pointer. When the block address is returned, you are guaranteed that the size, and only the size of the block as specified when the memory pool was created with the *K_Mem_FB_Create* function, is available for use.

Make sure you do not exceed the block size of that particular memory pool. This results in memory corruption. If all the blocks of a particular memory pool are in use, then no block will be released. The status will notify you that no memory block was free from the memory pool and the address sent to the pointer is invalid.

The block's memory, if one was available, is contiguous and not fragmented, but may contain garbage from when the block was previously in use. When you determine you are done with a particular block from a memory pool, you may release it back to the memory pool. This is described below.

This is an example of the *K_Mem_FB_Get* function:

**Called**

By tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Mem_FB_Get*(void *,byte **); /* this is the function prototype */

struct {/* previous created by *K_Mem_FB_Create* function */
unsigned char *dummy_ptr;
unsigned char pool_bytes[BLK_SIZE * NUM_BLOCKS];
} MEM_POOL1; /* this is memory pool1 */

unsigned char *BLOCK_ADDR; /* block pointer could be local or global */

unsigned char STATUS; /* should be local */

STATUS = *K_Mem_FB_Get*(&MEM_POOL1,&BLOCK_ADDR);

**Passed**

&MEM_POOL1 is the beginning address where this particular memory pool will reside in memory.

&BLOCK_ADDR is the address of the unsigned char pointer, in which the address of the fixed block will be placed.

#define BSIZE 10 /* should be a multiple of the alignment size if CPU cares about pointer alignment */

#define NBLOCKS 20 /* The number of blocks within memory pool */

struct {
unsigned char *head_ptr;
unsigned char body[BSIZE * NBLOCKS];
} MEM_POOL1;

```
void task2(void)
{
  unsigned char status;
  unsigned char *block_user_ptr; /* create a pointer to use as
  user wishes */
  unsigned char *block_ptr; /* create another pointer to keep
  memory address passed back to block_ptr, for use with
  K_Mem_FB_Release function */

  status = K_Mem_FB_Get(&MEM_POOL1,&block_ptr); /* load block
  pointer with memory address location of free block */
  if (status == K_OK)
    {
    block_user_ptr = block_ptr; /* save memory address of block
    because block_ptr probably will corrupt this address */
    }
  else
    {
    /* error, do what ever */
```

```
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: no free block within this memory pool.

If STATUS equals K_OK, then BLOCK_ADDR contains the block's address. Copy this address to another pointer, because when releasing this block, you will have to pass this address back.

### *The K_Mem_FB_Release function*

The *K_Mem_FB_Release* function releases a fixed block of memory back to a particular memory pool. You supply the address of the memory pool the block was retrieved from and the address the block was given when the *K_Mem_FB_Get* function was called. This address must be the same as the address received for this block.

When you release a block, it becomes free and added back into the free memory blocks of the particular memory pool from which it was taken. When a block is released, the block's memory may be released again as another block and the contents are destroyed.

The addresses supplied to the above functions are not tested and are assumed to be correct. If not, the memory could be corrupted, resulting in serious consequences.

This is an example of the *K_Mem_FB_Release* function:

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
void K_Mem_FB_Release(void *,byte *); /* this is the function prototype */

struct {/* previously created by K_Mem_FB_Create function */
unsigned char *dummy_ptr;
unsigned char pool_bytes[BLK_SIZE * NUM_BLOCKS];
} MEM_POOL1; /* this is memory pool 1 */

unsigned char *BLOCK_ADDR; /* previously declared, block pointer could be local
or global */

void K_Mem_FB_Release(&MEM_POOL1,BLOCK_ADDR);
```

**Passed**

&MEM_POOL1 is the beginning address where this particular memory pool will reside in memory.

BLOCK_ADDR is the contents of the BLOCK_ADDR address, which contains the address of the block that was retrieved by the *K_Mem_FB_Get* function.

#define BSIZE 10 /* should be a multiple of the alignment size if CPU cares about pointer alignment */

#define NBLOCKS 20 /* The number of blocks within memory pool */

struct {
unsigned char *head_ptr;
unsigned char body[BSIZE * NBLOCKS];
} MEM_POOL1;

```
void task2(void)
{
  unsigned char *block_user_ptr; /* create a pointer to use as
  user wishes */
  unsigned char *block_ptr; /* create another pointer to keep
  memory address passed back to block_ptr, for use with
  K_Mem_FB_Release function */

  /* user code here dealing with block_ptr */

  K_Mem_FB_Release(&MEM_POOL1,block_ptr); /* finished with this
  block of memory, release it back to memory pool */
}
```

**Returned**

No status is returned.

☞ *Ensure that the block address passed to this function is the same address received by the K_Mem_FB_Get function. No testing is performed to check the validity of this address.*

## MESSAGE MANAGER FUNCTIONS

The message manager is part of the CMX library and allows messages to pass from task to task. The message manager functions are listed below along with their reference pages.

*K_Mesg_Wait (Page 63)*
*K_Mesg_Ack_Sender (Page 64)*
*K_Mbox_Event_Set (Page 66)*

Just the address of the message is passed to the mailbox and not the message itself. This makes message passing very fast for the actual message data does not have to be copied into the mailbox when sent by the *K_Mesg_Send* or *K_Mesg_Send_Wait* functions, or out of the mailbox when retrieved by the *K_Mesg_Get* or *K_Mesg_Wait* functions. You tell the RTOS configuration module the total number of messages for all mailboxes. The memory needed is set aside for storing message addresses and really does not belong to any one mailbox.

As you send messages to mailboxes, the message memory (a message block) is given to the intended mailbox. CMX has already created and structured all of the message blocks at start up, which allows the message send and receive functions to quickly get a message block to store the message address and a few other CMX data items. Also, the messages which are passed to a particular mailbox are queued up as first in, first out. When all message blocks have been allocated to the mailboxes, then no further messages can be sent.

You also tell the RTOS configuration module the maximum number of mailboxes. Only tasks may have ownership rights to a mailbox and each task may "own" more than one. It is assumed that each mailbox will belong to only one task, as far as retrieving mail from that mailbox. Tasks and interrupts are free to send messages to any mailbox.

When a task decides it no longer needs a particular mailbox, another task may then assume ownership of that mailbox. This ownership theory only works when you have used the *K_Mbox_Event_Set* function. This function tells the mailbox to automatically set a particular task, an event number you define, when there is mail present in this mailbox. If this function is not used, or changed so that the task will not have an event set, then any task, at any time can use any mailbox as long as no other task is waiting on this mailbox.

### *The K_Mesg_Send function*

This function allows tasks and interrupts to send a message to a mailbox. Remember, the message itself is not sent, just the address of the message is passed to the mailbox. The message contents can be virtually anything as long as the sender and receiver agree on the format. This is extremely useful, as interrupts could send a message identifying a port's pin states for example. Also, a task may own several mailboxes, so a priority scheme could be set up with high priority message sent to mailbox one, lower priority messages sent to mailbox two, and lowest priority messages sent to mailbox three.

When the *K_Mesg_Send* function is called by a task or interrupt, the caller must supply the mailbox number this message will go to. In addition to the mailbox number, the caller must supply the address where the message resides in memory.

The message length may be any size since the mailbox just receives the address of the message. If at least one message block is free at the time of the call, the mailbox will receive the message. Each mailbox does not have a limit on the number of messages it receives. The only limitation is that a message block must be available to give to the mailbox. Once a message block is given to a mailbox, the message block is unusable to all other mailboxes until the message is passed to a task.

The interrupt or task that sent the message will be returned to immediately. However if a task is waiting for a message from this mailbox and its priority is higher than the task sending it, the scheduler will be notified to do an immediate task switch. In this case, the sending task or task running prior to an interrupt will become suspended and the task waiting on the message will become the new running task.

If the *K_Mbox_Event_Set* function has been enabled to set the task's event that owns this mailbox and the mailbox is empty when this message arrives, then the *K_Mesg_Send* function will automatically set the specified event of the task that owns this mailbox.

This is an example of the *K_Mesg_Send* function:

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Mesg_Send(byte,void *); /* this is the function prototype */

#define MBOX1 ??? /* numerical identifier that identifies the particular mailbox */

unsigned char SOURCE_BYTES[] = ??? /* could be global or local */

unsigned char STATUS; /* should be local to task */

STATUS = K_Mesg_Send(MBOX1,SOURCE_BYTES);
```

**Passed**

MBOX1 is the mailbox number to which the task would like to send messages. This number ranges from 0 to the maximum number of mailboxes specified minus 1.

SOURCE_BYTES is the address where the message bytes reside, which will be copied into the mailbox message's pointer.

```
#define MBOX1 1
```

```
void task2(void)
{
  unsigned char status;

  status = K_Mesg_Send(MBOX1,"This message for mailbox 1\n");
  /* send this message to mailbox 1 remember that the address of
  the message is really passed to the mailbox and not the
  contents. */
  if (status != K_OK) /* test status, see if good operation */
    {
    /* see why, maybe take corrective action */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: there are NO more message blocks available or the mailbox number is out of range.

If STATUS equals K_OK, then a message has been placed into the mailbox message block. Make note that each mailbox works as a FIFO (first in, first out) queue.

***The K_Mesg_Send_Wait function***

This function is identical to the *K_Mesg_Send* function but with this addition: the task that calls the *K_Mesg_Send_Wait* function will pass the amount of time it is willing to wait for the *K_Mesg_Ack_Sender* function from the receiving task. The amount of time may be zero, to wait indefinitely, or a value up to 65535. The task will be suspended until this message is retrieved out of the mailbox by a task and the task issues the *K_Mesg_Ack_Sender* function, or the specified time period expires.

This acts as an acknowledgment to the caller. The task that called *K_Mesg_Send_Wait* will automatically become READY to run when the destination task that receives the message calls the *K_Mesg_Ack_Sender* function. If the specified time period expires, then the task that sent the message will be automatically awakened and READY to resume. When this task becomes the RUNNING task, it will be notified that the time period expired, instead of being awakened by the receiving task calling the *K_Mesg_Ack_Sender* function.

Only tasks may call this function. If the receiving task for this message never runs or issues the *K_Mesg_Ack_Sender* function prior to retrieving another message, then the sender of the message will never run again unless the task specified a non zero time out period and the time period expires. The *K_Task_Wake_Force* function can also be used to forcefully wake this task.

This is an example of the *K_Mesg_Send_Wait* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Mesg_Send_Wait*(byte,word16,void *); /* this is the function prototype */

#define MBOX1 ??? /* numerical identifier that identifies the particular mailbox */

#define TIME_CNT ??? /* The period of time to wait or indefinitely. */

unsigned char SOURCE_BYTES[] = ??? /* could be global or local */

unsigned char STATUS; /* should be local to task */

STATUS = *K_Mesg_Send_Wait*(MBOX1,TIME_CNT,SOURCE_BYTES);

**Passed**

MBOX1 is the mailbox number which the task would like to test for messages. This number ranges from zero to the maximum number of mailboxes specified minus one.

TIME_PERIOD is the number of system ticks this task will wait for the *K_Mesg_Ack_Sender* function to wake it. A period of zero indicates to wait indefinitely. The period may range from zero to 65535.

SOURCE_BYTES is the address where the message bytes reside, which will be copied into the mailbox's message pointer.

#define MBOX1 1

unsigned char mesg1[] = {"hello task 1\n"};

```
void task2(void)
{
  unsigned char status;

  status = K_Mesg_Send_Wait(MBOX1,100,mesg1);
  /* send this message to mailbox 1 remember that the address of
  the message is really passed to the mailbox and not the
  contents.  Also wait up to 100 system TICKS for the
  K_Mesg_Ack_Sender function */
  if (status == K_ERROR) /* test status, see if error */
    {
    /* see why, maybe take corrective action */
    }
  if (status == K_TIMEOUT)
```

```
    {
    /* time out period specified has expired prior to
    K_Mesg_Ack_Sender */
    /* should we do any corrective action */
    }
  ..../* good acknowledgment by receiving task, happened with in
  time out period specified. */
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: there are no more message blocks available or the mailbox number is out of range.

K_TIMEOUT = Warning: That the time period expired before the *K_Mesg_Ack_Sender* function was used to wake this task.

If STATUS equals K_OK, then a message had been placed into the mailbox, a task then received this message and issued the *K_Mesg_Ack_Sender* function to notify this task (sender) that the message was received.

***The K_Mesg_Get function***

This function allows the calling task to retrieve a message from a mailbox, if the mailbox has any messages. The *K_Mesg_Get* function is only called by tasks.

The only parameter passed to this function is the mailbox number. If the mailbox has no messages in it at the time of this function, then the function will return a null pointer value to the caller indicating no message was available. If the mailbox has at least one message, then the first message address will be returned to the caller. Also the message block which contained this message address will automatically be returned to the free message blocks and marked free.

A task may own more than one mailbox and it can receive a message from any of its mailboxes. This is useful if you want to prioritize messages to a task and its mailbox.

When a mailbox gives the message address to the task, the receiving task automatically obtains which task sent the message (because the sending task used the *K_Mesg_Send_Wait* function). The receiver may then call the *K_Mesg_Ack_Sender* function to acknowledge it has received this message. The receiving task is free to call the *K_Mesg_Ack_Sender* function at any time prior to retrieving another message from any mailbox.

If a task calls the *K_Mesg_Get* function and specifies the same mailbox number another task has previous used in the *K_Mesg_Wait* function, and that task is currently waiting for a message from that mailbox, then an a null value is returned to this task. In other words, no more than one task may wait on a single mailbox, at any time.

The *K_Mesg_Get* function returns immediately to the caller regardless of a message being available or not. Messages are retrieved in the order they were received by the mailbox, which is first in, first out.

This is an example of the *K_Mesg_Get* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void * *K_Mesg_Get*(byte); /* this is the function prototype */

#define MBOX1 ??? /* numerical identifier that identifies the particular mailbox */

unsigned char *RECV_PTR; /* could be local or global */

RECV_PTR = *K_Mesg_Get*(MBOX1);

**Passed**

MBOX1 is the mailbox number in which the task would like to test for messages. This number ranges from zero to the maximum number of mailboxes specified minus one.

#define MBOX1 1

```
void task1(void)
{
  unsigned char *recv_ptr; /* create a pointer that will receive
  the address of message */

  recv_ptr = K_Mesg_Get(MBOX1); /* go get the message if one is
  available */
  if (recv_ptr != (unsigned char *)NULL) /* test, see if not NULL
  */
    {
    /* application code to deal with message */
    }
}
```

**Returned**

RECV_PTR is the pointer that will be given the address where the message bytes are located.

If RECV_PTR = NULL (0), then there was NOT a message in this mailbox, when this task called. If the RECV_PTR contains a non null value, this will be the address where the message bytes are.

### *The K_Mesg_Wait function*

The *K_Mesg_Wait* function performs just like the *K_Mesg_Get* function, but also gives the task the ability to wait for a message if one is not available for a specified time or indefinitely.

This function requires two parameters. The first is the mailbox number to check for messages. The second indicates the time period.

The *K_Mesg_Wait* function also allows a task to suspend itself waiting for a message if none were there at the time of the call, and automatically be put back into the READY state. This can be done in two ways. First, a message is sent to the mailbox. Second, the specified time period by the task expires.

The time period is the number of system ticks to wait for a message. This number may range from one through 65535. A time period of zero indicates this task will wait indefinitely for a message.

This is an example of the *K_Mesg_Wait* function:

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
void * K_Mesg_Wait(byte,word16); /* this is the function prototype */

#define MBOX1  ??? /* numerical identifier that identifies the particular mailbox */

#define TIME_CNT ???

unsigned char *RECV_PTR; /* could be local or global */

RECV_PTR = K_Mesg_Wait(MBOX1,TIME_CNT);
```

**Passed**

MBOX1 is the mailbox number which the task would like to test for messages. This number ranges from zero to the maximum number of mailboxes specified minus one.

TIME_CNT is the number of system ticks to wait for a message. The range is zero through 65535. If the value is zero, then the task will wait indefinitely for a message to arrive.

```
#define MBOX1 1

void task1(void)
{
  unsigned char *recv_ptr; /* create a pointer that will receive
  the address of message */

  recv_ptr = K_Mesg_Wait(MBOX1,100); /* go get a message if one
  is available, if not wait for 100 TICKs for message to arrive
  in mailbox */
  if (recv_ptr != (unsigned char *)NULL) /* test, see if not NULL
  */
    {
    /* application code to deal with message */
    }
}
```

**Returned**

RECV_PTR is the pointer that will be given the address where the message bytes are located.

If RECV_PTR = NULL (0), then either the time period specified expired prior to a message being retrieved or the mailbox number was out of range. If the RECV_PTR contains a non null value, this will be the address of the message bytes.

A null value returned to the caller by the *K_Mesg_Wait* function indicates the following: no messages were in the mailbox, the mailbox number is out of range or another task is waiting on this mailbox for a message to come in.

### *The K_Mesg_Ack_Sender function*

After using either the *K_Mesg_Get* or *K_Mesg_Wait* function to retrieve a message, the *K_Mesg_Ack_Sender* function should be used to acknowledge the sender and wake up the sending task. Only tasks may call this function.

If the sender used the *K_Mesg_Send* function and the task that received the message was unaware an acknowledgment was not necessary, the task may still call this function. The *K_Mesg_Ack_Sender* function knows the sender of the message is not waiting for a reply.

The task receiving the message must call this function to wake the suspended task (which used the *K_Mesg_Send_Wait* function) before retrieving another message or terminating its code. If not done, the sending task will be suspended until either the time out period expires (if the time period specified was non zero) or you use the *K_Task_Wake_Force* function, which forcefully wakes the sending task.

Any time a message is received, the receiving task should call the *K_Mesg_Ack_Sender* function to ensure the sending task is not suspended forever. If you know the sender did not use the *K_Mesg_Send_Wait* function, then the *K_Mesg_Ack_Sender* function need not be called.

This is an example of the *K_Mesg_Ack_Sender* function:

**Called**

Tasks

#include <cxfuncs.h> /* has function prototype */
byte *K_Mesg_Ack_Sender*(void); /* this is the function prototype */

unsigned char STATUS; /* should be local */

STATUS = *K_Mesg_Ack_Sender*();

**Passed**

Nothing.

#define MBOX1 1

```
void task1(void)
{
  unsigned char *recv_ptr; /* create a pointer that will receive
  the address of message */

  recv_ptr = K_Mesg_Get(MBOX1); /* go get the message if one is
  available */
  if (recv_ptr != (unsigned char *)NULL) /* test, see if not NULL
  */
    {
    /* application code to deal with message */
    K_Mesg_Ack_Sender(); /* go wake the task that sent this
    message, because the task that sent message used the
    K_Mesg_Send_Wait function. */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_NOT_WAITING = Warning: The task that sent the received message was not waiting.

If STATUS equals K_OK, then the task that sent the message had used the *K_Mesg_Send_Wait* function and the task was suspended is now placed into the READY state.

☞ *An immediate rescheduling will occur if the awakened task has a higher priority than the current running task.*

***The K_Mbox_Event_Set function***

This function will signal the specified task when message(s) are present in the selected mailbox. When the mailbox contains or receives a message, it will automatically use the *K_Event_Signal* function, setting a specific event bit of a particular task. This allows a task to wait on the events. When a message arrives in a mailbox, the mailbox will set the event, notifying the task that there are messages in the mailbox.

When you set up this function, you will pass three parameters to this function. The first parameter is the mailbox number. This number can range from zero to one less than the maximum number of mailboxes declared in the configuration file.

The next parameter is a particular task's slot number which may be any valid task created with the *K_Task_Create* function and not removed by the *K_Task_Delete* function. If the task slot number is zero, then no task will have its event set by this mailbox.

The third parameter is the event (bit) that will be set in the declared task when messages are available and present in this mailbox. The *K_Mbox_Event_Set* function works as follows. If the mailbox is empty when a message comes in, the mailbox will automatically call the *K_Event_Signal* function. The *K_Event_Signal* function will use the parameters set up in the *K_Mbox_Event_Set* function. Also the mode byte, declaring which mode the *K_Event_Signal* function will use, will be forced to zero. This identifies that only the specific task will have its event set.

When the task retrieves the message using the *K_Mesg_Get* or *K_Mesg_Wait* function, then the task’s event will be set by this mailbox if any messages are still present.

This allows tasks to wait on a variety of events and mailboxes. When a task is waiting on one or more mailboxes, the task will be notified that a message has arrived or there are more messages available at the mailbox. You would most likely set up the *K_Event_Wait* function so as the events occur, they will automatically be reset. When the task resumes running, the task would then retrieve the message. When the task retrieves the message, the mailbox automatically sets the task's event again if there are more messages present in its mailbox.

Mailboxes parameters can be changed at any time. This may be useful if you want another task to be the owner of this mailbox or change what event will be set when a message is present. Also, you may specify a task slot number of zero, so the mailbox will not set any task's event when a message is present.

Remember, any task can retrieve a message, if one is available, from any mailbox as long as there are no other tasks waiting on the mailbox. CMX feels only one task should own a mailbox which is why the *K_Mbox_Event_Set* function will only let a single task have its event set when there are messages present in a mailbox.

This is an example of the *K_Mbox_Event_Set* function:

**Called**

Before entering RTOS, tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Mbox_Event_Set*(byte,byte,word16); /* this is the function prototype */

#define MBOX1 ??? /* numerical identifier that identifies the particular mailbox */

unsigned char TASK_SLOT; /* Should be global */

#define EVENT ??? /* which event bit to set */

unsigned char STATUS; /* receives status of function. */

STATUS = *K_Mbox_Event_Set*(MBOX1,TASK_SLOT,EVENT);

**Passed**

MBOX1 is the mailbox number which the task would like to test for messages. This number ranges from zero to the maximum number of mailboxes specified minus one.

TASK_SLOT is the slot number of the task that will have an event bit set, when there are messages in this mailbox.

EVENT is an unsigned 16 bit variable or constant that determines which event bit will be set in the declared task.

unsigned char task1_slot; /* contains task 1 slot number that *K_Task_Create* returned */

#define MBOX1 1

#define MBOX2 2

#define TSK1_MB1_FLAG 0x0080 /* this is the event (bit) that the *K_Mbox_Event_Set* function will set when messages are present in its box. */

#define TSK1_MB2_FLAG 0x0040 /* this is the event (bit) that the *K_Mbox_Event_Set* function will set when messages are present in its box. */

```
void task1(void)
{
  unsigned char status;
  unsigned short event_bits; /* will indicate to task, which
  events are set by the K_Event_Wait function */

  unsigned char *recv_ptr; /* create a pointer that will receive
  the address of message */

  status = K_Mbox_Event_Set(MBOX1,task1_slot,TSK1_MB1_FLAG);
  /* this will tell mailbox 1 to set task1 event (bit 7) when a
  message is present */
  status = K_Mbox_Event_Set(MBOX2,task1_slot,TSK1_MB2_FLAG);
  /* this will tell mailbox 2 to set task1 event (bit 6) when a
  message is present */
  while(1)
    {
    event_bits = K_Event_Wait((TSK1_MB1_FLAG ||
    TSK1_MB2_FLAG),0,2);
    /* this will force task 1 to wait indefinitely for either
    mailbox 1 or mailbox 2 to signal that they have a message.
    Also when the task resumes because a message is present, the
    K_Event_Wait function will automatically clear the event bit
    that was set.  This way the mailbox will again set the event
    when the task retrieves the message, if any messages are still
    in its mailbox or when a message arrives. */
    if (event_bits & TSK1_MB1_FLAG)
      {
      recv_ptr = K_Mesg_Get(MBOX1); /* go get the message */
      /* remember that the mailbox will automatically set the
      event (TSK1_MB1_FLAG) if and when another message is
      present in its mailbox. */
      .../* process message */
      /* do the following if message sender used the
      K_Mesg_Send_Wait function. */
      K_Mesg_Ack_Sender(); /* go wake the task that sent this
      message, because the task that sent message used the
      K_Mesg_Send_Wait function. */
      }
    if (event_bits & TSK1_MB2_FLAG)
      {
      recv_ptr = K_Mesg_Get(MBOX2); /* go get the message */
      /* remember that the mailbox will automatically set the
      event (TSK1_MB2_FLAG) if and when another message is
      present in its mailbox. */
      .../* process message */
      /* do the following if message sender used the
      K_Mesg_Send_Wait function. */
      K_Mesg_Ack_Sender();
      /* go wake the task that sent this message, because the task
      that sent message used the K_Mesg_Send_Wait function. */
```

```
      }
    }/* will stay in while loop, processing messages from mailbox
    1 and mailbox 2. */
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the mailbox number is out of range.

☞ *Remember the mailbox will set the event the first time a message arrives and will also set the event each time the task retrieves a message, if more messages are in this mailbox. This function may be called more than once to specify a different event and/or task.*

## RESOURCE MANAGER FUNCTIONS

The resource manager is part of the CMX library and contains the functions listed below along with their reference pages. These functions allow only one task to have access to a particular resource at any one time. You may elect to have the task be put into a suspended state, to wait for this resource for a specified amount of time or indefinitely, if the resource is owned by another task. You can also not suspend the task if the resource is busy.

CMX has priority inheritance on resources. The highest priority task waiting for a resource will become the owner of the resource when the resource is released by the current owner. This is described in the next 2 paragraphs. Please keep these 2 items in mind, when writing and debugging your application code.

Priority inheritance is where the current task that owns a resource will be temporally bumped up to the priority level of the highest priority task waiting for the same resource. When the task releases the resource, then its priority will be restored to its original priority that it was prior to owning the resource.

When a resource is released by the current owner of the resource, the highest priority task waiting for this resource will become the new current owner. This is regardless of when it requested this resource in relationship to other task requesting this resource as well, if any. This means that resources do not work based on a first in, first out request like some RTOSs, but that they determine who will become the owner based on the priority of a particular task waiting for the resource.

*K_Resource_Get (Page 70)*
*K_Resource_Wait (Page 71)*
*K_Resource_Release (Page 73)*

### *The K_Resource_Get function*

The *K_Resource_Get* function is used only by tasks to request the use of a particular resource group. The task will supply the resource number. This number ranges from zero to one less than the maximum number of resources declared in the configuration module.

If the resource is free and not owned by any other task, then the task is given that resource and now owns it. If the resource is owned, the task that called this function will not be placed into the suspended state and will be notified that the resource is already owned.

The task that calls *K_Resource_Get* will not be suspended if the resource is owned, nor will it be placed into the suspended state while waiting for this resource to be free. Also if the task becomes the owner of this resource by this call, then the task must use the *K_Resource_Release* function when finished with the resource so other tasks may obtain ownership.

This is an example of the *K_Resource_Get* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Resource_Get*(byte); /* this is the function prototype */

#define RESOURCE_NUM ??? /* identifies resource number */

unsigned char STATUS; /* should be local */

STATUS = *K_Resource_Get*(RESOURCE_NUM);

**Passed**

RESOURCE_NUM is the number of the particular resource that this task would like to own.

#define RESOURCE1 1

```
void task2(void)
{
  unsigned char status;

  status = K_Resource_Get(RESOURCE1);
```

```
  /* pass resource 1 number to function, task 2 would like to
  become the owner of resource 1.  If resource 1 is not "owned"
  by another task, then task 2 will become the "owner" indicated
  by the value returned.  Task 2 will NOT be suspended if the
  resource is busy ("owned"), and the status returned will
  identify that the resource is busy. */
  if (status == K_OK) /* see if good operation */
    {
    /* task 2 now OWNS resource 1, when done with resource, must
    call the K_Resource_Release function to release it */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_RESOURCE_OWNED = Error: the resource is owned by another task.

K_ERROR = Error: the resource number was out of range.

If STATUS equals K_OK, then the task will own the resource. If STATUS equals K_RESOURCE_OWNED, or K_ERROR then the task does not own the resource.

☞ *It is up to you to ensure the task checks the return status to see if it owns the resource or not. If so, then the task may access this particular resource. If not, then the task should not access this resource, because another task is already using this resource. Contention (possibly corruption) could exist if both tasks try to manipulate this resource.*

***The K_Resource_Wait function***

This function works like the *K_Resource_Get* function. The difference is that if the resource is owned at the time a task calls the *K_Resource_Wait* function, then the task is suspended until the resource becomes free.

The resource wait queue works in the manner that when a resource is released by the current owner of the resource, the highest priority task waiting for this resource will become the new current owner. This is regardless of when it requested this resource in relationship to other task requesting this resource as well, if any. This means that resources do not work based on a first in, first out.

If the task was suspended and put into the queue, when the resource becomes free and ownership is passed to this task, the task will automatically be put into the READY to RESUME state.

Also, if the specified time period expires before the resource becomes free for this task, then the task will automatically be awakened and able to resume its code. It will also be notified that the time period expired and that this task does not own this resource.

When a task becomes the owner of this resource by this call, then the task must also use the *K_Resource_Release* function when finished to release this resource so other tasks may obtain ownership.

This is an example of the *K_Resource_Wait* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Resource_Wait*(byte,word16); /* this is the function prototype */

#define RESOURCE_NUM ??? /* identifies resource number */

#define TIME_PERIOD ??? /* The time period to wait for */

unsigned char STATUS; /* should be local */

STATUS = *K_Resource_Wait*(RESOURCE_NUM,TIME_PERIOD);

**Passed**

RESOURCE_NUM is the number of the particular resource this task would like to own.

TIME_PERIOD is the number of system ticks to wait for this resource, ranging from zero to 65535. If the value equals zero, then the task will wait indefinitely for this resource.

#define RESOURCE1 1

```
void task2(void)
{
  unsigned char status;

  status = K_Resource_Wait(RESOURCE1,100);
  /* pass resource 1 number to function, task 2 would like to
  become the owner of resource 1.  If resource 1 is not "owned"
  by another task and is free, then task 2 will become the "owner"
  indicated by value returned.  Task 2 will be suspended for 100
  TICKs if the resource is busy ("owned").  If the resource
  becomes free before the time period expires and this task is
  the next one in line (based on priority) to receive ownership,
```

```
  then this task will automatically become the "owner" of this
  resource.  If the time period expires, then the task will be
  removed from the resource wait queue and be returned
  identifying that the time period expired and this task does NOT
  "own" the resource. */

  if (status == K_OK) /* see if good operation */
    {
    /* task 2 now OWNS resource 1, when done with resource, must
    call the K_Resource_Release function to release it */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_TIMEOUT = Error: the time period expired.

K_ERROR = Error: the resource number out of range.

If STATUS equals K_OK, then the task now owns the resource. If any other value is returned to STATUS, then the task does not own the resource.

☞ *Make sure the STATUS byte is tested to see if the task owns the resource or not. If the task owns the resource, it may now use this resource knowing this task is the only one that has access to this resource. The tasks must also make sure they release the resource when done, using the K_Resource_Release function.*

***The K_Resource_Release function***

The *K_Resource_Release* function is used by the task to give up ownership of a resource. Only the task that has ownership of a resource is able to release that resource. The task will call this function specifying which resource number to release since a task may own more than one resource group.

It is your responsibility to ensure a task does not end without first releasing all the resources owned. Also, exercise caution when calling other functions that will suspend this task for a long period of time since other tasks may want access to this resource.

This is an example of the *K_Resource_Release* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Resource_Release*(byte); /* this is the function prototype */

#define RESOURCE_NUM ??? /* identifies resource number */

unsigned char STATUS; /* should be local */

STATUS = *K_Resource_Release*(RESOURCE_NUM);

**Passed**

RESOURCE_NUM is the number of the particular resource this task would like to release.

#define RESOURCE1 1

```
void task2(void)
{
  unsigned char status;
  /* task 2 owns resource group 1, and is done with this resource.
  task 2 will now release resource 1 */

  status = K_Resource_Release(RESOURCE1); /* task 2 is releasing
  resource 1 */
  if (status != K_OK) /* see if error, should not be */
    {
    /* If there was an error, take corrective action.  You should
    never get an error here, unless possibly when you called
    either the K_Resource_Get or K_Resource_Wait functions, they
    did not test status to see if the task truly owned the
    resource */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_RESOURCE_NOT_OWNED = Error: this task does NOT own this resource.

K_ERROR = Error: the resource number is out of range.

If STATUS equals K_OK, then the task now has released the particular resource.

☞ *The task that owns the resource must make sure it calls this function before finishing its code. If the task is going to use the K_Task_Delete function, then the task must first release the resource. If the task does not release the resource, the tasks waiting in the resource's queue will be SUSPENDED forever or until their time period expires.*

The tasks must make sure they test the STATUS byte, to see if the resource is released or not. If the STATUS is returned with an error value, then there is a coding error within the application code. If the task successfully releases a resource and the next task in the waiting queue for this resource has a higher priority, then an immediate rescheduling will occur, with that task becoming the RUNNING task.

## SEMAPHORE MANAGER FUNCTIONS

The semaphore manager is part of the CMX library and contains the functions listed below along with their reference pages. These functions allow one or more tasks to have access to a particular semaphore at any one time.

A task or interrupt can post to a semaphore, which increments the semaphore's counter. If a task is waiting for the semaphore, the waiting task will be placed into the RESUME state and the semaphore counter decremented.

A task can also pend on a semaphore. If the semaphore's counter is zero, depending on which semaphore pend function was used, either a status will be returned indicating that the semaphore has not been posted to, or the calling task will be suspended, indefinitely or for a given time period, waiting for a posting to the semaphore. If the counter is greater than zero, the semaphore's counter will be decremented and a status returned indicating that the task now owns the semaphore.

Semaphores are quite useful in a couple of ways. The first way would be that a single task would use a semaphore as a counter. Each time this semaphore was posted to, the semaphore's count would increase by one. A task could pend on a semaphore, waiting for a posting. Once the semaphore was posted to, the task would be placed into the RESUME state, identifying that a task or interrupt has posted to this semaphore. The task could then perform some action and loop until the semaphore count reached zero.

Another possible use would be to specify the number of tasks that can use the particular entity to which this semaphore is tied. Lets say a semaphore is initialized with a count of two. Two tasks could successfully obtain this semaphore. If a third task tried to obtain this semaphore, depending on which semaphore pend function was used, it would either receive a status indicating the semaphore was not available, or the task would be suspended until one of the other tasks posted to the semaphore, thus releasing it, or the task would wait until the specified time period expired.

*K_Semaphore_Create (Page 76)*
*K_Semaphore_Get (Page 77)*
*K_Semaphore_Wait (Page 78)*
*K_Semaphore_Post (Page 80)*

*K_Semaphore_Reset (Page 82)*

### *The K_Semaphore_Create function*

This function initializes a semaphore. Two parameters are needed by the *K_Semaphore_Create* function.

The first parameter is the semaphore number. This number ranges from zero to one less than the maximum number of semaphores declared in the configuration module.

The second parameter is the initial value of the counter for this semaphore. This counter is of type unsigned short, meaning it can range from 0 to 65535.

This is an example of the *K_Semaphore_Create* function:

**Called**

Before entering RTOS, tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Semaphore_Create*(byte,word16); /* this is the function prototype */

#define SEM_NUM ??? /* Identifies the semaphore. */

#define SEM_COUNT ??? /* determines the initial setting of the semaphore counter. */

unsigned char STATUS; /* should be local */

STATUS = *K_Semaphore_Create*(SEM_NUM,SEM_COUNT);

**Passed**

SEM_NUM is the number of the semaphore this function will work with. Values range from 0 to 1 less than maximum number of semaphores declared.

SEM_COUNT is the initial numerical value of the semaphore counter. Values range from 0 to 65535.

#define SEM1 1 /* define semaphore 1 */

#define SEM1_CNT 0 /* define initial semaphore count to be zero. */

```
void task1(void)
{
  unsigned char status;

  status = K_Semaphore_Create(SEM1,SEM1_CNT);
  /* Set up semaphore 1 with an initial count of zero */
  if (status != K_OK) /* see if error, should not be */
```

```
    {
    /* If there was an error, take corrective action.  You should
    never get an error here, unless possibly you have exceeded
    the maximum number of semaphores that was declared in the
    configuration module */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: semaphore number out of range.

### *The K_Semaphore_Get function*

The *K_Semaphore_Get* function is used by tasks to request the use of a particular semaphore. The task will supply the semaphore number. This number ranges from zero to one less than the maximum number of semaphores declared in the configuration module.

If the semaphore counter is greater than zero, the task is given that semaphore. If the semaphore counter is zero, the task that called this function will be returned to with a status indicating the semaphore is not available. The task will not be placed into the suspended state to wait for the semaphore.

If the task becomes the owner of this semaphore by this call, the task may need to use the *K_Semaphore_Post* function to notify other tasks when it is finished with the semaphore.

This is an example of the *K_Semaphore_Get* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Semaphore_Get*(byte); /* this is the function prototype */

#define SEM_NUM  ??? /* identifies semaphore number */

unsigned char STATUS; /* should be local */

STATUS = *K_Semaphore_Get*(SEM_NUM);

**Passed**

SEM_NUM is the number of the semaphore this function will work with. Values range from 0 to 1 less than maximum number of semaphores declared.

#define SEM1 1

```
void task2(void)
{
  unsigned char status;

  status = K_Semaphore_Get(SEM1);
  /* pass semaphore 1 number to function, task 2 would like to
  become the owner of semaphore 1.  If semaphore 1 is not "owned"
  by another task, then task 2 will become the "owner" indicated
  by the value returned.  Task 2 will NOT be suspended if the
  semaphore is busy ("owned"), and the status returned will
  identify that the semaphore is busy. */
  if (status == K_OK) /* see if good operation */
    {
    /* task 2 now OWNS semaphore 1, when done with semaphore, must
    call the K_Semaphore_Get function to release it */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_SEMAPHORE_NONE = Error: the semaphore is owned by another task.

K_ERROR = Error: the semaphore number was out of range.

If STATUS equals K_OK, then the task will own the semaphore. If STATUS equals K_SEMAPHORE_NONE or K_ERROR then the task does not own the semaphore.

☞ *It is up to you to ensure the task checks the return status to see if it owns the semaphore or not. If so, then the task may access this particular semaphore. If not, then the task should not access this semaphore, because another task is already using this semaphore. Contention (possibly corruption) could exist if both tasks try to manipulate this semaphore.*

***The K_Semaphore_Wait function***

This function works like the *K_Semaphore_Get* function. The difference is that the task can specify a time period to wait for the semaphore. If the time period expires the task will be placed into the RESUME state and notified that the time period expired and the semaphore was not available.

The semaphore wait queue works in the manner of first in, first out. Tasks that call this function and find the semaphore busy are placed into the wait queue. When the semaphore becomes free, the first task in the queue is the first task to own it.

If the task was suspended, when the semaphore becomes free and ownership is passed to this task, the task will automatically be put into the ready to RESUME state.

If the task becomes the owner of this semaphore by this call, the task may need to use the *K_Semaphore_Post* function to notify other tasks when it is finished with the semaphore.

This is an example of the *K_Semaphore_Wait* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Semaphore_Wait*(byte, word16); /* this is the function prototype */

#define SEM_NUM ??? /* identifies semaphore number */

#define TIME_PERIOD ??? /* The time period to wait for */

unsigned char STATUS; /* should be local */

STATUS = *K_Semaphore_Wait*(SEM_NUM,TIME_PERIOD);

**Passed**

SEM_NUM is the number of the semaphore this function will work with. Values range from 0 to 1 less than maximum number of semaphores declared.

TIME_PERIOD is the number of system ticks to wait for this semaphore, ranging from zero to 65535. If the value equals zero, then the task will wait indefinitely for this semaphore.

#define SEM1 1

```
void task2(void)
{
  unsigned char status;

  status = K_Semaphore_Wait(SEM1,100);
  /* pass semaphore 1 number to function, task 2 would like to
  become the owner of semaphore 1.  If semaphore 1 is not “owned”
  by another task and is free, then task 2 will become the “owner”
  indicated by value returned.  Task 2 will be suspended for 100
  TICKs if the semaphore is busy (“owned”).  If the semaphore
```

```
  becomes free before the time period expires and this task is
  the next one in line to receive ownership, then this task will
  automatically become the "owner" of this semaphore.  If the
  time period expires, then the task will be removed from the
  semaphore wait queue and be returned identifying that the time
  period expired and this task does NOT "own" the semaphore. */

  if (status == K_OK) /* see if good operation */
    {
    /* task 2 now OWNS semaphore 1, when done with semaphore, must
    call the K_Semaphore_Post function to release it */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_TIMEOUT = Error: the time period expired.

K_ERROR = Error: the semaphore number out of range.

If STATUS equals K_OK, then the task now owns the semaphore. If any other value is returned to STATUS, then the task does not own the semaphore.

☞ *It is up to you to ensure the task checks the return status to see if it owns the semaphore or not. If so, then the task may access this particular semaphore. If not, then the task should not access this semaphore, because another task is already using this semaphore. Contention (possibly corruption) could exist if both tasks try to manipulate this semaphore.*

***The K_Semaphore_Post function***

The *K_Semaphore_Post* function is used by the task to give up ownership of a semaphore. The task will call this function specifying which semaphore number to release since a task may own more than one semaphore group.

The *K_Semaphore_Post* function can also be used by tasks and interrupts as a counter. For example, an interrupt could post to a semaphore every time a data packet came in through a serial port. A task could be pending on the semaphore and process the packets.

It is your responsibility to ensure a task does not end without first releasing all the semaphores owned. Also, exercise caution when calling other functions that will suspend this task for a long period of time since other tasks may want access to this semaphore.

This is an example of the *K_Semaphore_Post* function:

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

#include <cxfuncs.h> /* has function prototype */
byte *K_Semaphore_Post*(byte); /* this is the function prototype */

#define SEM_NUM ??? /* identifies semaphore number */

unsigned char STATUS; /* should be local */

STATUS = *K_Semaphore_Post*(SEM_NUM);

**Passed**

SEM_NUM is the number of the semaphore this function will work with. Values range from 0 to 1 less than maximum number of semaphores declared.

#define SEM1 1

```
void task2(void)
{
  unsigned char status;

  status = K_Semaphore_Get(SEM1);/* get semaphore 1 */

  /* application code here */

  /* task 2 owns semaphore 1, and is done with this semaphore.
  task 2 will now release semaphore 1 */

  status = K_Semaphore_Post(SEM1); /* task 2 is releasing
  semaphore 1 */

  if (status != K_OK) /* see if error, should not be */
    {
    /* If there was an error, take corrective action.  You should
    never get an error here, unless possibly when you called the
    K_Semaphore_Get or K_Semaphore_Wait functions, they did not
    test status to see if the task truly owned the semaphore */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the semaphore number is out of range.

If STATUS equals K_OK, then the task now has released the particular semaphore.

☞ *The task that owns the semaphore must make sure it calls this function before finishing its code. If the task is going to use the K_Task_Delete function, then the task must first release the semaphore. If the task does not release the semaphore, the tasks waiting in the semaphore's queue will be SUSPENDED forever or until their time periods expire.*

The tasks must make sure they test the STATUS byte, to see if the semaphore is released or not. If the STATUS is returned with an error value, then there is a coding error within the application. If the task successfully releases a semaphore and the next task in the waiting queue for this semaphore has a higher priority, then an immediate rescheduling will occur, with that task becoming the RUNNING task.

***The K_Semaphore_Reset function***

This function flushes out a particular semaphore. The semaphore's wait queue is emptied and the semaphore's counter is reset to the value given to the *K_Semaphore_Create* function when initializing the semaphore. Two parameters are needed by the *K_Semaphore_Reset* function.

The first parameter is the semaphore number. This number ranges from zero to one less than the maximum number of semaphores declared in the configuration module.

The second parameter is the flush mode. A flush mode of zero means the semaphore will not be flushed if a task owns the semaphore. A flush mode greater than zero means the semaphore will be flushed whether a task owns the semaphore or not.

This is an example of the *K_Semaphore_Reset* function:

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Semaphore_Reset(byte,byte); /* this is the function prototype */

#define SEM_NUM ??? /* identifies semaphore number */

#define FLUSH_MODE ??? /* specifies whether to flush if semaphore is owned */

unsigned char STATUS; /* should be local */

STATUS = K_Semaphore_Reset(SEM_NUM,FLUSH_MODE);
```

**Passed**

SEM_NUM is the number of the semaphore this function will work with. Values range from 0 to 1 less than maximum number of semaphores declared.

FLUSH_MODE determines whether to flush the semaphore if the semaphore is currently owned by a task or not. A value of zero will only flush the semaphore if it is not owned by a task. Any other value will always flush the semaphore.

#define SEM1 1

```
void task2(void)
{
  unsigned char status;

  status = K_Semaphore_Reset(SEM1,0);
  /*  task2 requests a flush of semaphore 1 with mode zero.  If
  no task owns this semaphore it will be flushed and K_OK
  returned. If a task owns semaphore 1 K_ERROR will be returned
  and semaphore 1 will not be flushed. */

  if (status == K_OK) /* see if good operation */
    {
    /* semaphore 1 wait queue is empty and semaphore 1 counter is
    reset to the value that was passed to the K_Semaphore_Create
    function. */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the semaphore number out of range, or if the flush mode was zero, a task owns the semaphore.

If STATUS equals K_OK, then the semaphore was flushed. If any other value is returned to STATUS, then the semaphore was not flushed.

## CYCLIC TIMERS MANAGER FUNCTIONS

The cyclic timer manager is part of the CMX library and contains the functions listed below along with their page references.

The cyclic timers manager functions enables you to use timers to automatically execute the event signal function at the specified number of system ticks. You can also indicate whether the cyclic timer executes only once at the specified time or at each time period.

All cyclic timers are executed under the CMX timer task. The CMX timer task is the highest priority task. The cyclic timers are executed in the order of zero through one less than the maximum number of cyclic timers declared by in the configuration module if the particular cyclic timer is running.

With the cyclic timers, you can have events signaled at specific time intervals. Tasks can create timers by telling the timer what event to signal and what mode the *K_Event_Signal* function will perform. The event and mode may be changed at any time by a task. Also, tasks can start a cyclic timer, specifying the initial time period and the cyclic time period. Both tasks and interrupts are able to change a cyclic timer's initial time period, or cyclic time period, to restart a stopped cyclic time period or to stop a cyclic time period.

### *The K_Timer_Create function*

This function sets up a cyclic timer's event function. Four parameters are needed by the *K_Timer_Create* function.

The first parameter is the cyclic timer number. This number ranges from zero to one less than the maximum number of cyclic timers declared in the configuration module.

The second parameter is the mode that the *K_Event_Signal* function will operate in when the cyclic timer time period expires and calls the *K_Event_Signal* function. This number can be between zero through six, and identifies what the *K_Event_Signal* function will perform when it is called by the cyclic timer.

The third parameter is the task slot number or priority that the *K_Event_Signal* function will work with if needed, This is dictated by the mode number supplied. The mode chosen for the second parameter determines if this parameter contains a task slot number, a priority, or a value that is not used.

The fourth parameter is the event that may be set when the *K_Event_Signal* function is executed by this cyclic timer, depending on the mode selected.

Carefully study the chapter on Event Management which describes the *K_Event_Signal* function in full detail. Because the cyclic timer calls this function automatically when its time period expires, you should fully understand the *K_Event_Signal* function, so you properly set up the parameters of the *K_Timer_Create* function.

The cyclic timers are very useful. They may be used to synchronize tasks to specific time intervals or signal a task that it should change states of the port pins. Another use might be for an interrupt to continuously restart a cyclic timer's initial time period. If the interrupt did not occur within the time period, then the cyclic timer would time out, signaling that the interrupt has not arrived within the specified time period.

This is an example of the *K_Timer_Create* function:

**Called**

Before entering RTOS, tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Timer_Create*(byte,byte,byte,word16); /* this is the function prototype */

#define CYCLIC_NUM ??? /* Identifies the cyclic timer. */

#define MODE ??? /* determines the mode of the *K_Event_Signal* function. */

#define EVENT ??? /* determines which event to set. */

unsigned char TASK_PRI; /* works in conjunction of selected mode, specifies task or priority or is not used. */

unsigned char STATUS; /* should be local */

STATUS = *K_Timer_Create*(CYCLIC_NUM,MODE,TASK_PRI,EVENT);

**Passed**

CYCLIC_NUM is the numerical value of the cyclic timer this function will work with. Values range from 0 to 1 less than maximum number of cyclic timers declared.

MODE is the mode in which the *K_Event_Signal* function will execute.

TASK_PRI is the task's slot number, the priority, or unused that is determined by the MODE selected.

EVENT is the event bit that will be set when the cyclic timer executes and calls the *K_Event_Signal* function.

#define TMR1 1 /* define cyclic timer 1 */

#define TMR1_MODE 0 /* define what mode the *K_Event_Signal* function will use. This value indicates a specific task. */

#define TSK2_TM1_EVT 0x0002 /* this is the event that will be set by the cyclic timer, when its time period expires */

unsigned char task2_slot; /* will contain task 2 slot number */

```
void task1(void)
{
  unsigned char status;
  status =
  K_Timer_Create(TMR1,TMR1_MODE,task2_slot,TSK2_TM1_EVT);
  /* Set up cyclic timer 1 to automatically set task 2 event bit
  1, when its time period expires. */

  if (status != K_OK) /* see if error, should not be */
    {
    /* If there was an error, take corrective action.  User should
    never get an error here, unless possibly the user has exceeded
    the maximum number of cyclic timers that was declared in the
    configuration module */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: cyclic number out of range.

The status returned from the *K_Timer_Create* function indicates K_OK: it was successful or K_ERROR: an error occurred, because the cyclic timer number was out of range. Make note that this function does not test the task slot number, if passed, to see if it is valid or not. The *K_Event_Signal* function will test this parameter when the cyclic timer calls it.

### *The K_Timer_Start function*

This function is used to initially start a cyclic timer which is assumed to have been set up by the *K_Timer_Create* function. Only tasks may call this function. You supply the cyclic timer number of the cyclic timer you would like to start. This may be in the range from zero through one less than the maximum number of cyclic timers declared in the configuration module.

Also supplied is the initial time in system ticks required before the cyclic timer executes for the very first time. This time period may range any where from one to 65535. If you declare a value of zero for the initial time period, then this value is considered to be 65536.

At each system tick, the cyclic timers with a non-zero time period are decremented. When this time period reaches zero, the cyclic timer then calls the *K_Event_Signal* function. The time supplied is used only for the first execution of this cyclic timer and then the cyclic timer will execute automatically at the time specified by the cyclic time period unless that value has a time of zero, indicating a "one shot".

The last parameter supplied to the *K_Timer_Start* function is the cyclic time period the timer will use. This time period will automatically be loaded after the initial time period expires. The range of this value may be from zero through 65535. If the value is declared to be zero, then the cyclic timer will not be reloaded with the cyclic time period, stopping this cyclic timer. This way, the cyclic timer acts as a "one-shot" timer. If the cyclic time value is non-zero, then this value will automatically be loaded and be decremented by the system tick. Again, when this value reaches zero, the cyclic timer will call the *K_Event_Signal* function using the cyclic timer event parameters specified by the *K_Timer_Create* function.

You may, at any time, use the *K_Timer_Start* function to override the time remaining normally used to start the procedure. Also, the new cyclic time period that was specified would take effect after the new initial time period expired.

This is an example of the *K_Timer_Start* function:

**Called**

Before entering RTOS, tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Timer_Start(byte,word16,word16); /* this is the function prototype */

#define CYCLIC_NUM ??? /* Identifies what cyclic timer. */

#define INITIAL_PERIOD ??? /* initial time period. */

#define CYCLIC_PERIOD ??? /* cyclic time period. */

unsigned char STATUS; /* should be local */

STATUS = K_Timer_Start(CYCLIC_NUM,INITIAL_PERIOD,CYCLIC_PERIOD);
```

**Passed**

CYCLIC_NUM is the cyclic timer number the caller wants to work with. This may range from zero to one less than the maximum number of cyclic timers declared.

INITIAL_PERIOD is the number of system ticks the cyclic timer will use immediately in its time counters. This time is reduced at each system tick and when it becomes zero, the cyclic timer executes the *K_Event_Signal* function. This may range from one to 65535.

CYCLIC_PERIOD is the number of system ticks the cyclic timer will use at the recycle time period. This may range from zero to 65535.

#define TIMER_0 0 /* cyclic timer 0 number */

#define T0_INIT_TIME 100 /* specify that initial time be 100 */

#define T0_CYCLE_TIME 50 /* specify cycle time of 50 */

```
void task2(void)
{
  unsigned char status;

  status = K_Timer_Start(TIMER_0,T0_INIT_TIME,T0_CYCLE_TIME);
  /* start cyclic timer 0.  This cyclic timer will execute in 100
  system "ticks".  The cycle time of 50 will then be reloaded
  each time this cyclic timer executes.  So it will take 100
  system "ticks" to execute this cyclic timer the first time and
  then only 50 ticks from then on. */
  if (status != K_OK) /* see if error, should not be */
    {
    /* Take corrective action. */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: cyclic timer number out of range.

If STATUS equals K_OK, then the specified cyclic timer is started. The initial time period and cyclic period are used for this cyclic timer.

This function may be called more than once for a particular cyclic timer, to change both the cyclic timer's initial time and cyclic time.

### *The K_Timer_Initial function*

The *K_Timer_Initial* function is used to change a cyclic timer's initial time period. This value will take effect immediately, overriding the current time value left that would be used to execute this cyclic timer when it reaches zero.

This function may be called by tasks and interrupts. The cyclic timer's cycle time period is not changed in any way. The caller will identify to the *K_Timer_Initial* function which cyclic timer and the new initial time period that the cyclic timer should use. Again, this value may range from one through 65535.

The initial time period specified takes effect immediately. For example, if the time left before this cyclic timer executes is two system ticks. Then Task 1 calls this function specifying a new initial time period of 30. This value will replace the old value of two preventing the cyclic timer from executing for another 30 ticks instead of two.

This is a very useful feature. You could set up a software watchdog feature like a cyclic timer in a "one-shot" mode. For example, say the initial time period set by the *K_Timer_Start* function was 200. Task 1 should execute the *K_Timer_Initial* function (specifying a new initial time period of 100) approximately every 80 system ticks, identifying that it has completed some specific duty. If task 1 did not perform the specific duty it was suppose to for whatever reason, then the cyclic timer will time out and signal an event.

This function can also be used by interrupts to identify that a particular interrupt is not being received. For example, say the timer is currently stopped. When the interrupt occurs, it could use the *K_Timer_Initial* function to start the cyclic timer.

Another example is using the interrupt to clock in data from a pin of a port. Each time a data bit happened, it would also generate an interrupt. The interrupt would then take the data bit, stuff it into a circular bit buffer, update the bit pointer, call the *K_Timer_Initial* function and then exit. When the data bits stopped coming in, the cyclic timer would time out and call the *K_Event_Signal* function telling a task that data is now present to be processed by setting the event the task was waiting on.

This is an example of the *K_Timer_Initial* function:

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
 byte K_Timer_Initial(byte,word16); /* this is the function prototype */

#define CYCLIC_NUM ??? /* Identifies what cyclic timer. */

#define NEW_INITIAL_PERIOD ??? /* new initial period. */

unsigned char STATUS; /* should be local */

STATUS = K_Timer_Initial(CYCLIC_NUM,NEW_INITIAL_PERIOD);
```

**Passed**

CYCLIC_NUM is the number of the cyclic timer that the caller wants to work with. May range from zero to one less than maximum number of cyclic timers declared.

NEW_INITIAL_PERIOD is the number of system ticks that the cyclic timer will use a immediately in its time counters. This time is reduced at each system tick and when it becomes zero, the cyclic timer executes the *K_Event_Signal* function. May range from one to 65535.

#define TIMER_0 0 /* cyclic timer 0 number */

#define T0_INIT_TIME 100 /* specify that initial time be 100 */

```
void task2(void)
{
  unsigned char status;

  while(1)
    {
    .../* wait for, then process whatever, should happen every 60
    ticks or so */
    status = K_Timer_Initial(TIMER_0,T0_INIT_TIME);
    /* restart cyclic timer 0.  This cyclic timer will execute in
    100 system "ticks", unless task 2 restarts it again.  If so
    the cyclic timer has been programmed to set an event,
    notifying the watchdog task that task 2 did not execute when
    it was suppose to. */

    if (status != K_OK) /* see if error, should not be */
      {
      /* Take corrective action. */
      }
    }
}
```

**Returned**

A status will be returned to caller indicating K_OK: a successful operation, the specified cyclic timer has its initial time changed and also the cyclic timer is started if it was stopped or K_ERROR: the cyclic timer number was out of range.

This is very useful. A cyclic timer could continuously have this function called by a watchdog function. If the watchdog function did not execute or detected something wrong, the cyclic timer would eventually have its time counters decrement to zero, which could then notify a task to perform an orderly shutdown.

### *The K_Timer_Cyclic function*

The *K_Timer_Cyclic* function is very similar to the *K_Timer_Initial* function but is used to change a cyclic timer's cycle time period. This value will override the current cycle time and will be the time used to reload the cyclic timer when current time remaining for the cyclic timer reaches zero.

This function may be called by tasks and interrupts. The cyclic timer's current remaining time period will not change in any way. The caller will identify to the *K_Timer_Cyclic* function the new cycle time period that the cyclic timer should use. This value may range from zero through 65535. If you specify the cyclic time as zero, then the cyclic timer will act as a "one-shot", executing only once when the time remaining expires.

If the current cyclic timer is stopped when the *K_Timer_Cyclic* call is made, then the initial time out value for this cyclic timer will be 65536. After that, the cyclic time passed to this function will be used for the time out period.

This is an example of the *K_Timer_Cyclic* function:

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Timer_Cyclic(byte,word16); /* this is the function prototype */

#define CYCLIC_NUM ??? /* Identifies what cyclic timer. */

#define NEW_CYCLIC_PERIOD ??? /* new cyclic period. */

unsigned char STATUS; /* should be local */

STATUS = K_Timer_Cyclic(CYCLIC_NUM,NEW_CYCLIC_PERIOD);
```

**Passed**

CYCLIC_NUM is the number of the cyclic timer that the caller wants to work with. The number of the cyclic timer may range from 0 to 1 less than the maximum number of cyclic timers declared.

NEW_CYCLIC_PERIOD is the number of system ticks that the cyclic timer will use at the recycle time period. May range from 0 to 65535.

```
#define TIMER_0 0 /* cyclic timer 0 number */
```

```
#define T0_NEW_CYCLE_TIME 100 /* specify that new cycle time be 100 */

void task2(void)
{
  unsigned char status;

  status = K_Timer_Cyclic(TIMER_0,T0_NEW_CYCLE_TIME);
  /* Change cyclic timer 0 cycle time.  The new cycle time will
  not take effect until the current cyclic timer's time
  decrements to 0. */

  if (status != K_OK) /* see if error, should not be */
    {
    /* Take corrective action. */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: cyclic timer number out of range.

If STATUS equals K_OK, the specified cyclic timer has its cyclic time changed and the timer started, if it was stopped.

### *The K_Timer_Restart function*

The *K_Timer_Restart* function restarts a cyclic timer. The current cyclic timer's remaining time and cyclic time are untouched.

This function is very useful for restarting a cyclic timer that has been stopped by the *K_Timer_Stop* function. None of the time values are touched. If the cyclic timer was stopped because it was running in the "one-shot" mode, then when started, the initial time period used for the first execution will be 65536 system ticks.

This is an example of the *K_Timer_Restart* function:

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

#include <cxfuncs.h> /* has function prototype */
byte *K_Timer_Restart*(byte); /* this is the function prototype */

#define CYCLIC_NUM ??? /* Identifies the cyclic timer. */

unsigned char STATUS; /* should be local */

STATUS = *K_Timer_Restart*(CYCLIC_NUM);

**Passed**

CYCLIC_NUM is the number of the cyclic timer that the caller wants to work with. This may range from zero to one less than maximum number of cyclic timers declared.

#define TIMER_0 0 /* cyclic timer 0 number */

```
void task2(void)
{
  unsigned char status;
  status = K_Timer_Restart(TIMER_0);
  /* Restart cyclic timer 0.  None of the cyclic timer's time
  values will be touched. */
  if (status != K_OK) /* see if error, should not be */
    {
    /* Take corrective action. */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: cyclic timer number out of range.

If STATUS equals K_OK, then the specified cyclic timer is restarted if it was stopped. If already started, this function has no effect.

### *The K_Timer_Stop function*

The *K_Timer_Stop* function stops a cyclic timer once it is started. The cyclic timer values (the remaining time period before it executes and the cyclic time period) will not be touched by this function.

The caller only specifies the cyclic timer to stop. This parameter is in the range of zero through one less than the maximum number of cyclic timers declared in the configuration module.

When this function is called, the cyclic timer will be put into the stop mode and removed from the cyclic timer start list. The time remaining will have its value left at the time of the *K_Timer_Stop* function call. Also, the cyclic time period for this timer will not be touched so that when the cyclic timer is restarted using the *K_Timer_Restart* function, the time values for this cyclic timer will be used.

This is an example of the *K_Timer_Stop* function:

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

#include <cxfuncs.h> /* has function prototype */
byte *K_Timer_Stop*(byte); /* this is the function prototype */

#define CYCLIC_NUM ??? /* Identifies what cyclic timer. */

unsigned char STATUS; /* should be local */

STATUS = *K_Timer_Stop*(CYCLIC_NUM);

**Passed**

CYCLIC_NUM is the number of the cyclic timer that the caller wants to work with. This may range from zero to one less than maximum number of cyclic timers declared.

#define TIMER_0 0 /* cyclic timer 0 number */

```
void task2(void)
{
  unsigned char status;

  status = K_Timer_Stop(TIMER_0);
  /* Stop cyclic timer 0.  Cyclic timer 0 will immediately be
  stopped and removed from the cyclic timers run list.  Remember
  that the cyclic timer's time values, both the remaining
  execution time and cyclic time values will be untouched. */

  if (status != K_OK) /* see if error, should not be */
    {
    /* Take corrective action. */
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: cyclic timer number out of range.

If STATUS equals K_OK, then the specified cyclic timer is stopped or if already stopped, then has no effect.

## UART MANAGER FUNCTIONS

The UART manager is NOT part of the CMX library. It contains the following functions listed here with their page references.



These specialized UART (Universal Asynchronous Receiver and Transmitter) functions give you the ability to have the onboard serial UART(s) work in a true interrupt fashion. This way, both receiver and transmitter do not have to be polled while waiting to receive a character or the transmitter to be empty before the next character can be transmitted.

For each UART, there will be a receive buffer and transmit buffer. You determine the size of the buffers for the particular application requirements. The buffer size for the transmitter does not have to be the same size as the receiver. The following will explain how each of the buffers work and how CMX interfaces with them.

### The Receive Buffer

The receive buffer is a circular buffer with a head and tail pointer. The head pointer indicates where the next character will be placed into the buffer. The tail pointer indicates where to get the character from the receive buffer. The characters are retrieved in the order they have been received, this being a FIFO (first in, first out) structure. There is also a count byte, indicating the current number of characters residing in the receive buffer that have not been retrieved yet. If the count byte reaches the size of the receive buffer, then the receive buffer is marked full and no further characters will be placed into the buffer.

The receiver will also automatically wake a task that has requested a specific number of bytes from the receive buffer which are not there, if they become available. The receive buffer can only be "owned" by one task at a time. You will need to use a resource if more than one task wants to use this receiver at the same time.

### The Transmit Buffer

The transmit buffer is a linear type of buffer with a head and tail pointer. The head pointer points to the last character within the buffer that should be transmitted by the interrupt transmit routine. The tail pointer points to the character that will be transmitted out, when the transmitter is free to transmit. Each time the transmit buffer is used, the buffer is reset to the beginning.

The transmit buffer can only be "owned" by one task at a time. You will want to use a resource if more than one task wants to use this transmitter at the same time. If the transmit buffer has had characters loaded into it, then it is considered busy while it transmit these characters out. If this is so, any further request to load more characters into the transmit buffer will be blocked.

The source code of the CMX UART C module will need to be edited to set up the BAUD rate, number of data bits, parity and size of the receive and transmit buffers. Also you may have to set up the address of the specific CPU UART's address and other variables. You may want to add some specific functions to handle the possible receiver errors that may occur such as overrun error, parity error, framing error, etc. Because CMX cannot determine what you may want to do when an receiver error takes place or how you will want to handle this error, coding for this is up to you.

It should also be stated that these C code routines will add a greater amount of interrupt latency than normally would occur without these routines. This is because the UART interrupts can occur at any time and are completely asynchronous to "critical regions of code". Therefore the CMX UART functions must GLOBALLY disable and re-enable the interrupts. Depending upon the CPU and C compiler used, this may range for a few microseconds or more. It is up to you to determine whether the possible added interrupt latency introduced will be satisfactory or not.

You also determine whether the selected BAUD rate will be allowed. This depends on how long it takes these functions to execute and the latency that the disabling and enabling of interrupts will place on the application code and other interrupts.

These functions should meet the requirements of the application program. However, CMX really recommends you code the UART interrupt functions in assembly. This will speed up the time it takes for the UART interrupts to execute and reduce the amount of interrupt latency the above functions introduce. This can be done by setting up flags that indicate the state of a task.

For example, the assembly coded receiver routine could receive characters and place them into the receive buffer you create. When either a END_OF_PACKET character was received or the specified time period elapsed (detected by a timer interrupt routine), the receiver interrupt could call the interrupt pipe to trigger the particular task that will process the received packet of information. The receiver could receive multiple packets by having multiple buffers and then change the buffer pointers the receiver would use. CMX includes some examples for how to program UART interrupt routines in assembly.

### *The K_Init_Recv function*

The *K_Init_Recv* function will initialize the receive buffer by setting its associated pointers to the beginning of the receive buffer. Also the receive count_in variable, that indicates the number of bytes received but not retrieved yet, will be reset to zero. The receiver status flags will be reset indicating the receiver is fine and that no errors, full or otherwise, exist.

You are free to call the *K_Init_Recv* at any time, as long as a task is not waiting for the receiver, because the specified number of characters the task wants, are not yet present. Remember, if there are characters in the receive buffer that have not yet been retrieved, these characters will be lost. The receiver BAUD rate, number of data bits, parity and number of stop bits can be set here.

This is an example of the *K_Init_Recv* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_Init_Recv*(void); /* this is the function prototype */

*K_Init_Recv*();

**Passed**

```
main()
{
```

```
  K_Init_Recv(); /* go initialize the UART receiver */
  ...
}
```

**Returned**

Nothing is returned.

### *The K_Init_Xmit function*

The *K_Init_Xmit* function will initialize the transmit buffer by setting its associated pointers to the beginning of the transmit buffer. Also the transmit count_out variable, indicating the number of bytes which need to be transmitted, will be reset to zero. The transmitter status flags will be reset indicating the transmitter is fine and not busy. You should only call the *K_Init_Xmit* once. There is no reason to ever reinitialize the transmit buffer. The transmitter BAUD rate, number of data bits, parity and number of stop bits can be set here.

This is an example of the *K_Init_Xmit* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_Init_Xmit*(void); /* this is the function prototype */

*K_Init_Xmit*();

**Passed**

```
main()
{
  K_Init_Xmit(); /* go initialize the UART transmitter */
  ...
}
```

**Returned**

Nothing is returned.

### *The K_Update_Xmit function*

The *K_Update_Xmit* function is called by the transmitter interrupt. First the function will determine whether the transmitter empty flag is set. If so, then the transmit count_out variable will be tested for zero. If the count_out variable is non-zero, indicating that there are additional characters to transmit, the transmitter will automatically be loaded with the next buffer character, the count_out variable

decremented and the tail pointer incremented so it is pointing to the next character. If the count_out variable is zero, it will test to see if there is a task waiting to transmit more characters. Remember, only one task should have exclusive ownership of the transmitter (through the use of the resource functions). If the task is waiting, then the task will automatically be put into the RESUME state, indicating it may now put more characters into the transmit buffer when it becomes the highest priority task able to run.

### *The K_Update_Recv function*

If the receive buffer is not full, then the received character will be placed into the receive buffer and the receive head pointer will be incremented to point to the next storage location. The receiver count_in variable will also be incremented, indicating the number of bytes received. If a task is waiting on the receiver for a specified number of characters, then the *K_Update_Recv* function will test to see if the required number of characters are present now. If so, the task will automatically be put into the RESUME state, indicating the required number of characters are present.

### *The K_Put_Char function*

The *K_Put_Char* function allows the task that owns the transmitter to transmit one character. The parameter passed to this function is the address of the character the task wants to transmit. If the transmitter is already transmitting characters, then the character will not be put into the transmit buffer and the function will return a status indicating the transmitter was busy. If the transmitter is not transmitting, then the character will be placed into the transmit buffer. Then the transmitter will automatically be started. The task will be returned to immediately, indicating the operation was successful. The task will not have to wait for the character to be transmitted by the UART. The interrupt system will take care of that.

This is an example of the *K_Put_Char* function:

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Put_Char(void *); /* this is the function prototype */

unsigned char *src_PTR; /* this may be local or global, and will be the location that the
function gets the character. */

unsigned char STATUS; /* results of function. */

STATUS = K_Put_Char(src_PTR);
```

**Passed**

src_PTR = the address where the character resides in memory for this function.

unsigned char src_byte = '3'; /* This is the address to hold the character which will be sent out. Could also be local. */

```
void task1(void)
{
  unsigned char status;
  status = K_Put_Char(&src_byte); /* go send character. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
  or/* this shows another way. */
  status = K_Put_Char("3"); /* go send character. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_BUSY = Error: transmitter was busy.

If STATUS equals K_OK, then the task which called this function will have the character placed into the transmitter and the transmitter will automatically start, sending out the character.

### *The K_Put_Char_Wait function*

The *K_Put_Char_Wait* is like the above function. In addition this task will wait for the specified time period or indefinitely if the transmitter is busy. The parameters passed to this function are the address of the character to transmit and the amount of time it will wait for the transmitter to be free. This function allows the task that owns the transmitter to transmit one character.

If the transmitter is already transmitting characters, then the task will be placed into the suspended state. The task can specify to wait indefinitely by specifying a time out period of zero or to wait for one to 65535 system ticks. If the transmitter becomes free, then the task will automatically awaken, allowing it to then place the character into the transmit buffer. The task is returned to immediately indicating a successful operation. If the specified time period expires prior to the transmitter being free, then the character will not be placed into the transmit buffer and the task will be returned to with the status indicating the time period expired.

This is an example of the *K_Put_Char_Wait* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Put_Char_Wait*(void *,word16); /* this is the function prototype */

unsigned char *src_PTR; /* this may be local or global, and will be the location that the function gets the character. */

#define TIME_PERIOD ??? /* The time wait period. */

unsigned char STATUS; /* results of function. */

STATUS = *K_Put_Char_Wait*(src_PTR,TIME_PERIOD);

**Passed**

src_PTR = the address where the character resides in memory for this function.

TIME_PERIOD is the number of system ticks this task is willing to wait for the transmitter to be free. Range is zero to 65535.

unsigned char src_byte = '3';

/* This is the address to hold the character, which will be sent out. Could also be local. */

```
void task1(void)
{
  unsigned char status;
  status = K_Put_Char_Wait(&src_byte,100); /* go send character.
  Wait up to 100 system ticks for the transmitter to be free if
  need be. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
  or/* this shows another way. */
  status = K_Put_Char_Wait("3",100); /* go send character.
  Wait up to 100 system ticks for the transmitter to be free if
  need be. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
```

```
  . . .
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_TIMEOUT = Warning/error: The time period expired, transmitter still busy.

K_BUSY = Error: transmitter was busy.

If STATUS equals K_OK, then the task which called this function will have the character placed into the transmitter and the transmitter will automatically start, sending out the character.

### *The K_Put_Str function*

The *K_Put_Str* function allows the task that owns the transmitter to transmit one or more characters. The parameters passed to this function are the address of the string of characters to transmit and the count of the number of characters to send. The string of characters do not have to be terminated by any special character and the function allows tasks to send binary characters.

If the transmitter is already transmitting characters, then the string of characters will not be put into the transmit buffer and the function will return a status indicating the transmitter was busy. If the transmitter is not transmitting, then the string of characters will be placed into the transmit buffer according to the count parameter. Then the transmitter will automatically be started. The task will be returned to immediately indicating the operation was successful. The task will not have to wait for the characters to be transmitted by the UART. The interrupt system will take care of that. It is up to you to determine the correct count of characters you want copied from the string source to the transmit buffer.

This is an example of the *K_Put_Str* function:

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Put_Str(void *,word16); /* this is the function prototype */

unsigned char *src_PTR; /* this may be local or global, and will be the location that the
function gets the character. */

#define NUMBER ??? /* the number of characters to place into the transmit buffer. */
```

unsigned char STATUS; /* results of function. */

STATUS = *K_Put_Str*(src_PTR,NUMBER);

**Passed**

src_PTR = the address where the characters reside in memory for this function.

NUMBER is the number of characters to place into the transmit buffer.

unsigned char src_bytes[] = {"From task 1"};

/* This is the address to hold the string of characters, which will be sent out. Could also be local. */

```
void task1(void)
{
  unsigned char status;
  status = K_Put_Str(src_bytes,sizeof src_bytes);
  /* go send character.  Remember it is the second parameter that
  determines the number of characters that are actually copied
  to the transmit buffer. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if characters not sent. */
    }
  ...
  or/* this shows another way. */
  status = K_Put_Str("From task 1",12);
  /* go send character.  Remember it is the second parameter that
  determines the number of characters that are actually copied
  to the transmit buffer. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_BUSY = Error: transmitter was busy.

If STATUS equals K_OK, then the task which called this function will have a NUMBER of character(s) placed into the transmit buffer and the transmitter will automatically start sending out the characters.

☞ *The number of characters placed into the transmit buffer is determined by the NUMBER and not by the length of the string. It is up to you to determine the proper number of characters to transmit.*

***The K_Put_Str_Wait function***

The *K_Put_Str_Wait* is like the above function. In addition, this task will wait for the specified time period or indefinitely if the transmitter is busy. The parameters passed to this function are the address of the string of characters to transmit, the number of characters in the string and the amount of time it will wait for the transmitter to be free. This function allows the task that owns the transmitter to transmit one or more characters.

If the transmitter is already transmitting characters, then the task will be placed into the suspended state. The task can specify to wait indefinitely by specifying a time out period of zero or wait for one to 65535 system ticks. If the transmitter becomes free, then the task will automatically awaken, allowing it to place the characters into the transmit buffer according to the count parameter passed to this function. The task is returned to immediately indicating a successful operation. If the specified time period expires prior to the transmitter being free, then the characters will not be placed into the transmit buffer and the task will be returned to with the status indicating the time period expired.

This is an example of the *K_Put_Str_Wait* function:

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Put_Str_Wait(void *,word16,word16); /* this is the function prototype */

unsigned char *src_PTR; /* this may be local or global, and will be the location that the
function gets the character. */

#define NUMBER ??? /* the number of characters to place into the transmit buffer. */

#define TIME_PERIOD ??? /* The time wait period. */

unsigned char STATUS; /* results of function. */

STATUS = K_Put_Str_Wait(src_PTR,NUMBER,TIME_PERIOD);
```

**Passed**

src_PTR = the address where the character resides in memory for this function.

NUMBER is the number of characters to place into the transmit buffer.

TIME_PERIOD is the number of system ticks that this task is willing to wait for the transmitter to be free. Range is zero to 65535.

unsigned char src_bytes[] = {"Task1 transmitting this"};

/* This is the address to hold the character, which will be sent out. could also be local. */

```
void task1(void)
{
  unsigned char status;
  status = K_Put_Str_Wait(src_bytes,size of src_bytes, 100);
  /* go send characters.  Wait up to 100 system ticks for the
  transmitter to be free. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
  or/* this shows another way. */
  status = K_Put_Str_Wait("Task1 running",14,100);
  /* go send characters.  Wait up to 100 system ticks for the
  transmitter to be free. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_TIMEOUT = Warning/error: The time period expired, transmitter still busy.

K_BUSY = Error: transmitter was busy.

If STATUS equals K_OK, then the task which called this function will have the character placed into the transmitter and the transmitter will automatically start sending out the character.

### *The K_Get_Char function*

The *K_Get_Char* allows a task that owns the receiver to retrieve a character from the receive buffer if there is one available. The parameter passed to the *K_Get_Char* function is the address where the task would like the character to be copied to. If the receive buffer has one or more characters in its buffer, then only one character will be placed at the destination address.

If the receive buffer is empty, indicating either no characters have been received by the UART or that all characters that have been received and retrieved, then no character will be placed into the destination address.

The task will be returned to immediately with the value specifying the number of characters transferred. If the return value is zero, no character was retrieved. Otherwise the task will receive a count of one indicating one character was retrieved. Once a character has been retrieved, the tail pointer of the receive buffer is incremented and the receive buffer count_in variable is decremented.

This is an example of the *K_Get_Char* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
word16 *K_Get_Char*(void *); /* this is the function prototype */

unsigned char *DEST_PTR; /* this may be local or global, and will be the location that the function puts the character. */

unsigned short COUNT; /* this will specify the number of characters retrieved from buffer. */

COUNT = *K_Get_Char*(DEST_PTR);

**Passed**

DEST_PTR = the address where this function will place the character.

unsigned char recv_array[80];

/* this will receive up to 80 characters, could be local if user wants or could just be a depth of 1 in this case for this function only returns one character at the most. */

```
void task1(void)
{
  unsigned short count; /* local */
  unsigned char c;
```

```
  unsigned char *recv_ptr; /* could be local or global. */

  recv_ptr = recv_array; /* load pointer with address of
  recv_array. */
  count = K_Get_Char(recv_ptr); /* go get character. */
  if (count)
    {
    c = *recv_ptr++; /* get character and increment pointer. */
    ...
    }
  ...
  or/* this shows another way. */
  count = K_Get_Char(&c); /* go get character. */
  if (count)
    {
    .../* process c here, etc. */
    }
  ...
}
```

**Returned**

COUNT = the number of characters that were transferred from the UART receive buffer to the destination address.

If the COUNT value is zero, there were no characters in the UART receive buffer when this function was called. If the COUNT is non-zero, (which for this function would be 1), then one character was transferred to the destination address passed to this function. Make note that the task would just have to pass the address of a character type variable, instead of loading a pointer with the address and then passing the pointer (which really passes the address that the pointer has).

### *The K_Get_Char_Wait function*

The *K_Get_Char_Wait* function is like the above function. This function also has the ability to wait for a specific amount of time for the character to arrive if the receive buffer is empty. Also, this function allows a task that owns the receiver to retrieve a character from the receive buffer, if there is one available. If there is not a character in the receive buffer, then the task will wait for the specified time period. The parameters passed to this function are the address where the character is to be copied and the amount of time it is willing to wait if a character is not present. The time period may be zero indicating an indefinite wait or a time period of one to 65535 system ticks.

If the receive buffer has one or more characters in its buffer, then only one character will be placed at the destination address. The task will be returned to immediately with the return value of one indicating a good operation.

If the receive buffer is empty, indicating either no characters have been received by the UART or that all characters have been received and retrieved, then the task will be placed into the suspended state waiting for either the time period to expire or the receive buffer to receive a character. If the specified time period expires before a character is received, then the task will be returned to with the count being zero indicating the time period had expired and no character was transferred.

If the receiver receives a character while the task is suspended the receiver will automatically wake the task, copy the character to the destination address, and return with a value of one indicating a successful operation.

Once a character has been retrieved, the tail pointer of the receive buffer is incremented and the receive buffer count_in variable is decremented.

This is an example of the *K_Get_Char_Wait* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
word16 *K_Get_Char_Wait*(void *,word16); /* this is the function prototype */

unsigned char *DEST_PTR; /* this may be local or global, and will be the location that the function puts the character. */

#define TIME_PERIOD ??? /* The time wait period. */

unsigned short COUNT; /* this will specify the number of characters retrieved from buffer. */

COUNT = *K_Get_Char_Wait*(DEST_PTR,TIME_PERIOD);

**Passed**

DEST_PTR = the address where this function will place the character.

TIME_PERIOD is the number of system ticks that this task will wait for a character to be received. May range from zero to 65535, with zero indicating an indefinite wait.

unsigned char recv_array[80];

/* this will receive up to 80 characters, could be local if you want or could just be a depth of one in this case since this function only returns one character at the most. */

```
void task1(void)
{
  unsigned short count; /* local */
```

```
  unsigned char c;
  unsigned char *recv_ptr; /* could be local or global. */

  recv_ptr = recv_array; /* load pointer with address of
  recv_array. */
  count = K_Get_Char_Wait(recv_ptr,20);
  /* go get character, if there is no character, wait for 20
  system ticks for one to arrive. */
  if (count)
    {
    c = *recv_ptr++; /* get character and increment pointer. */
    ...
    }
  ...
  or/* this shows another way. */
  count = K_Get_Char_Wait(&c,20); /* go get character. */
  if (count)
    {
    .../* process c here, etc. */
    }
  ...
}
```

**Returned**

COUNT = the number of characters that were transferred from the UART receiver buffer to the destination address.

If the COUNT value is zero, there were no characters in the UART receiver buffer when this function was called, or the time period expired. If the COUNT is non-zero, (which for this function would be one), then one character was transferred to the destination address passed to this function.

### *The K_Get_Str function*

The *K_Get_Str* allows a task that owns the receiver to retrieve one or more characters from the receive buffer, if the desired number of characters is available. The parameters passed to the *K_Get_Str* function are the address where the task would like the characters to be copied to and the number of characters to retrieve from the receive buffer.

If the receive buffer has at least the desired number of characters in its buffer, then the characters will be copied to the destination address. If the receive buffer does not have the desired number of characters according to the count parameter, then no characters will be copied.

The task will be returned to immediately, with a value specifying the number of characters transferred. If the return value is zero, no characters were retrieved. If characters were retrieved, the return value will specify that number. Each time a character is copied to the destination address, the tail pointer of the receive buffer is incremented and the receive buffer count_in variable is decremented.

This is an example of the *K_Get_Str* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
word16 *K_Get_Str*(void *,word16); /* this is the function prototype */

unsigned char *DEST_PTR; /* this may be local or global, and will be the location that the function puts the character. */

#define NUMBER ??? /* number of characters to get. */

unsigned short COUNT; /* this will specify the number of characters retrieved from buffer. */

COUNT = *K_Get_Str*(DEST_PTR,NUMBER);

**Passed**

DEST_PTR = the address where this function will place the character.

NUMBER is the number of characters to retrieve from the UART receive buffer. Only the specified number will be transferred.

unsigned char recv_array[80];

/* this will receive up to 80 characters, could be local if you want or could just be a depth of the maximum number that this task would ask for. */

```
void task1(void)
{
  unsigned short count; /* local */
  unsigned char c;
  unsigned char *recv_ptr; /* could be local or global. */

  recv_ptr = recv_array; /* load pointer with address of
  recv_array. */
  count = K_Get_Str(recv_ptr,5); /* go get 5 characters. */
  while (count--)
    {
    c = *recv_ptr++; /* get character and increment pointer. */
```

```
    ...
    }
  ...
}
```

Comments

This allows a task to retrieve a specific number of characters from the UART receive buffer if that number is available. The task will not wait for that number of characters to come in.

**Returned**

COUNT = the number of characters that were transferred from the UART receiver buffer to the destination address.

If the COUNT value is zero, the required NUMBER of characters were not in the UART receive buffer when this function was called. If the COUNT is non-zero, then the value will represent the number of characters that were transferred to the destination address passed to this function. The COUNT value should match the NUMBER passed in this case.

***The K_Get_Str_Wait function***

The *K_Get_Str_Wait* function is like the above function. In addition, this function has the ability to wait for a specific amount of time for the characters to arrive if the receive buffer does not have the desired number of characters present.

This function allows a task that owns the receiver to retrieve a specified number of characters from the receive buffer if they are available. If the desired number of characters are not in the receive buffer, the task will wait for the specified time period to receive that number.

The first parameter passed to the *K_Get_Str_Wait* function is the address the task would like the characters to be copied to. The second parameter is the number of characters the task would like to retrieve from the receive buffer. Also passed is the amount of time it is willing to wait if the desired number of characters is not present. The time period may be zero indicating an indefinite wait or a time period of one to 65535 system ticks.

If the receive buffer has at least the desired number of characters in its buffer, that number of characters will be copied to the destination address. The task will be returned to immediately indicating the number of characters that were transferred.

If the receive buffer does not have the desired number of characters, the task will be placed into the suspended state waiting for either the time period to expire or the receive buffer to receive the specified number of characters. If the specified time period expires before the specified number of characters is received, then the task will be returned to with a value set at zero. This indicates the time period had expired and no character was transferred. If the desired number of characters are received before the time period expires, the receiver will automatically wake up the suspended task.

The *K_Get_Str_Wait* function will then copy the specified number of characters from the receive buffer to the destination address. The task will be returned to immediately with a count of the number of characters transferred (the same as the requested number) indicating a good operation.

Each time a character has been retrieved, the tail pointer of the receive buffer is incremented and the receive buffer count_in variable is decremented.

This is an example of the *K_Get_Str_Wait* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
word16 *K_Get_Str_Wait*(void *,word16,word16); /* this is the function prototype */

unsigned char *DEST_PTR; /* this may be local or global, and will be the location that the function puts the character. */

#define NUMBER ??? /* number of characters wanted. */

#define TIME_PERIOD ??? /* The time wait period. */

unsigned short COUNT; /* this will specify the number of characters retrieved from buffer. */

COUNT = *K_Get_Str_Wait*(DEST_PTR,NUMBER,TIME_PERIOD);

**Passed**

DEST_PTR = the address where this function will place the characters.

TIME_PERIOD is the number of system ticks that this task will wait for characters to be received. May range from zero to 65535, with zero indicating an indefinite wait.

NUMBER is the number of characters to retrieve from the UART receive buffer. Only the specified number will be transferred.

unsigned char recv_array[80];

/* this will receive up to 80 characters, could be local if you want or could just be a depth of the specified number that the task will want to retrieve. */

```
void task1(void)
{
  unsigned short count; /* local */
  unsigned char c;
  unsigned char *recv_ptr; /* could be local or global. */

  recv_ptr = recv_array; /* load pointer with address of
  recv_array. */
  count = K_Get_Str_Wait(recv_ptr,40,20);
  /* go get 40 characters, if there are not 40 characters, wait
  for up to 20 system ticks for 40 to arrive. */
  while (count--)
    {
    c = *recv_ptr++; /* get character and increment pointer. */
    ...
    }
  ...
}
```

**Returned**

COUNT = the number of characters that were transferred from the UART receive buffer to the destination address.

If the COUNT value is zero, the number of characters requested was not present in the UART receive buffer and the time period expired. If the COUNT is non-zero, then the number of characters requested was transferred to the destination address passed to this function.

### *The K_Get_Str_Wait_Return function*

The *K_Get_Str_Wait_Return* function is like the above function. In addition, this function retrieves the receive buffer characters that are present after the time out period expires, even if the specified count is not present.

If the desired number of characters are not in the receive buffer, then the task will wait for the specified time period for the receive buffer to receive that number.

The first parameter passed to the *K_Get_Str_Wait_Return* function is the address the task would like the characters to be copied to. The second parameter is the number of characters the task would like to retrieve from the receive buffer. Also passed is the amount of time it is willing to wait if the desired number of characters are not present. The time period may be zero, indicating an indefinite wait, or a time period of one to 65535 system ticks.

If the receive buffer has at least the desired number of characters in its buffer, then that number of characters will be copied to the destination address. The task will be returned to immediately indicating the number of characters that were transferred.

If the receive buffer does not have the desired number of characters, then the task will be placed into the suspended state, waiting for either the time period to expire or the receive buffer to receive the specified number of characters. If the specified time period expires before the characters are received, then the *K_Get_Str_Wait_Return* function will still copy the actual number of characters that have been received (which will be less than the specified number). The task will then be returned to with a value indicating the true number of characters that were transferred.

If the desired number of characters are received before the time period expires, the receiver will automatically wake up the suspended task. The *K_Get_Str_Wait_Return* function will then copy the specified number of characters from the receive buffer to the destination address supplied. The task will be returned to immediately with a count of the number of characters transferred (which will be the same as the requested number) indicating a good operation. This is a very powerful function and designed for protocols with variable length data packets that are sent and received.

Each time a character has been retrieved, the tail pointer of the receive buffer is incremented and the receive buffer count_in variable is decremented.

This is an example of the *K_Get_Str_Wait_Return* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
word16 *K_Get_Str_Wait_Return*(void *,word16,word16); /* this is the function prototype */

unsigned char *DEST_PTR; /* this may be local or global, and will be the location that the function puts the character. */

#define NUMBER ??? /* number of characters wanted. */

#define TIME_PERIOD ??? /* The time wait period. */

unsigned short COUNT; /* this will specify the number of characters retrieved from buffer. */

COUNT = *K_Get_Str_Wait_Return*(DEST_PTR,NUMBER,TIME_PERIOD);

**Passed**

DEST_PTR = the address where this function will place the characters.

TIME_PERIOD is the number of system ticks this task will wait for characters to be received. May range from zero to 65535, with zero indicating an indefinite wait.

NUMBER is the number of characters to retrieve from the UART receive buffer.

unsigned char recv_array[80];

/* this will receive up to 80 characters, could be local if user wants or could just be a depth of the specified number that the task will want to retrieve. */

```
void task1(void)
{
  unsigned short count; /* local */
  unsigned char c;
  unsigned char *recv_ptr; /* could be local or global. */

  recv_ptr = recv_array; /* load pointer with address of
  recv_array. */
  count = K_Get_Str_Wait_Return(recv_ptr,40,20);
  /* go get 40 characters, if there are not 40 characters, wait
  for up to 20 system ticks for 40 to arrive.  Still retrieve the
  number of characters in the buffer after time period expires.
  */
  if (count < 40)
    {
    .../* maybe test to see if count requested less than count
    received and possibly act on it in some way. */
    while (count--)
      {
      c = *recv_ptr++; /* get character and increment pointer. */
      ...
      }
    }
  ...
}
```

**Returned**

COUNT = the number of characters that were transferred from the UART receive buffer to the destination address.

If the COUNT value is zero, there were no characters in the UART receive buffer when the time period expired. If the COUNT is non zero, then the number of characters requested or present after time period expired were transferred to the destination address passed to this function.

### *The K_Get_Str_Return function*

The *K_Get_Str_Return* function allows a task to retrieve all the UART receive buffer characters present when this function call is performed. No time out is allowed and the task will not wait on any time period. The count returned will be the number of characters transferred to the destination address passed to this function from the task.

The task could call the *K_Recv_Count* function to determine the approximate number of characters in the receive buffer. (Approximate as the *K_Recv_Count* passes the true number of characters at the time of its call, but other characters may be received prior to the *K_Get_Str_Return* function executing.)

This is a very useful function. A task may call the *K_Task_Wait* function to wait indefinitely. When the UART receive interrupt determines the UART had received the proper number of characters (variable length packet), then the interrupt could use the *K_Task_Wake* function to wake the task that was suspended. When the task resumes running, it could then issue the *K_Get_Str_Return* function to retrieve the characters and process them. Also note the task could just be idle and the interrupt could use the *K_Task_Start* function to start the task.

This is an example of the *K_Get_Str_Return* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
word16 *K_Get_Str_Return*(void *); /* this is the function prototype */

unsigned char *DEST_PTR; /* this may be local or global, and will be the location that the function puts the character. */

unsigned short COUNT; /* this will specify the number of characters retrieved from buffer. */

COUNT = *K_Get_Str_Return*(DEST_PTR);

**Passed**

DEST_PTR = the address where this function will place the characters.

unsigned char recv_array[80];

/* the size of the destination the characters will be transferred to, should be as large as the receive buffer. This way the function will not transfer characters into the wrong memory locations. */

```
void task1(void)
```

```
{
  unsigned short count; /* local */
  unsigned char c;
  unsigned char *recv_ptr; /* could be local or global. */
  recv_ptr = recv_array; /* load pointer with address of
  recv_array. */
  count = K_Get_Str_Return(recv_ptr);
  /* go get what ever number of characters that are present in
  the UART receive buffer at the time of this call. */
  while (count--)
    {
    c = *recv_ptr++; /* get character and increment pointer. */
    ...
    }
  ...
}
```

☞ *This is very useful when a task is either IDLE or waiting indefinitely. The receiver interrupt could call the K_Task_Start or K_Task_Wake functions respectively, telling the task to now get the characters in the buffer since the variable length packet had arrived.*

**Returned**

COUNT = the number of characters that were transferred from the UART receive buffer to the destination address.

If the COUNT value is zero, there were no characters in the UART receive buffer when this function was called. If the COUNT is non-zero, then the number of characters present in the receive buffer were transferred to the destination address passed to this function.

***The K_Recv_Count function***

The *K_Recv_Count* function allows a task to obtain the number of characters that currently reside in the UART receive buffer.

This is an example of the *K_Recv_Count* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */

unsigned short COUNT; /* this will specify the number of characters in receiver buffer. */

COUNT = *K_Recv_Count*();

**Passed**

Nothing is passed.

```
void task1(void)
{
  count = K_Recv_Count();
  /* Go find out how many characters are currently in the UART
  receive buffer. */
  /* now possibly use a function to retrieve one or more
  characters from the receive buffer. */
  ...
}
```

☞ *The receiver may receive one or more characters after this function passes back the count.*

**Returned**

COUNT = the number of characters that are present in the UART receive buffer at the time of this function call.

If the COUNT value is zero, there were no characters in the UART receive buffer when this function was called. If the COUNT is non-zero, it is the number of characters currently in the receive buffer.

## THE OPERATING SYSTEM FUNCTIONS

These functions are part of the CMX operating system and are described on their respective pages, also listed.

### *The K_OS_Init function*

This function is called to initialize the CMX variables, parameters and configurable system maximums. The *K_OS_Init* function must be called before any other CMX functions are called. This is done in the user's start up code. The file containing this function should be compiled each time you change the "CXCONFIG.H" file, which declares the application's maximums.

This is an example of the *K_OS_Init* function:

**Called**

Before using any other CMX function calls.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Init*(void); /* this is the function prototype */

*K_OS_Init*();

**Passed**

Nothing.

**Returned**

Nothing.

☞ *Remember this function must be called before any other CMX function is called. If not, then disastrous results will occur.*

*K_OS_Init* Example:

```
void main(void)
{
  /* define any locals within main */

  K_OS_Init(); /* initialize CMX */
  ...
  /* now the user can access any CMX function, that is allowed
  to be accessed prior to entering the CMX operating system */
  ...
}
```

### *The K_OS_Start function*

The *K_OS_Start* function is called to invoke the CMX operating system. Once this function is called, the CMX operating system takes control of the CPU and determines when tasks should run and cyclic timers should execute. It is up to you to make sure at least one task is READY or will become READY by using the *K_Task_Start* function before calling *K_OS_Start*.

The *K_Task_Start* function call may be called within the start up code or by an interrupt after entering the CMX operating system. If none of the tasks become READY, then the CMX operating system will own all the CPU time except to allow interrupts and cyclic timers, if so started, to execute.

This is an example of the *K_OS_Start* function:

**Called**

When you want to enter the CMX operating system.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Start*(void); /* this is the function prototype */

*K_OS_Start*();

**Passed**

Nothing.

**Returned**

Never returns from the CMX operating system.

*K_OS_Start* Example:

```
void main(void)
{
  /* define any locals within main */

  K_OS_Init(); /* initialize CMX */
  ...
  /* now the user can access any CMX function, that is allowed
  to be accessed prior to entering the CMX operating system */
  ...
  /* Set up CMX by create tasks, cyclic timers, etc.  Possibly
  create queues, set up mailboxes, etc.  Start at least one task.
  */
  ...
  K_OS_Start(); /* enter into the CMX operating system */
  /* NOTE: will never return to this point */
```

```
}
```

### *The K_OS_Disable_Interrupts function*

Purpose: This function GLOBALLY disables the interrupts. Any non-maskable interrupt will not be immediately recognized. If the interrupt sets a latch, then this will not be prevented. This uses the particular CPU instruction that masks out all non-maskable interrupts from being acknowledged and processed.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Disable_Interrupts*(void); /* this is the function prototype */

*K_OS_Disable_Interrupts*();

**Passed**

Nothing is passed.

**Returned**

Nothing is returned.

Read the particular CPU manual that further describes the action and the effect it has on interrupts when the interrupts are GLOBALLY disabled.

*K_OS_Disable_Interrupts* example:

```
void task1(void)
{
  unsigned char status; /* local */
  .../* application code here */

  K_OS_Disable_Interrupts(); /* do critical region of code
  stuff. */
  ...
  K_OS_Enable_Interrupts(); /* re-enable interrupts. */

  .../* application code here */
}
```

Comments

Use this function sparingly, or not at all, if possible. ALL maskable interrupts will not be recognized until the interrupts are re-enabled. This will also add latency time to the interrupt processing its code. If used, the user must remember to re-enable interrupts using the *K_OS_Enable_Interrupts* function.

Some C vendor compilers have special instructions that will allow the interrupts to be GLOBALLY disabled and the code placed inline. This executes faster than calling the equivalent CMX function.

### *The K_OS_Enable_Interrupts function*

Purpose: This function GLOBALLY enables interrupts. It is used in conjunction with the *K_OS_Disable_Interrupts* function. All non-maskable interrupts that were GLOBALLY disabled, will be enabled. If any interrupt is pending, then the interrupt will now be recognized and processed according to the CPU interrupt hardware mechanism and priority scheme.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Enable_Interrupts*(void); /* this is the function prototype */

*K_OS_Enable_Interrupts*();

**Passed**

Nothing is passed.

**Returned**

Nothing is returned.

Read the particular CPU manual that further describes the action and the effect on interrupts when they are GLOBALLY enabled.

*K_OS_Enable_Interrupts* example:

```
void task1(void)
{
  unsigned char status; /* local */
  .../* application code here */

  K_OS_Disable_Interrupts(); /* do critical region of code
  items. */
  ...
  K_OS_Enable_Interrupts(); /* re-enable interrupts. */
```

```
  .../* application code here */
}
```

Comments

This function should be called as soon as possible after the *K_OS_Disable_Interrupts* function is called. Some C vendor compilers have special instructions that allow the interrupts to be GLOBALLY enabled and the code placed inline. This executes faster than calling the equivalent CMX function.

### *The K_OS_Intrp_Entry function*

The *K_OS_Intrp_Entry* function is used by most interrupts. The interrupt's first instruction is to call the *K_OS_Intrp_Entry* function. This informs CMX it should save the context of the CPU registers and swap in the interrupt stack when a interrupt occurs. See the chapter on Processor Specific Information for details on the particular CPU you are working with.

An interrupt does not have to call the *K_OS_Intrp_Entry* function. If it does not, then it is up to you to properly save and restore the contents of any register the interrupt will use. Also, that interrupt cannot use any CMX function calls. The interrupt's code must not call the *K_OS_Intrp_Exit* function when finished.

Please read the Processor Specific Information chapter which fully describes how an interrupt must call this function.

This is an example of the *K_OS_Intrp_Entry* function:

**Called**

Interrupts only.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Intrp_Entry*(void); /* this is the function prototype */

☞ *The call must be written in assembly language to the specific CPU instruction set.*

call *K_OS_Intrp_Entry* ;written in assembly some CPU'S

jsr *K_OS_Intrp_Entry* ;other CPU'S. etc.

**Passed**

Nothing.

**Returned**

Nothing.

☞ *Interrupts are disabled when this function returns. The user may allow this interrupt to re-enable interrupts or not. Interrupts such as the NMI cannot be disabled though. When this function is called, the task (if one is RUNNING) contexts will be saved. If this is the first interrupt (CMX allows nested interrupts), then the interrupt stack will be switched in.*

*K_OS_Intrp_Entry* example:

☞ *Written in assembly.*

```
INTERRUPT_X_HANDLER: ;this is where the specific interrupt will vector to, when
;the interrupt is recognized by the CPU.

; prologue code if need be.  DEPENDENT on CPU.

;

call K_OS_Intrp_Entry ;written in assembly some CPU'S

jsr K_OS_Intrp_Entry ;Hitachi H8/300 series
```

Now the interrupt may finish the code necessary to process this interrupt. Some CMX calls are allowed. The interrupt code could now call the interrupt function code written in C language. This is not recommended though, since interrupts should be as fast as possible. Also at this point this interrupt may re-enable interrupts. If you decide to re-enable interrupts, make sure that either this interrupt is masked out (won't be acknowledged again) or that the CPU hardware will not vector to this interrupt again until this interrupt has finished its code.

```
;interrupt's code ...

call K_OS_Intrp_Exit ;epilogue, must be called by interrupts that have used the
K_OS_Intrp_Entry call.
```

If the interrupt used the *K_OS_Intrp_Entry* call, then the CPU's instruction that indicates return from interrupt (RETI on 8051 CPU, RTE on the Hitachi H8/300 series CPU's) is not needed. CMX will inform the CPU that the interrupt is finished within the *K_OS_Intrp_Exit* code.

***The K_OS_Intrp_Exit function***

The *K_OS_Intrp_Exit* function is called by most interrupts. This function is the last instruction the interrupt's code will call. The interrupt that called the *K_OS_Intrp_Entry* should not use the instruction that signifies RETURN FROM INTERRUPT. When this function is called, CMX will automatically issue the RETURN FROM INTERRUPT instruction, informing the CPU that the interrupt has finished.

Also the CMX operating system will determine whether to restore the contexts of the task or interrupt, or to perform a rescheduling, allowing the highest priority task in the READY state to run.

For most processors, the *K_OS_Intrp_Entry* and *K_OS_Intrp_Exit* functions MUST be called from an assembler routine. A C function can be called to handle the bulk of the interrupt processing, if desired, after *K_OS_Intrp_Entry* is called.

Please read the Processor Specific Information chapter that fully describes how an interrupt must call this function.

This is an example of the *K_OS_Intrp_Exit* function:

**Called**

Interrupts only.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Intrp_Exit*(void); /* this is the function prototype */

☞ *Call must be written in assembly language to the specific CPU instruction set.*

call *K_OS_Intrp_Exit* ;written in assembly some CPU'S

jsr *K_OS_Intrp_Exit* ;other CPU'S. etc.

**Passed**

Nothing.

**Returned**

Does not return.

☞ *If the interrupt is nested (meaning this interrupt is at least the second interrupt) then this call will return to the prior interrupt's code. If this is the first interrupt, then this call will invoke the scheduler. The scheduler will determine whether the interrupt used a CMX call that requires a task swap, or that the task RUNNING prior to this interrupt should still be the RUNNING task.*

*K_OS_Intrp_Exit* example:

☞ *Written in assembly.*

INTERRUPT_X_HANDLER: ;this is where the specific interrupt will vector to, when
;the interrupt is recognized by the CPU.

; prologue code if need be. DEPENDENT on CPU.

;

;call *K_OS_Intrp_Entry* ;written in assembly some CPUs

;jsr *K_OS_Intrp_Entry* ;Hitachi H8/300 series

Now the interrupt may finish the code necessary to process this interrupt. Some CMX calls are allowed. The interrupt code could now call the interrupt function code written in C language. This is not recommended though, since interrupts should be as fast as possible. Also at this point this interrupt may re-enable interrupts. If you decide to re-enable interrupts, make sure that either this interrupt is masked-out (won't be acknowledged again) or that the CPU hardware will not vector to this interrupt again until this interrupt has finished its code.

;interrupt's code ...

call *K_OS_Intrp_Exit* ;epilogue, must be called by interrupts that have used the *K_OS_Intrp_Entry* call.

If the interrupt used the *K_OS_Intrp_Entry* call, then the CPU instruction that indicates return from interrupt (RETI on 8051 CPU, RTE on the Hitachi H8/300 series CPU's) is not needed. CMX will inform the CPU that the interrupt is finished within the *K_OS_Intrp_Exit* code.

### *The K_OS_Slice_On function*

The *K_OS_Slice_On* function allows you to turn on time slicing. This allows tasks with the same priority to be time sliced according to the TSLICE_SCALE value you declared within the "CXCONFIG.H" file. No parameters are passed to this function and only tasks may call it.

See the Time Slice Chapter on time slicing for a complete explanation on how time slicing works in the CMX operating system.

This is an example of the *K_OS_Slice_On* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Slice_On*(void); /* this is the function prototype */

*K_OS_Slice_On*();

**Passed**

Nothing is passed.

**Returned**

Nothing is returned.

Study the Time Slice Chapter to better understand how time slicing works.

*K_OS_Slice_On* example:

```
void task1(void)
{
  unsigned char status; /* local */

  K_OS_Slice_On(); /* enable time slicing. */

  .../* application code here */

  K_OS_Slice_Off();
  /* disable time slicing, if user wants. */
  ...
}
```

Comments

You can freely allow tasks to enable and disable time slicing when necessary. In most cases time slicing will not be needed.

***The K_OS_Slice_Off function***

The *K_OS_Slice_Off* function allows you to turn off time slicing. This will disable time slicing if it was enabled by the *K_OS_Slice_On* function. No parameters are passed to this function and only tasks may call it.

See the Time Slice Chapter on time slicing for a complete explanation on how time slicing works in the CMX operating system.

This is an example of the *K_OS_Slice_Off* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Slice_Off*(void); /* this is the function prototype */

*K_OS_Slice_Off*();

**Passed**

Nothing is passed.

**Returned**

Nothing is returned.

Study the Time Slice Chapter to better understand how time slicing works.

*K_OS_Slice_Off* example:

```
void task1(void)
{
  unsigned char status; /* local */

  K_OS_Slice_On(); /* enable time slicing. */

  .../* application code here */

  K_OS_Slice_Off(); /* disable time slicing. */
  ...
}
```

Comments

You can freely allow tasks to enable and disable time slicing when necessary. In most cases, time slicing will not be needed.

### *The K_OS_Tick_Update function*

This function is usually called by a timer interrupt. CMX needs one of the timer interrupts to use as a clock to perform scheduling for tasks which have used a function that invokes a time period, and for cyclic timers.

The *K_OS_Tick_Update* function MUST be called by an interrupt. This interrupt is selectable and should cause an interrupt at a specified time period. The interrupt's frequency should be constant since all time related activities are based on this frequency. The interrupt should first call the *K_OS_Intrp_Entry* function, then call the *K_OS_Tick_Update* function. The *K_OS_Tick_Update* function will decrement a counter that has been pre-loaded with the number of times the interrupt must call this function, before the scheduler flag is set. This counter is loaded with a value you select in the "CXCONFIG.H" file and the C define CMX_RTC_SCALE, as described in the RTOS Configuration File chapter.

When the count specified by the CMX_RTC_SCALE reaches zero, then the counter will be reloaded with the CMX_RTC_SCALE value, and the CMX time flag will be set if any task timers or cyclic timers need servicing. When the interrupt leaves the *K_OS_Intrp_Exit* function, the scheduler will determine whether to let the CMX timer task execute or to resume execution of the current running task. If the scheduler determines the CMX timer task should execute, then the task that was running prior to this interrupt will be put into the ready to RESUME state, saving its context. The scheduler will then let the CMX timer task execute. (See the Scheduler chapter for a more detailed description on how the scheduler works.)

If the current task prior to the interrupt used the CMX *K_Task_Lock* function, then the CMX time flag would be set and the counter reloaded according to the CMX_RTC_SCALE value. This would not invoke the scheduler when the interrupt leaves. This is because the privilege flag has been set by the *K_Task_Lock* function. The privilege flag will not allow any task switch until the privilege flag has been lowered (cleared), which must be done by the task that raised it. To lower the privilege flag, the task must call the CMX *K_Task_Unlock* function.

### *The K_OS_Low_Power_Func function*

The *K_OS_Low_Power_Func* function is called by the CMX scheduler module written in assembly. It is up to you to code this function to invoke the CPU power down mode.

You will most likely use the processor's instruction that reduces power, yet allows interrupts to wake the processor up. This is because the interrupt used to call the *K_OS_Tick_Update* function should be allowed to happen. This allows the interrupt to wake the processor up and the CMX timer task to execute and decrement any cyclic timers' time count and/or suspend tasks to decrement their time count also.

Be careful when coding this. Depending on the CPU and the power down mode selected, this function may do different things. The CMX assembly module assumes the function will return to the instruction that called it. You must ensure the power down mode exits properly and returns to the next instruction with all registers and the stack in the state they were prior to the *K_OS_Low_Power_Func* function being called.

You may write this function in assembly language and manipulate the registers and/or stack to ensure the processor returns to the instruction following the instruction that called the *K_OS_Low_Power_Func* function.

CMX provides an empty *K_OS_Low_Power_Func* function so no power down state is entered unless you code one. It is highly recommended this function be written after the application code has been fully written, debugged and tested.

This is an example of the *K_OS_Low_Power_Func* function:

**Called**

By the CMX scheduler.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Low_Power_Func*(void); /* this is the function prototype */

...

"in assembly coded scheduler module"

call *K_OS_Low_Power_Func*

**Passed**

Nothing.

**Returned**

Nothing.

☞ *Remember, you should code this function for the power down state you would like and to ensure the processor returns to the instruction following the call to K_OS_Low_Power_Func.*

*K_OS_Low_Power_Func* Example:

The scheduler will determine when NO tasks are able to run, because the tasks are either IDLE or SUSPENDED.

call *K_OS_Low_Power_Func* ;proper assembly code instruction by CMX within the scheduler assembly code.

```
void K_OS_Low_Power_Func(void)
{
  .../* user written routine here. */
  /* NOTE: may be coded in assembly if need be for the particular
  CPU and / or C vendor tools. */
}
```

***The K_OS_Task_Slot_Get function***

This function returns the task slot number of the currently RUNNING task. The slot number was assigned by the CMX *K_Task_Create* function when the task was created.

This is an example of the *K_OS_Task_Slot_Get* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_OS_Task_Slot_Get*(void); /* this is the function prototype */

unsigned char TASK_SLOT;  /* could be global or local */

TASK_SLOT = *K_OS_Task_Slot_Get*();

**Passed**

Nothing.

```
void task2(void)
{
  unsigned char task_slot;

  task_slot = K_OS_Task_Slot_Get();
  /*  returns the task slot number of the task that is currently
  in the RUNNING state */
}
```

**Returned**

TASK_SLOT is the task slot number of the currently RUNNING task.

***The K_OS_Tick_Get_Ctr function***

Every time the CMX *K_OS_Tick_Update* function is called by the timer interrupt, a global variable called K_tick_count is incremented. This variable contains the running total of the "true" system ticks. The current value of K_tick_count is returned by the *K_OS_Tick_Get_Ctr* function.

See the *K_OS_Tick_Update* section for a complete explanation on how the *K_OS_Tick_Update* function works in the CMX operating system.

This is an example of the *K_OS_Tick_Get_Ctr* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
word32 *K_OS_Tick_Get_Ctr*(void); /* this is the function prototype */

unsigned long TICK_COUNT;  /* could be global or local */

TICK_COUNT = *K_OS_Tick_Get_Ctr*();

**Passed**

Nothing.

```
void task2(void)
{
  unsigned long tick_count;

  tick_count = K_OS_Tick_Get_Ctr();
  /*  returns the total number of system ticks (updated by one
  each time K_OS_Tick_Update was called) that have occurred since
  the RTOS has started */
}
```

**Returned**

TICK_COUNT is the current total number of CMX system ticks. This value will be the total number of times the *K_OS_Tick_Update* function was called.

## THE CMX TIMER TASK

The CMX timer task is created by CMX. It must be created first and is done automatically by the CMX *K_OS_Init* function. (You should never use zero for any CMX function that requires a task slot number. If you do, an error will be returned and the CMX function called will not perform its code.)

The timer task is called by the scheduler when the do_timer_tsk flag is set. The do_timer_tsk flag is set by the *K_OS_Tick_Update* function when there is a need for the timer task to execute. This need arises when there are tasks waiting on a time out period and/or cyclic timers are running.

The timer task determines if a particular task timer (the task has used a CMX function to wait for a specific time period) needs to be reduced, and if so, will decrement the task timer. When and if the task timer reaches zero, (specifically reduced by the timer task), the timer task will automatically take the suspended task, wait at least the specified time period, and put the task into the READY state. The timer task will also notify the task when the task becomes the RUNNING task again and that the time period had elapsed.

Another job the timer task performs is executing the cyclic timers when their time out value decrements to zero, if there are any started. When a cyclic timer is started, and the specified time period decrements down to zero, then the cyclic timer will automatically execute, calling the *K_Event_Signal* function with its events' parameters.

The CMX timer task will execute, at your specified scheduling frequency. If the current RUNNING task calls the CMX *K_Task_Lock* function, then the timer task will be delayed (blocked) until the RUNNING task calls the CMX *K_Task_Unlock* function, which releases the block and allows the timer task to execute.

## STACKS IN GENERAL

This chapter will discuss both task and interrupt stacks. Since all C compilers place arguments (parameters) on the stack, functions (tasks are functions) use the stack to allocate local data space and save the return address, it is very important for you to understand this chapter.

On most microprocessors and microcomputers the stack resides in external memory, though some processors have an internal stack. There are slight differences among processors. The way processors work with the stack are described later in the Processor Specific Information chapter.

All processors have a stack pointer register (some are dedicated, others use a specific register according to the C compiler manufacturer). Also some C manufacturers create a frame pointer to access the locals created within a function and also to access the parameters passed to a function. Again the frame pointer is a dedicated register, or a register chosen by the C compiler manufacturer.

CMX lets you configure the size of external memory to set aside for all task stacks and the size for the interrupt stack. Each task will have its own stack. This allows CMX to save a task's context very fast. When a task's contexts are saved, just the registers have to be pushed on to the stack. Then the stack pointer gets loaded with another memory address. (The memory address swapped in will be the interrupt stack or another task's stack.)

It is up to you to ensure each task stack and the interrupt stack are large enough in size. The task's stack must be large enough to hold that particular task's local data area, all return addresses that are pushed on to the stack as each function is called and all called functions’ local data areas.

Each of the task's stacks must be able to hold all of the CPU's REGISTERS in case of an interrupt. If the CPU has a "non-maskable interrupt", then the task's stack must be capable of holding the additional number of bytes. This is so the CPU's REGISTERS will be saved when the non-maskable interrupt occurs.

We will show how to calculate the stack size needed and provide two examples. It is recommended you double the estimated size of all task's stacks and the interrupt stack, at least initially. As you become more proficient and code testing indicates the stack size may be decreased, then the stack sizes may be reduced.

Unpredictable results can occur when either a task's stack or the interrupt stack pushes into memory that is not a part of this stack's memory.

A warning about the examples. All C compilers are not exactly the same in the way a function is called (some use a stack frame, some do not) CMX highly recommends you study the C compiler manual that describes stack usage, locals and parameter passing, and the code (assembly source) produced by the C compiler to better understand the particular C compiler stack usage.

These are examples only. CMX does not guarantee this is exactly the true number of bytes needed for the tasks and interrupt stack. The examples below assume ints are (2) two bytes long. On some processors ints will be (4) four bytes long.

Example 1: (a lot of locals, no nested function calls)

```
void Task1(void) /* tasks may not receive nor return parameters)
{
  /* note that some C compilers create a stack frame which will
  use some of this task stack space to save the stack frame
  pointer and possibly other registers */

  int a; /* 2 bytes of local space */
  char b; /* 1 byte of local space *
  char *ptr; /* 2 bytes of local, for processors that address up
  to 65535 bytes and no bank switching.  Also NO alignment
  boundaries. */
```

☞ *Some processors need integers, pointers, etc. to be aligned properly to the processor alignment boundary. In some cases this is at an EVEN memory address, other times it at an ODD memory address. If the local is aligned incorrectly, according to the processors alignment rule, then an additional byte will be used in the local space. Be aware of this.*

```
  func_A(char,int); /* func_A is a function that will receive a
  character and an integer.  3 bytes worth of stack space PLUS 2
  bytes for the return address */
  func_B(int,int); /* func_B is a function that will receive an
  integer and an integer.  4 bytes worth of stack space PLUS 2
  bytes for the return address */
}

void func_A(char,int)
{
  char a; /* 1 byte of local really used from task 1 stack area */
  int b; /* 2 bytes of local, really used from task 1 stack area
  */
  ..../* func_A code */
}

void func_B(int,int)
{
  int a; /* 2 bytes of local really used from task 1 stack area */
  int b; /* 2 bytes of local, really used from task 1 stack area
  */
```

```
  ..../* func_B code */
}
```

In the above example, the number of bytes used from task 1 stack area would be five bytes for task 1 locals, four bytes for func_B arguments, two bytes for func_B return address and four bytes for func_B locals. Make note that func_A is not calculated because func_B uses more memory than func_A.

This calculation does not include saving the stack frame pointer, if one is used, nor ALIGNMENT if the CPU requires it.

Example 2: ( a lot of locals, some nested function calls)

```
void Task1(void) /* tasks may not receive nor return parameters)
{
  /* note that some C compilers create a stack frame which will
  use some of this task stack space to save the stack frame
  pointer and possibly other registers */

  int a; /* 2 bytes of local space */
  char b; /* 1 byte of local space *
  char *ptr; /* 2 bytes of local, for processors that address up
  to 65535 bytes and no bank switching.  Also NO alignment
  boundaries. */
```

☞ *Some processors need integers, pointers, etc. to be aligned properly to the processor alignment boundary. In some cases this is at an EVEN memory address, other times it at an ODD memory address. If the local is aligned incorrectly, according to the processors alignment rule, then an additional byte will be used in the local space. Be aware of this.*

```
  func_A(char,int); /* func_A is a function that will receive a
  character and an integer.  3 bytes worth of stack space PLUS 2
  bytes for the return address */

  func_B(int,int); /* func_B is a function that will receive an
  integer and an integer.  4 bytes worth of stack space PLUS 2
  bytes for the return address */
}

void func_A(char,int)
{
  char a; /* 1 byte of local really used from task 1 stack area */
  int b; /* 2 bytes of local, really used from task 1 stack area
  */
  ..../* func_A code */

  func_C(int,int); /* nested function call to func_C.  Func_C is
  a function that will receive 2 integers.  4 bytes worth of task
  1 stack space PLUS 2 bytes for the return address */
}
```

```
void func_B(int,int)
{
  int a; /* 2 bytes of local really used from task 1 stack area */
  int b; /* 2 bytes of local, really used from task 1 stack area
  */
  ..../* func_B code */
  }
  void func_C(int,int)
  {
  int a; /* 2 bytes of local really used from task 1 stack area */
  int b; /* 2 bytes of local, really used from task 1 stack area
  */
  ..../* func_C code */
}
```

In the above example, the number of bytes used from task 1 stack area would be five bytes for task 1 locals, three bytes for func_A arguments, two bytes for func_A return address, three bytes for func_A locals, four bytes for func_C arguments, two bytes for func_C return address, four bytes for func_C locals. Make note that func_B is not calculated because func_A that calls func_C uses more memory than func_B.

This calculation does not include saving the stack frame pointer, if one is used, nor ALIGNMENT if the CPU requires it.

As you can see, the task 1 stack gets used for task 1 locals and all functions that are called by task 1. This includes the functions' locals and the return address for all the functions that task 1 will call.

CMX suggests you enter a pattern into all the tasks' stack area, like the value AA hex, and then run the application code. Then you may view each particular task's stack area and see how much of the task's stack has really been used. If the stack pushes down pass its allocated memory, then serious negative results will occur, with the system crashing and corrupting memory.

## THE RTOS CONFIGURATION FILE

The RTOS must be preconfigured for each application written using CMX. This allows CMX to allocate the necessary memory and know the maximum number of particular items. This is done for speed and also to reduce the amount of code the CMX function would need to obtain memory on the fly (while your program was executing). CMX recommends this especially on an embedded real-time system.

It is not hard to pre-configure CMX and the values you set may be changed. The following symbolic constants must be in place before starting a new application. CMX recommends you initially set the values above what you originally project. Then when your application code is close to being done you may reduce the values.

You will encounter the following symbolic constants in the CMX file CXCONFIG.H, which is the configuration file. Each symbolic constant is followed by an explanation of what this value will be used for.

*CMX_MAX_TASKS*
*CMX_MAX_RESOURCES*
*CMX_MAX_CYCLIC_TIMERS*
*CMX_MAX_MESSAGES*
*CMX_MAX_QUEUES*
*CMX_MAX_MAILBOXES*
*CMX_MAX_SEMAPHORES*
*CMX_TASK_STK_SIZE*
*CMX_RTC_SCALE*
*CMX_TSLICE_SCALE*
*CMX_RAM_INIT*
*CMX_INTERRUPT_SIZE*
*CMX_PIPE_SIZE*
*CMXBUG_ENABLE*
*CMXTRACKER_ENABLE*
*CMXTRACKER_SIZE*

**#DEFINE CMX_MAX_TASKS (USER_VALUE).**

The CMX_MAX_TASKS is the maximum number of tasks allowed for this application, since memory must be allocated and set aside for all the tasks. Each task has its own "task control block". The maximum that the value can be is 255.

**#DEFINE CMX_MAX_RESOURCES (USER_VALUE).**

The CMX_MAX_RESOURCES is the maximum number of resource groups allowed for this application. The maximum number of resource groups is 255, and this is tested by the CMX resource functions. If no resource groups are going to be used, then you should set this value to zero.

**#DEFINE CMX_MAX_CYCLIC_TIMERS (USER_VALUE).**

The CMX_MAX_CYCLIC_TIMERS is the maximum number of cyclic timers allowed for this application. The maximum number of cyclic timers is 255, and this is tested by the CMX cyclic timer functions.

**#DEFINE CMX_MAX_MESSAGES (USER_VALUE).**

The CMX_MAX_MESSAGES is the maximum number of messages used by any and/ or all mailboxes. You set this value only for the allocation of memory. The maximum value can be the number of tasks times 65535. This value should be zero if you do not need to have the message capabilities.

**#DEFINE CMX_MAX_QUEUES (USER_VALUE).**

The CMX_MAX_QUEUES is the maximum number of queues you would like for this particular application. The maximum number of queues allowed is 255. If you will not be using queues, then set this value to zero.

**#DEFINE CMX_MAX_MAILBOXES (USER_VALUE).**

The CMX_MAX_MAILBOXES is the maximum number of mailboxes allowed for this application. The maximum number of mailboxes is 255, and this is tested by the CMX mailbox/message functions. If no mailboxes are going to be used, set this value to zero.

**#DEFINE CMX_MAX_SEMAPHORES (USER VALUE).**

The CMX_MAX_SEMAPHORES is the maximum number of semaphores allowed for this application. The maximum number of semaphores is 255, and this is tested by the CMX semaphore functions. If no semaphores are going to be used, set this value to zero.

**#DEFINE CMX_TASK_STK_SIZE (USER_VALUE).**

The CMX_TASK_STK_SIZE is the number of bytes set aside for all the tasks' stacks. This does not include the CMX timer task. You must ensure the total of all the stack sizes for the tasks you create does not exceed this value. No testing is done to see if the memory set aside for the tasks' stacks has been exceeded or not. CMX highly recommends you initially double the estimated size of memory reserved. The maximum size is 65535 bytes, though this maximum will probably never be needed.

**#DEFINE CMX_RTC_SCALE (USER VALUE)**

The CMX_RTC_SCALE counts the number of times the CMX function *K_OS_Tick_Update* is called by an interrupt before the *K_OS_Tick_Update* function sets the do_timer_tsk flag, if there are task timers and/or cyclic timers that need servicing. The do_timer_tsk flag notifies the scheduler that it should now save the contexts of the current RUNNING task and let the CMX timer task execute. The only way the scheduler will be blocked from letting the CMX timer task execute is if the current RUNNING task had called the CMX function *K_Task_Lock*. This CMX function will block all task scheduling until it is unblocked with the CMX function *K_Task_Unlock*.

The interrupt that calls the CMX *K_OS_Tick_Update* function should be an interrupt you can program to occur consistently at the time interval for this interrupt. Most CPU's have an onboard timer that can be programmed to cause an interrupt at a specified time.

To get a multiple of this interrupt's time before the do_timer_tsk flag is set, determine what multiple you want. The CMX_RTC_SCALE is a value from one to 255. For example, say a particular CPU has the ability to program one of its timers to cause an interrupt every 10 milliseconds accurately. Now you want to base all time related activities at every 30 millisecond interval. The value set for the CMX_RTC_SCALE would be three. The higher the system tick frequency (you determine the scheduling frequency), the more the CMX scheduler and CMX timer task will be invoked, and the less time spent on tasks running.

As another example, say it takes on average 100 microseconds to perform a task switch. (NOTE: some processors take a lot less time, some more, this depends on the number of tasks in the application code.) Also you have specified a scheduling frequency of 5 milliseconds. Therefore the CPU's time will be the following: 2% on scheduling and 98% on letting tasks RUN. The CMX timer task will also take away from the tasks execution time. The amount of time depends on the number of tasks waiting on at least time (they may be waiting on another entity also, such as an event) and the number of cyclic timers that are currently running and whether or not they are to execute their respective *K_Event_Signal* function. Also, keep in mind that interrupts will also steal the CPU time from tasks, the scheduler, CMX timer task and possibly other interrupts (if nesting of interrupts is allowed).

**#DEFINE CMX_TSLICE_SCALE (USER_VALUE)**

The CMX_TSLICE_SCALE counts the number of times the CMX function *K_OS_Tick_Update* is called by an interrupt before the *K_OS_Tick_Update* function sets the do_time_slice flag, if the current RUNNING task is a time sliced task. See the Time Slice Chapter for more details on how time slicing works. The value may range from one to 255.

**#DEFINE CMX_RAM_INIT (USER VALUE: SHOULD BE 0 OR 1)**

The CMX_RAM_INIT value enables or disables the initalization of CMX RAM variables by the *K_OS_Init* function. Initalization of variables is usually done by the C compiler start up code. If it is not, CMX_RAM_INIT MUST be enabled (set to 1).

**#DEFINE CMX_INTERRUPT_SIZE (USER_VALUE)**

☞ *Applies to most processors, but not all.*

The CMX_INTERRUPT_SIZE is the size of the stack used for first and subsequent interrupts acknowledged by the CPU. This stack is also used by the CMX scheduler, the CMX timer task and the CMX "interrupt pipe" when its contents are being executed. Remember that the pipe executes the CMX functions as though a task was calling them, and therefore would need enough stack space for the deepest CMX function the pipe may call. This is in addition to the nested interrupts stack use. The way to calculate this value is the maximum number of nested interrupts that could occur multiplied by the number and size of the registers for the particular processor. Also add the number

of bytes that each interrupt could push on the stack plus the number of bytes that a particular CMX function (largest stack use function) called by the "interrupt pipe" could use. This is a tough calculation and CMX highly recommends you double the calculated size initially. CMX feels a minimum number of 128 bytes is a good starting point.

The following #defines set up the CMX "interrupt pipe" size and also declares what CMX functions can be used indirectly by any one of the interrupts. This allows the CMX code to possibly be reduced in size if all the CMX functions can be called by interrupts indirectly are not used. Note that if the particular CMX function is not to be used by an interrupt, a task is still capable of calling that CMX function, without regard to the setting of the following defines.

**#DEFINE CMX_PIPE_SIZE (USER_VALUE)**

The CMX_PIPE_SIZE value defines the size of the CMX "interrupt pipe". This value MUST be between 1 and 255. The defined value identifies how many functions will be able to reside in this 'queue' at any one time. Remember this is really a circular buffer and ranges must be tested all the time to ensure the pointer in and pointer out don't exceed their maximum value. This can be accomplished very fast by adding the pointer value with a selected value less one.

If you enter a value other than the valid values shown here, the size will DEFAULT to 255. Also, if the pipe becomes full no error will be returned to the interrupt notifying that the particular CMX function was not put into the pipe. If the pipe fills frequently, either the pipe size should be increased to a larger value or the interrupt using a CMX function excessively should have its code modified to reduce the frequency of calls to the pipe.

**#DEFINE CMXBUG_ENABLE (USER VALUE: SHOULD BE 0 OR 1)**

The CMXBUG_ENABLE value enables or disables the initalization and setup of certain functions to work with CMXBUG.C module. The K_OS_Init.C module contains *setup_bug*, *K_Bug_Getchr* and *K_Bug_Putchr* functions which work with the CMXBUG.C module, if enabled. These three functions get compiled if this switch is set to a one. The *setup_bug* function will enable the UART, selecting 9600 BAUD, no parity. The *K_Bug_Getchr* and *K_Bug_Putchr* functions will be tailored to transmit characters and receive characters for the particular processor the user is using. The user may have to tailor these functions, especially the *setup_bug* function, depending upon the crystal selected to run the processor.

**#DEFINE CMXTRACKER_ENABLE (USER VALUE: SHOULD BE 0 OR 1)**

The CMXTRACKER_ENABLE value enables or disables the initalization and setup of certain functions to work with CMXTRACK.C module. The K_OS_Init.C module contains *setup_bug, K_Bug_Getchr* and *K_Bug_Putchr* functions which work with the CMXTRACK.C module, if enabled. These three functions get compiled if this switch is set to a one. The *setup_bug* function will enable the UART, selecting 9600 BAUD, no parity. The *K_Bug_Getchr* and *K_Bug_Putchr* functions will be tailored to transmit characters and receive characters for the particular processor the user is using. The user may have to tailor these functions, especially the *setup_bug* function, depending upon the crystal selected to run the processor.

**#DEFINE CMXTRACKER_SIZE (USER_VALUE)**

The CMXTRACKER_SIZE value defines the size of the CMX array for capturing the chronological flow of the RTOS. This value can range from 0 to 65535. The larger the size selected, the more of the chronological flow data can be captured.

## TIME SLICE CHAPTER

CMX allows time slicing if you feel it is necessary. CMX believes most applications will not need time slicing. However it is provided for those times it is needed and desirable.

To enable time slicing, call the *K_OS_Slice_On* function. This turns on time slicing. When you call this function the CMX_TSLICE_SCALE value will be used to determine a task's time slice count. All tasks will use this value.

When time slicing is enabled, the current running task's priority determines which tasks are time sliced for the moment. All tasks, starting with the next task with the same or lower priority than the current running task, will be time sliced. If a higher priority task preempts, then the priority of this task will be used to determine the starting priority of the tasks that are time sliced.

When time slicing is enabled, a task will automatically RUN for the number of TIME SLICE ticks. At the end of its time slice, a new time sliced task will be enabled if possible. A time sliced task has to be READY or ready to RESUME to be considered when the scheduler determines the next slice task to run.

All capable time sliced tasks will run in a “round robin” type of execution. Each sliced task will run for the number of slice ticks chosen. If a time slice task is blocked because it is waiting on some entity, that task will not be considered to run when the CMX scheduler decides what slice task to run next.

If a time slice task calls a CMX function that puts the task into the SUSPENDED state, then this task is assumed to have its slice time expired. This task will not resume until it comes out of the SUSPENDED state and is the next time sliced task in line to receive control.

Also note that if a task calls the *K_Task_Coop_Sched* function which does a cooperative scheduling, then the time slice period for the current running task will be lost. Also, the next task in the linked list of tasks capable of running, regardless of its priority, will become the new RUNNING task.

You may at any time have a task disable time slicing with the *K_OS_Slice_Off* function. Remember, only a task can disable time slicing and the task must be running to be able to call this function.

You must not assume that all tasks with the same priority or lower as the current RUNNING task will run. This is because higher priority tasks may constantly preempt the lower priority tasks before the SLICE TICK count expires. This would let the next task with the same priority execute.

## INTERRUPTS IN GENERAL

This chapter deals with CPU's interrupts and using the CMX "interrupt pipe". The Processor Specific Information chapter explains specifically the interrupts and how to use them for the particular processor you are using.

CMX allows nesting of interrupts. When an interrupt is acknowledged by the CPU hardware many things happen. This is a general description of what takes place. Different CPUs may do things slightly different.

First, the address being interrupted is pushed on the stack. Then some processors push either all or some of the CPU's registers on the stack, and some processors automatically clear the interrupt enable flag, blocking any more interrupts from being immediately acknowledged. This clearing does not block interrupts called "non-maskable interrupt".

Next the processor either jumps to the interrupt’s vector address, or retrieves the address in the code from this interrupt vector's address. The hardware loads the program counter with the address where you service this particular interrupt’s code.

CMX requires all interrupts that use any CMX functions indirectly (by using the "interrupt pipe"), or call the *K_OS_Tick_Update* function (which determines the scheduling frequency), to call the CMX interrupt service routines *K_OS_Intrp_Entry* and *K_OS_Intrp_Exit*. If the interrupt does not use the CMX interrupt service routines, you must save and restore all registers the interrupt's code will destroy. You must restore the CPU context in the state it was prior to the interrupt code executing. The interrupt cannot use any CMX calls, nor will any rescheduling be done by any action this interrupt may have performed. In addition, the interrupt code must not allow other interrupts to be acknowledged that would use the CMX interrupt service routines.

☞ *To avoid this, the interrupt routine can increase and decrease a CMX variable, understanding the "pipe" will not immediately execute its contents if a higher priority interrupt preempted, then use a CMX function indirectly. This varies from processor to processor and is described in the Processor Specific Information chapter in more detail.*

CMX lets interrupts call a limited number of CMX functions indirectly. If a task is in the middle of a CMX function where there is a "critical region of code" executing that must not be interrupted, then an interrupt can and will still be acknowledged.

This allows interrupts to happen without any latency or temporarily disabling the system for an undesirable amount of time. Also, this will reduce the amount of time needed for an interrupt to call a CMX function, allowing the interrupt to finish its code sooner.

## HOW INTERRUPTS INTERFACE WITH CMX FUNCTIONS

CMX creates an "interrupt pipe". This pipe is really a circular buffer, which contains the CMX function identifier and the particular CMX function parameters. You determine the size of this pipe within the CXCONFIG.H file.

The following will name the CMX interrupt functions and their equivalent CMX functions. The parameters that a function requires is EXACTLY the same as the non interrupt CMX functions. Each of the callable CMX interrupt functions just has a ‘K_Intrp_’ instead of the K_’ of the equivalent CMX functions.

Please review the table below for the equivalent CMX function and what parameters are needed.

| **CMX Interrupt Function** | **Equivalent non Interrupt Function** |
|---|---|
| *K_Intrp_Task_Wake* | *K_Task_Wake* |
| *K_Intrp_Task_Wake_Force* | *K_Task_Wake_Force* |
| *K_Intrp_Task_Start* | *K_Task_Start* |
| *K_Intrp_Timer_Stop* | *K_Timer_Stop* |

| | |
|---|---|
| *K_Intrp_Timer_Restart* | *K_Timer_Restart* |
| *K_Intrp_Timer_Initial* | *K_Timer_Initial* |
| *K_Intrp_Timer_Cyclic* | *K_Timer_Cyclic* |
| *K_Intrp_Mesg_Send* | *K_Mesg_Send* |
| *K_Intrp_Event_Signal* | *K_Event_Signal* |
| *K_Intrp_Semaphore_Post* | *K_Semaphore_Post* |

This pipe is also re-entrant so a higher priority interrupt can interrupt a lower interrupt, with the knowledge that the lower priority interrupt will resume where it left off once the higher priority interrupt finishes.

When the interrupt has finished its code (in most cases, it must have called the *K_OS_Intrp_Entry* routine) and calls the *K_OS_Intrp_Exit* routine the following will happen. First, the CMX interrupt handler determines whether another interrupt (lower priority interrupt) is in progress, and if so, returns to let that interrupt finish. If no more interrupts are waiting to finish (meaning that this was the first interrupt acknowledged) then the CMX interrupt handler will determine if the pipe was actually used. If the interrupt did not call the pipe placing a CMX function into it, then the task executing prior to the interrupt will resume running where it left off. If the interrupt did use the pipe, then the CMX handler will determine whether the pipe is already being processed, and if so, will continue to process the pipe or a task already within a CMX function.

If the task was within a CMX function, it will be allowed to finish. When the CMX function has finished, then the task will be suspended while the pipe executes its contents calling the CMX functions with their associated parameters that the interrupts wanted. If the task was not within a CMX function or the pipe was not currently being executed, then the interrupt handler will automatically start the execution of the pipe contents. Once the pipe has finished, either the suspended task will resume execution or a higher priority task will become the running task, if the CMX function had enabled this higher priority task to be in the running state.

Because of the features and structure of the CMX interrupt pipe, CMX needs only to disable the global interrupt enable flag in three to five places (depends on processor) in code and for a very small amount of time. This time depends on the processor in use and the crystal speed.

The CMX Processor Specific chapter explains in more detail how the interrupt(s) will use the CMX *K_OS_Intrp_Entry* and *K_OS_Intrp_Exit* routines explicitly.

It is highly recommended that the interrupts use the two CMX interrupt services calls. The *K_OS_Intrp_Entry* interrupt service routine is called by the interrupt when it first enters its code. The *K_OS_Intrp_Entry* routine saves the task contexts, including the particular CPU's registers, and loads in the interrupt stack. It will then return to the interrupt code.

Depending upon the particular CPU being used, this will add an additional number of microseconds of time. Once the *K_OS_Intrp_Entry* returns to the interrupt code, you are free to indirectly use those CMX calls that are allowed by interrupts. Also the disable interrupt flag is put into the proper state, not allowing any interrupts from being acknowledged except the interrupts that cannot be masked by this flag. You can re-enable the interrupt flag if you choose.

If a rescheduling (task switch) is to occur, then CMX will determine which task should run next, according to the priorities of the task, and whether the task can run or not. If a rescheduling does not occur, then the task suspended because of the interrupt will have its contexts restored and the task will then continue on as if the interrupt never occurred.

## THE CMX SCHEDULER CHAPTER

This chapter will explain in detail how the CMX scheduler works. The CMX scheduler is based on true preemption. This means that tasks and interrupts can cause an immediate task switch if a higher priority task becomes able to run as a result of a CMX function. Cooperative scheduling is also possible, so a task can let the next task (lower or same priority) run if so desired. True time slicing is also incorporated. The time slice mechanism still allows you to run higher priority tasks with the time sliced task regaining control (after higher priority task done) until its allocated slice time expires. This gives you complete control on how tasks will execute.

In CMX, a task will be in one of the following states:

### IDLE state

A task that has been created with the *K_Task_Create* function, but not started by the *K_Task_Start* function is in the IDLE state. A task that has completed its code and called the *K_Task_End* function is placed into the IDLE state if there are no outstanding triggers in its control block. A task that is in its IDLE state will not run.

### READY state

The READY state informs the scheduler that the task is ready to run, but NOT running. This allows the scheduler to determine what task to run when a scheduling takes place according to the task’s priority in relation to the other tasks’ priorities.

**RUN state**

The task that is executing is in the RUN state and owns the CPU time. Only one task may be in the RUN state at any one time.

**WAIT (suspended) state**

A task that suspended itself by a CMX function call is in the WAIT (suspended) state. There are many function calls that will suspend a task. The WAIT state consists of a task that is waiting on one or more of the following: time, events, flags, messages, on a reply, etc.

**RESUME state**

The RESUME state is treated the same as the READY state. The only difference is that it informs the scheduler that the task had been started, yet not finished, with its code. This means that a higher priority task has preempted and forced the original task that was running to become ready to RESUME, or that the task had suspended itself by a function call and now has removed itself out of the WAIT (suspended) state into the ready to RESUME state.

## CMX OPERATING FLAGS

The CMX operating system has specific flags set and cleared by the CMX functions and interrupt functions, notifying the scheduler what action to perform. The following describes these flags and how they become set and how the scheduler acts on them.

The *K_OS_Tick_Update* function, which is called by an interrupt, determines if any task timers are in use or what cyclic timers need servicing. This sets the CMX do_timer_tsk flag indicating that the scheduler should execute the CMX timer task. Also, if the running task is a time sliced task and its time slice count has expired, then the do_time_slice flag will be set. This indicates that the time slice task should relinquish control to the next time slice task that is READY to run.

Most of the CMX functions also set CMX flags. When a task calls a CMX function and that task becomes SUSPENDED, then the CMX preempted flag becomes set. If a task calls a CMX function that will possibly wake up a SUSPENDED task, then the CMX function will test to see if the formerly SUSPENDED task, which is now in the ready to RESUME state, has a higher priority than the current RUNNING task that called this function. If so, then the CMX preempted flag will be set.

The CMX do_coop_sched flag will be set by the CMX *K_Task_Coop_Sched* function. This indicates that the current RUNNING task would like the next task in the ready list to execute. This indicates the task would like to do a cooperative rescheduling.

The last flag CMX keeps track of is the do_int_pipe flag which indicates an interrupt has called the interrupt pipe routine wanting a CMX function to be performed. Interrupts can not call the CMX function directly because the CMX function may be within a "critical region of code", which would be disastrous. The CMX interrupt pipe routine automatically sets the do_int_pipe flag, indicating the interrupt pipe contents need execution.

There are also two counters CMX uses on most processors. The first counter is called the locked_out counter. This counter is increased and decreased respectively when a function enters and exits a "critical region of code". This is used instead of disabling and enabling the CPU GLOBAL interrupt enable flag or interrupt priority mask. This allows the CMX *return* function, which most functions call at the end of their code, and the CMX interrupt handler to determine if the scheduler may perform a task switch or if it is blocked because the current running task is within a "critical region of code". No task switch should occur at this moment. However, if the current running task is within the CMX function's "critical region of code", when the CMX function finishes the "critical region of code" a task switch will possibly occur.

The second counter is the CMX interrupt count. This is used by CMX to determine the nested level of interrupts. The CMX operating system will not perform a task switch when there are additional lower interrupts that have not finished their code because a higher priority interrupt occurred.

For the CMX scheduler to be called, either by the CMX interrupt handler or the *return* function, the following must occur. The CMX locked_out count must be zero, indicating no "critical region of code" exists. The CMX interrupt count must also be zero, indicating there are no more interrupts needing to finish their code because of nested interrupts. At least one of the five previous flags must be set: do_coop_sched, do_timer_tsk, preempted, do_time_slice and do_int_pipe.

This is so the CPU can have all the interrupts GLOBALLY enabled while within the CMX function’s "critical region of code". In a real-time environment, interrupts should be disabled for the shortest time possible without adding large amounts of interrupt latency times. For example, when an UART receives and transmits characters at 115K baud or greater, the CPU can receive an interrupt approximately every 87 microseconds (usecs). If the interrupts were GLOBALLY turned off, the UART could possibly receive an overrun error. Many other interrupts can occur in bursts of 20-50 microseconds (usecs).

Also, consider the interrupt process time. Because the interrupt does not call the CMX function directly, knowing the time it takes to complete the function is immaterial. Each function can take different amounts of time. You would have to analyze what functions the interrupt would call to truly know the time it takes to process the interrupt and its associated code.

You would always have to assume the worse case latency time that interrupts are GLOBALLY disabled. This latency time would be a variable depending on the CMX function's "critical region of code", the efficiency of the CPU instruction set, the speed of the CPU's crystal, etc. Since interrupts use the interrupt pipe to pass the function and its associated parameters and very low latency time is introduced by CMX, you can easily calculate the interrupt's time, which makes it very deterministic in nature.

When the scheduler is invoked, it determines what action to take based on the setting of the flags. The scheduler first saves the complete state of the current RUNNING task's context. This is so the task may RESUME running when it is the task's turn to execute again. The scheduler will then perform operations according to the flags that are set. The scheduler will do the CMX timer task if that flag is set.

Next, the scheduler will process the interrupt pipe if the pipe is not empty and needs processing. The scheduler will then determine whether to change the task slice pointer, indicating when the next time sliced task should run. If the preempted flag is set, then the scheduler will determine which of the tasks is the highest priority task able to run, letting that task become the RUNNING task. If the do_coop_sched flag is set, the next task able to run will become the RUNNING task.

The scheduler will determine whether the next running task is going to RESUME its code or begin at the task's opening brace. If the task is going to RESUME, then the scheduler will restore the task's context, manipulating the CPU registers, stack pointer, etc. This is so the task will execute exactly where it left off restoring the CPU registers in the state they were as if the task never suspended.

The scheduler will also determine if all tasks are either SUSPENDED or IDLE. If so, then no task is able to RUN at the current moment. The scheduler will then call the CMX POWER DOWN function, letting the processor enter the reduced power state if the processor supports this and you have enabled this capability. Ensure that the power down mode is one in which the interrupt used for the CMX system tick will allow the processor to resume and exit its power down state. This is because tasks waiting on a time out and cyclic timers that are running will need to decrement their respective time counts at the proper system tick frequency.

Carefully study the enclosed source code for the scheduler, which is written in assembly, for the specific processor that you are working with. Also, study the C source code module supplied with all the CMX functions within. (Note: this is for convenience only, as all CMX functions are individual modules, so they may be placed into a library.)

## QUICK REFERENCE

The following pages are a quick reference to all the CMX functions. The functions are listed within their respective manager in alphabetical order so you can easily look up the desired function for reference. These are intended to assist the user in syntax and use of the function.

[Before entering RTOS] means the call may be used prior to entering the CMX operating system. [Tasks] mean a task may use this CMX call. [Interrupts] mean interrupts may use this CMX call INDIRECTLY.

☞ *The K_OS_Init function must be called before any CMX function call may be used.*

Each CMX call also lists what CMX parameters or arguments are needed and passed, and a description indicating what should be loaded into these parameters or arguments. The CMX call will show the return STATUS parameter values most CMX functions return. If STATUS is returned, it is of unsigned character size.

An example is shown using the parameters or arguments needed for this CMX call, with the CMX call. For a detailed description of each CMX function, please refer to the appropriate function manager chapter at the beginning of this document.

## USING CMX WITH C COMPILERS

CMX is designed to work with most C compilers from different manufacturers. As you can see from the supplied source code, CMX does try to always follow true ANSI C specifications. There may be exceptions, to keep the code size as small as possible. Most C compilers will not generate an error, but will sometimes produce a warning message.

The C vendor compiler may use keywords, #pragma's, etc., for processor specific entities. One C vendor’s header files may not be compatible with another C vendor's tools.

If an error does occur, it is most likely one of the following: because the C compiler has a switch to detect incompatibilities from the ANSI standard. Most C compilers allow you to toggle, or invoke a switch when compiling to allow less strict use of C code.

There is a possibility the C compiler does not support the data types used by CMX, such as the void pointer or a function pointer. There is nothing CMX can do if the C compiler does not support these types of pointers.

You will notice CMX uses void pointers very often. In many of the CMX calls, especially the ones that move a block of data, such as message passing and receiving or queue adding and removing, these pointers will be used.

## CMX RETURN STATUS BYTE VALUES

| Symbol | Hex | Value | Explanation |
|---|---|---|---|
| K_OK | 00 | Good | CMX call was successful |
| K_TIMEOUT | 01 | Warning or Error | Time out occurred |
| K_NOT_WAITING | 02 | Error | Task not waiting for wake request |
| K_RESOURCE_OWNED | 05 | Error | Resource is already "owned" |
| K_RESOURCE_NOT_OWNED | 06 | Error | Resource not owned by calling task |
| K_QUE_FULL | 0A | Warning | Queue now full, slot was added |
| K_QUE_EMPTY | 0B | Warning | Queue now empty, slot was removed |
| K_SEMAPHORE_NONE | 0C | Error | Semaphore is not available |
| K_ERROR | FF | Error | General error, CMX call unsuccessful |

Some functions, such as *K_Mesg_Get*, may return a NULL pointer if there is no message available. This indicates a possible warning/error to the caller, for NO message was retrieved. In some cases a return value of zero indicates the CMX function is telling the caller the item it wanted was not there or a time out occurred.

## EVENT MANAGER FUNCTION

*K_Event_Reset*

Purpose: This function clears (resets) one or more specific events within a task. This does not change a task state, nor does it specify which events were set or not. In most cases, this function is not needed. The *K_Event_Wait* function automatically clears the events of a task, if so programmed.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Event_Reset*(byte tskid,word16 event); /* this is the function prototype */

unsigned char TASK_SLOT; /* should be global, slot number of task */

unsigned short EVENTS_TO_CLEAR; /* the event bits to clear, or could be a #define

#define EVENTS_TO_CLEAR ??? instead of variable. */

unsigned char STATUS; /* should be local */

STATUS = *K_Event_Reset*(TASK_SLOT,EVENTS_TO_CLEAR);

**Passed**

TASK_SLOT is the slot number of the task for which the event bit(s) will be cleared.

EVENTS_TO_CLEAR is a 16 bit wide variable or constant indicating the desired event bits to clear within this task.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task does not exist.

If STATUS equals K_OK, then the event bits were cleared according to the EVENTS_TO_CLEAR parameter passed.

☞ *This function clears just the event bits within the task that is being referenced. It does not change any of the other task event bits.*

*K_Event_Reset* example:

#include <cxfuncs.h> /* has function prototype */
byte *K_Event_Reset*(byte tskid,word16 event); /* this is the function prototype */

unsigned char task1_slot; /* should be global, slot number of task 1. */

#define EVENT1 0x0001 /* event bit location 0 */

#define EVENT16 0x8000 /* event bit location 15 */

```
void task1(void)
{
  unsigned char status; /* local */
  ...
  status = K_Event_Reset(task1_slot,EVENT1 | EVENT16);

  /* task 1 will clear task 1's event bits 0 and 15.  Does not
  notify whether the bits were set or reset. */

  if (status != K_OK)
    {
    /* maybe take corrective action. */
    }
  ...
}
```

## EVENT MANAGER FUNCTION

*K_Event_Signal*

Purpose: The *K_Event_Signal* function sets a specific event. This can be done in a variety of modes. This function may be called by tasks, cyclic timers, mailboxes or interrupts. The caller will select which event, the mode of event set it wants, and the task slot number or priority, depending upon the mode selected.

**Called**

Tasks, interrupts, cyclic timers and mailboxes

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Event_Signal(byte,byte,word16); /* this is the function prototype */

#define MODE ???

unsigned char TASK_PRI;

#define EVENT_TO_SET ???

unsigned char STATUS; /* should be local */

STATUS = K_Event_Signal(MODE,TASK_PRI,EVENT_TO_SET);
```

**Passed**

MODE is the mode in which this function will determine which tasks to work with. The values are as below.

TASK_PRI is either the task slot number or the priority in which this function will work with according to the MODE selected.

☞ *This parameter is not always used depending on the MODE.*

EVENT_TO_SET is the events to set.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task does not exist if MODE = 0, or the MODE is out of range.

If STATUS equals K_OK, then the function performed as the MODE indicated it should.

Remember that cyclic timers and mailboxes can be coded to automatically call this function. Interrupts may also call this function.

*K_Event_Signal* example:

unsigned char task2_slot; /* should be global. */

#define EVENT1 0x0001 /* define event bit 0 */

#define EVENT16 0x8000 /* define event bit 15 */

```
void task1(void)
{
  unsigned char status; /* local */
  ...
  status = K_Event_Signal(0,task2_slot,EVENT16);

  /* task 1 will tell the K_Event_Signal function to set event
  bit 15 of only task 2, whether task 2 is waiting or not for
  this event.  If task 2 is waiting, then task 2 will
  automatically resume. */

  if (status != K_OK)
  {
  /* take corrective action. */
  }
  status = K_Event_Signal(3,0,EVENT1);

  /* task 1 will tell the K_Event_Signal function to set event
  bit 0 of all tasks that have been created, whether the task is
  waiting or not for this event.  If the task is waiting, then
  the task will automatically resume.  Notice how TASK_PRI
  parameter not used, may contain any value. */

  if (status != K_OK)
    {
    /* take corrective action. */
    }
  ...
}
```

Comments

This function is very flexible and powerful. As you can see, the caller has the ability set an event within one or more tasks depending upon the MODE selected. Carefully read the chapter on Event Management for a thorough understanding of how events work.

## EVENT MANAGER FUNCTION

*K_Event_Wait*

Purpose: This function allows a task to wait for specific events with a specified time out if desired. The task will also specify the mode that indicates when to automatically clear the event bits that match. The task will be suspended until either the number of system ticks has expired or at least one event bit it is waiting for is set or becomes set. A time period of zero indicates the task will wait indefinitely for an event match. If there is a match as specified when the task calls this function, the task will not be suspended and will immediately be returned to.

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
word16 K_Event_Wait(word16,word16,byte); /* this is the function prototype */

#define MATCH ???

#define TIME_CNT ???

#define MODE ???

unsigned short EVENTS; /* should be local */

EVENTS = K_Event_Wait(MATCH,TIME_CNT,MODE);
```

**Passed**

MATCH is a 16 bit wide parameter indicating the specific events that this task would like to have set. More than 1 event bit may be specified.

TIME_CNT is the number of system ticks to wait for a match. If the value is 0 then the task will wait indefinitely for event match.

MODE is the mode in which this function will clear event bits, when they are set or become set. The values are below.

0 = do not clear the event bits.

1 = clear the event bit(s) according to the ones set within the MATCH parameter at BEGINNING of function.

2 = clear the event bit(s) according to the ones set within the MATCH parameter at END of function.

3 = do both modes 1 and 2.

**Returned**

EVENTS will either contain a zero indicating the time period specified expired before any of the events the task is waiting on became set, or the specific events were set according to the MATCH parameter.

☞ *Remember only events that are selected by the MATCH parameter are worked within this function. The MODE parameter allows powerful synchronization as to when the task's events are cleared.*

*K_Event_Wait* example:

```
void task1(void)
{
  unsigned short events; /* local */
  ...
  events = K_Event_Wait(0x0005,0x0010,2);

  /* task 1 will wait for only its event bits 0 or 2 to be set
  or to become set within 16 system ticks.  It is also telling
  the K_Event_Wait function to clear the task's event bits 0 and
  / or 2 if they are or become set within the time period of 16
  ticks.  Events variable will contain the states of only tasks
  event bits 0 and 2. */

  if (events)
    {
    if (events & 0x0001)
      {
      /* process code for task event bit 0 being set. */
      }
    if (events & 0x0004)
      {
      /* process code for task event bit 2 being set. */
      }
    }
  else
    {
    /* maybe take corrective action, for neither event bit was
    already set, nor became set. */
    }
  ...
}
```

☞ *There are other means to wait for an event. Please read the Event Manager Functions chapter that explains the differences of these functions.*

## UART FUNCTIONS

*K_Get_Char*

Purpose: This allows a task to retrieve a character from the UART receive buffer, if one is available. If there is at least one character, then that character will be placed at the address specified by the task and returned to immediately with the return count set to one. If there is no character, then the task will again be returned to immediately, with the return count set to zero indicating no character was transferred. Remember, CMX assumes only one task may have ownership to the UART receive buffer. If more than one task has access at the same time, a resource should be set up.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
word16 *K_Get_Char*(void *); /* this is the function prototype */

unsigned char *DEST_PTR; /* this may be local or global, and will be the location that the function puts the character. */

unsigned short COUNT; /* this will specify the number of characters retrieved from buffer. */

COUNT = *K_Get_Char*(DEST_PTR);

**Passed**

DEST_PTR = the address where this function will place the character.

**Returned**

COUNT = the number of characters that were transferred from the UART receive buffer to the destination address.

If the COUNT value is zero, there were no characters in the UART receive buffer when this function was called. If the COUNT is non-zero, (which for this function would be 1), then one character was transferred to the destination address passed to this function. Make note that the task would just have to pass the address of a character type variable, instead of loading a pointer with the address and then passing the pointer (which really passes the address that the pointer has).

*K_Get_Char* example:

unsigned char recv_array[80];

/* this will receive up to 80 characters, could be local if user wants or could just be a depth of 1 in this case for this function only returns one character at the most. */

```
void task1(void)
{
  unsigned short count; /* local */
  unsigned char c;
  unsigned char *recv_ptr; /* could be local or global. */

  recv_ptr = recv_array; /* load pointer with address of
  recv_array. */
  count = K_Get_Char(recv_ptr); /* go get character. */
  if (count)
    {
    c = *recv_ptr++; /* get character and increment pointer. */
    ...
    }
  ...
  or/* this shows another way. */
  count = K_Get_Char(&c); /* go get character. */
  if (count)
    {
    .../* process c here, etc. */
    }
  ...
}
```

## UART FUNCTIONS

*K_Get_Char_Wait*

Purpose: This allows a task to retrieve a character from the UART receive buffer, if one is available. If there is at least one character, then that character will be placed at the address specified by the task and returned to immediately with the return count set to one. If there is no character, then the task will wait for the desired time period specified. This time period may range from zero to 65535, with zero indicating an indefinite wait. When either a character is received or the specified time period specified, then the task will be returned to, identifying whether a character was retrieved or not by the count being non-zero or zero respectively. CMX assumes only one task may have ownership to the UART receive buffer. If more than one task has access at the same time a resource should be set up.

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
word16 K_Get_Char_Wait(void *,word16); /* this is the function prototype */

unsigned char *DEST_PTR; /* this may be local or global, and will be the location that
the function puts the character. */

#define TIME_PERIOD ??? /* The time wait period. */

unsigned short COUNT; /* this will specify the number of characters retrieved from
buffer. */

COUNT = K_Get_Char_Wait(DEST_PTR,TIME_PERIOD);
```

**Passed**

DEST_PTR = the address where this function will place the character.

TIME_PERIOD is the number of system ticks that this task will wait for a character to be received. May range from zero to 65535, with zero indicating an indefinite wait.

**Returned**

COUNT = the number of characters that were transferred from the UART receiver buffer to the destination address.

If the COUNT value is zero, there were no characters in the UART receiver buffer when this function was called, or the time period expired. If the COUNT is non-zero, (which for this function would be one), then one character was transferred to the destination address passed to this function.

*K_Get_Char_Wait* example:

unsigned char recv_array[80];

/* this will receive up to 80 characters, could be local if you want or could just be a depth of one in this case since this function only returns one character at the most. */

```
void task1(void)
{
  unsigned short count; /* local */
  unsigned char c;
  unsigned char *recv_ptr; /* could be local or global. */

  recv_ptr = recv_array; /* load pointer with address of
  recv_array. */
  count = K_Get_Char_Wait(recv_ptr,20);
  /* go get character, if there is no character, wait for 20
  system ticks for one to arrive. */
  if (count)
    {
    c = *recv_ptr++; /* get character and increment pointer. */
    ...
    }
  ...
  or/* this shows another way. */
  count = K_Get_Char_Wait(&c,20); /* go get character. */
  if (count)
    {
    .../* process c here, etc. */
    }
  ...
}
```

## UART FUNCTIONS

*K_Get_Str*

Purpose: This allows a task to retrieve a specific number of characters from the UART receive buffer if that specified number of characters are available. If there are at least the specified number of characters they will be placed at the address specified by the task and returned to immediately with the return count set to the number requested. If there is less than the requested number of characters, then the task will again be returned to immediately, with the return count set to zero, indicating no characters were transferred. CMX assumes that only one task has ownership to the UART receive buffer. If more than one task has access at the same time then a resource should be set up.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
word16 *K_Get_Str*(void *,word16); /* this is the function prototype */

unsigned char *DEST_PTR; /* this may be local or global, and will be the location that the function puts the character. */

#define NUMBER ??? /* number of characters to get. */

unsigned short COUNT; /* this will specify the number of characters retrieved from buffer. */

COUNT = *K_Get_Str*(DEST_PTR,NUMBER);

**Passed**

DEST_PTR = the address where this function will place the character.

NUMBER is the number of characters to retrieve from the UART receive buffer. Only the specified number will be transferred.

**Returned**

COUNT = the number of characters that were transferred from the UART receiver buffer to the destination address.

If the COUNT value is zero, the required NUMBER of characters were not in the UART receive buffer when this function was called. If the COUNT is non-zero, then the value will represent the number of characters that were transferred to the destination address passed to this function. The COUNT value should match the NUMBER passed in this case.

*K_Get_Str* example:

unsigned char recv_array[80];

/* this will receive up to 80 characters, could be local if you want or could just be a depth of the maximum number that this task would ask for. */

```
void task1(void)
{
  unsigned short count; /* local */
  unsigned char c;
  unsigned char *recv_ptr; /* could be local or global. */

  recv_ptr = recv_array; /* load pointer with address of
  recv_array. */
  count = K_Get_Str(recv_ptr,5); /* go get 5 characters. */
  while (count--)
    {
    c = *recv_ptr++; /* get character and increment pointer. */
    ...
    }
  ...
}
```

Comments

This allows a task to retrieve a specific number of characters from the UART receive buffer if that number is available. The task will not wait for that number of characters to come in.

## UART FUNCTIONS

*K_Get_Str_Return*

Purpose: This function allows the task to immediately receive whatever number of characters are presently in the UART receive buffer. The task will not wait on any time period. The count returned to the task will indicate the number of characters retrieved, if any. It is up to the task to ensure that the count will not exceed the depth of the destination where the characters will be placed. You can use the *K_Recv_Count* function to get an idea how many characters are present in the receive buffer, though this number could change before this function was executed.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
word16 *K_Get_Str_Return*(void *); /* this is the function prototype */

unsigned char *DEST_PTR; /* this may be local or global, and will be the location that the function puts the character. */

unsigned short COUNT; /* this will specify the number of characters retrieved from buffer. */

COUNT = *K_Get_Str_Return*(DEST_PTR);

**Passed**

DEST_PTR = the address where this function will place the characters.

**Returned**

COUNT = the number of characters that were transferred from the UART receive buffer to the destination address.

If the COUNT value is zero, there were no characters in the UART receive buffer when this function was called. If the COUNT is non-zero, then the number of characters present in the receive buffer were transferred to the destination address passed to this function.

*K_Get_Str_Return* example:

unsigned char recv_array[80];

/* the size of the destination the characters will be transferred to, should be as large as the receive buffer. This way the function will not transfer characters into the wrong memory locations. */

```
void task1(void)
{
  unsigned short count; /* local */
  unsigned char c;
  unsigned char *recv_ptr; /* could be local or global. */
  recv_ptr = recv_array; /* load pointer with address of
  recv_array. */
  count = K_Get_Str_Return(recv_ptr);
  /* go get what ever number of characters that are present in
  the UART receive buffer at the time of this call. */
  while (count--)
    {
    c = *recv_ptr++; /* get character and increment pointer. */
    ...
    }
  ...
}
```

☞ *This is very useful when a task is either IDLE or waiting indefinitely. The receiver interrupt could call the K_Task_Start or K_Task_Wake functions respectively, telling the task to now get the characters in the buffer since the variable length packet had arrived.*

## UART FUNCTIONS

*K_Get_Str_Wait*

Purpose: This allows a task to retrieve a specified number of characters from the UART receive buffer and wait for a specified amount of time if that number is not available. If there are at least the number of characters wanted, then those characters will be placed at the address specified by the task and returned to immediately with the return count set to the number requested. If the number of characters are not present, then the task will wait for the desired time period specified. This time period may range from zero to 65535, with zero indicating an indefinite wait. When either the number of characters is received or the time period specified expires, then the task will be returned to identifying whether the desired number of characters were retrieved or not, by the count being non-zero or zero respectively. CMX assumes only one task has ownership to the UART receive buffer. If more than one task has access at the same time, a resource should be set up.

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
word16 K_Get_Str_Wait(void *,word16,word16); /* this is the function prototype */

unsigned char *DEST_PTR; /* this may be local or global, and will be the location that
the function puts the character. */

#define NUMBER ??? /* number of characters wanted. */

#define TIME_PERIOD ??? /* The time wait period. */

unsigned short COUNT; /* this will specify the number of characters retrieved from
buffer. */

COUNT = K_Get_Str_Wait(DEST_PTR,NUMBER,TIME_PERIOD);
```

**Passed**

DEST_PTR = the address where this function will place the characters.

TIME_PERIOD is the number of system ticks that this task will wait for characters to be received. May range from zero to 65535, with zero indicating an indefinite wait.

NUMBER is the number of characters to retrieve from the UART receive buffer. Only the specified number will be transferred.

**Returned**

COUNT = the number of characters that were transferred from the UART receive buffer to the destination address.

If the COUNT value is zero, the number of characters requested was not present in the UART receive buffer and the time period expired. If the COUNT is non-zero, then the number of characters requested was transferred to the destination address passed to this function.

*K_Get_Str_Wait* example:

unsigned char recv_array[80];

/* this will receive up to 80 characters, could be local if you want or could just be a depth of the specified number that the task will want to retrieve. */

```
void task1(void)
{
  unsigned short count; /* local */
  unsigned char c;
  unsigned char *recv_ptr; /* could be local or global. */

  recv_ptr = recv_array; /* load pointer with address of
  recv_array. */
  count = K_Get_Str_Wait(recv_ptr,40,20);
  /* go get 40 characters, if there are not 40 characters, wait
  for up to 20 system ticks for 40 to arrive. */
  while (count--)
    {
    c = *recv_ptr++; /* get character and increment pointer. */
    ...
    }
  ...
}
```

## UART FUNCTIONS

*K_Get_Str_Wait_Return*

Purpose: This function is like the above *K_Get_Str_Wait* function, with the difference being that if the specified number of characters requested are not in the receive buffer after the time period, then the number of characters present will still be transferred. If there are at least the number of characters wanted, then those characters will be placed at the address specified by the task and returned to immediately with the return count set to the number requested. If the number of characters is not present, then the task will wait for the desired time period specified. This time period may range from zero to 65535, with zero indicating an indefinite wait. When either the number of characters is received or the specified time period expires, the task will be returned to with the number of characters received transferred.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
word16 *K_Get_Str_Wait_Return*(void *,word16,word16); /* this is the function prototype */

unsigned char *DEST_PTR; /* this may be local or global, and will be the location that the function puts the character. */

#define NUMBER ??? /* number of characters wanted. */

#define TIME_PERIOD ??? /* The time wait period. */

unsigned short COUNT; /* this will specify the number of characters retrieved from buffer. */

COUNT = *K_Get_Str_Wait_Return*(DEST_PTR,NUMBER,TIME_PERIOD);

**Passed**

DEST_PTR = the address where this function will place the characters.

TIME_PERIOD is the number of system ticks this task will wait for characters to be received. May range from zero to 65535, with zero indicating an indefinite wait.

NUMBER is the number of characters to retrieve from the UART receive buffer.

**Returned**

COUNT = the number of characters that were transferred from the UART receive buffer to the destination address.

If the COUNT value is zero, there were no characters in the UART receive buffer when the time period expired. If the COUNT is non zero, then the number of characters requested or present after time period expired were transferred to the destination address passed to this function.

*K_Get_Str_Wait_Return* example:

unsigned char recv_array[80];

/* this will receive up to 80 characters, could be local if user wants or could just be a depth of the specified number that the task will want to retrieve. */

```
void task1(void)
{
  unsigned short count; /* local */
  unsigned char c;
  unsigned char *recv_ptr; /* could be local or global. */

  recv_ptr = recv_array; /* load pointer with address of
  recv_array. */
  count = K_Get_Str_Wait_Return(recv_ptr,40,20);
  /* go get 40 characters, if there are not 40 characters, wait
  for up to 20 system ticks for 40 to arrive.  Still retrieve the
  number of characters in the buffer after time period expires.
  */
  if (count < 40)
    {
    .../* maybe test to see if count requested less than count
    received and possibly act on it in some way. */
    while (count--)
      {
      c = *recv_ptr++; /* get character and increment pointer. */
      ...
      }
    }
  ...
}
```

## UART FUNCTIONS

*K_Init_Recv*

Purpose: The *K_Init_Recv* function will initialize the receive buffer by setting its associated pointers to the beginning of the receive buffer. Also the receive count_in variable, that indicates the number of bytes received but not retrieved yet, will be reset to zero. The receiver status flags will be reset indicating the receiver is fine and that no errors, full or otherwise, exist.

You are free to call the *K_Init_Recv* at any time, as long as a task is not waiting for the receiver, because the specified number of characters the task wants, are not yet present. Remember, if there are characters in the receive buffer that have not yet been retrieved, these characters will be lost. The receiver BAUD rate, number of data bits, parity and number of stop bits can be set here.

This is an example of the *K_Init_Recv* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_Init_Recv*(void); /* this is the function prototype */

*K_Init_Recv*();

**Passed**

```
main()
{
  K_Init_Recv(); /* go initialize the UART receiver */
  ...
}
```

**Returned**

Nothing is returned.

## UART FUNCTIONS

*K_Init_Xmit*

Purpose: The *K_Init_Xmit* function will initialize the transmit buffer by setting its associated pointers to the beginning of the transmit buffer. Also the transmit count_out variable, indicating the number of bytes which need to be transmitted, will be reset to zero. The transmitter status flags will be reset indicating the transmitter is fine and not busy. You should only call the *K_Init_Xmit* once. There is no reason to ever reinitialize the transmit buffer. The transmitter BAUD rate, number of data bits, parity and number of stop bits can be set here.

This is an example of the *K_Init_Xmit* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_Init_Xmit*(void); /* this is the function prototype */

*K_Init_Xmit*();

**Passed**

```
main()
{
  K_Init_Xmit(); /* go initialize the UART transmitter */
  ...
}
```

**Returned**

Nothing is returned.

## MESSAGE MANAGER FUNCTION

*K_Mbox_Event_Set*

Purpose: This function sets up the mailbox. When the mailbox contains or receives a message, it will automatically use the *K_Event_Signal* function, setting a specific event bit of a particular task. This allows a task to wait on events. When a message arrives in a mailbox, the mailbox will set the event, notifying the task that there are messages in the mailbox.

### Called

Before entering RTOS, tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Mbox_Event_Set(byte,byte,word16); /* this is the function prototype */

unsigned char TASK_SLOT; /* Should be global */

#define EVENT ??? /* which event bit to set */

#define MBOX1 ??? /* numerical identifier that identifies the particular mailbox */

unsigned char STATUS; /* receives status of function. */

STATUS = K_Mbox_Event_Set(MBOX1,TASK_SLOT,EVENT);
```

### Passed

MBOX1 is the mailbox number in which the task would like to test for messages. This number ranges from zero to the maximum number of mailboxes specified minus one.

TASK_SLOT is the slot number of the task that will have an event bit set, when there are messages in this mailbox.

EVENT is the event identifier that determines which event bit will be set in the declared task.

### Returned

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the mailbox number is out of range.

☞ *Remember the mailbox will set the event the first time a message arrives and will also set the event each time the task retrieves a message, if more messages are in this mailbox. This function may be called more than once to specify a different event and/or task.*

*K_Mbox_Event_Set* example:

#define MBOX1 1 /* this is mailbox 1 identifier */

#define EVENT_MBOX1 0x4000 /* identify which event to set. */

unsigned char task2_slot; /* should be global. */

```
void task2(void)
{
  unsigned char status; /* local */
  unsigned short events;
  ...
  status = K_Mbox_Event_Set(MBOX1,task2_slot,EVENT_MBOX1);

  /* Task 1 will set up mailbox 1 to set task 2 event bit 14.
  When a message is placed into mailbox 1, then task 2 will have
  its event set.  This is very useful for tasks to wait on
  multiple events, then process the specified event when it
  happens.  Tasks can have many mailboxes and when the mailboxes
  receive a message, notify the task. */

  if (status)
    {
    /* error, mailbox number out of range */
    }
  while(1)
    {
    events = K_Event_Wait(EVENT_MBOX1,0,2); /* now let task wait
    indefinitely for an event to match.  Could wait on more than
    one event.  When task resumed, then could test events for what
    to do.  Event auto cleared when match happened. */
    if (events & EVENT_MBOX1)
      {
      /* go get message and process, etc. */
      }
    if (events & ???)
      {
      /* process this event setting, etc. */
      }
      ...  etc.
    }
  ...
}
```

## MEMORY MANAGER FUNCTION

*K_Mem_FB_Create*

Purpose: This function creates a fixed block memory pool. No memory contention is checked for by this function. It is assumed the pool's memory is free to be used and will not be used by code except through the memory function supplied.

You specify the number of fixed blocks, and the size of each block within this memory pool. You also pass the beginning address of memory where they would like this memory pool to reside. It is up to you to ensure the memory allocated to this block is large enough to support the memory requirements of this pool. The memory requirement for a pool is the number of blocks times the size of each block, plus the size of a character pointer. In addition, you must align this memory space to the alignment rules for the particular processor you are using.

**Called**

Before entering RTOS, tasks.

```
#include <cxfuncs.h> /* has function prototype */
void K_Mem_FB_Create(void *,word16,word16); /* this is the function prototype */

#define BLK_SIZE ??? /* size of each memory block */

#define NUM_BLOCKS ??? /* number of memory blocks */

struct {
unsigned char *dummy_ptr; /* will allocated space for CMX */
unsigned char pool_bytes[BLK_SIZE * NUM_BLOCKS]; /* allocate enough memory
for this memory pool. */
} MEM_POOL1; /* this is memory pool 1 */

K_Mem_FB_Create(&MEM_POOL1,BLK_SIZE,NUM_BLOCKS);
```

**Passed**

&MEM_POOL1 is the beginning address where this memory pool will reside in memory.

BLK_SIZE is the size in bytes, that each block within this memory pool will have. Maximum of 255 bytes

NUM_BLOCKS is the number of fixed blocks within this memory pool. Maximum is 65535.

**Returned**

No status is returned.

☞ *Remember you must ensure enough memory for this memory pool to exist, and no memory contention is tested for. Also, if the processor must have pointers residing on specific boundaries, like an even address, then the size of a block must be even. This is because CMX places pointers within the unused memory blocks, for internal use.*

*K_Mem_FB_Create* Example:

#define BLK_SIZE 10 /* each memory block to be 10 bytes */

#define NUM_BLOCKS 15 /* there will be 15 memory blocks */

struct {
unsigned char *dummy_ptr; /* this will allocated space for CMX */
unsigned char pool_bytes[BLK_SIZE * NUM_BLOCKS]; /* allocate enough memory for this memory pool. */
} MEM_POOL1; /* this is memory pool 1 */

```
void task2(void)
{
  ...
  K_Mem_FB_Create(&MEM_POOL1,BLK_SIZE,NUM_BLOCKS);

  /* task 2 will create a fixed block memory pool.  This memory
  pool will be created with each block having a size of 10, and
  there will be 15 blocks within this memory pool. */
  ...
}
```

## MEMORY MANAGER FUNCTION

*K_Mem_FB_Get*

Purpose: This function retrieves a fixed block of memory, if one is available and returns the address of this block. The fixed block memory is contiguous, but may have garbage left in the block's bytes from past usage of this block.

**Called**

By tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Mem_FB_Get(void *,byte **); /* this is the function prototype */

struct {/* previous created by K_Mem_FB_Create function */
unsigned char *dummy_ptr;
unsigned char pool_bytes[BLK_SIZE * NUM_BLOCKS];
} MEM_POOL1; /* this is memory pool 1 */

unsigned char *BLOCK_ADDR; /* block pointer could be local or global */

unsigned char STATUS; /* should be local */

STATUS = K_Mem_FB_Get(&MEM_POOL1,&BLOCK_ADDR);
```

**Passed**

&MEM_POOL1 is the beginning address where this particular memory pool will reside in memory.

&BLOCK_ADDR is the address of the unsigned char pointer, in which the address of the fixed block will be placed.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: no free block within this memory pool.

If STATUS equals K_OK, then BLOCK_ADDR contains the block's address. Copy this address to another pointer, because when releasing this block, you will have to pass this address back.

*K_Mem_FB_Get* Example:

struct {/* previously created by *K_Mem_FB_Create* function */
unsigned char *dummy_ptr;
unsigned char pool_bytes[BLK_SIZE * NUM_BLOCKS];
} MEM_POOL1; /* this is memory pool 1 */

```
void task2(void)
{
  unsigned char status/* local */
  unsigned char *blk_addr; /* local, for remembering address of
  block. */
  unsigned char *blk_ptr; /* local, for using memory block */
  ...
  status = K_Mem_FB_Get(&MEM_POOL1,&blk_addr);

  /* task 2 will retrieve a free block out of memory pool 1, if
  a free block exists. */

  if (status == K_OK)
    {
    blk_ptr = blk_addr; /* copy address of block into block
    pointer, now block pointer may be used freely. */
    ...
    }
  ...
}
```

Comments

You must ensure the address is passed back, held, and left untouched since it will be passed to the *K_Mem_FB_Release* function when you have finished with this block and would like to release it. Again, no memory testing is done to see if the memory pool's address is a valid one or not.

## MEMORY MANAGER FUNCTION

*K_Mem_FB_Release*

Purpose: This function releases the block of memory back to a particular pool, which will then be considered free and available.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_Mem_FB_Release*(void *,byte *); /* this is the function prototype */

struct {/* previously created by *K_Mem_FB_Create* function */
unsigned char *dummy_ptr;
unsigned char pool_bytes[BLK_SIZE * NUM_BLOCKS];
} MEM_POOL1; /* this is memory pool 1 */

unsigned char *BLOCK_ADDR; /* previously declared, block pointer could be local or global */

*K_Mem_FB_Release*(&MEM_POOL1,BLOCK_ADDR);

**Passed**

&MEM_POOL1 is the beginning address where this particular memory pool will reside in memory.

BLOCK_ADDR is the contents of the BLOCK_ADDR address, which contains the address of the block that was retrieved by the *K_Mem_FB_Get* function.

**Returned**

No status is returned.

☞ *Ensure that the block address passed to this function is the same address received by the K_Mem_FB_Get function. No testing is performed to check the validity of this address.*

*K_Mem_FB_Release* Example:

struct {/* previously created by *K_Mem_FB_Create* function */
unsigned char *dummy_ptr;
unsigned char pool_bytes[BLK_SIZE * NUM_BLOCKS];
} MEM_POOL1; /* this is memory pool 1 */

```
void task2(void)
{
```

```
  unsigned char *blk_addr; /* local, for remembering address of
  block. */
  unsigned char *blk_ptr; /* local, for using memory block */
  ...
  K_Mem_FB_Release(&MEM_POOL1,blk_addr);

  /* task 2 will now release this particular block back to its
  particular memory pool.  When this block is released, it is
  then considered free, and may be again released to another
  task, or even again to this task, when a block is requested out
  of this memory pool. */
  ...
}
```

## MESSAGE MANAGER FUNCTION

*K_Mesg_Ack_Sender*

Purpose: This function wakes a task that sent a message using the *K_Mesg_Send_Wait* function. The task that retrieves the message must issue this call prior to retrieving the next message. The task may always call this function if it is not sure whether the message retrieved was sent with either the *K_Mesg_Send_Wait* or *K_Mesg_Send* functions. This function is intelligent enough to determine whether a task is waiting for this call or not.

**Called**

Tasks

#include <cxfuncs.h> /* has function prototype */
byte *K_Mesg_Ack_Sender*(void); /* this is the function prototype */

unsigned char STATUS; /* should be local */

STATUS = *K_Mesg_Ack_Sender*();

**Passed**

Nothing.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_NOT_WAITING = Warning: The task that sent the received message was not waiting.

If STATUS equals K_OK, then the task that had sent the message had used the *K_Mesg_Send_Wait* function and the task was suspended is now placed into the READY state.

☞ *An immediate rescheduling will occur if the awakened task has a higher priority than the current running task.*

*K_Mesg_Ack_Sender* example:

#define MBOX1 1 /* mailbox 1 identifier. */

```
void task2(void)
{
```

```
  unsigned char status; /* local */
  unsigned char *recv_ptr; /* local pointer to receive address
  of message. */
  ...
  recv_ptr = K_Mesg_Get(); /* go get message if one is available
  */

  if (recv_ptr) /* not NULL, must be message */
    {
    status = K_Mesg_Ack_Sender();

    /* Task 2 will wake task 1, which is the one that sent message
    with the K_Mesg_Send_Wait function. */

    .../* process message.  NOTE: the K_Mesg_Ack_Sender function
    can be called any time prior to calling another function that
    retrieves a message, such as the K_Mesg_Get or K_Mesg_Wait
    functions. */
    }
  ...
}
```

## MESSAGE MANAGER FUNCTION

*K_Mesg_Get*

Purpose: This function allows a task to retrieve a message address from a mailbox, if one is available. The task will not be suspended whether there is a message or not.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void * *K_Mesg_Get*(byte); /* this is the function prototype */

#define MBOX1 ??? /* numerical identifier that identifies the particular mailbox */

unsigned char *RECV_PTR; /* could be local or global */

RECV_PTR = *K_Mesg_Get*(MBOX1);

**Passed**

MBOX1 is the mailbox number in which the task would like to test for messages. This number ranges from zero to the maximum number of mailboxes specified minus one.

**Returned**

RECV_PTR is the pointer that will be given the address where the message bytes are located.

If RECV_PTR = NULL (0), then there was NOT a message in this mailbox when this task called. If the RECV_PTR contains a non null value, this will be the address of the message bytes.

*K_Mesg_Get* example:

#define MBOX1 1 /* this is mailbox 1 identifier */

unsigned char *recv_ptr; /* create a pointer to receive the message address, could also be local variable to task */

```
void task1(void)
{
  unsigned char status; /* local */
  ...
  recv_ptr = K_Mesg_Get(MBOX1);
  /* Task 1 will try to get a message from mailbox 1.  If a
  message is available, then the address where the message is
  will be returned back to the caller. */
```

```
  if (recv_ptr) /* see if non NULL value */
    {
    /* process message code here */
    }
  ...
}
```

## MESSAGE MANAGER FUNCTION

*K_Mesg_Send*

Purpose: This function allows tasks and interrupts to send a message to a mailbox. The calling task or interrupt will not wait for a *K_Mesg_Ack_Sender* request. Remember, the message itself is not sent, just the address of the message is passed to the mailbox. The message contents can be virtually anything as long as the sender and receiver agree on the format. This is extremely useful, as interrupts may send a message identifying a port's pin states for example. Also, a task may own several mailboxes, so a priority scheme could be set up with high priority message sent to mailbox one, lower priority messages sent to mailbox two, and lowest priority messages sent to mailbox three.

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Mesg_Send(byte,void *); /* this is the function prototype */

#define MBOX1 ??? /* numerical identifier that identifies the particular mailbox */

unsigned char SOURCE_BYTES[] = ??? /* could be global or local */

unsigned char STATUS; /* should be local to task */

STATUS = K_Mesg_Send(MBOX1,SOURCE_BYTES);
```

**Passed**

MBOX1 is the mailbox number in which the task would like to test for messages. This number ranges from 0 to the maximum number of mailboxes specified minus 1.

SOURCE_BYTES is the address where the message bytes reside, which will be copied into the mailbox's message pointer.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: there are NO more message blocks available or the mailbox number is out of range.

If STATUS equals K_OK, then a message has been placed into the mailbox message block. Make note that each mailbox works as a FIFO (first in, first out) queue.

*K_Mesg_Send* example:

unsigned char global_message[] = {"global message\n"}; /* this is a global message. */

#define MBOX1 1 /* numerical identifier that identifies the particular mailbox */

```
void task2(void)
{
  unsigned char status; /* local */
  ...
  status = K_Mesg_Send(MBOX1,global_mesg);

  /* Task 2 will send a message to mailbox 1.  Just the address
  where this message resides, is placed into the mailbox message
  block. */
  if (status != K_OK)
    {
    /* an error occurred because either there were no more message
    blocks available or the mailbox specified is out of range. */
    ...
    }
  status = K_Mesg_Send(MBOX1,"could pass a message like this");

  /* This shows the user another way in which a message could be
  passed. */
}
```

☞ *Mailboxes do not have a predetermined depth. Mailboxes can be of any depth, for the message blocks are the actual carrier of the message address. As long as a message block is free, it will be given to any mailbox that needs it.*

## MESSAGE MANAGER FUNCTION

*K_Mesg_Send_Wait*

Purpose: This function sends a message to a mailbox. The calling task will also wait for either the time period to expire or for the *K_Mesg_Ack_Sender* function to wake it. If there is a message block free, it will be linked into the mailbox and the address of the message bytes being sent will be placed into the message block. The task will also wait for the task that receives this message to issue the *K_Mesg_Ack_Sender* function to wake this task. The task can specify the time period it is willing to wait. This can be indefinitely or for a number of system ticks.

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Mesg_Send_Wait(byte,word16,void *); /* this is the function prototype */

#define MBOX1 ??? /* numerical identifier that identifies the particular mailbox */

#define TIME_PERIOD ??? /* amount of time to wait for */

unsigned char SOURCE_BYTES[] = ???; /* could be global or local */

unsigned char STATUS; /* should be local to task */

STATUS = K_Mesg_Send(MBOX1,TIME_PERIOD,SOURCE_BYTES);
```

**Passed**

MBOX1 is the mailbox number which the task would like to test for messages. This number ranges from zero to the maximum number of mailboxes specified minus one.

TIME_PERIOD is the number of system ticks this task will wait for the *K_Mesg_Ack_Sender* function to wake it. A period of zero indicates to wait indefinitely. The period may range from zero to 65535.

SOURCE_BYTES is the address where the message bytes reside, which will be copied into the mailbox's message pointer.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: there are no more message blocks available or the mailbox number is out of range.

K_TIMEOUT = warning: That the time period expired before the *K_Mesg_Ack_Sender* function was used to wake this task.

If STATUS equals K_OK, then a message had been placed into the mailbox, a task then received this message and issued the *K_Mesg_Ack_Sender* function to notify this task (sender) that the message was received.

*K_Mesg_Send_Wait* example:

unsigned char global_message[] = {"global message\n"}; /* this is a global message. */

#define MBOX1 1 /* numerical identifier that identifies the particular mailbox */

```
void task2(void)
{
  unsigned char status; /* local */
  ...
  status = K_Mesg_Send_Wait(MBOX1,0,global_mesg);

  /* Task 2 will send a message to mailbox 1.  Task 2 then will
  be suspended indefinitely, waiting for the task that receives
  this message to use the K_Mesg_Ack_Sender function, notifying
  task 2 that the message was received. */
  if (status != K_OK)
    {
    if (status == K_TIMEOUT) /* because of time out */
      {
      ...
      }
    else
      {
      /* an error occurred because either there were no more
      message blocks available or the mailbox specified is out of
      range. */
      }
    }
  ...
  status = K_Mesg_Send(MBOX1,30,"hello world");

  /* Task 2 will now wait for only 30 system ticks for task that
  receives this message to use K_Mesg_Ack_Sender function. */
}
```

## MESSAGE MANAGER FUNCTION

*K_Mesg_Wait*

Purpose: This function allows a task to wait for a message from a specific mailbox for a specified period of time. The task will specify the number of system ticks to wait for a message. A time period of zero will suspend the task indefinitely waiting for a message. The task will remain suspended until either the specified number of ticks expire or a message is received, at which time the task will be put into the READY state.

### Called

Tasks.

#include <cxfuncs.h> /* has function prototype */
void * *K_Mesg_Wait*(byte,word16); /* this is the function prototype */

#define MBOX1 ??? /* numerical identifier that identifies the particular mailbox */

#define TIME_CNT ???

unsigned char *RECV_PTR; /* could be local or global */

RECV_PTR = *K_Mesg_Wait*(MBOX1,TIME_CNT);

### Passed

MBOX1 is the mailbox number in which the task would like to test for messages. This number ranges from zero to the maximum number of mailboxes specified minus one.

TIME_CNT is the number of system ticks to wait for a message. The range is zero through 65535. If the value is zero, then the task will wait indefinitely for a message to arrive.

### Returned

RECV_PTR is the pointer that will be given the address where the message bytes are located.

If RECV_PTR = NULL (0), then either the time period specified expired prior to a message being retrieved or the mailbox number was out of range. If the RECV_PTR contains a non null value, this will be the address where the message bytes are.

*K_Mesg_Wait* example:

#define MBOX1 1 /* this is mailbox 1 identifier */

unsigned char *recv_ptr; /* create a pointer to receive the message address, could also be local variable to task */

```
void task1(void)
{
  unsigned char status; /* local */
  ...
  recv_ptr = K_Mesg_Wait(MBOX1,100);

  /* Task 1 will wait for a message.  If a message is available,
  then task 1 will not be suspended and the recv_ptr will have
  the message address on return.  Otherwise task 1 will wait for
  100 system ticks for a message to arrive.  If a message arrives
  within this time frame then task 1 will automatically be put
  into the READY state.  If the time period expires, then the
  task will be put into the READY state and notified that no
  message was received */

  if (recv_ptr) /* see if non NULL value */
    {
    /* process message code here */
    }
  ...
}
```

☞ *If a task is waiting for a message and one comes in prior to the time period elapsing, the task is immediately put back into the READY to resume state.*

## OPERATING SYSTEM FUNCTION

*K_OS_Disable_Interrupts*

Purpose: This function GLOBALLY disables the interrupts. Any non-maskable interrupt will not be immediately recognized. If the interrupt sets a latch, then this will not be prevented. This uses the particular CPU instruction that masks out all non-maskable interrupts from being acknowledged and processed.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Disable_Interrupts*(void); /* this is the function prototype */

*K_OS_Disable_Interrupts*();

**Passed**

Nothing is passed.

**Returned**

Nothing is returned.

Read the particular CPU manual that further describes the action and the effect it has on interrupts when the interrupts are GLOBALLY disabled.

*K_OS_Disable_Interrupts* example:

```
void task1(void)
{
  unsigned char status; /* local */
  .../* application code here */

  K_OS_Disable_Interrupts(); /* do critical region of code
  stuff. */
  ...
  K_OS_Enable_Interrupts(); /* re-enable interrupts. */

  .../* application code here */
}
```

Comments

Use this function sparingly or not at all if possible. ALL maskable interrupts will not be recognized until the interrupts are re-enabled. This will also add latency time to the interrupt processing its code. If used, the user must remember to re-enable interrupts using the *K_OS_Enable_Interrupts* function.

Some C vendor compilers have special instructions that will allow the interrupts to be GLOBALLY disabled and the code placed inline. This executes faster than calling the equivalent CMX function.

## OPERATING SYSTEM FUNCTION

*K_OS_Enable_Interrupts*

Purpose: This function GLOBALLY enables interrupts. It is used in conjunction with the *K_OS_Disable_Interrupts* function. All non-maskable interrupts that were GLOBALLY disabled, will be enabled. If any interrupt is pending, then the interrupt will now be recognized and processed according to the CPU interrupt hardware mechanism and priority scheme.

### Called

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Enable_Interrupts*(void); /* this is the function prototype */

*K_OS_Enable_Interrupts*();

### Passed

Nothing is passed.

### Returned

Nothing is returned.

Read the particular CPU manual that further describes the action and the effect on interrupts when they are GLOBALLY enabled.

*K_OS_Enable_Interrupts* example:

```
void task1(void)
{
  unsigned char status; /* local */
  .../* application code here */

  K_OS_Disable_Interrupts(); /* do critical region of code
  items. */
  ...
  K_OS_Enable_Interrupts(); /* re-enable interrupts. */

  .../* application code here */
}
```

Comments

This function should be called as soon as possible after the *K_OS_Disable_Interrupts* function is called. Some C vendor compilers have special instructions that allow the interrupts to be GLOBALLY enabled and the code placed inline. This executes faster than calling the equivalent CMX function.

## OPERATING SYSTEM FUNCTION

*K_OS_Init*

Purpose: This function is called to set up all the memory needed for CMX and initialize the CMX variables, parameters and configurable system maximums. The *K_OS_Init* function must be called before any other CMX functions are called. This is done in the start up code. The file containing the function *K_OS_Init* should be compiled each time you change the "CXCONFIG.H" file, which declares the application's maximums.

**Called**

Before using any other CMX function calls.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Init*(void); /* this is the function prototype */

*K_OS_Init*();

**Passed**

Nothing.

**Returned**

Nothing.

☞ *Remember this function must be called before any other CMX function is called. If not, then disastrous results will occur.*

*K_OS_Init* Example:

```
void main(void)
{
  /* define any locals within main */

  K_OS_Init(); /* initialize CMX */
  ...
  /* now the user can access any CMX function, that is allowed
  to be accessed prior to entering the CMX operating system */
  ...
}
```

## OPERATING SYSTEM FUNCTION

*K_OS_Intrp_Entry*

Purpose: The *K_OS_Intrp_Entry* function is used by most interrupts. The interrupt's first instruction is to call the *K_OS_Intrp_Entry* function. This informs CMX it should save the context of the CPU registers and swap in the interrupt stack when a interrupt occurs. See the chapter on Processor Specific Information for details on the particular CPU you are working with.

Because some C compilers generate code (such as setting up the frame pointer for a function), the interrupt handler, for most processors, should be written in assembly language. The interrupt handler follows the guidelines for the particular CPU calling the *K_OS_Intrp_Entry* routine. After this is done, the interrupt handler may call the interrupt processing code written in C, or just continue with the assembly language interrupt processing code.

It is recommended to write all interrupt code in assembly, since greater speed is achieved. If the interrupt does not call this function, then the interrupt must save and restore all registers used, and this interrupt may not use any CMX calls. Please read the Processor Specific Information chapter on interrupts and the additional interrupt notes supplied for the CPU you are working with.

**Called**

Interrupts only.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Intrp_Entry*(void); /* this is the function prototype */

☞ *The call must be written in assembly language to the specific CPU instruction set.*

call *K_OS_Intrp_Entry* ;written in assembly some CPU'S

jsr *K_OS_Intrp_Entry* ;other CPU'S. etc.

**Passed**

Nothing.

**Returned**

Nothing.

☞ *Interrupts are disabled when this function returns. The user may allow this interrupt to re-enable interrupts or not. Interrupts such as the NMI cannot be disabled though. When this function is called, the task (if one is RUNNING) contexts will be saved. If this is the first interrupt (CMX allows nested interrupts), then the interrupt stack will be switched in.*

*K_OS_Intrp_Entry* example:

☞ *Written in assembly.*

INTERRUPT_X_HANDLER: ;this is where the specific interrupt will vector to, when ;the interrupt is recognized by the CPU.

; prologue code if need be. DEPENDENT on CPU.

;

call *K_OS_Intrp_Entry* ;written in assembly some CPU'S

jsr *K_OS_Intrp_Entry* ;Hitachi H8/300 series

Now the interrupt may finish the code necessary to process this interrupt. Some CMX calls are allowed. The interrupt code could now call the interrupt function code written in C language. This is not recommended though, since interrupts should be as fast as possible. Also at this point this interrupt may re-enable interrupts. If you decide to re-enable interrupts, make sure that either this interrupt is masked out (won't be acknowledged again) or that the CPU hardware will not vector to this interrupt again until this interrupt has finished its code.

;interrupt's code ...

call *K_OS_Intrp_Exit* ;epilogue, must be called by interrupts that have used the *K_OS_Intrp_Entry* call.

If the interrupt used the *K_OS_Intrp_Entry* call, then the CPU's instruction that indicates return from interrupt (RETI on 8051 CPU, RTE on the Hitachi H8/300 series CPU's) is not needed. CMX will inform the CPU that the interrupt is finished within the *K_OS_Intrp_Exit* code.

## OPERATING SYSTEM FUNCTION

*K_OS_Intrp_Exit*

Purpose: Any interrupt that used the *K_OS_Intrp_Entry* call to have CMX save the current task context and switch in the interrupt stack, must use this function when the interrupt has finished its code. This takes the place of the normal CPU's return from interrupt instruction. This function is the last instruction of the interrupt's code.

Please read the Processor Specific Information chapter on interrupts and the additional interrupt notes supplied for the CPU you are working with.

**Called**

Interrupts only.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Intrp_Exit*(void); /* this is the function prototype */

☞ *Call must be written in assembly language to the specific CPU instruction set.*

call *K_OS_Intrp_Exit* ;written in assembly some CPU'S

jsr *K_OS_Intrp_Exit* ;other CPU'S. etc.

**Passed**

Nothing.

**Returned**

Does not return.

☞ *If the interrupt is nested (meaning this interrupt is at least the second interrupt) then this call will return to the prior interrupt's code. If this is the first interrupt, then this call will invoke the scheduler. The scheduler will determine whether the interrupt used a CMX call that requires a task swap, or that the task RUNNING prior to this interrupt should still be the RUNNING task.*

*K_OS_Intrp_Exit* example:

☞ *Written in assembly.*

INTERRUPT_X_HANDLER: ;this is where the specific interrupt will vector to, when
;the interrupt is recognized by the CPU.

; prologue code if need be. DEPENDENT on CPU.

;

;call *K_OS_Intrp_Entry* ;written in assembly some CPUs

;jsr *K_OS_Intrp_Entry* ;Hitachi H8/300 series

Now the interrupt may finish the code necessary to process this interrupt. Some CMX calls are allowed. The interrupt code could now call the interrupt function code written in C language. This is not recommended though, since interrupts should be as fast as possible. Also at this point this interrupt may re-enable interrupts. If you decide to re-enable interrupts, make sure that either this interrupt is masked-out (won't be acknowledged again) or that the CPU hardware will not vector to this interrupt again until this interrupt has finished its code.

;interrupt's code ...

call *K_OS_Intrp_Exit* ;epilogue, must be called by interrupts that have used the *K_OS_Intrp_Entry* call.

If the interrupt used the *K_OS_Intrp_Entry* call, then the CPU instruction that indicates return from interrupt (RETI on 8051 CPU, RTE on the Hitachi H8/300 series CPU's) is not needed. CMX will inform the CPU that the interrupt is finished within the *K_OS_Intrp_Exit* code.

## OPERATING SYSTEM FUNCTION

*K_OS_Low_Power_Func*

Purpose: Lets the processor enter reduced power state. The *K_OS_Low_Power_Func* function is called by the CMX scheduler module written in assembly. It is up to you to code this function to invoke the CPU power down mode.

In most cases, you select the state that allows the processor to resume normal activity when the interrupt used for the "system tick" occurs. This way the tasks and cyclic timers that need their time periods reduced and tested will be able to do so. Also it is up to you that the processor returns properly to the next instruction within the assembly module that called the *K_OS_Low_Power_Func* function. You can move this function to the assembly module if need be. Remember, CMX assumes that when the CPU comes out of the reduced power state it will return properly to the instruction following the instruction that called this function.

**Called**

By the CMX scheduler.

```
#include <cxfuncs.h> /* has function prototype */
void K_OS_Low_Power_Func(void); /* this is the function prototype */

...

"in assembly coded scheduler module"

call K_OS_Low_Power_Func
```

**Passed**

Nothing.

**Returned**

Nothing.

☞ *Remember, you should code this function for the power down state you would like and to ensure the processor returns to the instruction following the call to K_OS_Low_Power_Func.*

*K_OS_Low_Power_Func* Example:

The scheduler will determine when NO tasks are able to run, because the tasks are either IDLE or SUSPENDED.

call *K_OS_Low_Power_Func* ;proper assembly code instruction by CMX within the scheduler assembly code.

```
void K_OS_Low_Power_Func(void)
{
  .../* user written routine here. */
  /* NOTE: may be coded in assembly if need be for the particular
  CPU and / or C vendor tools. */
}
```

## OPERATING SYSTEM FUNCTION

*K_OS_Slice_Off*

Purpose: This function disables time slicing if it had been previously enabled by the *K_OS_Slice_On* function. Tasks will no longer be time sliced. When called, this function takes effect immediately.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Slice_Off*(void); /* this is the function prototype */

*K_OS_Slice_Off*();

**Passed**

Nothing is passed.

**Returned**

Nothing is returned.

Study the Time Slice Chapter to better understand how time slicing works.

*K_OS_Slice_Off* example:

```
void task1(void)
{
  unsigned char status; /* local */

  K_OS_Slice_On(); /* enable time slicing. */

  .../* application code here */

  K_OS_Slice_Off(); /* disable time slicing. */
  ...
}
```

Comments

You can freely allow tasks to enable and disable time slicing when necessary. In most cases, time slicing will not be needed.

## OPERATING SYSTEM FUNCTION

*K_OS_Slice_On*

Purpose: This function enables time slicing. Tasks will become time sliced according to their priority with other tasks. The CMX_TSLICE_SCALE value defined in the "CXCONFIG.H" file will be loaded into the slice_count variable and be used to determine when to perform a time slice task change. The change will take effect immediately.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Slice_On*(void); /* this is the function prototype */

*K_OS_Slice_On*();

**Passed**

Nothing is passed.

**Returned**

Nothing is returned.

Study the Time Slice Chapter to better understand how time slicing works.

*K_OS_Slice_On* example:

```
void task1(void)
{
  unsigned char status; /* local */

  K_OS_Slice_On(); /* enable time slicing. */

  .../* application code here */

  K_OS_Slice_Off();
  /* disable time slicing, if user wants. */
  ...
}
```

Comments

You can freely allow tasks to enable and disable time slicing when necessary. In most cases time slicing will not be needed.

## OPERATING SYSTEM FUNCTION

*K_OS_Start*

Purpose: The *K_OS_Start* function is called to invoke the CMX operating system. Once this function is called, the CMX operating system takes control of the CPU and determines when tasks should run and cyclic timers should execute. It is up to you to make sure at least one task is READY or will become READY by using the *K_Task_Start* function before calling *K_OS_Start*. Once you enter the CMX operating system, there is no way to exit the CMX operating system.

**Called**

When you want to enter the CMX operating system.

#include <cxfuncs.h> /* has function prototype */
void *K_OS_Start*(void); /* this is the function prototype */

*K_OS_Start*();

**Passed**

Nothing.

**Returned**

Never returns from the CMX operating system.

*K_OS_Start* Example:

```
void main(void)
{
  /* define any locals within main */

  K_OS_Init(); /* initialize CMX */
  ...
  /* now the user can access any CMX function, that is allowed
  to be accessed prior to entering the CMX operating system */
  ...
  /* Set up CMX by create tasks, cyclic timers, etc.  Possibly
  create queues, set up mailboxes, etc.  Start at least one task.
  */
  ...
  K_OS_Start(); /* enter into the CMX operating system */
  /* NOTE: will never return to this point */
}
```

## OPERATING SYSTEM FUNCTION

*K_OS_Task_Slot_Get*

Purpose: This function returns the task slot number of the currently RUNNING task. The slot number was assigned by the CMX *K_Task_Create* function when the task was created.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_OS_Task_Slot_Get*(void); /* this is the function prototype */

unsigned char TASK_SLOT;  /* could be global or local */

TASK_SLOT = *K_OS_Task_Slot_Get*();

**Passed**

Nothing.

**Returned**

TASK_SLOT is the task slot number of the currently RUNNING task.

*K_OS_Task_Slot_Get* Example:

```
void task2(void)
{
  unsigned char task_slot;

  task_slot = K_OS_Task_Slot_Get();
  /*  returns the task slot number of the task that is currently
  in the RUNNING state */
}
```

## OPERATING SYSTEM FUNCTION

*K_OS_Tick_Get_Ctr*

Purpose: Every time the CMX *K_OS_Tick_Update* function is called by the timer interrupt, a global variable called cmx_tick_count is incremented. This variable contains the running total of the "true" system ticks. The current value of cmx_tick_count is returned by the *K_OS_Tick_Get_Ctr* function.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
word32 *K_OS_Tick_Get_Ctr*(void); /* this is the function prototype */

unsigned long TICK_COUNT; /* could be global or local */

TICK_COUNT = *K_OS_Tick_Get_Ctr*();

**Passed**

Nothing.

**Returned**

TICK_COUNT is the current total number of CMX system ticks. This value will be the total number of times the *K_OS_Tick_Update* function was called.

*K_OS_Tick_Get_Ctr* Example:

```
void task2(void)
{
  unsigned long tick_count;

  tick_count = K_OS_Tick_Get_Ctr();
  /*  returns the total number of system ticks (updated by one
  each time K_OS_Tick_Update was called) that have occurred since
  the RTOS has started */
}
```

## OPERATING SYSTEM FUNCTION

*K_OS_Tick_Update*

Purpose: This function is usually called by a timer interrupt. CMX needs one of the timer interrupts to use as a clock to perform scheduling for tasks which have used a function that invokes a time period, and for cyclic timers.

The *K_OS_Tick_Update* function MUST be called by an interrupt. This interrupt is selectable and should cause an interrupt at a specified time period. The interrupt's frequency should be constant since all time related activities are based on this frequency. The interrupt should first call the *K_OS_Intrp_Entry* function, then call the *K_OS_Tick_Update* function. The *K_OS_Tick_Update* function will decrement a counter that has been pre-loaded with the number of times the interrupt must call this function, before the scheduler flag is set. This counter is loaded with a value you select in the "CXCONFIG.H" file and the C define CMX_RTC_SCALE, as described in the RTOS Configuration File chapter.

When the count specified by the CMX_RTC_SCALE reaches zero, then the counter will be reloaded with the CMX_RTC_SCALE value, and the CMX time flag will be set if any task timers or cyclic timers need servicing. When the interrupt leaves the *K_OS_Intrp_Exit* function, the scheduler will determine whether to let the CMX timer task execute or to resume execution of the current running task. If the scheduler determines the CMX timer task should execute, then the task that was running prior to this interrupt will be put into the ready to RESUME state, saving its context. The scheduler will then let the CMX timer task execute. (See the Scheduler chapter for a more detailed description on how the scheduler works.)

If the current task prior to the interrupt used the CMX *K_Task_Lock* function, then the CMX time flag would be set and the counter reloaded according to the CMX_RTC_SCALE value. This would not invoke the scheduler when the interrupt leaves. This is because the privilege flag has been set by the *K_Task_Lock* function. The privilege flag will not allow any task switch until the privilege flag has been lowered (cleared), which must be done by the task that raised it. To lower the privilege flag, the task must call the CMX *K_Task_Unlock* function.

## UART FUNCTIONS

*K_Put_Char*

Purpose: This allows a task to put a character into the UART transmitter buffer. If the transmitter is busy, the character will not be placed into the transmitter and the task will be notified that the transmitter was busy. Remember, more than one task may try to use the transmitter at the same time.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Put_Char*(void *); /* this is the function prototype */

unsigned char *src_PTR; /* this may be local or global, and will be the location that the function gets the character. */

unsigned char STATUS; /* results of function. */

STATUS = *K_Put_Char*(src_PTR);

**Passed**

src_PTR = the address where the character resides in memory for this function.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_BUSY = Error: transmitter was busy.

If STATUS equals K_OK, then the task which called this function will have the character placed into the transmitter and the transmitter will automatically start, sending out the character.

*K_Put_Char* example:

unsigned char src_byte = '3';

/* This is the address to hold the character which will be sent out. could also be local. */

```
void task1(void)
{
  unsigned char status;
```

```
  status = K_Put_Char(&src_byte); /* go send character. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
  or/* this shows another way. */
  status = K_Put_Char("3"); /* go send character. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
}
```

## UART FUNCTIONS

*K_Put_Char_Wait*

Purpose: This allows a task to put a character into the UART transmitter buffer. If the transmitter is busy, then the task will wait up to a specified time period for it to become free. The specified time period may be zero through 65535 with zero indicating an indefinite wait. If the transmitter is busy, then the task will be suspended until the specified time period. If the transmitter is free or becomes free within the time period, the character will be placed into the transmit buffer and the transmitter started. If the transmitter is busy after the time expires, then the character will not be placed into the transmit buffer and the task will be notified that the transmitter was busy. Remember that more than one task may try to use the transmitter at the same time.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Put_Char_Wait*(void *,word16); /* this is the function prototype */

unsigned char *src_PTR; /* this may be local or global, and will be the location that the function gets the character. */

#define TIME_PERIOD ??? /* The time wait period. */

unsigned char STATUS; /* results of function. */

STATUS = *K_Put_Char_Wait*(src_PTR,TIME_PERIOD);

**Passed**

src_PTR = the address where the character resides in memory for this function.

TIME_PERIOD is the number of system ticks this task is willing to wait for the transmitter to be free. Range is zero to 65535.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_TIMEOUT = Warning/error: The time period expired, transmitter still busy.

K_BUSY = Error: transmitter was busy.

If STATUS equals K_OK, then the task which called this function will have the character placed into the transmitter and the transmitter will automatically start, sending out the character.

*K_Put_Char_Wait* example:

unsigned char src_byte = '3';

/* This is the address to hold the character, which will be sent out. Could also be local. */

```
void task1(void)
{
  unsigned char status;
  status = K_Put_Char_Wait(&src_byte,100); /* go send character.
  Wait up to 100 system ticks for the transmitter to be free if
  need be. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
  or/* this shows another way. */
  status = K_Put_Char_Wait("3",100); /* go send character.
  Wait up to 100 system ticks for the transmitter to be free if
  need be. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
}
```

## UART FUNCTIONS

*K_Put_Str*

Purpose: This allows a task to put a string of characters into the UART transmitter buffer. If the transmitter is busy, the characters will not be placed into the transmitter and the task will be notified that the transmitter is busy. Remember, more than one task may try to use the transmitter at the same time.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Put_Str*(void *,word16); /* this is the function prototype */

unsigned char *src_PTR; /* this may be local or global, and will be the location that the function gets the character. */

#define NUMBER ??? /* the number of characters to place into the transmit buffer. */

unsigned char STATUS; /* results of function. */

STATUS = *K_Put_Str*(src_PTR,NUMBER);

**Passed**

src_PTR = the address where the characters reside in memory for this function.

NUMBER is the number of characters to place into the transmit buffer.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_BUSY = Error: transmitter was busy.

If STATUS equals K_OK, then the task which called this function will have a NUMBER of character(s) placed into the transmit buffer and the transmitter will automatically start sending out the characters.

☞ *The number of characters placed into the transmit buffer is determined by the NUMBER and not by the length of the string. It is up to you to determine the proper number of characters to transmit.*

*K_Put_Str* example:

unsigned char src_bytes[] = {"From task 1"};

/* This is the address to hold the string of characters, which will be sent out. Could also be local. */

```
void task1(void)
{
  unsigned char status;
  status = K_Put_Str(src_bytes,sizeof src_bytes);
  /* go send character.  Remember it is the second parameter that
  determines the number of characters that are actually copied
  to the transmit buffer. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if characters not sent. */
    }
  ...
  or/* this shows another way. */
  status = K_Put_Str("From task 1",12);
  /* go send character.  Remember it is the second parameter that
  determines the number of characters that are actually copied
  to the transmit buffer. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
}
```

## UART FUNCTIONS

*K_Put_Str_Wait*

Purpose: This allows a task to put a string of characters into the UART transmitter buffer. If the transmitter is busy, then the task will wait up to the specified time period for it to become free. The specified time period may be zero through 65535 with zero indicating an indefinite wait. If the transmitter is busy, the task will be suspended until the time period specified. If the transmitter is free or becomes free within the time period, the characters will be placed into the transmit buffer and the transmitter started. If the transmitter is busy after time expires, then the characters will not be placed into the transmit buffer and the task will be notified that the transmitter was busy. Remember, more than one task may try to use the transmitter at the same time.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Put_Str_Wait*(void *,word16,word16); /* this is the function prototype */

unsigned char *src_PTR; /* this may be local or global, and will be the location that the function gets the character. */

#define NUMBER ??? /* the number of characters to place into the transmit buffer. */

#define TIME_PERIOD ??? /* The time wait period. */

unsigned char STATUS; /* results of function. */

STATUS = *K_Put_Str_Wait*(src_PTR,NUMBER,TIME_PERIOD);

**Passed**

src_PTR = the address where the character resides in memory for this function.

NUMBER is the number of characters to place into the transmit buffer.

TIME_PERIOD is the number of system ticks that this task is willing to wait for the transmitter to be free. Range is zero to 65535.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_TIMEOUT = Warning/error: The time period expired, transmitter still busy.

K_BUSY = Error: transmitter was busy.

If STATUS equals K_OK, then the task which called this function will have the character placed into the transmitter and the transmitter will automatically start sending out the character.

*K_Put_Str_Wait* example:

unsigned char src_bytes[] = {"Task1 transmitting this"};

/* This is the address to hold the character, which will be sent out. could also be local. */

```
void task1(void)
{
  unsigned char status;
  status = K_Put_Str_Wait (src_bytes,sizeof src_bytes, 100);
  /* go send characters.  Wait up to 100 system ticks for the
  transmitter to be free. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
  or/* this shows another way. */
  status = K_Put_Str_Wait("Task1 running",14,100);
  /* go send characters.  Wait up to 100 system ticks for the
  transmitter to be free. */
  if (status != K_OK) /* test status */
    {
    /* maybe do something if character not sent. */
    }
  ...
}
```

## QUEUE MANAGER FUNCTION

*K_Que_Add_Bottom*

Purpose: This function fills the bottom free slot of a queue, if one is available, and copies the source bytes dictated by the source pointer into the slot. The number of bytes copied is the size of the slot as indicated when this queue was created.

**Called**

Before entering RTOS by tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Que_Add_Bottom(byte,void *); /* this is the function prototype */

#define QUE_NUM ???

unsigned char *SOURCE_POINTER; /* could be local or global depending upon
application and user. */

unsigned char STATUS; /* should be local */

STATUS = K_Que_Add_Bottom(QUE_NUM,SOURCE_POINTER);
```

**Passed**

QUE_NUM is the queue number, which identifies a particular queue created with the *K_Que_Create* function. Can be from zero to one less than the maximum configured.

SOURCE_POINTER is a pointer that should contain the address where the source bytes are in memory.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: queue was full, no slot available.

K_QUE_FULL = Warning: queue is now full, slot was filled.

If the STATUS equals K_OK, the source contents were copied into the queue slot. This is also true if the STATUS equaled K_QUE_FULL, indicating that the queue is now full.

*K_Que_Add_Bottom* example:

#define QUE1 1 /* this is the numeric identifier for queue 1, which has already been created with a each slot size being 6 bytes */

unsigned char *source_pointer = {"hello"); /* string that will be copied into slot */

```
void task2(void)
{
  unsigned char status; /* local */
  ...
  status = K_Que_Add_Bottom(QUE1,source_pointer);

  /* task 2 will add to queue 1.  The bottom slot, if available
  will be the slot that receives the source bytes.  The number
  of bytes copied, in this case is 6. */

  if (status != K_OK) /* test status */
    {
    /* see if status indicated queue is now full and the contents
    copied or that the queue was full before this call, and that
    the contents were not copied. */
    }
  ...
}
```

Comments

Once the source bytes are copied into the slot you may reuse the source byte memory locations for other uses. It is highly recommended you keep the slot size to a minimum because of the time it takes to copy bytes from the source to the destination. You may want to place just the pointers to the source bytes into the queue's slot. When the slot is retrieved, the task then will actually get the source bytes. The source bytes' memory locations may not be used for other purposes until the slot pointer has been removed.

Also you should send messages, which just pass pointers and not the contents, if speed is of importance and the source bytes do not have to be copied.

## QUEUE MANAGER FUNCTION

*K_Que_Add_Top*

Purpose: This function fills the top free slot of a queue, if one is available, and copies the callers' source bytes dictated by the source pointer into the slot. The number of bytes copied is the size of the slot as indicated when this queue was created.

**Called**

Before entering RTOS and tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Que_Add_Top(byte,void *); /* this is the function prototype */

#define QUE_NUM ???

unsigned char *SOURCE_POINTER; /* could be local or global depending upon
application and user. */

unsigned char STATUS; /* should be local */

STATUS = K_Que_Add_Top(QUE_NUM,SOURCE_POINTER);
```

**Passed**

QUE_NUM is the queue number, which identifies a particular queue, that was created with the *K_Que_Create* function. This can be from zero to one less than the maximum configured.

SOURCE_POINTER is a pointer that should contain the address where the source bytes are in memory.

**Returned**

The status of this operation will return one these status codes:

K_OK: good operation, function was successful

K_QUE_FULL: the operation was good and now the queue is full

K_ERROR: an error indicting the queue was full or that the queue number was out of range or not created.

☞ *Remember the pointer may point to anything. This may be done by casting, so actually longs could be passed, other pointers, etc. Remember that it is up to you to ensure that the queue number to this function is the queue in which they want to add this to, and that the queue had been created.*

*K_Que_Add_Top* example:

#define QUE1 1 /* this is the numeric identifier for queue 1, which has already been created with a each slot size being 6 bytes */

unsigned char *source_pointer = {"hello"); /* string that will be copied into slot */

```
void task2(void)
{
  unsigned char status; /* local */
  ...
  status = K_Que_Add_Top(QUE1,source_pointer);

  /* task 2 will add to queue 1.  The top slot, if available will
  be the slot that receives the source bytes.  The number of bytes
  copied, in this case is 6. */

  if (status != K_OK) /* test status */
    {
    /* see if status indicated queue is now full and the contents
    copied or that the queue was full before this call, and that
    the contents were not copied. */
    }
  ...
}
```

Comments

Once the source bytes are copied into the slot, you may reuse the source byte memory locations for other uses. It is highly recommended that the slot size be kept to a minimum, because of the time it takes to copy bytes from the source to destination. You may want to place just the pointers to the source bytes into the queue's slot, then when the slot is retrieved, the task will actually get the source bytes. The source bytes' memory locations may not be used for other purposes, until the slot pointer has been removed.

You should send messages, which just pass pointers and not the contents, if speed is of importance and the source bytes do not have to be copied.

## QUEUE MANAGER FUNCTION

*K_Que_Create*

Purpose: This function is used to create a circular queue. This queue may contain up to 32767 slots with each slot able to hold up to 255 bytes. It is recommended that the slot size be kept at a minimum, because the larger the slot, the longer it will take to transfer the source bytes into the slot. You must have the memory set aside for this queue, or use the memory functions to allocate enough memory. The queue address supplied to this function is never tested for any type of memory contention. The memory needed for this queue is the number of slots times the size of the slot.

**Called**

Before entering RTOS and tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Que_Create(sign_word16,byte,byte *,byte); /* this is the function prototype */

#define NUM_SLOTS ???

#define SIZE_SLOT ???

#define QUE_NUM ???

unsigned char QUE_NAME[NUM_SLOTS * SIZE_SLOT]; /* should be global */

unsigned char STATUS;

STATUS = K_Que_Create(NUM_SLOTS,SIZE_SLOT,QUE_NAME,QUE_NUM);
```

**Passed**

NUM_SLOTS is the number of slots that this particular queue will have and the maximum is 32767.

SIZE_SLOT is the number of bytes that each slot will hold (size of each slot) within this queue.

QUE_NAME is the beginning address where this queue will reside in memory.

QUE_NUM is the queue identification number that all queue functions will use in determining the queue's memory location.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful, queue created.

K_ERROR = Error: queue number out of range.

☞ *Remember that it is up to you to ensure enough memory for this queue exists. No memory contention is tested for.*

*K_Que_Create* example:

#define Q1_SLOTS 100 /* this queue will have 100 slots */

#define Q1_SIZE_SLOT 6 /* each slot size is 6 bytes in length */

#define QUE1 /* identifier that we want to work with queue number 1. */

unsigned char QUE1_BYTES[Q1_SLOTS * Q1_SIZE_SLOT]; /* memory */

```
void task2(void)
{
  unsigned char status; /* local */
  ...
  status =
  K_Que_Create(Q1_SLOTS,Q1_SIZE_SLOT,QUE1_BYTES,QUE1);

  /* task 2 will create a queue.  This queue will have 100 slots,
  and the size of each slot will be six bytes.  Any further
  reference to this queue will use the QUE1 identifier. */
  ...
}
```

## QUEUE MANAGER FUNCTION

*K_Que_Get_Bottom*

Purpose: This function removes the bottom most recently used slot of a queue, if one is available, and copies the slot bytes into the callers destination memory area as dictated by the address passed. The number of bytes copied is the size of the slot as indicated when this queue was created. The supplied queue address is not tested to ensure that it is a valid queue address.

**Called**

Before entering RTOS and tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Que_Get_Bottom(byte,void *); /* this is the function prototype */

#define QUE_NUM ???

unsigned char *DEST_POINTER; /* could be local or global depending upon
application and user. */

unsigned char STATUS; /* should be local */

STATUS = K_Que_Get_Bottom(QUE_NUM,DEST_POINTER);
```

**Passed**

QUE_NUM is the queue number, which identifies a particular queue was created with the *K_Que_Create* function. Can be from zero to one less than the maximum configured.

DEST_POINTER is a pointer that contains the address where the slot bytes will be copied to in memory.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: queue was empty, no slot available.

K_QUE_EMPTY = Warning: queue is now empty, the slot contents were transferred.

If the STATUS equals K_OK, the slot contents were copied to the destination's address. This is also true if the STATUS equaled K_QUE_EMPTY, indicating that now the queue is empty.

☞ *Remember the slots may contain anything, bytes, integers, pointers, etc. Once the contents from a slot are removed, you may cast those bytes into what you would like. It is up to you to ensure the queue number of this function is the same queue you added this to, and that it has been created.*

*K_Que_Get_Bottom* example:

#define QUE1 1 /* this is the numeric identifier for queue 1, which has already been created with a each slot size being 6 bytes */

unsigned char dest_array[6]; /* create a destination area that will be loaded with contents of the slot from a queue. Could be declared local or global */

```
void task2(void)
{
  unsigned char status; /* local */
  ...
  status = K_Que_Get_Bottom(QUE1,dest_array);

  /* task 2 will remove the bottom slot contents from queue1.
  The bottom slot, if available will have its contents
  transferred to the memory locations as specified by this
  function.  The number of bytes copied, in this case is six. */

  if (status != K_OK) /* test status */
    {
    /* see if status indicated queue is now empty and the contents
    copied or that the queue was empty before this call, and that
    the contents were not copied. */
    }
  ...
}
```

Comments

Once the bottom slot contents are removed, the pointer maintaining the bottom of this queue will be incremented to point to the new bottom slot of this queue. The queue count that represents the number of slots used will be reduced by one.

## QUEUE MANAGER FUNCTION

*K_Que_Get_Top*

Purpose: This function removes the top most recently used slot of a queue, if one is available, and copies the slot contents bytes into the caller's destination memory area as dictated by the address passed. The number of bytes copied is the size of the slot as indicated when this queue was created. The supplied queue address is not tested to ensure that it is a valid queue address.

**Called**

Before entering RTOS and tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Que_Get_Top(byte,void *); /* this is the function prototype */

#define QUE_NUM ???

unsigned char *DEST_POINTER; /* could be local or global depending upon
application and user. */

unsigned char STATUS; /* should be local */

STATUS = K_Que_Get_Top(QUE_NUM,DEST_POINTER);
```

**Passed**

QUE_NUM is the queue number, which identifies a particular queue, that was created with the *K_Que_Create* function. This can be from zero to one less than the maximum configured.

DEST_POINTER is a pointer that should contain the address where the slot bytes will be copied to in memory.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: queue was empty, no slot available.

K_QUE_EMPTY = Warning: queue is now empty, the slot contents were transferred.

If the STATUS equals K_OK, the slot contents were copied to the destination's address. This is also true if the STATUS equaled K_QUE_EMPTY, indicating that now the queue is empty.

☞ *Remember the slots may contain anything, bytes, integers, pointers, etc. Once the contents from a slot are removed, you may cast those bytes into what you would like. It is up to you to ensure the queue number of this function is the same queue you added this to, and that it has been created.*

*K_Que_Get_Top* example:

#define QUE1 1 /* this is the numeric identifier for queue 1, which has already been created with a each slot size being 6 bytes */

unsigned char dest_array[6]; /* create a destination area that will be loaded with contents of the slot from a queue. Could be declared local or global */

```
void task2(void)
{
  unsigned char status; /* local */
  ...
  status = K_Que_Get_Top(QUE1,dest_array);

  /* task 2 will remove the top slot contents from queue1.  The
  top slot, if available will have its contents transferred to
  the memory locations as specified by this function.  The number
  of bytes copied, in this case is six. */

  if (status != K_OK) /* test status */
    {
    /* see if status indicated queue is now empty and the contents
    copied or that the queue was empty before this call, and that
    the contents were not copied. */
    }
  ...
}
```

Comments

Once the top slot contents are removed, the pointer maintaining the top of this queue will be incremented to point to the new top slot of this queue. The queue count that represents the number of slots used will be reduced by one.

## QUEUE MANAGER FUNCTION

*K_Que_Reset*

Purpose: This function will free all used slots in a queue and reset all pointers accordingly. The queue number supplied must be created by the *K_Que_Create* function and is not tested to ensure this is so.

**Called**

Before entering RTOS and by tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Que_Reset*(byte); /* this is the function prototype */

#define QUE_NUM ???

unsigned char STATUS; /* should be local */

STATUS = *K_Que_Reset*(QUE_NUM);

**Passed**

QUE_NUM is the queue number, which identifies a particular queue that was created with the *K_Que_Create* function. This can be from zero to one less than the maximum configured.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: queue was empty, no slot available.

*K_Que_Reset* example:

#define QUE1 1 /* this is the numeric identifier for queue 1, which has already been created with a each slot size being 6 bytes */

```
void task2(void)
{
  unsigned char status; /* local */
  ...
  K_Que_Reset(QUE1);

  /* task 2 will reset this queue.  All used slots will now be
  flushed and the queue pointers will be reset to point to the
  first slot */
```

```
  ...
}
```

Comments

When a queue is reset the slot contents remain. Just the pointers that maintain this queue are reset to point to the first slot. So the slots may or may not contain garbage left over.

## UART FUNCTIONS

*K_Recv_Count*

Purpose: This function allows the task to get the count of the number of characters presently available in the UART receive buffer. No characters will be retrieved. Remember the receive count could change if another character was received after this function call.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */

unsigned short COUNT; /* this will specify the number of characters in receiver buffer. */

COUNT = *K_Recv_Count*();

**Passed**

Nothing is passed.

**Returned**

COUNT = the number of characters that are present in the UART receive buffer at the time of this function call.

If the COUNT value is zero, there were no characters in the UART receive buffer when this function was called. If the COUNT is non-zero, it is the number of characters currently in the receive buffer.

*K_Recv_Count* example:

```
void task1(void)
{
  count = K_Recv_Count();
  /* Go find out how many characters are currently in the UART
  receive buffer. */

  /* now possibly use a function to retrieve one or more
  characters from the receive buffer. */
  ...
}
```

## RESOURCE MANAGER FUNCTION

*K_Resource_Get*

Purpose: This function allows a task to request a particular resource, if the resource is free. The task will not be put into the suspended state if this resource is busy. If the resource is free at the time of this call, the caller will now own the resource.

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Resource_Get(byte); /* this is the function prototype */

#define RESOURCE_NUM ??? /* identifies resource number */

unsigned char STATUS; /* should be local */

STATUS = K_Resource_Get(RESOURCE_NUM);
```

**Passed**

RESOURCE_NUM is the resource number of the particular resource, that this task would like to own.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_RESOURCE_OWNED = Error: the resource is owned by another task.

K_ERROR = Error: the resource number was out of range.

If STATUS equals K_OK, then the task will own the resource. If STATUS equals K_RESOURCE_OWNED, or K_ERROR then the task does not own the resource.

☞ *It is up to you to ensure the task checks the return status to see if it owns the resource or not. If so, then the task may access this particular resource. If not, then the task should not access this resource, because another task is already using this resource. Contention (possibly corruption) could exist if both tasks try to manipulate this resource.*

*K_Resource_Get* example:

```
#define RESOURCE2 2 /* resource 2 identification number */
```

```
void task1(void)
{
  unsigned char status; /* local */
  ...
  status = K_Resource_Get(RESOURCE2);

  /* Task 1 will try to "own" resource 2.  If resource was free,
  then task 1 will now become the owner.  If the resource was
  busy, then task 1 will not be suspended (put into resource's
  queue) and be notified that the resource was already "owned"
  by another task */

  if (status == K_OK)
    {
    /* task 1 "owns" this resource.  Task 1 may now access this
    resource, knowing that only this task may manipulate this
    resource, and that no other task has access to this resource.
    When the task is finished with this resource, it must call
    the K_Resource_Release function to notify the resource this
    task will now give up ownership of this resource so another
    task may now become the owner. */
    ...
    K_Resource_Release(RESOURCE2); /* release resource */
    }
  ...
}
```

## RESOURCE MANAGER FUNCTION

*K_Resource_Release*

Purpose: The task that owns a resource must use this function to release ownership. The task may own more than one resource so it must supply the resource identification number. All the tasks in this resource wait queue are waiting, and will not run until the resource becomes free or the respective time period expires. When this function is called, the next task in this resource wait queue will automatically own the resource and be put into the READY state.

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Resource_Release(byte); /* this is the function prototype */

#define RESOURCE_NUM ??? /* identifies resource number */

unsigned char STATUS; /* should be local */

STATUS = K_Resource_Release(RESOURCE_NUM);
```

**Passed**

RESOURCE_NUM is the number of the particular resource this task would like to release.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_RESOURCE_NOT_OWNED = Error: this task does NOT own this resource.

K_ERROR = Error: the resource number is out of range.

If STATUS equals K_OK, then the task now has released the particular resource.

☞ *The task that owns the resource must make sure it calls this function before finishing its code. If the task is going to use the K_Task_Delete function, then the task must first release the resource. If the task does not release the resource, the tasks waiting in the resource's queue will be SUSPENDED forever or until their time period expires.*

The tasks must make sure they test the STATUS byte, to see if the resource is released or not. If the STATUS is returned with an error value, then there is a coding error within the application code. If the task successfully releases a resource and the next task in the waiting queue for this resource has a higher priority, then an immediate rescheduling will occur, with that task becoming the RUNNING task.

*K_Resource_Release* example:

#define RESOURCE2 2 /* resource 2 identification number */

```
void task2(void)
{
  unsigned char status; /* local */
  ...
  status = K_Resource_Release(RESOURCE2);

  /* Task 1 will now release the resource 2.  Task 1 should test
  the status byte, to see if this function was successful. */

  if (status != K_OK)
    {
    /* A definite application coding error */
    critical_error(); /* jump to critical handler, and see what
    is wrong with the application code.*/
    ...
    }
  ...
}
```

## RESOURCE MANAGER FUNCTION

*K_Resource_Wait*

Purpose: This function allows a task to obtain the requested resource if the resource is free, and if it is not, place the task that called this function in the suspended state. The time period will indicate to wait for so many system ticks or indefinitely. The resource wait queue works in the manner that when a resource is released by the current owner of the resource, the highest priority task waiting for this resource will become the new current owner.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Resource_Wait*(byte,word16); /* this is the function prototype */

#define RESOURCE_NUM ??? /* identifies resource number */

#define TIME_PERIOD ??? /* The time period to wait for */

unsigned char STATUS; /* should be local */

STATUS = *K_Resource_Wait*(RESOURCE_NUM,TIME_PERIOD);

**Passed**

RESOURCE_NUM is the number of the particular resource this task would like to release.

TIME_PERIOD is the number of system ticks to wait for this resource, ranging from zero to 65535. If the value equals zero, then the task will wait indefinitely for this resource.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_TIMEOUT = Error: the time period expired.

K_ERROR = Error: the resource number out of range.

If STATUS equals K_OK, then the task now owns the resource. If any other value is returned to STATUS, then the task does not own the resource.

☞ *Make sure the STATUS byte is tested to see if the task owns the resource or not. If the task owns the resource, it may now use this resource knowing this task is the only one that has access to this resource. The tasks must also make sure they release the resource when done, using the K_Resource_Release function.*

*K_Resource_Wait* example:

#define RESOURCE2 2 /* resource 2 identification number */

```
void task2(void)
{
  unsigned char status;

  status = K_Resource_Wait(RESOURCE1,100);
  /* pass resource 1 number to function, task 2 would like to
  become the owner of resource 1.  If resource 1 is not "owned"
  by another task and is free, then task 2 will become the "owner"
  indicated by value returned.  Task 2 will be suspended for 100
  TICKs if the resource is busy ("owned").  If the resource
  becomes free before the time period expires and this task is
  the next one in line (based on its priority) to receive
  ownership, then this task will automatically become the "owner"
  of this resource.  If the time period expires, then the task
  will be removed from the resource wait queue and be returned
  identifying that the time period expired and this task does NOT
  "own" the resource. */

  if (status == K_OK) /* see if good operation */
    {
    /* task 2 now OWNS resource 1, when done with resource, must
    call the K_Resource_Release function to release it */
    }
}
```

## SEMAPHORE MANAGER FUNCTION

*K_Semaphore_Create*

Purpose: This function sets up a semaphore's initial count. The first parameter is the semaphore number. This number ranges from zero to one less than the maximum number of semaphores declared in the configuration module. The second parameter is the initial value of the counter for this semaphore. This counter is of type unsigned short, meaning it can range from 0 to 65535.

**Called**

Before entering RTOS, tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Semaphore_Create(byte,word16); /* this is the function prototype */

#define SEM_NUM ??? /* Identifies the semaphore. */

#define SEM_COUNT ??? /* determines the initial setting of the semaphore counter. */

unsigned char STATUS; /* should be local */

STATUS =K_Semaphore_Create(SEM_NUM,SEM_COUNT);
```

**Passed**

SEM_NUM is the number of the semaphore this function will work with. Values range from 0 to 1 less than maximum number of semaphores declared.

SEM_COUNT is the initial numerical value of the semaphore counter. Values range from 0 65535.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: semaphore number out of range.

*K_Semaphore_Create* example:

```
#define SEM1 1 /* define semaphore 1 */

#define SEM1_CNT 0 /* define initial semaphore count to be zero. */
```

```
void task1(void)
{
  unsigned char status;
  status = K_Semaphore_Create(SEM1,SEM1_CNT);
  /* Set up semaphore 1 with an initial count of zero */

  if (status != K_OK) /* see if error, should not be */
    {
    /* If there was an error, take corrective action.  User should
    never get an error here, unless possibly the user has exceeded
    the maximum number of semaphores that was declared in the
    configuration module */
    }
}
```

## SEMAPHORE MANAGER FUNCTION

*K_Semaphore_Get*

Purpose: The *K_Semaphore_Get* function is used by tasks to request the use of a particular semaphore. The task will supply the semaphore number. This number ranges from zero to one less than the maximum number of semaphores declared in the configuration module. If the semaphore is not available the task will not be placed into the SUSPENDED state to wait for it.

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Semaphore_Get(byte); /* this is the function prototype */

#define SEM_NUM ??? /* identifies semaphore number */

unsigned char STATUS; /* should be local */

STATUS = K_Semaphore_Get(SEM_NUM);
```

**Passed**

SEM_NUM is the number of the semaphore this function will work with. Values range from 0 to 1 less than maximum number of semaphores declared.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_SEMAPHORE_NONE = Error: the semaphore is owned by another task.

K_ERROR = Error: the semaphore number was out of range.

If STATUS equals K_OK, then the task will own the semaphore. If STATUS equals K_SEMAPHORE_NONE, or K_ERROR then the task does not own the semaphore.

☞ *It is up to you to ensure the task checks the return status to see if it owns the semaphore or not. If so, then the task may access this particular semaphore. If not, then the task should not access this semaphore, because another task is already using this semaphore. Contention (possibly corruption) could exist if both tasks try to manipulate this semaphore.*

*K_Semaphore_Get* example:

```
#define SEM1 1

void task2(void)
{
  unsigned char status;

  status = K_Semaphore_Get(SEM1);
  /* pass semaphore 1 number to function, task 2 would like to
  become the owner of semaphore 1.  If semaphore 1 is not “owned”
  by another task, then task 2 will become the “owner” indicated
  by the value returned.  Task 2 will NOT be suspended if the
  semaphore is busy (“owned”), and the status returned will
  identify that the semaphore is busy. */
  if (status == K_OK) /* see if good operation */
    {
    /* task 2 now OWNS semaphore 1, when done with semaphore, must
    call the K_Semaphore_Post function to release it */
    }
}
```

## SEMAPHORE MANAGER FUNCTION

*K_Semaphore_Post*

Purpose: The task that owns a semaphore must use this function to release ownership. An interrupt could also use this function as a counter. A task may own more than one semaphore so it must supply the semaphore identification number. All the tasks in this semaphore's wait queue will not run until the semaphore becomes free or their respective time periods expire. When this function is called, the next task in this semaphore wait queue will automatically own the semaphore and be put into the READY state.

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Semaphore_Post(byte); /* this is the function prototype */

#define SEM_NUM ??? /* identifies semaphore number */

unsigned char STATUS; /* should be local */

STATUS = K_Semaphore_Post(SEM_NUM);
```

**Passed**

SEM_NUM is the number of the semaphore this function will work with. Values range from 0 to 1 less than maximum number of semaphores declared.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the semaphore number is out of range.

If STATUS equals K_OK, then the task now has released the particular semaphore.

☞ *The task that owns the semaphore must make sure it calls this function before finishing its code. If the task is going to use the K_Task_Delete function, then the task must first release the semaphore. If the task does not release the semaphore, the tasks waiting in the semaphore's queue will be SUSPENDED forever or until their time periods expire.*

The tasks must make sure they test the STATUS byte to see if the semaphore is released or not. If the STATUS is returned with an error value, then there is a coding error within the application code. If the task successfully releases a semaphore and the next task in the waiting queue for this semaphore has a higher priority, then an immediate rescheduling will occur, with that task becoming the RUNNING task.

*K_Semaphore_Post* example:

```
#define SEM1 1

void task2(void)
{
  unsigned char status;

  status = K_Semaphore_Get(SEM1);/* get semaphore 1 */

  /* application code here */

  /* task 2 owns semaphore 1, and is done with this semaphore.
  task 2 will now release semaphore 1 */

  status = K_Semaphore_Post(SEM1); /* task 2 is releasing
  semaphore 1 */

  if (status != K_OK) /* see if error, should not be */
    {
    /* If there was an error, take corrective action.  You should
    never get an error here, unless possibly when you called the
    K_Semaphore_Get or K_Semaphore_Wait functions, they did not
    test status to see if the task truly owned the semaphore */
    }
}
```

## SEMAPHORE MANAGER FUNCTION

*K_Semaphore_Reset*

Purpose: This function flushes out a particular semaphore. The semaphore's wait queue is emptied and the semaphore's counter is reset to the value given to the *K_Semaphore_Create* function when initializing the semaphore. Two parameters are needed by the *K_Semaphore_Reset* function.

The first parameter is the semaphore number. This number ranges from zero to one less than the maximum number of semaphores declared in the configuration module.

The second parameter is the flush mode. A flush mode of zero means the semaphore will not be flushed if a task owns the semaphore. A flush mode greater than zero means the semaphore will be flushed whether a task owns the semaphore or not.

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Semaphore_Reset(byte,byte); /* this is the function prototype */

#define SEM_NUM ??? /* identifies semaphore number */

#define FLUSH_MODE ??? /* specifies whether to flush if semaphore is owned */

unsigned char STATUS; /* should be local */

STATUS = K_Semaphore_Reset(SEM_NUM,FLUSH_MODE);
```

**Passed**

SEM_NUM is the number of the semaphore this function will work with. Values range from 0 to 1 less than maximum number of semaphores declared.

FLUSH_MODE determines whether to flush the semaphore if the semaphore is currently owned by a task or not. A value of zero will only flush the semaphore if it is not owned by a task. Any other value will always flush the semaphore.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the semaphore number out of range, or if the flush mode was zero, a task owns the semaphore.

*K_Semaphore_Reset* example:

```
#define SEM1 1

void task2(void)
{
  unsigned char status;

  status = K_Semaphore_Reset(SEM1,0);
  /*  task2 requests a flush of semaphore 1 with mode zero.  If
  no task owns this semaphore it will be flushed and K_OK
  returned. If a task owns semaphore 1 K_ERROR will be returned
  and semaphore 1 will not be flushed. */

  if (status == K_OK) /* see if good operation */
    {
    /* semaphore 1 wait queue is empty and semaphore 1 counter is
    reset to the value that was passed to the K_Semaphore_Create
    function. */
    }
}
```

## SEMAPHORE MANAGER FUNCTION

*K_Semaphore_Wait*

Purpose: This function allows a task to obtain the requested semaphore if the semaphore is free, and if it is not, place the task that called this function in the suspended state. The task specifies a time period to wait for the semaphore. If the time period expires the task will be placed into the RESUME state and notified that the time period expired and the semaphore is not available.

The tasks wait as first in, first out, meaning the first task waiting for the semaphore is the task that will own it when it is free.

If the task was suspended, when the semaphore becomes free and ownership is passed to this task, the task will automatically be put into the ready to RESUME state.

**Called**

Tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Semaphore_Wait(byte,word16); /* this is the function prototype */

#define SEM_NUM ??? /* identifies semaphore number */

#define TIME_PERIOD ??? /* The time period to wait for */

unsigned char STATUS; /* should be local */

STATUS = K_Semaphore_Wait(SEM_NUM,TIME_PERIOD);
```

**Passed**

SEM_NUM is the numerical value of the semaphore this function will work with. Values range from 0 to 1 less than maximum number of semaphores declared.

TIME_PERIOD is the number of system ticks to wait for this semaphore, ranging from zero to 65535. If the value equals zero, then the task will wait indefinitely for this semaphore.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_TIMEOUT = Error: the time period expired.

K_ERROR = Error: the semaphore number out of range.

If STATUS equals K_OK, then the task now owns the semaphore. If any other value is returned to STATUS, then the task does not own the semaphore.

☞ *It is up to you to ensure the task checks the return status to see if it owns the semaphore or not. If so, then the task may access this particular semaphore. If not, then the task should not access this semaphore, because another task is already using this semaphore. Contention (possibly corruption) could exist if both tasks try to manipulate this semaphore.*

*K_Semaphore_Wait* example:

```
#define SEM1 1

void task2(void)
{
  unsigned char status;

  status = K_Semaphore_Wait(SEM1,100);
  /* pass semaphore 1 number to function, task 2 would like to
  become the owner of semaphore 1.  If semaphore 1 is not “owned”
  by another task and is free, then task 2 will become the “owner”
  indicated by value returned.  Task 2 will be suspended for 100
  TICKs if the semaphore is busy (“owned”).  If the semaphore
  becomes free before the time period expires and this task is
  the next one in line to receive ownership, then this task will
  automatically become the “owner” of this semaphore.  If the
  time period expires, then the task will be removed from the
  semaphore wait queue and be returned identifying that the time
  period expired and this task does NOT “own” the semaphore. */

  if (status == K_OK) /* see if good operation */
    {
    /* task 2 now OWNS semaphore 1, when done with semaphore, must
    call the K_Semaphore_Post function to release it */
    }
}
```

## TASK MANAGER FUNCTION

*K_Task_Coop_Sched*

Purpose: This function performs cooperative rescheduling. This allows a task the ability to let a task of the same or lower priority become the current RUNNING task. Usually there is little need for this because of the preemptive nature of the CMX operating system. However, there may be times you would want a lower priority task to execute before this task would normally do so.

### Called

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_Task_Coop_Sched*(void); /* this is the function prototype */

*K_Task_Coop_Sched*();

### Passed

Nothing is passed. Only tasks can call this function.

### Returned

No status is returned.

*K_Task_Coop_Sched* example:

void task2(void); /* prototype task2 */

```
void task2(void)
{

  /* task's application code ... */
  K_Task_Coop_Sched();

  /* task 2 has decided to release control to the next same
  priority or lower priority task able to run. */
  ...
}
```

## TASK MANAGER FUNCTION

*K_Task_Create*

Purpose: This function creates a task and defines it to CMX. CMX then sets up the task control block for this task. The task is put into the IDLE state.

**Called**

Before entering RTOS and tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Task_Create(byte,byte *,K_FP,word16); /* this is the function prototype */

#define PRIORITY ???

unsigned char TASK_SLOT; /* should be global */

void TASK(void); /* prototype of task function */

#define STACK_SIZE ???

unsigned char STATUS; /* should be local */

STATUS = K_Task_Create(PRIORITY,&TASK_SLOT, TASK,STACK_SIZE);
```

**Passed**

PRIORITY is the priority for this task. The lower the number the higher the priority.

&TASK_SLOT is the address where CMX will put task slot number. Must be used for all references to this task.

TASK is the task address where it resides in code. When task begins execution, this is where CMX will vector to.

STACK_SIZE is the number of bytes set aside for this task stack area. You must make sure stack size is large enough for all the levels of nesting, and depth of one interrupt.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: no free task control block available.

If STATUS equals K_OK, then TASK_SLOT contains the task identification number assigned by CMX. This identification number must be used for all CMX function calls that deal with this task.

*K_Task_Create* example:

void task1(void); /* prototype task1 */

unsigned char task1_slot; /* global storage for task I.D. number */

```
void user_code(void)
{
  unsigned char status; /* local */
  ...
  status = K_Task_Create(10,&task1_slot,task1,128);

  /* create task 1, will have a priority of 10, task1_slot will
  contain the task's slot number for all further references to
  this task and have a stack size depth of 128 bytes. */

  if (status != K_OK)
    {
    /* take corrective action. */
    }
  ...
}

void task1(void)
{
  /* declare locals if need be */
  /* task 1 code... */
  K_Task_End();
  /*** TASK MUST HAVE THIS FUNCTION HERE IF IT WOULD NORMALLY
  TERMINATE BY HITTING ITS RIGHT END BRACE.  ***/
}
```

## TASK MANAGER FUNCTION

*K_Task_Create_Stack*

Purpose: The CMX *K_Task_Create_Stack* function is used to create a task. The main difference between this create task function and the *K_Task_Create* function, is that you pass the address of the task's stack address to the *K_Task_Create_Stack* function versus the size of the how much stack space is required when you use *K_Task_Create* function. When you use the *K_Task_Create* function and 'kill' a task, the stack space does not get reclaimed. Thus if you created and killed a fair number of tasks, then you would most likely run out of stack space. The new *K_Task_Create_Stack* function avoids this, for now you can use memory allocation functions to gain and then free stack space needed for many tasks. This function is very useful for embedded systems that may run a TCP/IP stack, where multiple clients may need servicing. This way you can allocate stack space when needed when creating a new task and when that task is no longer needed, you can then kill it and reclaim that stack space. You can create a task before entering the CMX operating system or dynamically while running under the CMX operating system. The creation tells CMX where the task's execution code resides in ROM, the stack starting address, the priority of this task, and the address of the slot number CMX assigns for this task.

The parameters you need to send are the following.

The priority for this task. Priority numbers may range from zero to 254. The priority tells CMX the order in which to run tasks when they become READY. The lower the number, the higher the priority. At rescheduling the highest priority task (lowest priority number) that is READY to run becomes the RUNNING task. If tasks have the same priority then it is determined by the order of creation, the first task created with the same priority as another task created later is given the first option to run, then the later one. The priority also is used by the CMX time slicing mechanism. Tasks will be time sliced, starting with tasks with the same or lower priority as the current task, if time slicing is enabled by calling the *K_OS_Slice_On* function. See the chapter about time slicing for more detailed information on time sliced tasks and how they work.

Another parameter is the address of an unsigned character to put the slot number CMX will assign to this task. The task slot number is used for ALL CMX function calls that require the task number. It is up to the user to make sure that they do not destroy or corrupt this slot number. If the task is removed, then the slot number is no longer valid. If another task is created after a task is removed, then the newly created task may have the "old" slot number, which the previously removed task had.

The next parameter is the address where the task's code will begin execution. This address is where CMX will start the task's code when the task switches from the READY state to the RUNNING state.

The last parameter supplied is the stack address for this task. Note that it is up to you to pass the address of memory, that will be large enough to be used by this task and to handle the number of nested function calls, locals, saving of registers, etc. Also on most processors the stack walks downward, so the user must ensure that they pass the stack address pointing to the 'top' of the stack space that they have freed up and not the bottom. Of course if you are using a processor where the stack grows upward, then you would pass the base address of the stack space. Since insufficient stack size is one of the most common causes of system crashes and corrupt memory, it is recommended you double the estimated size. As you become more knowledgeable and actually test your code, then the size may be reduced. See the chapter on stacks for more information on how to calculate the size of the stack for a particular task.

This is an example of the *K_Task_Create_Stack* function:

**Called**

Before entering RTOS and tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Task_Create_Stack(byte, byte *, K_FP, word16); /* this is the function
prototype */

#define PRIORITY ???

unsigned char TASK_SLOT; /* should be global */

void TASK(void); /* prototype of task function */

unsigned ??? STACK_ADDRESS; /* The beginning stack address for this task, must
be aligned to match the stack pointer alignment criteria. */

unsigned char STATUS; /* should be local */

STATUS =
K_Task_Create_Stack(PRIORITY,&TASK_SLOT,TASK,&STACK_ADDRESS);
```

**Passed**

PRIORITY is the priority for this task. The lower the number the higher the priority.

&TASK_SLOT is the address where CMX will put task slot number. Must be used for all references to this task.

TASK is the address where the task resides in code. When task begins execution, this is where CMX will vector to.

&STACK_ADDRESS is the task stack. You must make sure the stack memory size that you have allocated is large enough for all the levels of nesting, and the depth of one interrupt.

void task1(void); /* function prototype, show that task1 does not receive nor return parameters */

unsigned char task1_slot; /* create storage for CMX to return task1 slot number */

There are many ways to create a stack for a task, we will show you a few ways. Please ensure that you pass the top of the memory allocated to the task stack, if the stack pointer grows downward.

```
struct {
unsigned int task_stk[1000];
unsigned int dummy;
} task1_stack;

void main(void)
{
  unsigned char status; /* create a local status byte */

  status =
  K_Task_Create_Stack(5,&task1_slot,task1,task1_stack.dummy);

  /* call CMX function K_Task_Create_Stack with task1 having a
  priority of 5, the address for storage of task1 slot number,
  the address of task1, and finally a stack size of 128 bytes */
  if (status != K_OK) /* check status, make sure good function
  call */
    {
    error_handler(); /* go to error handler */
    }
}
```

Another way.

```
void *alloc;

void main(void)
{
  unsigned char status; /* create a local status byte */

  if ((alloc = malloc(1000)) != NULL)
    {
    status = K_Task_Create_Stack(5,&task1_slot,task1,((alloc) +
    998));
```

```
    /* call CMX function K_Task_Create_Stack with task1 having a
    priority of 5, the address for storage of task1 slot number,
    the address of task1, and finally a stack size of 128 bytes */

    if (status != K_OK) /* check status, make sure good function
    call */
      {
      error_handler(); /* go to error handler */
      }
    }
  else
    {
    Handle memory allocation error
    }
}
```

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: no free task control block available.

If STATUS equals K_OK, then TASK_SLOT contains the task identification number assigned by CMX. This identification number must be used for all CMX function calls that deal with this task.

CMX returns the status of the *K_Task_Create_Stack* function call indicating whether the call was successful or not. If the status is good then the slot number returned must be used from now on for any CMX function calls dealing with this task. Usually this slot number will be stored in external RAM. When a task is created, CMX puts the task into the IDLE state. This means that the task is loaded but will never RUN until started.

## TASK MANAGER FUNCTION

*K_Task_Delete*

Purpose: This function deletes a task and removes its task control block. No further reference to this task is allowed. The task must have been created prior to calling this function.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Task_Delete*(byte); /* this is the function prototype */

unsigned char TASK_SLOT; /* global, declared earlier */

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Delete*(TASK_SLOT);

**Passed**

TASK_SLOT is the name where the particular task's slot number resides.

**Returned**

STATUS returned is one of the following:

☞ *Only if the task is not removing itself*

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist or the task was in the WAIT state.

If STATUS equals K_OK, then the task has been removed successfully. If the task is removing itself, then an immediate task switch will occur.

*K_Task_Delete* example:

void task1(void); /* prototype task1 */

unsigned char task1_slot; /* global storage for task I.D. number */

```
void task2(void)
{
  unsigned char status; /* local */
```

```
  ...
  status = K_Task_Delete(task1_slot);

  /* remove task 1 */

  if (status != K_OK)
    {
    /* take corrective action. */
    }
  ...
}

void task1(void)
{
  /* declare locals if need be */
  /* task 1 code... */
}
```

## TASK MANAGER FUNCTION

*K_Task_End*

Purpose: This function allows a task to terminate itself either prematurely or at the end of its code. This function must be called by all tasks that would normally execute their right-end brace. Once called, all variables on the stack will be lost. Make sure other tasks are not waiting on this task. Never call this function if the task owns a resource. Only the calling task can terminate itself early.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_Task_End*(void); /* this is the function prototype */

*K_Task_End*();

**Passed**

Nothing is passed. Only tasks can call this function.

**Returned**

No status is returned.

☞ *An immediate rescheduling will occur when this function is called. If the task has additional trigger (K_Task_Start) requests, then the task will be put into the READY state, otherwise the task will become IDLE. Remember, all local variables will be lost. Other tasks may be waiting for this task to use a CMX function call to wake it. A task that owns a resource should never call this function before releasing the resource.*

*K_Task_End* example:

void task2(void); /* prototype task2 */

```
 void task2(void)
{
  /* task's application code ... */
  ...
}

void task2(void)
{
  /* task's application code ... */
  K_Task_End();
```

```
  /* task 2 has terminated itself early.  Normally this call
  should not be used except in critical error situations, if at
  all */
  ...
  K_Task_End(); /* ALL TASKS MUST HAVE this function, if normally
  would execute their right end brace. */
}
```

Comments

In example 1, task 2 has normally reached its right end brace and MUST execute the *K_Task_End* function to properly terminate itself. In example 2, task 2 has terminated itself early. All locals on stack will be lost. If a task must use this function, it is wise to make sure that no other tasks may be waiting on this task. If a task is waiting, the task about to terminate itself may either forcefully wake that task with the *K_Task_Wake_Force* function or send a message or flag to another task, so that task may wake the waiting task.

## TASK MANAGER FUNCTION

*K_Task_Lock*

Purpose: This function raises the privilege flag for the calling task. Once this flag is raised, the calling task owns all the CPU time and will not be preempted by any other tasks. The task calling this function should never call any other function that would permanently suspend this task. Only the task calling this function can lower the privilege flag using the *K_Task_Unlock* function.

**Called**

Tasks only.

#include <cxfuncs.h> /* has function prototype */
void *K_Task_Lock*(void); /* this is the function prototype */

*K_Task_Lock*();

**Passed**

Nothing is passed. Only tasks can call this function.

**Returned**

No status is returned.

Interrupts can still execute but the CMX timer task (which executes cyclic timers and handles the tasks timers) will not execute when the privilege flag is raised.

*K_Task_Lock* example:

void task2(void); /* prototype task2 */

```
 void task2(void)
{

  /* task's application code ... */
  K_Task_Lock();

  /* task 2 has raised the privilege flag, task 2 must lower the
  privilege flag before the task's code ends */
  ...
  K_Task_Unlock(); /* make sure task lowers privilege flag */
}

void task1(void)
{
  /* declare locals if need be */
```

```
  /* task 1 code... */
}
```

## TASK MANAGER FUNCTION

*K_Task_Name*

Purpose: This function gives you the ability to name a task, thus helping the user to know the name of the task, when working with CMX add on modules, such as CMXBug and/or CMXTracker. The function is called with the slot number of the task to be named and a pointer to the task's name that the user wants to name it. Note that the tasks name can be as long as the user would like, but only the first 12 characters of the task's name will be displayed by CMXBug and CMXTracker. This is an example of the *K_Task_Name* function:

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Task_Name*(byte, char *); /* this is the function prototype */

unsigned char TASK_SLOT; /* global, declared earlier */

char *TASK_NAME; /* The address of the user defined task name */

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Delete*(TASK_SLOT,TASK_NAME);

**Passed**

TASK_SLOT is the name where the particular task's slot number resides.

TASK_NAME is the address of where the user defined task name resides in memory. Note that the name may be any number of characters, but only the first 12 characters of the task's name will be shown by CMXBug and CMXTracker.

unsigned char task2_slot; /* defined earlier, contains task 2 slot number */

```
void task2(void)
{
  /* application code here */
  K_Task_Name(task2_slot,"TASK2"); /* Name task 2, utilized by
  CMXBug and CMXTracker when displaying tasks */
}
```

**Returned**

STATUS returned is one of the following:

☞ *Only if the task is not removing itself*

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist or the task was in the WAIT state.

If STATUS equals K_OK, then the task has been successfully named.

## TASK MANAGER FUNCTION

*K_Task_Priority*

Purpose: This function allows any task priority to be changed. The new priority is in effect immediately after this function call. The task must have been created prior to the call.

**Called**

Before entering RTOS and tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Task_Priority*(byte,byte); /* this is the function prototype */

unsigned char TASK_SLOT; /* global, declared earlier */

#define NEW_PRIORITY ???

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Priority*(TASK_SLOT,NEW_PRIORITY);

**Passed**

TASK_SLOT is the name where the particular task's slot number resides.

NEW_PRIORITY is the new priority for the task. The lower the number, the higher the priority The value may range from zero to 254.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task I.D. number does not exist.

If STATUS equals K_OK, then the task has the new priority for its priority. Note that the new priority will become effective immediately but will not cause a rescheduling.

*K_Task_Priority* example:

void task1(void); /* prototype task1 */

unsigned char task1_slot; /* global storage for task I.D. number */

```
void task2(void)
```

```
{
  unsigned char status; /* local */
  ...
  status = K_Task_Priority(task1_slot,20);
  /* go change task 1 priority */

  if (status != K_OK)
    {
    /* take corrective action. */
    }
  ...
}

void task1(void)
{
  /* declare locals if need be */
  /* task 1 code... */
}
```

Comments

Task 2 is changing the priority of task 1. This function will be used very sparingly, if at all. It is best to design the application code so task priorities need not be changed from their initial values. The new priority becomes effective immediately after this function call but does not cause a rescheduling.

## TASK MANAGER FUNCTION

*K_Task_Start*

Purpose: This function takes a task from the IDLE state into the READY state. If the task is in the READY state when this function is called, then this trigger will queue up and when the task finishes its code, the task will automatically become READY again. This function will queue a maximum of 255 triggers.

**Called**

Before entering RTOS, tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Task_Start(byte); /* this is the function prototype */

unsigned char TASK_SLOT; /* global, declared earlier */

unsigned char STATUS; /* should be local */

STATUS = K_Task_Start(TASK_SLOT);
```

**Passed**

TASK_SLOT is the name where the particular task's slot number (I.D. number) resides.

**Returned**

An immediate task switch will occur if the priority of task being triggered is higher than the current RUNNING task.

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist.

If STATUS equals K_OK, then the task is put into the READY state. If it is in the READY state already, then the trigger will be put into the task's trigger queue.

*K_Task_Start* example:

```
void task1(void); /* prototype task1 */
```

unsigned char task1_slot; /* global storage for task I.D. number */

```
void user_code(void)
{
  unsigned char status; /* local */
  ...
  status = K_Task_Start(task1_slot);

  /* go trigger (start) task 1 */

  if (status != K_OK)
    {
    /* take corrective action. */
    }
  ...
}

void task1(void)
{
  /* declare locals if need be */
  /* task 1 code... */
}
```

Comments

We are triggering (starting) task 1. This will put task 1 into the READY state. This means that the task is able to run. The task will become the running task when it is the highest priority task able to run. If task 1 had already been started, then the trigger will increment the start (trigger) byte of the task's control block for use when the task finishes its code.

## TASK MANAGER FUNCTION

*K_Task_Unlock*

Purpose: This function lowers the privilege flag for the calling task. This function can be called only by the task that used the *K_Task_Lock* function, which raises the privilege flag. Once the privilege is lowered, the CMX scheduler performs normally again.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
void *K_Task_Unlock*(void); /* this is the function prototype */

*K_Task_Unlock*();

**Passed**

Nothing is passed. Only tasks can call this function.

**Returned**

No status is returned.

*K_Task_Unlock* example:

void task2(void); /* prototype task2 */

```
 void task2(void)
{
  /* task's application code ... */
  K_Task_Lock();
  /* task 2 has raised the privilege flag, task 2 must lower the
  privilege flag before the task's code ends */
  ...
  K_Task_Unlock(); /* make sure task lowers privilege flag */
}

void task1(void)
{
  /* declare locals if need be */
  /* task 1 code... */
}
```

## TASK MANAGER FUNCTION

*K_Task_Wait*

Purpose: This function allows a task to suspend itself for a specified period of time or indefinitely. It is very useful for synchronization with time, interrupts, and other tasks.

**Called**

Tasks.

#include <cxfuncs.h> /* has function prototype */
byte *K_Task_Wait*(word16); /* this is the function prototype */

#define TIME_CNT ???

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Wait*(TIME_CNT);

**Passed**

TIME_CNT is number of system ticks that this task will suspend itself. If the value is zero then the task will be suspended indefinitely until the *K_Task_Wake* function is used. If the value is non-zero, then the task will be suspended for that number of system ticks. The *K_Task_Wake* function may be used prior to the time left, to wake this function and put it back into the READY state. The maximum value that TIME_CNT may be is 65535.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist.

K_TIMEOUT = Warning: the time specified has elapsed.

If STATUS equals K_OK, then the task has RESUMED execution because the *K_Task_Wake* or *K_Task_Wake_Force* function was used to wake this task. If STATUS equals K_TIMEOUT, then the time period specified has expired and this is why the task was awakened.

*K_Task_Wait* example:

void task1(void); /* prototype task1 */

unsigned char task1_slot; /* global storage for task I.D. number */

```
void task1(void)
{
  unsigned char status; /* local */
  ...
  status = K_Task_Wait(100);

  /* task 1 will suspend itself for 100 system ticks, unless the
  K_Task_Wake function is used to wake this task prior to time
  period elapsing */

  if (status == K_ERROR)
    {
    /* ERROR, take corrective action. */
    }
  if (status == K_TIMEOUT)
    {
    /* time out occurred.  In some cases this is what you want
    for this task at this time.  If the task was not expecting
    the time period to expire, then corrective action could take
    place here */
    }
  ...
  status = K_Task_Wait(0);

  /* task 1 will now suspend itself indefinitely until the
  K_Task_Wake function is used to wake this task. */

  if (status == K_ERROR)
    {
    /* ERROR, take corrective action. */
    }
  ...
}
```

## TASK MANAGER FUNCTION

*K_Task_Wake*

Purpose: This function wakes a task that had put itself in a suspended state. The task must have used the *K_Task_Wait* function to suspend itself. If the task that "wakes up" has a higher priority than the currently RUNNING task, then an immediate rescheduling (task switch) will occur.

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Task_Wake(byte); /* this is the function prototype */

unsigned char TASK_SLOT; /* global, declared earlier */

unsigned char STATUS; /* should be local */

STATUS = K_Task_Wake(TASK_SLOT);
```

**Passed**

TASK_SLOT is the name where the particular task's slot number resides.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist.

K_NOT_WAITING = Error: the task specified was not waiting.

If STATUS equals K_OK, then the task specified has been "awakened" and put into the READY state. If STATUS equals K_NOT_WAITING, then the task specified was not waiting for this function call.

*K_Task_Wake* example:

```
void task1(void); /* prototype task1 */

 unsigned char task1_slot; /* global storage for task I.D.  number */
```

```
void task2(void)
{
  unsigned char status; /* local */
  ...
  status = K_Task_Wake(task1_slot);
  /* task 2 is waking task 1 */

  if (status != K_OK)
    {
    /* ERROR, take corrective action. */
    }
  ...
}

void task1(void)
{
  /* task1 code ... */
  K_Task_Wait(0);
  /* more task 1 code ... */
}
```

Comments

The *K_Task_Wake* function is very useful when used with the *K_Task_Wait* function. Since interrupts may call this function, this allows you to write application code where interrupts can wake a task. Also other tasks may wake a task that has suspended itself either indefinitely or for a specific time period.

## TASK MANAGER FUNCTION

*K_Task_Wake_Force*

Purpose: This function is like the above *K_Task_Wake* function, with the exception that it will wake any task suspended by the following CMX functions: *K_Task_Wait*, *K_Mesg_Wait*, *K_Event_Wait* and *K_Mesg_Send_Wait*. This function will not wake a task that is waiting for a resource. If the awakened task has a higher priority than the currently RUNNING task, an immediate rescheduling (task switch) will occur.

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Task_Wake_Force(byte); /* this is the function prototype */

unsigned char TASK_SLOT; /* global, declared earlier */

unsigned char STATUS; /* should be local */

STATUS = K_Task_Wake_Force(TASK_SLOT);
```

**Passed**

TASK_SLOT is the name where the particular task's slot number resides.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist.

K_NOT_WAITING = Error: the task specified was not waiting.

If STATUS equals K_OK, then the task specified has been "awakened" and put into the READY state. If STATUS equals K_NOT_WAITING, then the task specified was not waiting for this function call.

*K_Task_Wake_Force* example:

```
void task1(void); /* prototype task1 */
```

unsigned char task1_slot; /* global storage for task I.D. number */

```
void task2(void)
{
  unsigned char status; /* local */
  ...
  status = K_Task_Wake_Force(task1_slot);
  /* task 2 is forcefully waking task 1 */
  /* Normally this function is not needed */

  if (status != K_OK)
    {
    /* ERROR, take corrective action. */
    }
  ...
}

void task1(void)
{
  /* task1 code ... */
  K_Task_Wait(0);
  /* more task 1 code ... */
}
```

## CYCLIC TIMERS MANAGER FUNCTION

*K_Timer_Create*

Purpose: This function sets up a cyclic timer's event function. When the cyclic timer executes because its programmed time period expires, it calls the *K_Event_Signal* function using the parameters programmed here. The *K_Event_Signal* function mode, the task slot number or priority, and the event to set are sent to this function.

**Called**

Before entering RTOS, tasks.

```
#include <cxfuncs.h> /* has function prototype */
byte K_Timer_Create(byte,byte,byte,word16); /* this is the function prototype */

#define  CYCLIC_NUM ??? /* Identifies the cyclic timer. */

#define MODE ??? /* determines the mode of the K_Event_Signal function. */

#define EVENT ??? /* determines which event to set. */

unsigned char TASK_PRI; /* works in conjunction of selected mode, specifies task or
priority or is not used. */

unsigned char STATUS; /* should be local */

STATUS = K_Timer_Create(CYCLIC_NUM,MODE,TASK_PRI,EVENT);
```

**Passed**

CYCLIC_NUM is the numerical value of which cyclic timer that this function will work with. Range in value from 0 to 1 less than maximum number of cyclic timers declared.

MODE is the mode in which the *K_Event_Signal* function will execute.

TASK_PRI is the tasks slot number, the priority, or unused that is determined by the MODE selected.

EVENT is the event bit that will be set when the cyclic timer's executes and calls the *K_Event_Signal* function.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: cyclic number out of range.

*K_Timer_Create* example:

#define CYCLIC1 1 /* identifies cyclic timer 1. */

#define CY1_EVENT 0x0002 /* this is the event bit that will be set. */

#define CYCLIC1_MODE 0 /* mode declaration for *K_Event_Signal* function */

unsigned char task2_slot; /* previous declared, global. */

```
void task1(void)
{
  unsigned char status; /* local */
  ...
  status =
  K_Timer_Create(CYCLIC1,CYCLIC1_MODE,task2_slot,CY1_EVENT);
  /* task 1 will now set up cyclic timer 1, so when its time
  period expires and executes, will automatically call the
  K_Event_Signal function, which will perform in the following
  manner.  The K_Event_Signal function will set task 2's event
  bit 1.  The setting of this event within task 2 will occur every
  time cyclic timer 1 has its time period expire. */

  if (status != K_OK)
    {
    /* take corrective action. */
    }
  ...
}

void task2(void)
{
  unsigned short events; /* local */
  events = K_Event_Wait(CY1_EVENT | MBOX1_EVENT,0);

  /* task 2 can now wait until the CY1_EVENT or MBOX1_EVENT to
  become set waking this task.  CY1_EVENT will become set when
  the associated cyclic timer, becomes running and its time
  period expires, calling the K_Event_Signal function which will
  set the CY1_EVENT. */
  if (events & CY1_EVENT)
    {
    /* process this event. */
    }
  if (events & MBOX1_EVENT)
    {
    /* get message from mailbox, process. */
    }
```

```
}
```

Comments

The *K_Timer_Create* function can be called at any time to change the set up parameters this function will use when it calls the *K_Event_Signal* function. Since cyclic timers are either "one-shots" or cyclic in nature, they are very useful in telling a task when a time period has elapsed.

## CYCLIC TIMERS MANAGER FUNCTION

*K_Timer_Cyclic*

Purpose: This function is used to change a cyclic timer's cycle time period. You supply the cyclic time period the system ticks that must expire before this cyclic timer executes. The current remaining time left before this cyclic timer executes is left untouched. The new cyclic time period takes effect when the cyclic timer executes and reloads its time counter with the cyclic time. The new time can range from zero to 65535 and if zero will make this cyclic timer a "one-shot" timer. Also, the cyclic timer is started if it was stopped when this function is called.

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Timer_Cyclic(byte,word16); /* this is the function prototype */

#define CYCLIC_NUM ??? /* Identifies the cyclic timer. */

#define NEW_CYCLIC_PERIOD ??? /* new cyclic period. */

unsigned char STATUS; /* should be local */

STATUS = K_Timer_Cyclic(CYCLIC_NUM,NEW_CYCLIC_PERIOD);
```

**Passed**

CYCLIC_NUM is the number of the cyclic timer that the caller wants to work with. This may range from 0 to 1 less than the maximum number of cyclic timers declared.

NEW_CYCLIC_PERIOD is the number of system ticks that the cyclic timer will use at the recycle time period. May range from 0 to 65535.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: cyclic timer number out of range.

If STATUS equals K_OK, the specified cyclic timer has its cyclic time changed and the timer started, if it was stopped.

*K_Timer_Cyclic* example:

#define CYCLIC1 1 /* This identifies cyclic timer 1. */

```
void task1(void)
{
  unsigned char status; /* local */
  ...
  status = K_Timer_Cyclic(CYCLIC1,100);

  /* task 1 will change cyclic timer 1 cyclic time to 100.  Also
  cyclic timer 1 will be started again, if it had been stopped
  by the K_Timer_Stop function or it was a "one-shot" cyclic
  timer and it had executed once. */

  if (status != K_OK)
    {
    /* take corrective action. */
    }
}
```

Comments

This new cyclic time period does not take effect until the time remaining for this cyclic timer expires. Then the new cyclic time will be used and put into the cyclic time counters. If the cyclic time period is zero, then the cyclic timer will be stopped since the timer is programmed to operate as a "one-shot" timer.

## CYCLIC TIMERS MANAGER FUNCTION

*K_Timer_Initial*

Purpose: This function is used to change a cyclic timer's initial time period. You supply the initial time period with the system ticks that must expire before this cyclic timer executes. The current remaining time left before this cyclic timer executes is immediately replaced with this time, overriding the "normal" time left. The cyclic time period that was programmed is left untouched. This function also restarts the cyclic timer, if it had been stopped.

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Timer_Initial(byte,word16); /* this is the function prototype */

#define CYCLIC_NUM ??? /* Identifies what cyclic timer. */

#define NEW_INITIAL_PERIOD ??? /* new initial period. */

unsigned char STATUS; /* should be local */

STATUS = K_Timer_Initial(CYCLIC_NUM,NEW_INITIAL_PERIOD);
```

**Passed**

CYCLIC_NUM is the number of the cyclic timer that the caller wants to work with. May range from zero to one less than maximum number of cyclic timers declared.

NEW_INITIAL_PERIOD is the number of system ticks that the cyclic timer will use a immediately in its time counters. This time is reduced at each system tick and when it becomes zero, the cyclic timer executes the *K_Event_Signal* function. May range from one to 65535.

**Returned**

A status will be returned to caller indicating K_OK: a successful operation, the specified cyclic timer has its initial time changed and also the cyclic timer is started if it was stopped or K_ERROR: the cyclic timer number was out of range.

This is very useful. A cyclic timer could continuously have this function called by a watchdog function. If the watchdog function did not execute or detected something wrong, the cyclic timer would eventually have its time counters decrement to zero, which could then notify a task to perform an orderly shutdown.

*K_Timer_Initial* example:

#define CYCLIC1 1 /* This identifies cyclic timer 1. */

```
void task1(void)
{
  unsigned char status; /* local */
  ...
  status = K_Timer_Initial(CYCLIC1,50);

  /* task 1 will change cyclic timer 1 initial time to 50.  Also
  cyclic timer 1 will be started again, if it had been stopped
  by the K_Timer_Stop function or it was a "one-shot" cyclic
  timer and it had executed once.  Lets say there was a count of
  3 remaining in the cyclic time counters, when this function was
  called.  The cyclic time counters would immediately be loaded
  with 50, overriding the previous value of 3. */

  if (status != K_OK)
    {
    /* take corrective action. */
    }
}
```

## CYCLIC TIMERS MANAGER FUNCTION

*K_Timer_Restart*

Purpose: This function restarts a cyclic timer whether it was stopped or is still running. The time remaining in the cyclic timers time counters and the cyclic time period is left untouched. This works in conjunction with the *K_Timer_Stop* function which stops cyclic timers. This function is very useful to restart a cyclic timer that had been stopped without changing any parameters of the cyclic timer.

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

#include <cxfuncs.h> /* has function prototype */
byte *K_Timer_Restart*(byte); /* this is the function prototype */

#define CYCLIC_NUM ??? /* Identifies the cyclic timer. */

unsigned char STATUS; /* should be local */

STATUS = *K_Timer_Restart*(CYCLIC_NUM);

**Passed**

CYCLIC_NUM is the number of the cyclic timer that the caller wants to work with. This may range from zero to one less than maximum number of cyclic timers declared.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: cyclic timer number out of range.

If STATUS equals K_OK, then the specified cyclic timer is restarted if it was stopped. If already started, this function has no effect.

*K_Timer_Restart* example:

#define CYCLIC1 1 /* This identifies cyclic timer 1. */

```
void task1(void)
{
```

```
  unsigned char status; /* local */
  ...
  status = K_Timer_Restart(CYCLIC1);
  /* task 1 will restart cyclic timer 1.  If cyclic timer 1 was
  currently started, then this function would have no effect.  If
  the cyclic timer was stopped, then the cyclic timer would be
  started, using the current time periods within it. */

  if (status != K_OK)
    {
    /* take corrective action. */
    }
}
```

## CYCLIC TIMERS MANAGER FUNCTION

*K_Timer_Start*

Purpose: This function is used start a cyclic timer which is assumed to have been set up by the *K_Timer_Create* function. Both the initial time and cyclic time periods are programmed into the cyclic timer. Also the cyclic timer is started. The initial time period is used first by the cyclic timer. After the initial time period expires, the cyclic timer will be used from then on.

**Called**

Before entering RTOS, tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

```
#include <cxfuncs.h> /* has function prototype */
byte K_Timer_Start(byte,word16,word16); /* this is the function prototype */

#define CYCLIC_NUM ??? /* Identifies what cyclic timer. */

#define INITIAL_PERIOD ??? /* initial time period. */

#define CYCLIC_PERIOD ??? /* cyclic time period. */

unsigned char STATUS; /* should be local */

STATUS = K_Timer_Start(CYCLIC_NUM,INITIAL_PERIOD,CYCLIC_PERIOD);
```

**Passed**

CYCLIC_NUM is the cyclic timer number the caller wants to work with. This may range from zero to one less than the maximum number of cyclic timers declared.

INITIAL_PERIOD is the number of system ticks the cyclic timer will use immediately in its time counters. This time is reduced at each system tick and when it becomes zero, the cyclic timer executes the *K_Event_Signal* function. This may range from one to 65535.

CYCLIC_PERIOD is the number of system ticks the cyclic timer will use at the recycle time period. This may range from zero to 65535.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: cyclic timer number out of range.

If STATUS equals K_OK, then the specified cyclic timer is started. The initial time period and cyclic period are used for this cyclic timer.

☞ *This function may be called more than once for a particular cyclic timer, to change both the cyclic timer's initial time and cyclic time.*

*K_Timer_Start* example:

#define CYCLIC1 1 /* This identifies cyclic timer 1. */

```
void task1(void)
{
  unsigned char status; /* local */
  ...
  status = K_Timer_Start(CYCLIC1,10,50);

  /* task 1 will start cyclic timer 1.  The initial time period
  will be 10 and cyclic time period will be 50.  This means that
  the cyclic time counter will be loaded with 10 initially.
  After this initial time period expires (because of 10 system
  ticks), the cyclic timer will call the K_Event_Signal
  automatically with the parameters programmed by the
  K_Timer_Create function.  Then the cyclic time counter will be
  loaded with the cyclic time period from then on and that time
  will be used by the time counters.  If the cyclic time period
  is 0, then the cyclic timer will execute only once "one-shot"
  mode and then stopped.  Otherwise the cyclic time will be used
  to determine the frequency in which this cyclic timer will
  execute. */

  if (status != K_OK)
    {
    /* take corrective action. */
    }
}
```

Comments

To fully understand how flexible and powerful cyclic timers are, and how many different ways they may be used, please read the Cyclic Timers Manager Functions chapter.

## CYCLIC TIMERS MANAGER FUNCTION

*K_Timer_Stop*

Purpose: This function stops a cyclic timer. If this function is called, the cyclic timer will remain stopped (not execute its code, the *K_Event_Signal* function) until started again by the *K_Timer_Start*, *K_Timer_Restart*, *K_Timer_Cyclic*, or *K_Timer_Initial* functions. The cyclic time periods currently in effect and their time amounts are left untouched by this function.

**Called**

Tasks and interrupts.

☞ *Interrupts can call this function, but only indirectly. See the Processor Specific section on how to do this.*

#include <cxfuncs.h> /* has function prototype */
byte *K_Timer_Stop*(byte); /* this is the function prototype */

#define CYCLIC_NUM ??? /* Identifies what cyclic timer. */

unsigned char STATUS; /* should be local */

STATUS = *K_Timer_Stop*(CYCLIC_NUM);

**Passed**

CYCLIC_NUM is the number of the cyclic timer that the caller wants to work with. This may range from zero to one less than maximum number of cyclic timers declared.

**Returned**

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: cyclic timer number out of range.

If STATUS equals K_OK, then the specified cyclic timer is stopped or if already stopped, then has no effect.

*K_Timer_Stop* example:

#define CYCLIC1 1 /* This identifies cyclic timer 1. */

```
void task1(void)
{
```

```
  unsigned char status; /* local */
  ...
  status = K_Timer_Stop(CYCLIC1);

  /* task 1 will now stop cyclic timer 1.  Cyclic timer 1 will
  not execute its code again ( the K_Event_Signal function),
  until this cyclic timer gets started again with one of the
  cyclic timer functions that start or restart a cyclic timer. */

  if (status != K_OK)
    {
    /* take corrective action. */
    }
  ...
}
```

## UART FUNCTIONS

*K_Update_Recv*

Purpose: If the receive buffer is not full, then the received character will be placed into the receive buffer and the receive head pointer will be incremented to point to the next storage location. The receiver count_in variable will also be incremented, indicating the number of bytes received. If a task is waiting on the receiver for a specified number of characters, then the *K_Update_Recv* function will test to see if the required number of characters are present now. If so, the task will automatically be put into the RESUME state, indicating the required number of characters are present.

## UART FUNCTIONS

*K_Update_Xmit*

Purpose: The *K_Update_Xmit* function is called by the transmitter interrupt. First the function will determine whether the transmitter empty flag is set. If so, then the transmit count_out variable will be tested for zero. If the count_out variable is non-zero, indicating that there are additional characters to transmit, the transmitter will automatically be loaded with the next buffer character, the count_out variable decremented and the tail pointer incremented so it is pointing to the next character. If the count_out variable is zero, it will test to see if there is a task waiting to transmit more characters. Remember, only one task should have exclusive ownership of the transmitter (through the use of the resource functions). If the task is waiting, then the task will automatically be put into the RESUME state, indicating it may now put more characters into the transmit buffer when it becomes the highest priority task able to run.

**Symbols**



**A**



**C**



**E**



**I**



**K**

**M**