CMX–RTOS

中文使用手册

郭 伟

2005年5月9日

CMX 多任务执行体

欢迎来到实时多任务操作系统(RTOS)的世界。CMX 为创建一个由高效的 C 语言或者汇编语言编写的、设计良好的多任务应用程序提供了必要的函数调用功能和操作系统平台。

一个处理器可以通过分时执行、在不同的任务之间互相切换，从而使人感觉到这些任务好象是在同时运行的，这就是所谓的多任务执行方式。CMX 是一个实时的多任务操作系统，它提供了一系列函数调用功能和一个实时多任务操作系统内核。它可以完成：

- ✧ 对任务执行进程的控制
- ✧ 发送和接收消息
- ✧ 处理事件
- ✧ 资源访问控制
- ✧ 用不同的方法进行定时器控制
- ✧ 内存管理功用
- ✧ 任务切换和中断

1、CMX 任务调度程序

CMX 操作系统能根据所接收到的输入事件实时地进行任务切换，CMX 操作系统的核心就是其任务调度程序。这个调度程序基于真正的抢占(preemption)机制，这意味着如果因为调用某个 CMX 函数使得具有更高优先级的任务变为可运行状态(就绪态)，那么当前任务和中断服务子程序就会产生瞬时的任务切换，执行目前优先级最高的任务。

除强行中断当前任务的抢占方式外，还有协作任务调度方式，也就是在必要的情况下，当前运行任务可以让另一个任务(优先级相同或较低)先运行。CMX 操作系统能够实现真正的分时操作。分时调度机制允许一个优先级较高的任务抢占当前正在运行的任务。当该高优先级的任务完成后，原先的分时任务重新获得 CPU 控制权，直到分配给它的时间片耗尽为止。

调度程序跟踪与任务相关的一些变量，调度程序何时执行任务切换取决于这些变量的状态。任务或中断可能会引发抢占式任务调度，它通知调度程序有一个优先级较高的任务需要运行。也有可能会发生决定*系统 Tick*中断，即所有与时间相关活动的基础。CMX 的时间 tick 函数将决定是否有任何与时间相关的活动需要处理，它也会通知调度程序调用 CMX 定时器任务，定时器任务将处理定时工作。

如果发生了任务抢占，正在运行的任务的上下文会被立即保存。调度程序为决定下一个运行的任务会调入所有与该任务相关的信息。如果一个任务被挂起，不管是被函数调用所阻塞还是被优先级较高的任务挂起，在执行完强行抢占的优先级较高的任务后，原先的任务都将重新恢复运行，就好象从没有被挂起过。原先的任务将恢复运行时，所有被保存的任务变量将被重新载入，所有寄存器也都恢复成原先在任务中的值。

如果你想获得更多的信息，请参看 CMX RTOS 用户手册调度程序这章，该章详细介绍了调度程序的工作机制。这一章解释了调度程序将用到的各种标记，以及调度程序对于中断驱动 cmx_tick 函数、任务、CMX 函数和其它中断的接口。

1.1 任务的中断

一个任务可能会因为进行了某个 CMX 函数调用而把自身挂起，迫使调度程序立即发生重新调度，而不考虑指定的系统时间间隔。当然把一个任务从挂起状态中脱离、或开始一个空闲态的任务的 CMX 函数调用也会发生强制性的任务重新调度，假如这个被挂起的任务或新任务比当前运行的任务优

先级高的话。

CMX 提供了处理单个或多重(嵌套)中断的能力。当发生中断嵌套时，因为中断可能会调用许多 CMX 函数，CMX 为此提供了必要的中断函数用来保存和恢复任务或中断的上下文。

当一个任务(或中断)的上下文被保存时，所有与该任务或中断有关的参数(所有的 CPU 寄存器，局部变量以及用于传递参数的变量)能够确保被准确地恢复，就好象这个任务或中断从来没有被挂起过。

1.2 任务的状态

一个任务可以处于几种可能的状态之一，但同一时刻一个任务只能有一种状态。这些状态是：空闲态、就绪态、运行态、等待(挂起)态、恢复态。

(1) 空闲态

一个用 cxtcre 函数创建但未由 cxtrig 函数激活的任务处于在空闲态。一个已经执行完毕并调用 cxtend 函数，而且在该任务的控制块中没有明显的触发设置，那么该任务也处于空闲态。处于空闲态的任务是不会运行的。

(2) 就绪态

处于就绪态的任务，调度程序已为其准备就绪，只是当前未运行而已。当发生任务调度操作时，调度程序会根据任务之间的级别关系来决定运行哪个任务。

(3) 运行态

正在执行的任务处于运行态，并占有 CPU 时间。任何时刻只能有一个任务处于运行态。

(4) 等待(挂起)态

一个因调用 CMX 函数而被挂起的任务处于等待(悬挂)状态。有许多函数调用会使任务挂起。处于挂起态的任务包括因等待下列情况之一而被挂起的任务：时间、事件、标志、消息、应答等。

(5) 恢复态

恢复态和就绪态是基本相同的，唯一的不同点是，它会告知调度程序任务的代码曾经执行过，但是任务没有完成。这说明有优先级较高的任务实现了抢占，并迫使正在运行的任务转变成恢复态，或者该任务因函数调用被挂起，并且现在已经从等待(挂起)态转变为恢复态。

1.3 任务的设计

每个编写的任务都是用来完成某些特定的功能，用户所要做的就是正确地创建任务，并使得任务以有序的方式来完成特定的工作。组织一个与其它任务相关联的任务，同时提供正确的中断处理程序、且该中断处理程序与其它中断和任务相关，这项工作可能是在编写实时多任务应用代码时最具挑战性的工作。

当着手一个新的工程及其相关代码时，用户必须告诉 CMX RTOS 系统一些特定的信息。这些信息在此只给出简略的介绍，后续章节将给出更详细的内容。刚开始的时候，在工程被详细、完整地定义之前，CMX 建议所选择的各种参数值大于其估计值，从而使得这些参数不必因为预留不足而经常改变。

在利用 CMX RTOS 开发平台进行实时多任务应用开发时，需要向系统提供的信息包括：

- ***任务的数目；***
- ***循环定时器的数目；***
- ***邮箱的数目；***
- ***消息的数目；***
- ***资源的数目；***
- ***所有任务的堆栈空间；***

- ***中断的堆栈空间；***
- ***队列的数目；***

CMX 多任务应用程序在进入操作系统之前要分配所有需要用到的内存空间，这使得在使用一些涉及内存操作的函数时可以节约 CPU 时间，CMX 认为这是创造最快的代码执行时间的最好办法。如果在操作系统动态分配内存时发现没有可用内存时，它会做些什么呢?任务必须知道如何处理这种情况并且不断测试 CMX 函数是否返回了表明无可用内存的值。

对于那些用汇编语言编写程序的用户, 参数的传递与编写 C 的程序代码时相同。你可以试编译一个调用 CMX 函数的 C 函数，然后检查由编译器产生的调用特定 CMX 函数的汇编代码。

在试图使用 CMX 系统之前，CMX 极力建议用户完整地阅读其用户手册以及其它任何可能附加的文件如 CMXREAD.DOC。你应该意识到学用 CMX 系统就好象学用其它任何新的软件一样具有一条学习曲线。你对它使用的越多，你就会对 CMX 软件理解的越深，而且能更好地把它加入到你的应用程序代码中。

1.4 CMX 返回状态字节值

标识	HEX	类别	含义
AOK	00	正确	CMX 函数调用正确完成
AERTIME	01	警告/错误	发生超时
AERNOWAIT	02	错误	任务没有在等待唤醒的请求
AERRESFUL	03	错误	资源队列满错误
AERRESOWNED	05	错误	资源已被占用
AERRSNOWN	06	错误	资源不属于调用任务
AERNOMSG	09	警告	没有检索到任何消息
AERQUEFUL	0A	警告	因加入队列项而使队列满
AERQUEEMPTY	0B	警告	因删去队列项而使队列空
AERR	FF	错误	普通错误，CMX 函数调用异常

有些函数，例如 cmxsget，如果不存在可用的消息可能会返回一个空指针，这是由于不能检索到任何消息而对调用者发出的出错警告。在某些情况下返回值为 0 表明了 CMX 函数所需要的内容不存在或者发生了超时。

1.5 CMX 数据类型

CMX 在其头文件 cxdefine.h 中预定义了下列数据类型，以下列出的只是其中的一部分。这些数据类型的定义随不同的处理器和不同公司的 C 语言编译器会有所不同，请仔细查看 cxdefine.h 文件以确定你当前所用的处理器和特定的 C 语言编译器。

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

1.6 函数的编排格式

对于每一个函数都有详细的函数功能介绍，以及函数的参数使用说明和函数的返回值，而且还有可能包括其它一些有用的信息。下面是一个典型的函数编排格式：

调用

在进入 RTOS 之前，任务，中断

[在进入 RTOS 之前] 就是指该函数可以在进入 CMX 操作系统之前被调用。[任务]指这个函数可以被一个任务调用。[中断]指这个函数可以被中断间接地调用。

cmx_init 函数必须在调用其它任何 CMX 函数之前被调用。

用户首先会看到调用该函数所需要的头文件。当然该函数所需要的参数将会在函数的头文件说明之后以示例的形式给以介绍，中间会有一些简单的注释，而且每个参数都给出了示例性的用法。以下是一个例子：

```
#include <cxfuncs.h>            /* 头文件 */
```

因为许多参数以常数的形式进行传送，所以我们只需要用#defines 来声明这些参数，而其后跟随的三个问号(即 ???)表明该值需要由用户来决定。当然我们还是需要一些描述性的文字来指明该参数所代表的意义。你也可以用任何一个你喜欢的文字串来命名该参数，当然 CMX 建议该文字串最好具有一些与参数有关的意义。

```
#define   PRIORITY    ???
unsigned char   TASK_SLOT;   /* 该参数必须是全局变量 */
void    TASK(void);              /* TASK 函数原型 */
#define   STACK_SIZE    ???
unsigned char     STATUS;       /* 该参数必须是局部变量 */

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

上述样例中各参数意义：

PRIORITY：是这个任务的优先级。此值越小则优先级越高；

&TASK_SLOT：是 CMX 存放该任务槽号变量的地址。该值会被所有涉及该任务的 CMX 函数所引用；

TASK：是该任务代码所驻留的起始地址。当任务开始运行时，这是 CMX 将要导向的地址；

STACK_SIZE：是任务堆栈区所需要的字节数。必须保证该堆栈空间的大小对于可能发生的多级嵌套以及可能发生的中断是足够的。

1.7 函数调用的返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERR = Error：因没有空闲的任务控制块而出错。

如果函数的返回状态 STATUS 为 AOK，那么 TASK_SLOT 将包含一个由 CMX 赋予的任务标识

号。该标识号可以被任何一个涉及该任务的 CMX 函数所调用。

下面为 RTOS 中创建一个任务的例子。

```
void  task1(void);                /* 函数说明，可见 task1 不接受也不返回任何参数。*/
unsigned char   task1_slot;       /* 为 CMX 返回 task1 的槽号创建存储空间。*/
void  main(void)
{
   unsigned char   status;              /* 声明局部字节变量 status  */
   status  =  cxtcre(5 , &task1_slot , task1 , 128);
   /* 以 5 的优先级、存放 task1 槽号空间的地址、task1 的起始地址以及 128 字节的堆栈空间调用
      CMX 函数。*/
   if (status != AOK)       /* 检查返回状态，以确认是否正确返回 */
      {
         error_handler();   /* 进行错误处理 */
      }
 }
```

在此或许会有一些附加的注释示例如下，这些注释会对该函数的使用有一定的帮助。

注解: *CMX 关于 cxtcre 函数调用的返回状态值表明调用是否成功。如果状态是 good，那么从现在开始，任何处理这个任务的 CMX 函数都需要用此返回的任务槽号来引用该任务。通常这个任务槽号将存在外部 RAM 中。当一个任务建立时，CMX 将这个任务置为空闲态。这意味着任务已经被载入，但它不会运行除非任务受到触发而开始。*

2、任务管理函数

任务管理函数是 CMX 函数库中的一部分，它为用户管理任务提供了必要的函数。任务管理函数如下：

cxtcre 、cxttrig 、cxtpri 、cxtwatm 、cxtwake 、cxtwakf 、cxprvr 、cxprvl 、cxsched 、cxtend 、cxtrmv

2.1 cxtcre 创建任务函数

CMX 的 cxtcre 函数通常被用来创建任务，可以在进入 CMX 操作系统前创建任务，也可以在 CMX 操作系统运行时动态地创建任务。为了使系统能以最快的速度运行，而且可以提前知道要使用什么样的任务，CMX 建议在进入实时多任务操作系统前创建所有的任务。

创建任务的过程将告诉 CMX 系统，任务的执行代码处于 ROM 中的什么位置，任务所需堆栈大小(每一个任务需要一个自己的堆栈)，任务的优先级，以及存放 CMX 系统指定的任务号的地址。

该函数所需要传送的参数如下：

任务的优先级

任务优先级的可能值域为 0 到 254。如果任务都处于就绪态，那么优先级将告诉 CMX 这些任务运行的顺序。优先级的值越小，优先级别就越高。当发生重新调度时，处于就绪态中的优先级最高(优先级的值最小)的任务将被允许进入运行态。如果任务的优先级相同，则其运行的顺序取决于任务被创建

的顺序。在相同优先级的的情况下，CMX 将优先运行先创建的任务，然后才是后创建的任务。优先级也被 CMX 的分时调度机制所使用。如果已经通过调用 enable_slice 函数启动了分时调度机制。与当前任务具有相同或较低优先级的任务将按照时间片的原则进行分时运行。关于分时间片的操作的详细内容请参看分时任务的有关章节，这里暂不作详细介绍。

存放该任务槽号变量空间的地址

CMX 用一个无符号的字符变量来存放分配给这个任务的槽号值。这个任务的槽号值将被任何要引用该任务的 CMX 函数所用。对用户来说必须保证不去破坏或修改这个任务的槽号值。如果一个任务被移去了，那么任务的槽号也不将失效。如果又有一个任务在该任务被移去之后被创建，那么新创建的任务可能会拥有与刚才被移去的任务相同的槽号值。

任务入口

任务入口就是该任务代码开始执行的地址。当任务从就绪态转变到运行态时，CMX 就是从这个地址开始执行任务代码。

任务的堆栈

由于堆栈空间的不足而引起系统内部发生存储冲突甚至系统被破坏是很普遍的，所以估计了堆栈的空间之后，CMX 建议开辟双倍的堆栈空间。等到有了更多的经验并确切的检测了任务代码所需的堆栈空间之后，方可减小堆栈空间，以提高存储器空间的利用效率。有关对于一个特定任务如何计算堆栈空间的大小的详细信息，请参看关于堆栈的章节。

cxtcre 函数应用举例

```
#include <cxfuncs.h>                    /* 头文件 */

#define    PRIORITY    ???
unsigned char    TASK_SLOT;         /* 该参数必须是全局变量 */
void    TASK(void);                  /* 函数说明 */
#define   STACK_SIZE    ???
unsigned char     STATUS;             /* 该参数必须是局部变量 */
STATUS  =  cxtcre(PRIORITY , &TASK_SLOT , TASK , STACK_SIZE);
```

其中 PRIORITY：是这个任务的优先级。此值越小则优先级越高。

&TASK_SLOT：是 CMX 存放该任务槽号变量的地址。该值会被所有涉及该任务的 CMX 函数所引用。

TASK：是该任务代码所驻留的起始地址。当任务开始运行时，这是 CMX 将要导向的地址。

STACK_SIZE：是任务堆栈区所需要的字节数。必须保证该堆栈空间的大小对于可能发生的多级嵌套以及可能发生的中断是足够的。

```
void  task1(void);            /* 函数说明，可见 task1 不接受也不返回任何参数。*/
unsigned char  task1_slot;  /* 为 CMX 返回 task1 的槽号创建存储空间。*/
void   main(void)
```

```
{
   unsigned char   status;        /* 声明局部字节变量 status  */
   status  =  cxtcre(5 , &task1_slot , task1 , 128);
   /* 以 5 的优先级、存放 task1 槽号空间的地址、task1 的起始地址以及 128 字节的堆栈空间调用
      CMX 函数。*/
   if (status != AOK)                  /* 检查返回状态，以确认是否正确返回 */
      {
         error_handler();              /* 进行错误处理 */
      }
 }
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERR = Error：因没有空闲的任务控制块而出错。

如果函数的返回状态 STATUS 为 AOK，那么 TASK_SLOT 将包含一个由 CMX 赋予的任务标识号。该标识号可以被任何一个涉及该任务的 CMX 函数所调用。

CMX 关于 cxtcre 函数调用的返回状态值表明调用是否成功。如果状态是 good，那么从现在开始，任何处理这个任务的 CMX 函数都需要用此返回的任务槽号来引用该任务。通常这个任务槽号将保存在外部 RAM 中。当一个任务建立时，CMX 将这个任务置为空闲态。这意味着任务已经被载入，但它不会运行除非任务受到触发而开始。

2.2 cxtcre1 堆栈空间可控制的任务创建函数

CMX 的 cxtcre1 函数也是用于创建一个任务，这个函数与 cxtcre 函数之间最大的不同之处在于需要把任务堆栈的起始地址作为参数传递给 cxtcrel 函数。这个地址所指向的堆栈空间大小，相当于在使用 cxtcre 函数创建任务时所需的堆栈空间大小。当用户使用 cxtcre 函数创建一个任务最后又删除该任务时，任务所占有的堆栈空间不会被收回。因此如果使用 cxtcre 函数创建了一定数量的任务并且最后又删除了相当数量的任务，那么用户很可能会用完所有的堆栈空间。

使用最新的 cxtcrel 函数来创建任务就可以避免这一情况的发生，对于任务所需的堆栈空间用户可以使用内存管理函数来申请或释放。这个函数对于需要运行一个 TCP/IP 堆栈的嵌入式系统来说是非常有用的，因为在此系统中可能有多个客户需要服务。通过这种方法，在创建一个新任务时，用户可以将其所必需的堆栈空间分配给新任务；而当不需要该任务的时候，用户可以删除它同时收回其所占有的堆栈空间。

用户可以在进入 CMX 操作系统前创建任务，也可以在 CMX 操作系统运行过程中动态地创建任务。任务的创建需要提供 CMX 任务可执行代码驻留在 ROM 中的位置，以及任务的堆栈起始地址，任务的优先级以及存放 CMX 分配给这个任务的槽号的地址。

以下为该函数所需要的参数说明：

第一个参数是任务的优先级。优先级的取值范围从 0 到 254。优先级决定了 CMX 就绪队列中任务运行的顺序。优先级的值越小，任务的优先级就越高。在任务重新调度过程中，就绪态任务中优先级最高(优先级的值最小)的任务将变为运行态的任务。如果就绪态任务的优先级相同，则它们的执行顺序取决于任务创建的先后顺序，对于优先级相同的任务来说先创建的任务先运行，然后是再执行下一个被创建的任务。优先级同样也被用于 CMX 分时调度机制中。如果通过调用 enable_slice 函数启动了

分时调度，所有的任务将被分时运行，所有与当前运行任务的优先级相同或相比来说较低的任务将参与分时调度。有关分时调度机制的详细说明请查阅分时调度一节。

第二个参数是一个用于存放 CMX 将赋予任务的槽号的无符号字符变量的地址，所有 CMX 函数都将使用这个任务槽号来引用该任务。用户不可以破坏这个地址中的内容，即任务槽号。如果任务已被删除，那么此任务的槽号将失效。如果一个任务被删除以后又创建了一个任务，那么这个新创建的任务很可能会取得刚被删除的任务的槽号值。

另一个参数是任务可执行代码的入口地址，这个地址指明了当任务从就绪态转换到运行态时，CMX 执行该任务代码的起始地址。

最后一个参数是这个任务的堆栈地址，用户必须把具有足够空间的堆栈地址作为参数传递给该函数，因为不仅仅这个任务会使用该堆栈，而且还要考虑到函数嵌套调用、局部变量所需的空间、寄存器的保存等等。在大多数处理器上堆栈是向下增长的，用户必须保证所传递的堆栈起始地址指向堆栈空间的顶部而不是指向底部；当然如果所使用的处理器的堆栈空间是向上增长的，那么情况将会相反，所传递的堆栈起始地址需要指向堆栈空间的底部。堆栈空间的不足是引起大多数系统崩溃的原因，所以 CMX 建议把所估计的堆栈空间加倍；当用户确认并实际测试了应用程序实际需要的堆栈空间之后，用户才可以缩小这个堆栈空间。对于一个特定的任务如何计算其所需的堆栈空间大小的具体细节请参阅堆栈一节。

以下是一个 cxtcre1 函数的应用实例：

调用

在进入 RTOS 之前，任务。

```
#include <cxfuncs.h>                                  /* 头文件 */
byte  cxtcre1(byte , byte * , CMX_FP , word16);  /* 函数声明 */
#define   PRIORITY   ???
unsigned char   TASK_SLOT;          /* 该变量必须是局部变量 */
void   TASK(void);                        /* 任务函数声明 */
unsigned  ??? STACK_ADDRESS;  /* 任务堆栈的起始地址，必须放在内存地址对齐的位置。
                                          STACK_ADDRESS 应为偶数*/
unsigned char   STATUS;               /* 该变量必须是局部变量 */

STATUS = cxtcre1(PRIORITY , &TASK_SLOT , TASK , &STACK_ADDRESS);
```

参数传递

PRIORITY：是这个任务的优先级。该值越小，则优先级越高。

&TASK_SLOT：是 CMX 将放置任务槽号的地址，所有对这个任务的引用都将使用这个槽号。

TASK 是任务代码驻留在内存的地址。当任务开始执行的时候，CMX 将导向这个起始地址。

&STACK_ADDRESS：是任务堆栈。要求所分配的堆栈空间足够大以能够满足各种函数嵌套以及中断嵌套的需求。

```
Void  task1(void);                /* 函数声明，可见任务 1 不接收也不返回任何参数 */
unsigned char    task1_slot;     /* 创建用于存放 CMX 返回的任务槽号的变量空间 */
```

为任务创建一个堆栈有许多方法，下面将介绍几个这种方法。如果堆栈是向下生长的，则必须保证所传递的堆栈起始地址指向堆栈的顶部。

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

void   main(void)
{
   unsigned char   status;        /* 创建局部变量 status */

   status  =  cxtcre1(5 , &task1_slot , task1 , task1_stack , dummy);
   /* 使用指定任务 1 的优先级为 5，存放任务 1 槽号的地址，任务 1 的入口地址以及 128 字节的
      堆栈的参数调用 CMX 的 cxtcre1 函数。*/
   if (status != AOK) /* 检查返回状态，以确认函数调用是否成功 。*/
      {
         error_handler(); /* 错误处理 */
       }
 }
```

另一种方法：

```
void  *alloc;

void   main(void)
{
   unsigned char    status; /* 创建局部变量 status */
   if ((alloc = malloc(1000)) != NULL)
      {
         status  =  cxtcre1(5，&task1_slot , task1 , ((alloc) + 998));
         /* 使用指定任务 1 的优先级为 5，存放任务 1 槽号的地址，任务 1 的入口地址以及 128 字
            节的堆栈的参数调用 CMX 的 cxtcre1 函数。*/
         if (status != AOK) /* 检查返回状态，以确认函数调用是否成功。 */
            {
               error_handler();   /* 错误处理 */
               }
       }
    else
       {
        处理内存分配错误。
       }
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。
AERR = Error：没有空闲的任务控制块而出错返回。

如果 STATUS 值为 AOK， 那么 TASK_SLOT 中包含了 CMX 所分配的任务标识号。这个标识号能够为所有的 CMX 函数引用这个任务时所使用。CMX 返回 cxtcrel 函数调用的状态，以表明调用是否成功。如果返回的状态值表明函数调用成功，那么从现在开始任何的 CMX 函数都必须使用返回的任务槽号来引用这个任务。通常这个任务槽号存放在外部 RAM 中。当一个任务被创建时，CMX 把任务置于空闲态，这意味着任务已被装载但不会运行直到被触发启动。

2.3 cxttrig 任务触发函数

cxtcre 函数将刚创建的任务置于空闲态，而 cxttrig 函数则负责触发并开始一个任务的运行。这个函数可以在任何时候被调用。cxttrig 函数把一个任务置于就绪态，并且当这个已经准备好运行的任务具有最高优先级时就允许它成为当前运行任务。只要任务处于空闲态，该函数调用会把它置为就绪态，虽然该任务并不立即就成为当前运行任务，但一旦任务脱离空闲态，则任何关于此任务的其它 cxttrig 调用将被放入触发队列，这种调用最多可达 255 次。

当任务执行完所有代码正常结束(调用了任务结束函数)并自动回到空闲态后，任何对任务的触发操作都会使任务自动地回到就绪态。任务的槽号会作为参数传送给该函数，以确定要触发哪一个任务。该函数可以在进入 CMX 操作系统之前调用，也可以在进入操作系统之后调用，还可以被中断调用。对于任何一个任务调用，cxttrig 函数触发的最大数量为 255，这些调用都将放入此任务的触发队列。

这是一个 Cxttrig 函数的实例：

调用

在进入 RTOS 前，任务，中断。
注：中断也可以调用该函数，但只能是间接调用。具体调用方法请参看处理器详细说明一章。

```
#include <cxfuncs.h>            /* 头文件 */
unsigned char  TASK_SLOT; /* 该参数必须是全局变量 */
unsigned char   STATUS;      /* 该参数必须是局部变量*/
STATUS  =  cxttrig(TASK_SLOT);
```

参数传递

TASK_SLOT ：是存放指定任务槽号的变量名。

```
unsigned char   task1_slot; /* 定义任务 1 的槽号变量 */
Void  task2(void)
{
   unsigned char   status;
   /* 触发任务 1，若此调用是第一次触发那么该任务状态转变为就绪态 */
   status  =  cxttrig(task1_slot);
   if (status != AOK) /* 检查函数调用返回状态，以确定是否正常返回 */
      {
         error_handler();   /* 进行错误处理 */
      }
```

}

返回值

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERR = Error：因任务标识号(槽号)不存在而出错

如果 STATUS 为 AOK，那么任务状态已转变为就绪态。如果任务早已经是就绪态，那么该触发操作将会被放入到任务的触发队列。

此返回的状态也指出了该 CMX 函数 cxttrig 的调用是否已正常完成。如果被启动的任务相对于当前处于运行态的任务具有更高的优先级(较小的优先级值)，那么会立即发生任务切换，这种任务的切换不是由系统滴答引发的正常任务调度。

2.4 cxtpri 改变任务优先级函数

Cxtpri 函数用于改变一个任务的优先级，可以在任何时候被调用。此函数有两个参数：所想要改变的任务的槽号和该任务新的优先级。在调用此函数之前要求该任务已被创建，否则此函数调用会出错返回。

当一个任务的优先级比其它任务的优先级低的时候，这是一个非常有用的函数，因为当它需要变成运行态时，它的优先级就需要变得高一些。另外用户还可以根据任务以及外部变量处理过程中的条件动态地改变任务的优先级。*在一个设计良好的系统，这个函数一般很少或根本不需要调用*。

如果一个任务的优先级比当前任务的优先级高，而且允许分时操作，那么该任务将会变为分时任务，分时操作将以新的优先级进行。有关分时操作以及如何使用的详细内容请参看时间分片的章节。

这是一个 cxtpri 函数的实例：

调用

在进入 RTOS 之前和任务

```
#include <cxfuncs.h>                    /* 头文件 */
unsigned char   TASK_SLOT;              /* 该变量必须是全局变量 */
#define    NEW_PRIORITY     ???
unsigned char    STATUS;                /* 该变量必须是局部变量 */
STATUS  =  cxtpri(TASK_SLOT , NEW_PRIORITY);
```

参数传递

TASK_SLOT：是存放指定任务槽号的变量名。
NEW_PRIORITY：是为该任务设定的新的优先级值，该值越小任务的优先级就越高。该值的取值范围是 0 到 254。

```
unsigned char   task1_slot;    /* 定义任务 1 的槽号变量 */

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

```
    status = cxtpri(task1_slot , 3);     /* 改变任务的优先级为 3 */
    if (status != AOK)                   /* 检查函数调用返回状态，以确定是否正常返回 */
       {
          error_handler();               /* 进行错误处理 */
       }
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERR = Error：因任务标识号(槽号)不存在而出错

如果函数的返回状态 STATUS 为 AOK，那么该任务的优先级已经成功地改变。注意该任务的新的优先级会立即发生作用，但不会引发任务的重新调度。

2.5 cxtwatm 任务挂起函数

cxtwatm 函数能使一个任务在指定的时间内或永远地被挂起。该函数使得一个任务可以与其他任务、中断或系统滴答函数同步，让一个任务在一段指定的时间内将自身挂起，并且当这段时间结束的时候将恢复自身，这种功能是非常有用的。这个函数也允许一个任务限时或无限期地等待一个中断来唤醒它，并通知它某个特定事件的发生。(事件管理程序可以使每一个任务处理多个事件，这部分内容将在以后详细介绍。)只有任务可以调用这个函数。

时间参数以系统滴答的倍数形式传递给这个函数。这个指定的时间值就是需要让任务处于挂起状态的系统滴答的数量。这个系统滴答数量的取值可以从 0 到 65535(十六进制为 0 到 FFFF)。若该值取为 0，那么将会使任务进入无限期的等待，直到出现 cxtwake 或者是 cxtwakf 函数的调用唤醒该任务。

当调用了 cxtwatm 函数之后，任务会自动在一段指定的时间(如果指定的时间非 0)内挂起，并且在该时间耗尽后任务又会进入就绪态以准备恢复运行。cxtwake 或者 cxtwakf 函数可以用来在指定挂起时间耗尽之前唤醒这个任务。cxtwatm 函数返回的状态字节将会指出任务被唤醒的方式。记住，其它任务和中断可以使用 cxtwake 或者 cxtwakf 函数来唤醒这个挂起的任务。

这个函数调用后任务挂起时间的精确度由系统滴答的精度决定。例如，假设现在系统滴答为 20 毫秒，如果某个任务需要 10 个系统滴答并且调用了 cxtwatm 函数，那么将会发生下列情况：指定的时间值随着每一次系统的滴答而不断减少，当这个时间值变为 0 的时候，任务将会自动进入就绪态以准备恢复运行。由此可见，根据任务在一次系统滴答期间何时调用 cxtwatm 函数，任务挂起时间大致为 180 到 200 个毫秒。

以下是 cxtwatm 函数的一个实例：

调用任务

```
#include <cxfuncs.h>              /* 头文件 */
#define   TIME_CNT   ???
unsigned char    STATUS;      /* 该变量必须是局部变量 */
STATUS  =  cxtwatm(TIME_CNT);
```

参数传递

TIME_CNT 是任务需要挂起自身的系统滴答的数量，如果该值取为 0，那么这个任务将会无限期地挂起，直到 cxtwake 函数来唤醒它。如果该值非 0，那么任务将在指定数量的系统滴答所规定的时间内挂起。cxtwake 函数可以在该挂起时间耗尽之前被调用来唤醒这个任务，并把这个任务放入就绪队列。TIME_CNT 可取的最大值为 65535。

```
Void  task2(void)
{
   unsigned char   status;
   status  =  cxtwatm(100);  /* 使任务 2 挂起等待 100 个系统滴答 */
   if (status != AOK)         /* 检查返回状态，以确认任务是否在挂起时间耗尽前被唤醒 */
      {
        /* 如果挂起时间已经耗尽那么采取可行的正确操作，除非任务想要与系统滴答函数进行同
           步 */
      }
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERR = Error：因任务标识号(槽号)不存在而出错
AERTIME = Warning：指定的时间已经耗尽

如果函数的返回状态 STATUS 为 AOK，那么该任务是被 cxtwake 或 cxtwakf 函数唤醒的，并且已经恢复执行。如果函数的返回状态 STATUS 为 AERTIME，那么说明指定的挂起时间已经耗尽而使任务被唤醒。

：这个函数不会自动测试函数的调用者是否是任务，因此必须保证只有任务才可以调用这个函数。

该函数的返回状态将会指出任务是因为指定挂起时间(TIME_CNT 非 0)耗尽而被唤醒，还是被 cxtwake 或 cxtwakf 函数唤醒的。

2.6 cxtwake 唤醒任务函数

cxtwake 函数的功能是唤醒处于挂起态的任务。任务和中断都可以调用这个函数。函数的调用者需要把任务的槽号传递给 cxtwake 函数，然后此函数根据所提供的任务槽号负责把该任务唤醒、并把它放入就绪态准备恢复运行。

注意被唤醒的任务因为其优先级的原因可能不会立即成为运行态的任务。如果此任务的优先级比当前处于运行态的任务优先级要高，那么会立即发生任务的切换而不考虑系统滴答函数的因素。

以下是 cxtwake 函数的一个实例：

调用

任务，中断。

注：中断也可以调用该函数但只能是间接调用。具体调用方法请参看处理器详细说明一章。

```
#include <cxfuncs.h>              /* 头文件 */
unsigned char   TASK_SLOT; /* 该变量必须是全局变量。*/
unsigned char   STATUS;       /* 该变量必须是局部变量。*/
STATUS  =  cxtwake(TASK_SLOT);
```

参数传递

TASK_SLOT：是存放指定任务槽号的变量名。

```
unsigned char   task1_slot; /* 定义任务 1 的槽号变量。*/
void  task2(void)
{
   unsigned char   status;
   status  =  cxtwake(task1_slot); /* 唤醒任务 1。 */
   if (status  !=  AOK) /* 检查返回状态以确定任务是否在等待。*/
      {
         /* 如果任务 1 未曾处于等待(挂起)状态，采取相应的操作。 */
      }
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERR = Error：因任务标识号(槽号)不存在而出错。
AERNOWAIT = Error：指定的任务未处于等待状态(挂起态)。
如果函数的返回状态 STATUS 为 AOK，那么指定的任务已经被唤醒并且已经成为就绪态的任务。如果函数的返回状态 STATUS 为 AERNOWAIT，那么表明指定的任务在该函数调用的时候未处于挂起态。

2.7 cxtwakf 强制唤任务函数

cxtwakf 函数和 cxtwake 函数特别相似，它被用来强制唤醒因为各种因素而被挂起的任务。如果想提前结束任务的挂起状态，那么这个函数可以胜任这项工作。cxtwakf 函数可以被任务或中断调用，而且除非特定环境需要以及紧急情况，一般不要轻易使用它。

cxtwakf 函数和 cxtwake 函数的不同点在于，后者只能唤醒因等待时间而挂起的任务，而前者不论任务是因为什么而挂起，都能被唤醒。

对于因调用 cxtwatm、cxewatm、cxmswatm、cxrsrsv、cxmssenw 函数而挂起的任务，cxtwakf 函数都能将它唤醒。因为任务一旦调用了这些函数的其中任何一个都会被挂起，直到发生任务所等待的相应函数调用或者超时(time out)为止该任务才能继续运行。

这是一个关于 cxtwakf 函数的例子：

调用

任务，中断。

注：中断也可以调用该函数但只能是间接调用。具体调用方法请参看处理器详细说明一章。

```
#include <cxfuncs.h> /* 头文件 */
unsigned char   TASK_SLOT; /* 该变量必须是全局变量 */
unsigned char   STATUS; /* 该变量必须是局部变量 */
STATUS  =  cxtwakf(TASK_SLOT);
```

参数传递

TASK_SLOT：是存放指定任务槽号值的变量名。

```
unsigned char   task1_slot;  /* 定义任务 1 的槽号变量 */
void  task2(void)
{
   unsigned char   status;
   status  =  cxtwakf(task1_slot);  /* 强制唤醒任务 1 */
   if (status != AOK)  /* 检查函数状态，以确定函数是否在等待。 */
      {
        /* 如果任务 1 未处于挂起态，采取相应的操作。 */
      }
 }
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERR = Error：因任务标识号(槽号)不存在而出错。

AERNOWAIT = Error：指定的任务未处于等待状态(挂起态)。

如果函数的返回状态 STATUS 为 AOK，那么指定的任务已经被唤醒并且已经成为就绪态的任务。如果函数的返回状态 STATUS 为 AERNOWAIT，那么表明指定的任务在该函数调用的时候未处于挂起态。

：只有在紧急情况下，该函数才能被使用 。一个设计优良的系统基本上不需要使用这个函数。

2.8 cxprvr 设置特权标志函数

cxprvr 函数的功能是非常强大的，调用此函数时需特别小心，因为此函数会为调用它的任务设置特权标志。当一个任务拥有特权标志时，任务会占有所有的 CPU 时间，此时即使有更高优先级的任务可以运行，也不会发生任务切换。

拥有特权标志的任务决不会挂起，因为系统不允许其他任何任务运行。只有调用了 cxprvr 函数的任务才是可以取消该特权标志的唯一任务。当特权标志出现后，中断照常被处理，系统滴答函数将照常执行，但 CMX 定时器任务(它会减少任务定时器的定时以及执行其它与时间相关的操作)的执行将被推迟，直到取消了特权标志。

应该意识到当特权标志被设置之后，所有任务定时器的定时都不会被减少和任务超时(timeout)也

将不会被测试，直到定时器任务运行。在这种情况下，中断仍旧会被正常地处理，并且系统滴答函数也将照常发生，但是如果中断管道中有需要执行内容的话，这些内容是不会被执行的。在这种情况下最显著的一点是当 CMX 调度程序正常触发一个较高优先级的任务进入就绪态时，这个高优先级的任务是不可能进入运行态的。cxprvr 函数一般很少使用或者不用，而且这个函数只有任务方可调用。

这是一个关于 cxprvr 函数的例子。

调用任务

```
#include <cxfuncs.h> /* 头文件 */
void   cxprvr(void);   /* 函数说明 */
cxprvr();
```

参数传递

不需要传递任何参数。只有任务才能调用这个函数。

```
Void   task2(void)
{
   cxprvr(); /* 任务 2 设置特权标志，它将占有所有的 CPU 时间，并且不允许发生任务调度。*/
   /* 执行任务代码，决不能将自身挂起 。*/
   cxprvl(); /* 取消特权标志 */
}
```

返回值

不返回任何状态值。

当设置了特权标志时，CMX 的定时器任务(该任务负责执行循环定时器和处理任务定时器)将不会执行。

：这个函数不会自动测试调用者是否是任务，因此必须确保只有任务才能调用它。

2.9 cxprvl 取消特权标志函数

这个函数的作用就是取消特权标志。只有调用 cxprvr 函数建立特权标志的任务，才可以调用这个函数。一旦特权标志被取消，CMX 调度程序的操作将恢复正常。

这是一个关于 cxprvl 函数的例子：

调用任务

```
#include <cxfuncs.h> /* 头文件 */
void  cxprvl(void);     /* 函数声明 */
cxprvl();
```

参数传递

不需要传递任何参数。只有任务才能调用这个函数。

```
void   task2(void)
{
   cxprvr();        /* 设置特权标志 */
   /* 执行应用程序代码 */
   cxprvl();        /* 任务 2 取消特权标志，从而允许发生正常的调度 。*/
 }
```

返回值

不返回任何状态值。

警告：这个函数不会自动测试调用者是否是任务，因此必须确保只有任务才能调用它。

2.10 cxsched 协作调度函数

使用 cxsched 函数可以实现协作式调度(cooperative schedule)。一般只有当一个具有更高优先级的任务处于就绪态时，CMX 才会发生调度。然而当任务调用了这函数，CMX 将不管任务的优先级而直接调度下一个任务。

这使得任务允许让其它具有相同或较低优先级的任务变为运行任务(如果这个任务已处于就绪态)。调用这个函数任务的状态将立即从运行态转变为恢复态。因为此实时操作系统是建立在抢占式调度的基础上的(意味着当前运行任务是可运行的优先级最高的任务)，所以在大多数情况下不要使用这个函数。任务可以通过调用 cxtwatm 函数等待一个指定时间，这也会使得与此任务具有相同优先级的任务变为运行任务。只有任务才可以调用这个函数。

这是一个关于 cxsched 函数的例子：

调用任务

```
#include <cxfuncs.h>     /* 头文件 */
void   cxsched(void);     /* 函数声明 */
cxsched();
```

参数传递

不需要传递任何参数。只有任务才能调用这个函数。

```
void   task2(void)
{
   /* 执行应用程序代码 */
   cxsched();   /*现在允许发生重新调度，而不必等待正常的抢占调度机制来实现一次正常的度。
                  注意：当一个处于就绪态队列中最高优先级的任务准备运行时，被完整保存的任
                  务的上下文将被恢复。*/
}
```

返回值

不返回任何状态值。

2.11 cxtend 任务结束函数

cxtend 函数允许一个任务提前结束或者在执行完代码时结束。cxtend 函数必须被所有能够正常结束的任务在结束之前调用。一旦任务在结束前调用了这个函数，那么这个任务的所有变量以及在堆栈上的操作都将被忽略。这个函数对于任务退出任何严重的甚至是不可恢复的错误是非常有用的。在这种情况下，任务在调用 cxtend 函数之前最好先给其它任务发送一个消息，以说明哪一个任务以及错误的类型等。

当任务调用了 cxtend 函数后，任务将自动终止同时把栈指针和程序指针重新设回到任务的开始处(变成空闲态)。只要任务被重新启动或者在该任务的触发队列非空(有触发请求)，任务依然还能重新执行。只有任务可以调用这个函数。当任务的触发队列中有触发者时，任务会自动重新启动并进入就绪态，并且当它变为就绪态中优先级最高的任务时，就会立即进入运行态。

如果任务使用了无限循环而永远不会结束运行，那么在该任务的代码中可以不使用 cxtend 函数。

如果某任务在调用 cxtend 函数之前从邮箱收到了消息，并且发送任务使用了 cxmssenw 函数(意味着它将等待接收任务的应答)，那么在调用 cxtend 函数前必须调用 cxmsack 函数向发送任务发出一个应答(这将会唤醒消息发送任务)。

这是一个关于 cxtend 函数的例子：

调用任务

```
#include <cxfuncs.h>     /* 头文件 */
void  cxtend(void);      /* 函数声明 */
cxtend();
```

参数传递

不需要传递任何参数。只有任务才能调用这个函数。

```
Void  task2(void)
{
   ……   /* 执行应用程序代码  */
   cxtend(); /* 这必须是在任务结束括号之前的最后一个 C 语言的语句 */
 }
```

或者可以这么使用：

```
 void   task2(void)
 {
   ……   /* 以下是应用程序代码 */
   if  (???) /* 严重的甚至是不可恢复的错误 */
      {
         cxtend(); /* 因为严重的甚至是不可恢复的错误，结束任务 2 。*/
      }
   …… /* 执行其余的代码 */
   cxtend();  /* 函数正常退出 */
```

```
}
```

或者还可以这么使用，如果一个任务不可能到达它的结束括号，示例如下：

```
void   task2(void)
{
   while(1)
        {
          …… /* 执行应用程序代码 */
        }
   /* 这里不再需要 cxtend 函数 */
}
```

返回值

不返回任何状态值。

注：当这个函数被调用时，将会立即发生任务的重新调度(甚至有可能这个任务重新从头开始执行)。如果这个任务有任何的触发请求，那么这个任务将被置于就绪态，否则这个任务将变为空闲态。不要忘了，这个任务所有的局部变量都已丢失，其余的任务可能在等待这个任务使用 CMX 函数去唤醒它们。一个拥有资源的任务在其释放资源前不应该调用这个函数。

警告：*所有的任务在结束之前(执行到任务结束括号处)必须调用这个函数。如果任务因为某种原因而不可能到达它的结束括号，那么此任务可以不使用这个函数。CMX 建议所有的任务都应该有这个函数，不管任务是否执行它。*

2.12 cxtrmv 删除任务函数

cxtrmv 函数可以把任务从任务控制块队列里永久地删除。调用该函数的任务需要把希望删除的任务槽号作为参数(如果需要，它也可以可能传递自己的槽号)传递给 cxtrmv 函数。

如果需要被删除的任务处于挂起态，那么删除操作失败，函数调用将会返回错误状态，否则指定任务将被真正地删除。任务被删除以后将不可能再运行，任何指向该任务的操作都将导致失败。

如果调用该函数的任务删除的任务正是它本身，那么该任务会被立即删去并发生任务切换。值得注意的是，当任务被删除时，所有处于因等待该任务而挂起的其它任务并不会被告知该任务已删除，它们将永远保持挂起态。比如：如果任务 1 通过 cxmssenw 函数向任务 2 发送了一个消息，任务 2 在调用 cxmsack 函数发送应答(同时唤醒任务 1)前就被任务 3 所删除，任务 1 将会因未收到应答而会处于死等。当然还可以用 cxtwakf 函数强制唤醒任务 1。

这是一个关于 cxtrmv 函数的例子：

调用任务

```
#include <cxfuncs.h>              /* 头文件 */
byte  cxtrmv(byte);               /* 函数声明 */
unsigned char   TASK_SLOT;    /* 该变量必须是全局变量 */
unsigned char   STATUS;         /* 该变量必须是局部变量 */
STATUS  =  cxtrmv(TASK_SLOT);
```

参数传递

TASK_SLOT：是存放指定任务槽号值的变量名。

```
unsigned char   task2_slot;    /* 定义任务 2 的槽号变量 */
void   task2(void)
{
   /* 执行应用程序代码 */
   cxtrmv(task2_slot);    /* 删除任务 2，以后任何关于任务 2 的引用都将失败。 */
}
```

返回值

不包含任务删除自身的情况。

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERR = Error：因任务标识号(槽号)不存在或者任务此时处于挂起态而出错

如果状态为 AOK, 那么任务已被成功地删除。如果任务删除的是自身，那么将会立即发生任务切换。

警告：*确保将被删除的任务不拥有任何系统资源。这个函数不会检测是否是任务调用了它，因此必须确保只有任务才可以调用这个函数。*

3、事件管理函数

事件管理函数是 CMX 库中的一部分，它为控制事件提供必需的函数。事件管理函数如下面所列：

cxewatm；

cxesig；

cxerst；

CMX 事件函数是非常先进的，它功能强大而且容易使用。每一个任务都可以有 16 个事件。每个事件被定义为一位 ，该位可以被设置或清除。某一位被设置为 1，那么表示该位对应的事件发生。任务可以等待事件位的任意组合，即同时等待多个事件位。当任务正在等待的任何事件发生时，任务被唤醒并得知哪个事件发生了。任务可以指定一个时间值来等待，或死等这些事件。

在等待事件之前，任务可以设置为自动把要等待的事件位清零(意味着任务将挂起)，或者不清除。而当事件发生或任务被唤醒时，任务也可以设置为自动把事件位清零，或者不清除。任务也可在任何时候调用使事件清除(置 0)的函数。

事件的处理非常灵活，它可以被任务、中断、循环定时器及邮箱处理(操作)。你可用七种不同的方法(即模式，对应值为 0-6)设置一事件位：

1、 设置指定任务；

2、 设置优先级最高的任务；

3、 设置正在等待该事件的优先级最高的任务；

4、 同时设置所有的任务；

5、 同时设置正在等待该事件的所有任务；

6、 同时设置有相同优先级的任务；

7、 同时设置正在等待该事件的具有相同优先级的所有任务；

3.1 cxewatm 等待事件函数

这个函数允许一个任务在一段指定的超时时间(time out)内等待特定事件的发生，如果需要的话。这个函数非常灵活而且功能强大，因为它允许一个任务等待一个或多个事件的发生。它也允许一个任务指定它所能够愿意等待的时间，因此它能够在所等待的事件未发生时采取正确的操作。使用清除模式命令提供给任务真正同步的能力，而不必考虑设置事件的实体，例如其它任务，邮箱，循环定时器以及中断等。

任务需要给 cxewatm 函数提供三个参数：

(1)、所等待的事件。

(2)、一个命令值用以指明是否在测试任务所需要等待的事件发生之前先自动清除它，还是在事件发生之后清除，还是在事件发生之前之后都清除，或者根本不清除。

(3)、一个时间段指明任务为匹配某事件所希望等待的时间或者是无限期的等待，时间值为 0 意味着这个任务将会无限期地等待这个事件的发生。

当这个函数被调用时指定的事件将被测试。当 cxewatm 函数被调用时，如果指定的事件位有一个或多个被设置，则任务将立即返回并指明发生了哪些事件。当任务调用这个函数时，它所期望的事件并没有发生，则任务将等待这些事件的发生。当所指定的事件至少有一个发生了或指定等待时间耗尽时，任务将自动恢复(被唤醒)。当然，可以通过 cxtwakf 函数来强制唤醒这个任务，而不管事件是否发生或时间是否耗尽。

清除模式命令

方式 0：不管事件发生与否都不清除。
方式 1：在调用该函数之后，在事件发生之前将清除事件位。
方式 2：事件发生之后将清除。
方式 3：事件发生之前或之后都清除。

清除模式命令有四个可能值：

0 ：表明当 cxewatm 函数被调用以及使用过程中，根本不清除任务的事件标志。这意味着任务所等待的各事件将仍保持当前的状态。(指调用该函数时的状态)

1 ：表明 cxewatm 函数将自动清除任务所等待的事件。这意味着如果这个任务将要等待事件 0 和事件 2，那么在测试是否存在匹配的事件之前，这两个事件都将被清除。这将导致任务由于不存在任何匹配的事件而挂起。

2：表明当出现与任务所等待事件相匹配的事件之后，cxewatm 函数将自动清除这些事件。该任务将被通知它所等待的什么事件已发生，同时这些事件将被清除。当然如果指定的等待时间耗尽或者任务被 cxtwakf 函数强制唤醒，那么这些事件的状态将保持它们原有的状态，处于原来的清除态。

3：指明 cxewatm 函数同时完成清除模式值为 1 和 2 时的功能。这意味着这个任务在等待事件之前将强制置这些事件于清除态，而当任务因为它所等待的事件至少有一个发生而恢复时，这些事件又再次被清除。清除模式值为 1 和 2 的情况已在上面两段中有过解释。

记住，cxewatm 函数工作和操作的对象只是调用该函数的任务所需要的事件(匹配的事件)。这意味着这个任务的所有别的事件将处于它们原有的状态。同样记住，每个任务可以有 16 个事件，并且 cxewatm 函数不会去操作其它任务的事件。

只有任务才能调用这个函数。当调用了该函数之后，任务将被挂起直到与所要求事件发生匹配或者发生超时(time out，如果时间值非 0)。当任务所等待的某个事件位被置位时，任务将自动进入恢复态，准备再次重新开始运行。函数的返回值将指明发生了任务所等待的哪一个事件。值得注意的是当任务恢复时，可能不止一个事件位被设置，这是因为任务由于其优先级的原因可能不会立即变为运行态的任务。

如果指定的等待时间耗尽，任务将进入恢复态，返回值 0 表明发生了这种超时(time out)情况。当

发生超时时，任务需要采取适当的操作，因为在指定的时间内事件并没有发生。

如果一个任务使用清除模示 0 或 2 调用 cxewatm 函数之前事件位已被设置，那么这个任务将不会被挂起。如果你希望任务仅仅等待一个事件，那么可以使用 cxtwatm 和 cxtwake 函数。这些函数能够稍稍加快任务的执行，因为 cxtwake 函数只处理一个特定的任务，而 cxesig 函数需要处理多个任务。

这是一个关于 cxewatm 函数的例子：

调用任务

```
#include <cxfuncs.h> /*头文件 */
word16  cxewatm(word16 , word16 , byte); /* 函数声明 */

#define   MATCH       ???
#define   TIME_CNT    ???
#define   MODE        ???
unsigned short     EVENTS; /* 该变量必须是局部变量 */

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

参数传递

MATCH：是一个 16 位的参数，它指明这个任务希望等待的特定的事件。可以使用多于一个的事件位。

TIME_CNT：是为获得一次匹配所希望等待的系统滴答的数量。如果它的值为 0，那么这个任务将为事件的匹配而无限制的等待下去。TIME_CNT 的最大值是 65535。

MODE ：决定当任务在进行匹配所等待的事件位之前或之后，这个函数是否采取清除的方式。具体说明如下所示：

0 = 不清除事件位
1 = 在函数进行匹配测试之前清除事件位
2 = 在函数获得事件匹配之后清除事件位
3 = 同时执行模示 1 和 2 的操作

```
#define  TSK2_EVENT1    0x01
#define  TSK2_EVENT3    0x04
#define  TSK2_EVT1 AND 3 TSK2_EVENT1 | TSK2_EVENT3

void task2(void)
{
   unsigned short   events;
   ……    /* 执行应用程序代码 */
   /* 注意：可以使用 3 种方法中的任何一种来识别事件 */
   events  =  cxewatm(TSK2_EVENT1 | TSK2_EVENT3 , 100 , 2);
   events  =  cxewatm(TSK2_EVT1 AND 3 , 100 , 2);
   events  =  cxewatm(0x05 , 100 , 2);
   /* 任务 2 将等待事件 0 和(或)2 的发生。任务 2 将为这个匹配的发生而等待 100 个系统滴答。同
      样任务 2 使用了清除模示值为 2，这表明当发生一个匹配并且任务恢复时，事件 0 和 2 将被
```

```
    自动清除。如果指定的等待时间耗尽，则该函数返回的值 0，这表明在事件为 0 或 2 被设置
    之前等待时间已经耗尽。*/
  if (events = = 0)  /* 测试是否出错 */
    {
      /* 在指定的等待时间内事件位 0 或 2 没有被设置，或者被 cxtwakf 函数所唤醒，在此处采
      取正确的操作。 */
    }
  else
  {
     if (events  &  TSK2_EVENT1)
       {
         /* 执行处理发生事件 0 的应用程序代码 */
       }
     if (events  &  TSK2_EVENT3)
       {
         /* 执行处理发生事件 2 的应用程序代码 */
       }
   }
}

/* 注意：可以通过以下的操作来代替以上的 if-else 语句所完成的操作。 */
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;
      }
    }
```

返回值

EVENTS 的值为 0 表明在任务所等待的任何一个事件发生之前，指定的等待时间耗尽，否则发生了由 MATCH 参数选择的一个或一些事件。

记住只有通过 MATCH 参数选择的事件才被该函数使用。MODE 参数根据何时清除任务的事件位实现了强大的同步能力。

3.2 cxesig 事件设置函数

cxesig 函数用于设置一个特定的事件，该函数能够以多种模式工作。这个函数可以被任务、循环定时器、邮箱或中断所调用。调用者将选择哪一个事件、该函数所工作的哪一种模式、以及由工作模式所决定的任务槽号还是任务的优先级。一些循环定时器和邮箱函数分别使用 cxesig 函数来标识时间的耗尽和邮件到达邮箱。

通过 cxesig 函数每次只有一个事件被设置。从技术上讲，可以调用 cxesig 函数来设置多个事件，但 CMX 建议不要这样做，而且实际上也很少需要这么做。

Cxesig 函数的模式值参数

调用者依据所选择的模式值来指定 cxesig 函数所需要的参数是任务的槽号，还是任务的优先级，还是一个未被使用的无效值。

调用者还需要指定 cxesig 函数的工作模式。此函数一共有七种不同的工作模式，如下所示：

模式#	**作用范围**(响应事件的对象范围)	**任务槽号或者优先级**
0	指定的某任务	任务槽号
1	最高优先级任务	无效值(任意值)
2	等待事件的优先级最高的任务	无效值(任意值)
3	所有的任务	无效值(任意值)
4	等待事件的所有任务	无效值(任意值)
5	具有某相同优先级的所有任务	优先级
6	等待事件的具有某相同优先级的所有任务	优先级

模式值 0：在这种模式下，该函数将设置由任务槽号指定的任务的特定事件。指定的任务不必正在等待这个事件。这是设置事件的最快方式，因为 cxesig 函数不需要测试许多变量或循环。以这种模式工作的 cxesig 函数能够有效地被中断、循环定时器和邮箱使用。然而，邮箱只能够用这种方式去通知一个特定的任务有消息(邮件)在邮箱中。(欲知详细细节请查阅消息管理这一章)。

模式值 1：在这种模式下，该函数将设置最高优先级任务的特定事件。cxesig 函数能自动寻找所创建的最高优先级的任务，同时设置这个任务的指定事件。如果有两个或多个任务具有相同的最高优先级，那么最先被 cxtcre 函数创建的任务的事件将得到设置。(注意一般用来指定任务槽号或优先级的参数没有被使用，它可以是任意值。)

模式值 2：在这种模式下，cxesig 函数将会寻找正在等待这个事件的最高优先级的任务。这会使得函数从最高优先级的任务处开始寻找、并检验这个任务是否正在等待这个事件(因为任务调用了 cxewatm 函数)，如果没有找到的话，函数将不断地检验下一个优先级最高的任务，直到函数因为找到符合要求的任务(正在等待这个事件的任务)、或者已检验完所有的任务都没找到而退出。(再次提醒用于指定任务槽号或优先级的参数可以是任意值，因为在这种情况下该参数未被使用。)

模式值 3：在这种模式下，cxesig 函数将对所有的任务都设置指定的事件。函数将循环运行，直至所有的任务都设置了指定的事件，而不管任务是否在等待这个事件。需要注意的是该操作将依赖已创建任务的数目来决定所需花费的时间。(用来指明任务槽号或优先级的参数未被使用，它可以是任意值。)

模式值 4：在这种模式下，此函数将为正在等待指定事件的所有任务，都设置指定的事件。cxesig 函数将循环测试所有的任务直到检测完所有的任务，如果某任务正在等待这个事件，那么该任务所等待的这个事件将被设置。(用来指明任务槽号或优先级的参数未被使用，它可以是任意值。)

模式值 5：在这种模式下，cxesig 函数将设置与指定任务优先级相同的所有任务的特定事件。用于指定任务槽号或优先级的参数必须是调用者希望设置事件的那些任务的优先级，而不管任务是否在等待这个事件。函数将循环检测每个任务是否与指定任务优先级相同，如果相同，那么这个任务的特定事件将会被设置。注意如果不存在与指定的任务优先级相同优先级的任务，那么函数将不设置任何任务的事件。

模式值 6：在这种模式下，该函数将为任何拥有指定优先级相同、并且正在等待指定事件的任务设置指定的事件。cxesig 将检测每个任务是否符合这个标准，满足标准的任务的状态将会从就绪态转换到恢复态。

当 cxesig 函数被调用时，如果任务正在等待指定的事件，同时函数调用调度模式值决定了将选择这个任务，那么将会发生下列情况：正在等待这个事件的任务的该事件位将被设置，然后这个任务将从就绪态变为恢复态，表明该任务可以从其挂起的地方重新开始执行。如果任务因为所等待的事件发生而被唤醒并且此任务的优先级高于正在运行任务的优先级，那么在 cxesig 函数执行完成后，系统将进行重新调度以实现任务的切换。如果某任务的指定事件被设置而任务并没有正在等待该事件，那么它的状态将保持不变。

注意调用 cxesig 函数的任务、循环定时器、邮箱或者中断，都不必知道是否有任务正在等待指定的事件，也不必了解是一个还是多个任务在等待该指定事件。如果只需要实现对单个事件的同步，用户可以使用 cxtwatm 和 cxtwake 函数来实现，这种同步机制在速度上会稍微快一些。

以下是一个关于 cxesig 函数的例子：

调用

任务、中断、循环定时器和邮箱。

注：中断也可以调用该函数但只能是间接调用。具体调用方法请参看处理器详细说明一章。

```
#include <cxfuncs.h>                    /* 头文件 */
byte  cxesig(byte , byte , word16);  /* 函数声明 */

#define    MODE                ???
#define    EVENT_TO_SET    ???
unsigned char    TASK_PRI;
unsigned char    STATUS;  /* 该变量必须是局部变量 */

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

参数传递

MODE 是 cxesig 函数的工作模式值，该值将决定函数需要处理的任务。该参数的取值如下所列：

模式	需要处理的任务
0	由 TASK_PRI 来指定的某个特定任务
1	最高优先级的任务，不包括 CMX 定时器任务
2	等待该事件的优先级最高的任务
3	所有的任务，将设置所有已创建的任务的事件
4	等待这个事件的所有任务
5	具有由 TASK_PRI 指定优先级的所有任务
6	所有具有由 TASK_PRI 指定优先级的所有正在等待该事件的任务

TASK_PRI 由 MODE 参数所确定的模式值来决定，该值可以是任务的槽号或优先级。 (该参数在 MODE 取某些模式值时将是无效参数。)

EVENT_TO_SET：是一个 16 位宽的无符号变量或常量，代表将要被设置的事件。

```
#define   TSK2_EVENT1   0x01
#define   Mode_0              0x00;
unsigned char   task2_slot;

void   task1(void)
{
   unsigned char   status;

   ……    /* 执行应用程序代码 */

   status  =  cxesig(Mode_0 , task2_slot , TSK2_EVENT1);
   /* 任务 1 将设置任务 2 的事件(0x01)，而不管任务 2 是否正在等待该事件。如果任务 2 正在等待
      这个事件，那么任务 2 将自动恢复。*/
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERR = Error：模式值为 0 时指定任务不存在，或者模式值取值错误。
如果 STATUS 等于 AOK, 那么 cxesig 函数已经完成了根据模式值所指定的操作。

注意：循环定时器和邮箱可以设置成自动调用的方式调用该函数。中断也可以调用这个函数。

上述的两个函数 cxewatm 和 cxesig 使得一个任务或者中断可以向其它任务或中断发信号、而不需了解它们是不是正在等待这个信号。

例如，如果任务 1 正在等待一个气阀关闭的信号以继续它的工作，同时任务 2 在其能够继续执行前需要等待任务 1 收到这个信号，那么任务 2 可以调用 cxtwatm 函数实现无限期的等待，直到一个任务或中断唤醒它。 然后任务 1 将调用 cxewatm 函数指明某事件对应气阀被关闭这个信号，该信号将被一个外部的传感器感知、并且其信号的输出将触发一个中断。当传感器探知阀门已被关闭将会引发一个中断，中断代码将调用 cxesig 函数，以表明该事件的发生、并设置任务 1 的该事件位。cxesig 函数调用完成后，任务 1 将自动从挂起态恢复为就绪态，然后任务 1 可以使用 cxtwake 函数通知任务 2 已经收到了信号，同时为任务 2 继续处理而设置某些全局变量的正确值。有可能在中断没有发生、而任务 1 指定的等待时间非 0 的情况下该时间耗尽，那么任务 1 将变为就绪态。当任务 1 恢复运行时，它可以通过查看返回值(如返回值为 0 即意味着超时的发生)来采取合适的操作，如警报等。

3.3 cxerst 清除事件标志函数

cxerst 函数利用清除事件位的方式清除一个或多个事件，这意味着一个任务能够清除其它某一个任务或它自己本身的指定事件。如果不想用 cxewatm 函数的清除模式命令来清除事件，那么可以使用这个函数。注意：因为任务不会等待事件清除操作，所以各个任务除非清除的是它自己的事件，否则不知道它的事件中的某一个已被清除。

以下是一个关于 cxerst 函数的例子：

调用任务

```
#include <cxfuncs.h>                        /* 头文件 */
byte   cxerst(byte tskid , word16 event);   /* 函数声明 */
unsigned char    TASK_SLOT;                 /* 该变量必须是全局变量，即任务的槽号 */
unsigned short    EVENTS_TO_CLEAR; /* 需要清除的事件位，或者可以是一个宏定义，如 #define
                                        EVENTS_TO_CLEAR ???    来代替以上的变量定义 */
unsigned char   STATUS;                     /* 该变量必须是局部变量 */
STATUS  =  cxerst(TASK_SLOT , EVENTS_TO_CLEAR);
```

参数传递

TASK_SLOT：是事件位将要被清除的任务槽号。

EVENTS_TO_CLEAR：是一个无符号的 16 位宽的变量或常量，它代表在这个任务中希望被清除的事件位。

```
#define   TSK1_EVENT1     0x0001
unsigned char   task1_slot;
void   task1(void)
{
    unsigned char   status;
    status  =  cxerst(task1_slot , TSK1_EVENT1); /* 任务 1 需要清除它自己的事件位位 0 */
    ……              /* 任务 1 的其它代码 */
    cxtend();         /* 通知 CMX 该任务已经完成 */
 }
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERR = Error：指定任务不存在而出错。
如果 STATUS 等于 AOK, 那么由 EVENTS_TO_CLEAR 参数指定的事件位已被清除。

注意该函数仅仅清除由任务槽号指定任务的事件位，它不影响其它任务的任何事件位。

4、队列管理函数

队列管理是 CMX 库的一部分并且提供了必要的函数来实现队列的管理。队列管理函数如下所列：

cxqcre 函数；
cxqrst 函数；
cxqatt 函数；
cxqatb 函数；
cxqrft 函数；
cxqrfb 函数；

4.1 cxqcre 创建循环队列函数

这个函数可以用来创建一个循环队列，在使用这个函数的时候需要提供该队列希望拥有的元素或记录的数量。每个队列最多拥有 32767(0 – 7FFFH)个元素。

另一个参数是分配给每个队列元素的字节数目，队列中每个元素的大小由用户按实际情况决定。每个队列元素的最大尺寸必须小于 255 个字节，同时这个尺寸的大小将是在这个特定队列中所有队列元素的尺寸大小。在进入 CMX 操作系统之前或进入以后都可以调用 cxqcre 函数。

用户必须提供该驻留在外部 RAM 中的队列的起始地址。

对于要求指针或整型数必须驻留在一个偶或奇地址边界的 CPU 中，提供给 cxqcre 函数的队列地址也必须驻留在那个特定的地址边界。每个队列元素的字节数可能也必须保证是两个这种边界所限制的最少字节数的倍数。

调用 cxqcre 函数来构建一个特定的队列所需要的内存，是其队列元素数与每个队列元素所需字节数的乘积。CMX 不会检测是否已经为这个队列分配了所需要的正确的字节数，如果队列的大小小于队列元素的大小与队列元素数的乘积，那么可能会出现不可预知后果。

该函数不检测内存争用问题。系统假定对这个队列所需要的内存任何时候都是可用的。这个函数允许任务在任何时候从队列中删除或增加队列元素。

另外还需要提供一个队列号，所有队列函数将使用该队列号来引用该队列。这是一个用来识别函数将要引用的队列的数，这使得函数可以用它来正确地处理 CMX 的队列结构。这个队列号的范围从 0 到小于队列数量最大值的一个数，此最大值由用户在 CMX 配置文件中定义。

必须认识到队列的维护方式是，每一个队列元素都被当作一个无符号的字符数组，它允许字符、整数、指针以及长整型等形式的存储。

如果需要传递许多字节到一个队列，CMX 建议只需要将指向这些数据的指针传递给队列即可，此时队列元素的大小只需要是一个指针的大小。这样将使队列的操作速度变得非常快，但是在使用这些数据并且将指针删除之前必须保证源数据的内容不能有任何的更改。

队列中每个元素的大小假如是 20 字节，但这并不是说每次将数据放入一个队列元素中时所有的 20 个字节都会用到。例如，某变长记录，记录的最大长度是 19 个字节，可以建立这样一个结构，其

第一个字节是这个记录的字节数，而其余的字节是记录的数据内容。这样就可以把这个结构的地址传递作为参数传递给队列函数。

CMX 的队列管理函数将拷贝全部 20 个字节，而不管其最后的几个字节可能是空的还是属于其它数据项的。任务从队列中接收数据时需要首先检测该记录的计数字节，然后再处理后续指定长度的字节数据。再次提醒用户注意：CMX 建议尽可能地使用指针操作来代替使用大的队列元素队列操作，虽然实际有时还是需要这种大量数据拷贝和传递的操作。

以下是一个 cxqcre 函数的例子：

调用

进入 RTOS 前、任务。

```
#include <cxfuncs.h>                                /* 头文件 */
byte   cxqcre(sign_word16 , byte , byte * , byte);   /* 函数声明 */

#define   NUM_SLOTS   ???
#define   SIZE_SLOT     ???
#define   QUE_NUM     ???
unsigned char   STATUS;
unsigned char   QUE_NAME[NUM_SLOTS  *  SIZE_SLOT]; /* 该变量必须是全局变量 */

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

参数传递

NUM_SLOTS：是这个队列将拥有的队列元素总数，其最大值为 32767。
SIZE_SLOT ：是在这个队列中每个队列元素将占有的字节数(每个队列元素的大小)。
QUE_NAME：是这个队列将驻留在内存中的起始地址。
QUE_NUM：是队列的标识(队列号)，所有的队列函数都将用它来引用队列。

```
#define   QUE1_SLOTS   50  /* 这个队列将拥有 50 个队列元素 */
#define   QUE1_SIZE      10  /* 每个队列元素是 10 个字节长 */
#define   QUE1    1              /* 队列号为 1，以便于 CMX 队列函数引用队列 */
unsigned char   queue1[QUE1_SLOTS  *  QUE1_SIZE];     /* 为队列分配存储区 */

void  task2(void)
{
   unsigned char   status;
   ……          /* 任务的其它应用代码 */
   status  =  cxqcre(QUE1_SLOTS , QUE1_SIZE , queue1,QUE1);
    /* 创建一个循环队列，在这个队列中有 50 个队列元素，每个队列元素最多可存储 10 个字节长
       的信息。同样队列在内存中的起始地址也作为参数传递给了函数。*/
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回，队列已被创建。
AERR = Error：队列号超出范围而出错。
注意一定要为队列分配足够的内存，函数不检测内存争用问题。

4.2 cxqrst 队列复位函数

cxqrst 函数可以把队列复位，重新设置为空状态。当调用该函数时，队列中所有的队列元素全被释放，并且队列元素中所有包含的信息将被永久地删除。该函数只允许被任务调用。

cxqrst 函数需要的唯一的参数是队列号。该标识号的取值范围从 0 到 CMX 规定的队列的最大数量。

以下是 cxqrst 函数的一个实例：

调用

进入 RTOS 之前、任务。

```
#include   <cxfuncs.h>          /* 头文件 */
byte   cxqrst( byte );          /* 函数声明 */

#define    QUEUE_NUM      ???
unsigned char   STATUS;   /* 该变量必须是局部变量 */
STATUS  =  cxqrst(QUEUE_NUM);
```

参数传递

QUEUE_NUM：是队列号，用于确定由 cxqcre 函数创建的某个队列。该队列号可以是从 0 到配置文件中所定义的最大队列数。

```
#define   QUE1   1    /* 队列 1 的标识号 */
void   task2(void)
{
   unsigned char   status;
   ……    /* 其它应用程序代码 */
   status  =  cxqrst(QUE1);    /* 初始化队列 1，所有的队列元素处于空闲状态 */
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERR = Error：队列是空的，没有任何队列元素(未创建)。

4.3 cxqatt 数据载入队列顶部函数

这个函数可以将数据填入指定队列顶端的空闲的队列元素中，调用者将它们想要操作的队列号作为参数传递给 cxqatt 函数，同时作为参数传递的还有源数据的起始地址，而不管源数据是什么，例如是一个字节，还是整型数、指针、长整型数等。

如果创建了一个其每个队列元素都是两字节长的队列，而使用时传递了一个长整型数(4 byte)的地址，那么这个长整形数只有头两个字节被拷贝入队列。同样道理，如果只是传递了某个字节数据的地址，那么这个字节及其相邻的字节将被拷贝入队列。(每次只能按元素大小拷贝一组数据)

如果在这个循环队列中有空闲的队列元素， 那么 cxqatt 函数将取出其中位于队列顶端的空闲的队列元素，并按照所提供的数据首地址将数据拷贝到这个队列元素中(所拷贝的数据的长度由 cxqcre 函数创建队列时所提供的队列元素的大小决定)。这种数据的拷贝操作将使原存放数据的存储区成为可用的存储区 (释放被拷贝的数据存储区) 。

cxqatt 函数检测队列号的有效性，同时也检测队列元素的大小是否大于 0。如果 C 编译器能自动将所有外部未被初始化的 RAM 全部清除为 0(创建队列后，应将其清“ 空 ”)，那么这样就可以确保队列已被建立。这个函数所用的队列元素总是取自于循环队列的最顶端。因为队列是循环队列(环型)，所以要防止队列数据的回卷。只有任务才可以调用这个函数。

以下是一个关于 cxqatt 函数的例子：

调用

进入 RTOS 之前、任务。

```
#include <cxfuncs.h>            /* 头文件 */
byte cxqatt(byte , void *);     /* 函数声明 */

#define   QUEUE_NUM    ???
unsigned char   *SOURCE_POINTER; /* 局部变量或全局变量，由应用和用户决定。*/
unsigned char   STATUS;                  /* 该变量必须是局部变量 */

STATUS  =  cxqatt(QUEUE_NUM , SOURCE_POINTER);
```

参数传递

QUEUE_NUM：是队列号，这是一个由 cxqcre 函数所创建的特定队列的标识。这个值的范围是从 0 到配置信息中规定的最大值。

SOURCE_POINTER：是一个指向源字节数据的指针。

```
#define   QUE1   1  /* 队列 1 的队列号 */

void   task2(void)
{
   unsigned char   status;
   status  =  cxqatt(QUE1 , "hello world");   /* 将数据拷贝入队列顶端的空闲队列元素中 */
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK ：该函数调用成功正常返回。

AERQUEFUL：该函数调用成功，但现在队列已满。

AERR：函数调用错误，表明队列已满或者队列号超出许可范围或者队列未创建。

记住指针可以指向任何对象。必须通过计算来保证指针指向的对象能完整地被复制，如长整型数据以及指针等等。用户一定要确保这个队列已被创建，并且这个函数使用的队列号确实是希望被操作的队列的标识号。

4.4 cxqatb 数据载入队列底部函数

这个函数可以将数据填入指定队列底部的空闲的队列元素中。调用者将它们想要操作的队列号作为参数传递给 cxqatb 函数。同时作为参数传递的还有源数据的起始地址，而不管源数据是什么，是一个字节，还是整型数、指针、长整型数等。

如果创建了一个其每个队列元素都是两字节长的队列，而使用时传递了一个长整型数的地址，那么这个长整形数只有头两个字节被拷贝入队列。同样道理，如果只是传递了某个字节数据的地址，那么这个字节及其相邻的字节将被拷贝入队列。

如果在这个循环队列中有空闲的队列元素， 那么 cxqatb 函数将取出其中位于队列底部的空闲的队列元素，并按照所提供的数据首地址将数据拷贝到这个队列元素中(所拷贝的数据的长度由 cxqcre 函数创建队列时所提供的队列元素的大小决定)。这种数据的拷贝操作将使原存放数据的存储区成为可用的存储区。

cxqatb 函数检测队列号的有效性。这个函数所用的队列元素总是取自于循环队列的最底部。只有任务可以调用这个函数。

以下是一个关于 cxqatb 函数的例子：

调用

进入 RTOS 之前、任务。

```
#include <cxfuncs.h>            /* 头文件 */
byte   cxqatb(byte , void *);   /* 函数声明 */

#define   QUEUE_NUM    ???
unsigned char   *SOURCE_POINTER;  /* 局部变量或全局变量，由应用和用户决定。*/
unsigned char   STATUS;                  /* 该变量必须是局部变量 */

STATUS  =  cxqatt(QUEUE_NUM , SOURCE_POINTER);
```

参数传递

QUEUE_NUM：是队列号，这是一个由 cxqcre 函数所创建的特定队列的标识。这个值的范围是从 0 到配置信息中规定的最大值。

SOURCE_POINTER：是一个指向源字节数据的指针。

```
#define   QUE1    1 /* 队列 1 的队列号 */
```

```
void  task2(void)
{
   unsigned char   status;
   status  =  cxqatb(QUE1 , "12345\n"); /* 将数据拷贝入队列底部的空闲队列元素中 */
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：
AOK =Good：该函数调用成功正常返回。
AERR = Error：队列已满，没有可用的队列元素。
AERQUEFUL= Warning：该函数调用成功，但现在队列已满。

如果 STATUS 为 AOK，那么源数据的内容已经被拷贝到队列中。如果 STATUS 为 AERQUEFUL，那么源数据的内容也已经被拷贝到队列中并且队列中无可用队列元素。

4.5 cxqrft 数据从队列顶端移出函数

这个函数可以将数据从指定队列的某个队列元素中移出。调用者从这个队列元素中将数据移出并返回给调用者使用，该队列元素就是被 cxqatt 函数操作的最后一个处于队列顶端的队列元素。

调用者利用队列号和拷贝数据的目的地址两个参数来调用 cxqrft 函数，然后这个函数将根据队列元素大小，将最后一个被 cxqatt 函数操作的、处于队列顶端的队列元素中的数据内容拷贝到指定的目的地址，然后释放这个队列元素，以使它能够被再次使用。

函数不进行内存的测试，所以必须确保所提供的地址确实是一个队列的地址。

以下是一个关于 cxqrft 函数的例子：

调用

进入 RTOS 之前、任务。

```
#include <cxfuncs.h>                  /* 头文件 */
byte cxqrft( byte , void *);          /* 函数声明 *／

#define  QUEUE_NUM     ???

unsigned char   *DEST_POINTER;   /* 局部变量或全局变量，由应用和用户决定。*/
unsigned char    STATUS;          /* 该变量必须是局部变量 */

STATUS  =  cxqrft(QUEUE_NUM , DEST_POINTER);
```

参数传递

QUEUE_NUM：是队列号，这是一个由 cxqcre 函数所创建的特定队列的标识。这个值的范围是从 0 到配置信息中规定的最大值。
DEST_POINTER：是一个指向队列元素中数据希望被拷贝到的内存地址的指针。

```
#define  QUE1   1

void  task2(void)
{
   unsigned char   status;
   unsigned char  dest[20];          /* 为存放队列元素中数据分配空间 */
   status  =  cxqrft(QUE1 , dest);   /* 从队列中最后一个被使用的处于顶端的队列元素中拷出数据*/
   ……                               /* 处理所得到的数据 */
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK =Good：该函数调用成功正常返回。

AERR = Error：队列空，没有可用的队列元素。

AERQUEEMPTY = Warning：该函数调用成功，数据已被拷贝但现在队列已空。

如果 STATUS 为 AOK，那么队列元素中的数据已被拷贝到目的地址中。如果 STATUS 为 AERQUEEMPTY，则表明现在队列已空。

注意队列元素中可能是任何内容，字节、整数或指针等等。只要数据从队列元素中移出，用户可以将这些数据放在任何地方。用户必须确保这个函数使用的队列已经被创建而且该队列就是曾经放入数据的队列。

4.6 cxqrfb 数据从队列底部移出函数

cxqrfb 函数完成的操作与上述的 cxqrft 函数除了一点外基本相同。数据的内容从队列中的某个队列元素中被拷贝出来并返回给调用者，但该函数所操作的队列元素是队列中最后一个被加入到队列底部的队列元素而不是 cxqrtf 函数操作的队列顶端的队列元素。这个函数使得一个任务可以从指定队列中取出所需要的数据。

调用者利用队列号和拷贝数据的目的地址两个参数来调用 cxqrfb 函数。然后这个函数将根据队列元素大小将最后一个被 cxqatb 函数操作的处于队列底端的队列元素中的数据内容拷贝到指定的目的地址，然后释放这个队列元素，以使它能够被再次使用。

函数不进行内存的测试，所以必须确保所提供的地址确实是一个队列的地址。

队列函数使得任务可以传递参数给其它的任务。用户可以用很多方式来操作一个队列，例如 FIFO(先进先出)，LIFO(后进先出)或者是两种方式的结合。一个队列元素中的数据可能是：整型数、指针、给命令处理任务的命令、给错误处理任务的错误代码以及需要任务处理的其余参数的地址等等。通常处理一个队列时，可以将 cxqrft 和 cxqrfb 函数混合或配合使用。

记住，当从队列的顶端或底部移去一个队列元素时，这个元素就被认为是空的并且队列将它标记为空闲的队列元素。这时没有任何函数能够访问这个队列元素，直到它再次被赋值使用。

队列中每个队列元素的尺寸越大，那么将数据从源地址中拷贝到一个队列元素，或从一个队列元素中拷贝到目的地址所需发费的时间也就越长。CMX 建议设置队列元素的尺寸较小，而且对大量的数据传送使用指针操作，那么只需要传递一个指针到队列即可。

以下是一个关于 cxqrfb 函数的例子：

调用

进入 RTOS 之前、任务。

```
#include <cxfuncs.h>                /* 头文件 */
byte   cxqrfb( byte , void *);      /* 函数声明 */

#define   QUEUE_NUM    ???
unsigned char    *DEST_POINTER;     /* 局部变量或全局变量，由应用和用户决定。*/
unsigned char    STATUS;            /* 该变量必须是局部变量 */
STATUS  =  cxqrfb(QUEUE_NUM , DEST_POINTER);
```

参数传递

QUEUE_NUM：是队列号，这是一个由 cxqcre 函数所创建的特定队列的标识。这个值的范围是从 0 到配置信息中规定的最大值。

DEST_POINTER：是一个指向队列元素中数据希望被拷贝到的内存地址的指针。

```
#define  QUE1   1

void  task2(void)
{
   unsigned char   status;
   unsigned char   dest[20];        /* 为存放队列元素中数据分配空间 */
   status  =  cxqrfb(QUE1 , dest);  /* 从队列中最后一个被使用的处于底部的队列元素中拷出数据*/
   ……                               /* 处理所得到的数据 */
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK =Good：该函数调用成功正常返回。

AERR = Error：队列空，没有可用的队列元素。

AERQUEEMPTY = Warning：该函数调用成功，数据已被拷贝但现在队列已空。

如果 STATUS 为 AOK，那么队列元素中的数据已被拷贝到目的地址中。如果 STATUS 为 AERQUEEMPTY，则表明现在队列已空。

注意队列元素中可能是任何内容，字节、整数或指针等等。只要数据从队列元素中移出，用户可以将这些数据放任何地方。用户必须确保这个函数使用的队列已经被创建而且该队列就是曾经放入数据的队列。

5、UART 管理函数

UART 管理不是 CMX 库中的一部分。它提供了以下所列的函数：

init_xmit、init_recv、uart_update_xmit、uart_update_recv、put_uart_chr、put_uart_chr_w、put_uart_str、put_uart_str_w、get_uart_chr、get_uart_chr_w、get_uart_str、get_uart_str_w、get_uart_str_wg、get_uart_str_v、get_uart_recv_cnt

这些 UART 函数可以以实时中断的方式进行在线的串行 UART 工作，当在等待接收一个字符或检查发送缓冲器是否为空以便发送下一个字符时，接收程序和发送程序都不必使用查询(poll)方式。

每一个 UART 都有一个接收缓冲器和发送缓冲器。根据应用程序要求可指定缓冲器的大小，而且两个缓冲器可以大小不一。下面将解释每一个缓冲器的是如何工作的，并且介绍 CMX 是如何与它们衔接的。

5.2 UART 中的发送、接收缓冲器

5.2.1 接收缓冲器

接收缓冲器是一个带有头指针和尾指针的循环(circular)缓冲。头指针表明下一个字符放在缓冲区的位置；尾指针表明可以从接收缓冲器中获得字符的位置。这些字符按照它们收到的先后顺序被取出，是一个先进先出的结构(FIFO)。同时缓冲器中有一个计数变量，它表示当前贮存在接收缓冲器中还没有被取出的字节数目。如果计数字节达到了接收缓冲器的最大允许值，那么接收缓冲器将被标记为状态满，接收缓冲器将不允许再接收字符。

如果有一个任务需要从接收缓冲器中获得一定数量的字节数据，而接收缓冲器中并没有这么多的数据，那么该任务将被阻塞。一旦接收缓冲器发现数据量够了，它回自动唤醒此睡眠任务。任何时候一个接收缓冲器只能被一个任务占用，如果在同一时刻不止一个任务需要使用接收缓冲器，那么必须使用资源来加以控制。

5.2.2 发送缓冲器

发送缓冲器是带有头指针和尾指针的线性缓冲区，头指针指向缓冲器里最后一个字符，该字符应该被发送中断程序所发送。当发送缓冲器可以进行发送操作时，尾指针指向当前应该向外发送的字节。每次使用发送缓冲器时，缓冲器都会重置到其初始状态。

发送缓冲器任何时候只能被一个任务所占用。如果在同一时刻有许多任务要使用发送缓冲器，那么必须使用资源来加以控制。如果发送缓冲器已经装载了字符，那么当它往外发送字符时，它处于忙状态。如果发送缓冲器处于忙状态，任何装载字符的要求都会被阻塞。

CMX UART 的 C 模块源代码需要重新编辑以便设置波特率、数据位数、接收缓冲器和发送缓冲器的大小以及奇偶校验，还须设置指定的 CPU 的 UART 地址和其它相关变量，同时还需增加一些特殊的函数来处理可能发生的接收错误，比如超载(overrun)，奇偶校验错，帧错误等等，因为 CMX 不知道当接收器错误发生时程序将如何处理。

用 C 编写的这些函数将增加中断延迟时间，这个延迟时间将会比没有这些函数时长的多。因为 UART 中断可在任何时候发生，并和*程序临界区*完全异步，因此在 CMX UART 函数中，必须完全关闭中断然后又重新打开中断。根据不同的 CPU 和 C 编译器，这个延迟时间可达微秒级甚至更长，因此必须确定由此而引入的中断延迟时间是否能满足 UART 的需要。

必须确定所选择的波特率是允许的，这依赖于执行这些函数需花多少时间，也依赖于应用程序代码和其它中断中因为开关中断而引起的中断延迟时间。

这些 UART 函数必须能够满足应用程序的要求。CMX 建议用汇编语言编写 UART 中断函数，因为这会使 UART 中断执行的速度更快，而且还可以减少上面介绍的中断延迟时间。这种功能可以通过设置表明任务状态的标志来实现。

例如，用汇编语言编写的接收子程序可以接收字符并把它们放入接收缓冲器中。当收到 END_OF_PACKET(数据包结束标志)字符或者指定的时间周期耗尽时，接收中断会调用中断管道来触发指定任务去处理收到的数据。接收子程序也可以使用多个接收缓冲器存放数据，只需适当改变接收缓冲器的指针即可。CMX 库中有一些关于如何用汇编语言编辑 UART 中断程序的例子可供参考。

5.3 init_recv 初始化接收缓冲器函数

init_recv 函数用于初始化接收缓冲器，其具体操作是将接收缓冲器的相关指针设置到缓冲的起始位置。同时用于表明已接收到、但未被处理的字符数计数变量 count_in 将被重置为 0。接收缓冲器的状态标志也将被重置以表明缓冲器处于良好的状态，没有错误或者缓冲器已满等其它错误的存在。

只要没有任务因其需要的指定数量的字符未被满足而在等待接收缓冲器，用户就可以在任何时候自由地调用 init_recv 函数。注意，如果接收缓冲器中存在没有被检索取走的字符，那么在 init_recv 函数执行后这些字符将会丢失。用户可以在这个函数中设置接收器的 BAUD(波特)率，数据的位数，校验的奇偶性以及停止位的数量。

5.4 init_xmit 初始化发送缓冲器函数

init_xmit 函数用于初始化发送缓冲器，其具体操作是将发送缓冲器的相关指针设置到缓冲的起始位置，同时用于表明需要被传送字节数的计数变量 count_out 将被重置为 0。发送器的状态标志也将被重置，以表明发送器状态良好、并处于空闲状态。一般只需要调用一次 init_xmit 函数，因为一般情况下没有任何原因使得需要重新初始化发送缓冲器。用户可以在这个函数中设置发送器的 BAUD(波特)率，数据的位数，校验的奇偶性以及停止位的数量。

5.5 uart_update_xmit 函数

uart_update_xmit 函数由发送中断调用。首先函数将检测发送缓冲器的空闲标志是否已被设置；如果是这样的话，那么函数将测试发送计数变量 count_out 是否为 0。如果发送计数变量 count_out 非 0，表明还有需要发送的字符存在，发送器将自动装载下一个缓冲器中的字符，同时计数变量 count_out 将减 1，并且尾指针将自动加 1 以指向下一个需要发送的字符。如果计数变量 count_out 为 0，函数将检测是否有任务正在等待发送缓冲器以传送字符。注意，任何时候只允许有一个任务占有发送缓冲器(可以通过资源管理函数来控制)。如果函数检测到有任务正在等待，那么该任务将被自动置为恢复态，表明当任务变为能够运行的最高优先级的任务时，它就可以将所需要发送的字符放入传送缓冲器中。

5.6 uart_update_recv 函数

如果接收缓冲器没有满，那么接收到的字符将被放入接收缓冲器中，同时接收器的头指针将加 1，以指向下一个可存储字符的位置。接收计数变量 count_in 也将加 1，该变量表明所接收到的在缓冲器中的字符数。如果有任务正在等待接收器以接收指定数目的字符，那么 uart_update_recv 函数将检测目前所接收的字符是否已足够。如果所接收的字符已经足够，那么该任务将被自动成为恢复态，同时指明该任务所需要的字符已被接收。

5.7 put_uart_chr 函数

put_uart_chr 函数允许占有发送缓冲器的任务发送一个字符。这个函数的参数是任务想要发送的字符的地址。如果发送器正在传送字符，那么这个字符不会被放入到发送缓冲器中，同时函数将返回一个状态表明发送器忙。如果发送器处于空闲状态，那么这个字符将被放入到发送缓冲器中，然后发送器将自动开始工作。函数将立即返回一个值表明调用成功。任务在 UART 发送字符时不需要等待，中断系统自动完成所需要的操作。

以下是一个关于 put_uart_chr 函数的例子：

调用

任务。

```
#include <cxfuncs.h>          /* 头文件 */
byte   put_uart_chr(void *);  /* 函数声明 */
unsigned char   *SRC_PTR; /* 可以是局部变量，也可以是全局变量，表明函数获得字符的地址。*/
unsigned char    STATUS;    /* 函数返回值 */

STATUS  =  put_uart_chr(SRC_PTR);
```

参数传递

SRC_PTR：为传递给这个函数的字符的驻留在内存中的地址。

```
unsigned char  src_byte  =  '3';        /* 存储着需要被发送的字符的地址，可以是局部变量。*/

void  task1(void)
{
   unsigned char   status;
   status  =  put_uart_chr(&src_byte); /* 开始发送字符 */
   if (status != AOK)                        /* 检测返回值 */
      {
        /* 对发送不成功的处理*/
      }
   ……
   /* 或者可以用另外一种格式编写这段程序。*/
   status  =  put_uart_chr("3");  /* 开始发送字符 */
   if (status != AOK)              /* 检测返回值 */
      {
        /* 对发送不成功的处理 */
      }
   ……
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
UART_BUSY = Error：因发送器忙而返回。
如果函数的返回状态 STATUS 为 AOK，那么调用这个函数的任务已经把字符放入发送器中，同时发送器将自动开始发送字符。

5.8 put_uart_chr_w 函数

put_uart_chr_w 函数与上述的函数类似。如果发送器忙，那么这个任务将等待一段指定的时间或者进行无限期的等待。这个函数的参数是需要被传送的字符的地址和希望等待发送器成为空闲的一定数量的时间。这个函数允许拥有发送缓冲器的任务发送一个字符。

如果发送器正在发送字符，那么任务将被挂起。任务的挂起时间的长短由任务调用函数的第二个参数值决定，该值为 0 表明任务将进行无限期的等待，否则任务将等待指定数量(该数量的取值范围是 0 到 65535)的系统滴答时间。一旦发送器成为空闲状态，那么该挂起任务将自动被唤醒，然后它就可以将所需要发送的字符放入发送缓冲器中。函数将立即返回一个值以表明操作成功。如果在指定时间内发送器未成为空闲状态，那么所需要发送的字符将不会被放入到发送缓冲器中，同时任务将获得一个返回值表明时间耗尽。

以下是一个关于 put_uart_chr_w 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                    /* 头文件 */
byte  put_uart_chr_w(void * , word16); /* 函数声明 */
unsigned char   *SRC_PTR; /* 可以是局部变量，也可以是全局变量，表明函数获得字符的地址。*/
#define  TIME_PERIOD   ???              /* 指定等待的时间 */
unsigned char   STATUS;                 /* 函数返回值 */

STATUS  =  put_uart_chr_w(SRC_PTR , TIME_PERIOD);
```

参数传递

SRC_PTR：为传递给这个函数的字符的驻留在内存中的地址。
TIME_PERIOD：是任务希望为发送器成为空闲而可以等待的系统滴答数，其范围从 0 到 65535 。

```
unsigned char  src_byte  =  '3';   /* 存储着需要被发送的字符的地址，可以是局部变量。*/

void  task1(void)
{
   unsigned char   status;
   status  =  put_uart_chr_w(&src_byte , 100);  /* 开始发送字符，如果需要的话，可以为发送器成为
                                                   空闲而等待 100 个系统滴答时间 */
   if (status != AOK)      /* 检测返回值*/
      {
        /* 对发送不成功的处理*/
      }
   ……
   /* 或者可以用另外一种格式编写这段程序。*/
   status  =  put_uart_chr_w("3", 100); /* 开始发送字符，如果需要的话，可以为发送器成为空闲而
                                           等待 100 个系统滴答时间 */
   if (status != AOK)   /*检测返回值*/
      {
```

```
        /* 对发送不成功的处理*/
      }
    ……
}
```

返回参数

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERTIME = Warning/error：指定等待时间耗尽时发送器依然忙。
UART_BUSY = Error：因发送器忙而返回。
如果函数的返回状态 STATUS 为 AOK，那么调用这个函数的任务已经把字符放入发送器中，同时发送器将自动开始发送字符。

5.9 put_uart_str 函数

put_uart_str 函数允许占有发送缓冲器的任务发送一个或多个字符，这个函数的参数是需要发送的字符串的地址和字符串的字符数。字符串没有必要用任何指定字符标识字符串的结束，这使得函数允许任务传送二进制的字符。

如果发送器正在发送字符，那么所需要发送的字符串将不会被送入发送缓冲器中，同时函数将返回一个状态以表明发送器忙。如果当前发送器处于空闲状态，那么该字符串将依据串长计数参数所确定的长度被送入发送缓冲器中，此后发送器将自动开始工作。任务将立即获得一个返回值，表明发送操作成功。任务在 UART 发送字符时不需要等待发送操作的完成，中断系统自动完成所需要的操作。用户必须确保从源字符串地址拷贝到发送缓冲器中的字符的数量是正确的。

以下是一个关于 put_uart_str 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                    /* 头文件 */
byte  put_uart_str( void * , word16);  /* 函数声明 */
unsigned char   *SRC_PTR;               /* 可以是局部变量，也可以是全局变量，表明函数获得字符的
                                           地址。*/
#define    NUMBER    ???                /* 需要放入发送缓冲器的字符数 */
unsigned char    STATUS;                /* 函数返回值 */

STATUS  =  put_uart_str(SRC_PTR , NUMBER);
```

参数传递

SRC_PTR：为传递给这个函数的字符串的驻留在内存中的地址。
NUMBER：是所需要送入发送缓冲器的字符数。

```
unsigned char   src_bytes[]  =  {"From task 1"};   /* 存储着需要被发送的字符的地址，可以是局部变
                                                      量。*/
void   task1(void)
```

```
{
  unsigned char   status;
  status  =  put_uart_str(src_bytes , sizeof src_bytes);/* 开始发送字符，注意第二个参数将决定实际
                                                          上被拷贝入发送缓冲器的字符数。*/
  if (status != AOK)  /* 检测函数返回值 */
    {
      /* 对发送不成功的处理*/
    }
  ……
  /* 或者可以用另外一种格式编写这段程序。*/
  status  =  put_uart_str("From task 1" , 12); /* 开始发送字符，注意第二个参数将决定实际上被拷贝
                                                  入发送缓冲器的字符数。*/
  if (status != AOK)   /* 检测函数返回值 */
    {
      /* 对发送不成功的处理*/
    }
  ……
}
```

返回参数

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

UART_BUSY = Error：因发送器忙而返回。

如果函数的返回状态 STATUS 为 AOK，那么调用这个函数的任务已经把指定数量的字符放入发送器中，同时发送器将自动开始发送字符。

送入发送缓冲器中的字符数量是由变量 NUMBER 所决定的，而不是由字符串的长度所决定。对于用户来说必须保证需要发送的字符数量是正确的。

5.10 put_uart_str_w 函数

put_uart_str_w 函数与上述的函数类似。如果发送器忙，那么这个任务将等待一段指定的时间或者进行无限期的等待。这个函数的参数是所需要发送的字符串的地址，字符串的字符数以及任务为发送器成为空闲状态而愿意等待的时间。这个函数允许占有发送缓冲器的任务发送一个或多个字符。

如果发送器正在发送字符，那么发送任务将被挂起。任务的挂起时间的长短由任务调用函数的第二个参数值决定，该值为 0 表明任务将进行无限期的等待，否则任务将等待指定数量(该数量的取值范围是 0 到 65535)的系统滴答时间。一旦发送器成为空闲状态，那么该挂起任务将自动被唤醒，然后允许它把数据传递给这个函数的计数参数值所确定长度的需要发送的字符串放入发送缓冲器中。任务将立即获得一个返回值，表明操作已经成功。如果在指定的等待时间内发送器未成为空闲，那么该字符串将不会被放入发送缓冲器中，同时任务将获得一个返回值，表明等待时间耗尽。

以下是一个关于 put_uart_str_w 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                                    /* 头文件 */
byte   put_uart_str_w(void * , word16 , word16);    /* 函数声明 * /
unsigned char   *SRC_PTR;     /* 可以是局部变量，也可以是全局变量，表明函数获得字符的地
                                    址。*/
#define   NUMBER          ???  /* 需要放入发送缓冲器的字符数 */
#define   TIME_PERIOD   ???  /* 指定等待的时间 */
unsigned char    STATUS;         /* 函数返回值 */

STATUS  =  put_uart_str_w(SRC_PTR , NUMBER , TIME_PERIOD);
```

参数传递

SRC_PTR：为传递给这个函数的字符串的驻留在内存中的地址。
NUMBER：是所需要送入发送缓冲器的字符数。
TIME_PERIOD：是任务为发送器成为空闲而愿意等待的系统滴答数。这个值的范围从 0 到 65535。

```
unsigned char   src_bytes[] = {"Task1 transmitting this"}; /* 存储着需要被发送的字符的地址，可以是
                                                              局部变量。*/
void   task1(void)
{
   unsigned char   status;
   /* 开始发送字符，如果需要的话，可以为发送器成为空闲而等待 100 个系统滴答时间 */
   status  =  put_uart_str_w(src_bytes , sizeof src_bytes , 100);
   if (status != AOK)      /* 检测返回值*/
      {
        /* 对发送不成功的处理*/
      }
   ……

   /* 或者可以用另外一种格式编写这段程序。*/

   /* 开始发送字符，如果需要的话，可以为发送器成为空闲而等待 100 个系统滴答时间 */
   status  =  put_uart_str_w("Task1 running" , 14 , 100);
   if (status != AOK)  /*检测返回值*/
      {
        /* 对发送不成功的处理*/
      }
   ……
}
```

返回参数

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERTIME = Warning/error：指定等待时间耗尽时发送器依然忙。

UART_BUSY = Error：因发送器忙而返回。

如果函数的返回状态 STATUS 为 AOK，那么调用这个函数的任务已经把指定数量的字符放入发送器中，同时发送器将自动开始发送字符。

5.11 get_uart_chr 函数

get_uart_chr 允许占有接收器的任务在接收缓冲器中检索一个字符，如果接收缓冲器中存在字符的话。get_uart_chr 函数的参数是任务中用于存放所取得的字符的地址。如果接收缓冲器中有字符，那么只有一个字符将被取出并放入目的地址。

如果接收缓冲器是空的，表明没有字符被 UART 所接收或者是已接收所有的字符并都已被检索并被取走，那么将不会有字符被送入目的地址。

这个任务将立即获得一个返回值，以指明获得的字符数。如果返回值为 0，表明没有字符被取回。否则任务将获得一个为 1 的返回值，表明有一个字符被取回。一旦有一个字符被取走，接收缓冲器的尾指针就减 1，同时接收缓冲器的计数变量 count_in 也将减 1。

以下是一个关于 get_uart_chr 函数的例子：

调用

任务。

```
#include <cxfuncs.h>              /* 头文件  */
word16  get_uart_chr( void *);  /* 函数声明 */
unsigned char    *DEST_PTR;   /* 可以是局部变量，也可以是全局变量，指明函数用于存放取回的
                                 字符的地址。*/
unsigned short    COUNT;        /* 指明从缓冲器中取回的字符数 */
COUNT  =  get_uart_chr(DEST_PTR);
```

参数传递

DEST_PTR：是这个函数用来存放取回的字符的地址。

```
/* 该数组最多可以存放接收的 80 个字符，该数组可以是局部变量，在以下的例子中可以将其长度
   设为 1，因为在这种情况下函数至多只能返回一个字符。*/
unsigned char   recv_array[80];

void   task1(void)
{
   unsigned short    count;       /* 声明局部变量 */
   unsigned char     c;
   unsigned char    *recv_ptr;  /* 可以是局部变量也可以是全局变量 */

   recv_ptr  =  recv_array;      /* 将 recv_array 数组的地址值传给指针 */
   count  =  get_uart_chr(recv_ptr);       /* 开始检索字符 */
   if (count)
      {
         c = *recv_ptr++;         /* 获得一个字符同时指针加 1 */
```

```
        ……
      }
    ……

    /* 或者可以用另外一种格式编写这段程序。*/
    count  =  get_uart_chr(&c);       /* 开始检索字符 */
    if (count)
      {
        ……                          /* 处理获得的 c 变量中的字符 */
      }
    ……
}
```

返回值

COUNT：是从 UART 接收缓冲器中传送到目的地址中的字符数。

如果 COUNT 值等于 0，表明当函数被调用时 UART 接收缓冲器中没有字符。如果 COUNT 值不等于 0(对于这个函数来说应为 1)，那么已有一个字符被传送到作为参数传递给这个函数的目的地址中。注意任务将可以把一个字符型变量的地址传递给函数，也可以将这个地址赋予一个指针，然后再将这个指针传递给函数(这也会真正地把指针的值传递给函数)。

5.12 get_uart_chr_w 函数

get_uart_chr_w 函数与上述的函数类似。如果接收缓冲器是空的话，这个函数可以为字符的到达而等待一段指定的时间。如果接收缓冲器中有字符的话，函数将允许占有接收缓冲器的任务从其中检索并取回一个字符。如果接收缓冲器中没有字符，那么任务将等待一段指定的时间。这个函数的参数是任务中用于存放所取得的字符的地址，以及如果缓冲器中没有字符时任务希望等待的时间。这个时间值为 0 表明任务将进行无限期的等待，否则任务将等待一个由该值指定的介于 1 和 65535 之间的系统滴答数。

如果接收缓冲器中有字符，那么只有一个字符将被取出并放入目的地址。任务将立即返回一个为 1 的值表明操作成功。

如果接收缓冲器是空的，表明没有字符被 UART 所接收或者是已接收所有的字符并都已被检索并被取走，那么任务将被挂起，直到指定等待时间耗尽或者接收缓冲器收到一个字符为止。如果在收到一个字符前指定的等待时间耗尽，那么任务将获得一个为 0 的返回值，表明等待时间已耗尽，同时没有字符被取回。

如果当任务挂起时接收器收到一个字符，那么接收器将自动唤醒这个任务，同时将字符拷贝到目的地址中，并返回一个为 1 的值表明操作成功。

每当一个字符被检索而取回，接收缓冲器的尾指针将加 1，同时缓冲器计数变量 count_in 将减 1。

以下是一个关于 get_uart_chr_w 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                          /* 头文件  */
word16   get_uart_chr_w( void * , word16);  /* 函数声明 */
```

```
/* 可以是局部变量，也可以是全局变量，指明函数用于存放取回的字符的地址。*/
unsigned char   *DEST_PTR;
#define   TIME_PERIOD   ??? /* 指定等待的时间 */
unsigned short   COUNT;        /* 指明从缓冲器中取回的字符数 */

COUNT  =  get_uart_chr_w(DEST_PTR , TIME_PERIOD);
```

参数传递

DEST_PTR：是这个函数用来存放取回的字符的地址。

TIME_PERIOD：是这个任务可以为接收到一个字符而愿意等待的系统滴答数。其值范围从 0 到 65535，值为 0 时表明任务将进行无限期的等待。

```
/*该数组最多可以存放接收的 80 个字符，该数组可以是局部变量，在以下的例子中可以将其长度
   设为 1，因为在这种情况下函数至多只能返回一个字符。*/
unsigned char recv_array[80];

void   task1(void)
{
   unsigned short   count;                    /* 声明局部变量 */
   unsigned char   c;
   unsigned char   *recv_ptr;                 /* 可以是局部变量也可以是全局变量 */
   recv_ptr  =  recv_array;                   /* 将 recv_array 数组的地址值传给指针 */
   count  =  get_uart_chr_w(recv_ptr , 20); /*开始检索字符，如果缓冲器中没有字符，任务将为接收
                                               到一个字符而等待 20 个系统滴答时间。*/
   if (count)
      {
        c  =  *recv_ptr++;        /* 获得一个字符同时指针加 1 */
        ……
      }
   ……
   /* 或者可以用另外一种格式编写这段程序。*/
   count  =  get_uart_chr_w(&c , 20);    /* 开始检索字符 */
   if (count)
      {
        ……    /* 处理获得的 c 变量中的字符 */
      }
   ……
}
```

返回参数

COUNT：是从 UART 接收缓冲器中传送到目的地址中的字符数。

如果 COUNT 值等于 0，表明当函数被调用时 UART 接收缓冲器中没有字符，或者任务指定的等待时间耗尽。如果 COUNT 值不等于 0(对于这个函数来说应为 1)，那么已有一个字符被传送到作为参数传递给这个函数的目的地址中。

5.13 get_uart_str 函数

get_uart_str 函数允许占有接收器的任务从接收缓冲器中检索并取回一个或多个字符，假如缓冲器中有足够数目的字符的话。get_uart_str 函数的参数是存放取回的字符串的地址以及希望从接收缓冲器中取回的字符数目。

如果接收缓冲器中有足够数目的字符，那么这些字符将被拷贝到目的地址。如果接收缓冲器中的字符数量不够参数所指定的值，那么将不会有字符被拷贝到目的地址。

任务在调用函数之后将立即得到一个返回值，该返回值指明了实际拷贝到的字符数。如果返回值为 0，表明没有字符被取回。如果有字符被取回，返回值将指明得到的字符数量。每次当一个字符被拷贝到目的地址时，接收器的尾指针将自动加 1，同时接收缓冲器的字符计数变量 count_in 将减 1。

以下是一个关于 get_uart_str 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                    /* 头文件 */
word16  get_uart_str(void , word16);    /* 函数声明 */
unsigned char   *DEST_PTR;              /* 可以是局部变量，也可以是全局变量，指明函数用于存放
                                           取回的字符的地址。*/
#define   NUMBER   ???                  /* 指定需要的字符数 */
unsigned short   COUNT;                 /* 指明从缓冲器中取回的字符数 */

COUNT  =  get_uart_str(DEST_PTR , NUMBER);
```

参数传递

DEST_PTR：是这个函数用来存放取回的字符的地址。

NUMBER：是函数希望从 UART 接收缓冲器中取回的字符数。假如接收缓冲器中有足够数量的字符的话，函数只能得到该指定数目的字符。

```
unsigned char   recv_array[80];
/* 最多可存放 80 个字符，如果需要的话也可以定义为局部变量，当然该数组的大小也可以就是这
   个任务所希望得到的字符的数量。 */

void   task1(void)
{
   unsigned short   count;                    /* 声明局部变量 */
   unsigned char   c;
   unsigned char   *recv_ptr;            /* 可以是局部变量也可以是全局变量 */
   recv_ptr  =  recv_array;              /* 将 recv_array 的地址值赋予指针 */
   count  =  get_uart_str(recv_ptr , 5);  /* 从接收缓冲器中检索 5 个字符 */

   while (count--)
        {
          c  =  *recv_ptr++;           /* 取出一个刚检索回的字符同时指针值加 1 */
```

```
            ……
        }
    ……
}
```

注：如果缓冲器中有足够数目的字符，函数将允许任务立即从 UART 接收缓冲器中取回指定数目的字符。假如字符数量不够，任务不会等待缓冲器接收剩余的字符而直接返回。

返回值

COUNT：是从 UART 接收缓冲器实际拷贝到目的地址的字符数。

如果返回值 COUNT 的值等于 0，表明当函数被调用时 UART 接收缓冲器中没有足够数量的字符。如果 COUNT 值不等于 0，那么这个值表明实际被传递到函数地址参数所指定的目的地址中的字符数。在这种情况下，COUNT 变量的值将与 NUMBER 参数的值相同。

5.14 get_uart_str_w 函数

get_uart_str_w 函数与上述的函数完成的功能基本一致。只是在接收缓冲器中没有足够数目的字符时，此函数可以为缓冲器接收这些字符而等待一段指定的时间。

如果接收缓冲器中有足够数目的字符，则函数允许占有接收器的任务从中检索并取回指定数目的字符。如果接收器中的字符数量不够，那么任务将为接收器接收到足够数目的字符而等待一段指定的时间。

get_uart_str_w 函数的第一个参数是任务用于存放取回的字符串的地址，第二个参数是任务希望从接收缓冲器中取回的字符数目，同时传递给函数的还有当缓冲器中没有足够数目的字符时任务可以等待的时间值。这个值为 0 表明任务将进行无限期的等待，或者是一个介于 1 和 65535 之间的值，指明等待的系统滴答数目。

如果接收缓冲器中有足够数目的字符，那么这些字符将被拷贝到目的地址。任务将立即得到一个返回值，表明任务实际得到的字符数量。

如果接收缓冲器中没有足够数目的字符，任务将被挂起直到而等待的时间耗尽或者直到缓冲器接收到了指定数目的字符。如果在接收缓冲器接收到指定数目的字符之前，任务等待的指定时间已经耗尽，那么函数返回值将被置为 0。这表明发生超时同时任务没有得到任何字符。如果在接收缓冲器接收到指定数目的字符之前，缓冲器接收到了足够数目的字符，则接收器将自动唤醒被挂起的任务。

get_uart_str_w 函数能把指定数目的字符从接收缓冲器拷贝到目的地址中。并且函数将立即返回一个值，表明任务实际得到的字符数量(等于所希望检索回的字符数)，这意味着函数调用成功。

每当一个字符被检索回时，接收缓冲器的尾指针将加 1，同时接收缓冲的字符计数变量 count_in 将减 1。

以下是一个关于 get_uart_str_w 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                                    /* 头文件 */
word16   get_uart_str_w(void * , word16 , word16); /* 函数声明 */
unsigned char   *DEST_PTR;  /* 可以是局部变量，也可以是全局变量，指明函数用于存放取回的字
                                    符的地址。*/
```

```
#define   NUMBER          ???   /* 指定需要的字符数 */
#define   TIME_PERIOD     ???   /* 指定所希望等待的时间 */
unsigned short   COUNT;         /* 指明从缓冲器中取回的字符数 */

COUNT  =  get_uart_str_w(DEST_PTR , NUMBER , TIME_PERIOD);
```

参数传递

DEST_PTR：是这个函数用来存放取回的字符的地址。

TIME_PERIOD：是任务为接收缓冲器接收到足够数量的字符而等待的系统滴答数。其值的范围为从 0 到 65535，当值为 0 时表明任务将进行无限期的等待。

NUMBER：是函数希望从 UART 接收缓冲器中取回的字符数。假如接收缓冲器中有足够数量的字符的话，函数只能得到该指定数目的字符。

```
unsigned char   recv_array[80];
/* 最多可存放 80 个字符，如果需要的话也可以定义为局部变量，当然该数组的大小也可以就是这
   个任务所希望得到的字符的数量。 */
void   task1(void)
{
   unsigned short   count;          /* 声明局部变量 */
   unsigned char   c;
   unsigned char   *recv_ptr; /* 可以是局部变量也可以是全局变量 */
   recv_ptr  =  recv_array;   /* 将 recv_array 的地址值赋予指针 */
   count  =  get_uart_str_w(recv_ptr , 40 , 20);
   /* 从接收缓冲器中检索 40 个字符，如果在接收缓冲器中不够 40 个字符，则任务将可以为缓冲
      器接收到 40 个字符而等待 20 个系统滴答时间。*/
   while (count--)
         {
            c  =  *recv_ptr++;   /* 取出一个刚检索回的字符同时指针值加 1 */
            ……
         }
   ……
}
```

返回值

COUNT：是从 UART 接收缓冲器实际拷贝到目的地址的字符数。

如果返回值 COUNT 的值等于 0，表明当函数被调用时 UART 接收缓冲器中没有足够数量的字符。如果 COUNT 值不等于 0，那么这个值表明实际被传递到函数地址参数所指定的目的地址中的字符数。

5.15 get_uart_str_wg 函数

get_uart_str_wg 函数与上述的函数完成的功能基本一致。不同的只是在等待时间耗尽后，即使缓冲器中没有足够数量的字符，函数也会从接收缓冲器中取回字符。

如果接收器中的字符数量不够，那么任务将为接收器接收到足够数目的字符而等待一段指定的时间。

get_uart_str_wg 函数的第一个参数是任务用于存放取回的字符串的地址，第二个参数是任务希望从接收缓冲器中取回的字符数目，同时传递给函数的还有当缓冲器中没有足够数目的字符时任务可以等待的时间值。这个值可以为 0 表明任务将进行无限期的等待，或者是一个介于 1 和 65535 之间的值，指明等待的系统滴答数目。

如果接收缓冲器中有足够数目的字符，那么这些字符将被拷贝到目的地址。任务将立即得到一个返回值，表明任务实际得到的字符数量。

如果接收缓冲器中没有足够数目的字符，任务将被挂起直到而等待的时间耗尽或者直到缓冲器接收到了指定数目的字符。如果在接收缓冲器接收到指定数目的字符之前，任务等待的指定时间已经耗尽，那么 get_uart_str_wg 函数将会把实际已接收到的一定数目(这个数将小于所希望得到的字符数目)的字符拷贝到目的地址。任务将得到一个返回值，表明实际得到的字符数目。

如果在接收缓冲器接收到指定数目的字符之前，缓冲器接收到了足够数目的字符，则接收器将自动唤醒被挂起的任务。随后 get_uart_str_wg 函数将从接收缓冲器中把一定数目的字符拷贝到参数所指定的目的地址中。任务将立即得到一个返回值，即任务所得到的字符数(这个数与所希望得到的字符数相同)，表明函数调用成功。这是一个功能非常强大的函数，可以实现变长数据包的接收和发送。

每当一个字符被检索回时，接收缓冲器的尾指针将加 1 同时接收缓冲的字符计数变量 count_in 将减 1。

以下是一个关于 get_uart_str_wg 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                                    /* 头文件 */
word16  get_uart_str_wg( void * , word16 , word16 );  /* 函数声明 */
unsigned char   *DEST_PTR;    /* 可以是局部变量，也可以是全局变量，指明函数用于存放取回的
                                 字符的地址。*/
#define   NUMBER          ???   /* 指定需要的字符数 */
#define   TIME_PERIOD   ???   /* 指定所希望等待的时间 */
unsigned short   COUNT;         /* 指明从缓冲器中取回的字符数 */

COUNT  =  get_uart_str_wg(DEST_PTR , NUMBER , TIME_PERIOD);
```

参数传递

DEST_PTR：是这个函数用来存放取回的字符的地址。

TIME_PERIOD：是任务为接收缓冲器接收到足够数量的字符而等待的系统滴答数。其值的范围为从 0 到 65535，当值为 0 时表明任务将进行无限期的等待。

NUMBER：是函数希望从 UART 接收缓冲器中取回的字符数。

```
unsigned char   recv_array[80];
```

```
/* 最多可存放 80 个字符，如果需要的话也可以定义为局部变量，当然该数组的大小也可以就是这
   个任务所希望得到的字符的数量。 */
void   task1(void)
{
   unsigned short   count;            /* 声明局部变量 */
   unsigned char   c;
   unsigned char   *recv_ptr;   /* 可以是局部变量也可以是全局变量 */
   recv_ptr  =  recv_array;     /* 将 recv_array 的地址值赋予指针 */
   count  =  get_uart_str_wg(recv_ptr , 40 , 20);
   /* 从接收缓冲器中检索 40 个字符，如果在接收缓冲器中不够 40 个字符，则任务将可以为缓冲
      器接收到 40 个字符而等待 20 个系统滴答时间。若等待的指定时间耗尽，那么函数将会把缓
      冲器中实际已接收到的一定数目的字符取回。*/
   if (count  <  40)
      {
        ……/* 检测取回的字符数是否就是所希望得到的数量，若不是则采取适当的操作。 */
        while (count--)
                {
                  c  =  *recv_ptr++;  /* 取出一个刚检索回的字符同时指针值加 1 */
                  ……
                }
      }
   ……
}
```

返回值

COUNT：是从 UART 接收缓冲器实际拷贝到目的地址的字符数。

如果返回值 COUNT 等于 0，表明当等待时间耗尽时，UART 接收缓冲器中没有字符。如果 COUNT 值不等于 0，那么这个值表明实际被传递到函数地址参数所指定的目的地址中的字符数，该值要么是所希望得到的指定的字符数或者是小于该数的一个值，表明在等待时间耗尽时接收缓冲器中存在的字符数量。

5.16 get_uart_str_v 函数

get_uart_str_v 函数允许一个任务从 UART 接收缓冲器中检索回所有已经接收到的字符，该函数不进行任何超时操作，可见任何时候调用该函数的任务都不会等待。任务得到的返回值是实际被拷贝到由函数参数指定目的地址的字符数。

任务可以先调用 get_uart_recv_cnt 函数来确定接收缓冲器中的字符数目。(这样在调用 get_uart_recv_cnt 函数实际得到的字符数就会与调用 get_uart_recv_cnt 函数得到的字符数基本一致，当然可能在 get_uart_str_v 函数执行前又有新的字符被接收。)

这是一个非常有用的函数。一个接收任务调用 cxtwatm 函数来无限期的等待数据的到来。当 UART 中断确定 UART 已接收到确定数目的字符(变长的数据包)时，中断就可以使用 cxtwake 函数来唤醒被挂起的任务。当挂起任务恢复运行时，它就可以使用 get_uart_str_v 函数从缓冲器中取回并处理所需要的字符。当然除了使用 cxtwatm 函数实现之外接收任务还可能是处于空闲态，在这种情况下中断就应该使用 cxttrig 函数触发并开始这个任务。

以下是一个关于 get_uart_str_v 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                /* 头文件 */
word16   get_uart_str_v( void * );  /* 函数声明 */
unsigned char   *DEST_PTR;        /*可以是局部变量，也可以是全局变量，指明函数用于存放取回
                                    的字符的地址。*/
unsigned short   COUNT;           /* 指明从缓冲器中取回的字符数 */

COUNT  =  get_uart_str_v(DEST_PTR);
```

参数传递

DEST_PTR：是这个函数用来存放取回的字符的地址。

```
unsigned char   recv_array[80];
/* 将要用来存放取回字符的目的地址的大小应该和接收缓冲器一样大，这样函数就不会把字符存
   放到其它错误的内存地址中。*/
void   task1(void)
{
   unsigned short  count;           /* 声明局部变量 */
   unsigned char   c;
   unsigned char  *recv_ptr;  /* 可以是局部变量也可以是全局变量 */

   recv_ptr  =  recv_array;    /* 将 recv_array 的地址值赋予指针 */
   count  =  get_uart_str_v(recv_ptr);
   /* 调用函数检索当前在 UART 接收缓冲器中的所有字符。*/
   while(count--)
        {
          c = *recv_ptr++;        /* 取出一个刚检索回的字符同时指针值加 1 */
          ……
        }
   ……
}
```

这个函数非常有用，因为接收任务可以处于空闲或者以无限期等待的方式来等待数据的到来，这样接收中断在接收到变长的数据包后可以分别调用 cxttrig 函数或者 cxtwake 函数来通知任务可以从缓冲器中取走字符。

返回值

COUNT 是从 UART 接收缓冲器实际拷贝到目的地址的字符数。

如果返回值 COUNT 为 0，表明当函数执行时 UART 接收缓冲器中没有字符。如果 COUNT 不为 0，那么函数执行时接收缓冲器中所有的字符都已经被拷贝到由函数参数所指定的目的地址中。

5.17 get_uart_recv_cnt 函数

任务可以使用 get_uart_recv_cnt 函数来获知当前在 UART 接收缓冲器中的字符数量。
以下是一个关于 get_uart_recv_cnt 函数的例子：

调用
任务。

```
#include <cxfuncs.h>                    /* 头文件 */
unsigned short   COUNT;                 /* 指明从缓冲器获知的字符的数量 */
COUNT  =  get_uart_recv_cnt();
```

参数传递

该函数没有参数。

```
Void  task1(void)
{
   count  =  get_uart_recv_cnt();    /* 查询当前在 UART 接收缓冲器中的字符数。*/
   /* 现在可以调用某个函数从接收缓冲器中取回一个或多个字符。*/
   ……
}
```

在这个函数返回 COUNT 值指明接收缓冲器中当前的字符数量之后，接收任务可以根据情况接收一个或多个字符。

返回值

COUNT 值表明任务调用这个函数时，接收缓冲器中当前的字符数量。
如果返回值 COUNT 的值为 0，表明当函数被调用时 UART 接收缓冲器中没有任何字符。如果 COUNT 值不为 0，那么这个值就是当前在接收缓冲器中的字符数量。

6、内存管理函数

内存管理函数是 CMX 库的一部分，它允许用户创建一个尺寸固定的内存池，可以从一个内存池中取出一个空闲块，也可以把一个内存块释放回到内存池中。内存管理函数如下：
cxbfcre 、cxbfget 、cxbfrel

6.1 cxbfcre 内存池创建函数

这个函数用于建立一个内存池。注意这个函数不检测内存争用问题。该函数认为内存池中的内存是空闲的且能够被自由地使用，同时认为这些内存除了内存操作函数可以使用外，不会被别的代码所使用。
在调用 cxbfcre 函数时需要提供以下的参数，第一个参数是这个内存池的开始地址，第二个参数是这个内存池中内存块的大小，最后一个参数是内存池中包含内存块的数目。
如果用户正在使用的 CPU 要求整数或指针必须存放在与内存奇地址或内存偶地址对齐的位置，那

么内存块的起始地址也应该放在相应的对齐位置，同时内存块的大小必须是偶数。这是因为指针会被维持在每个内存池中下一个空闲内存块所驻留的地址。

当创建内存池时，需要一个额外的由字符指针的大小所决定的字节数。例如，某用户确定每个内存块的大小需要 50 字节，共需内存块的数目为 10 个，另外还需一个 2 字节的字符指针。那么这个内存池一共需要 502 个字节。根据上面所指出的大小声明一个无符号字符的数组，那么这个数组的起始地址就是内存池的起始地址。用户可以创建任意多的内存池，只要提供给内存池的内存保证不会被其它操作所使用。

用户必须根据 CPU 的要求来安排内存池的位置，以使内存池的起始地址开始于可以存储整数的正确的存储边界。因为当调用 cxbfget 函数时，传递给指针的地址必须遵循 CPU 的存储边界要求。同样，内存块的大小也需要遵循 CPU 对内存地址安排的要求。

例如，Intel 公司的 80196 处理器规定整数必须驻留在偶数边界的地址处，这表明内存的使用必须从一个偶数地址开始，同时内存块的尺寸必须为 2 的倍数。另外，其它一些处理器可能要求内存块的尺寸必须为 4 的倍数。

CMX 要求内存缓冲池小于 64K 字节。另外用户可以自由地创建内存池结构，但必须确保内存池的地址开始于正确的 CPU 地址边界。

CMX 认为所提供的参数一定是正确的，并且在任何情况下都不会检测参数的正确性。当内存池创建以后，用户就可以从这个内存池中获取或释放内存块。

以下是一个关于 cxbfcre 函数的例子：

调用

进入 RTOS 之前，任务。

```
#include <cxfuncs.h>                        /* 头文件 */
void   cxbfcre( void * , word16 , word16 );   /* 函数声明 */

#define   BLK_SIZE          ???             /* 定义每个内存块的大小 */
#define   NUM_BLOCKS   ???                  /* 定义内存块的数目 */
struct {
        unsigned char    *dummy_ptr;        /* 为 CMX 分配空间 */
        /* 为这个内存池分配足够的内存*/
        unsigned char   pool_bytes[BLK_SIZE  *  NUM_BLOCKS]; /* BLK_SIZE  *  NUM_BLOCKS
                                                                不可超过 64Kbyte */
    }MEM_POOL1;  /* 这是内存池 1 */

void   cxbfcre(&MEM_POOL1 , BLK_SIZE , NUM_BLOCKS);
```

参数传递

&MEM_POOL1：是这个内存池驻留在内存中的起始地址。
BLK_SIZE：是内存池中每个内存块所需的字节数，其最大值为 255 个字节。
NUM_BLOCKS：是在这个内存池中大小固定的内存块的数量，最大值为 65535。

```
#define   BSIZE        10   /* 若 CPU 对指针的存放有限制，那么该值应该是两个相邻的存储边界之
                               间尺寸的倍数。*/
#define   NBLOCKS   20  /* 内存池中的内存块数 */
```

```
struct { /* 该结构声明必须在进入 RTOS 之前进行 */
          unsigned char   *head_ptr;
          unsigned char   body[BSIZE  *  NBLOCKS];
       } MEM_POOL1;

void   task2(void)
{
   cxbfcre(MEM_POOL1 , BSIZE , NBLOCKS);
   /* 建立一个有 20 个固定尺寸为 10 字节的内存池 */
}
```

返回值

该函数没有任何状态返回值。

注意：用户必须确保这个内存池中有足够的内存，并且函数不检测内存竞争问题。如果处理器要求指针必须驻留在特定的存储边界上，如偶地址边界，那么这个内存块的尺寸必须为偶数。这是因为 CMX 将指针放置在空闲的内存块里作为内部使用。

6.2 cxbfget 获取内存块函数

如果内存池里存在空闲内存块，那么 cxbfget 函数将从内存池中得到一个固定的空闲内存块。这个函数有两个参数，第一个参数是希望从中得到内存块的内存池的起始地址。用户需要把与传递给 cxbfcre 函数用于创建内存池相同的起始地址传递给函数。第二个参数是用于存放得到的内存块起始地址的指针地址。

cxbfget 函数将确定在指定内存池中是否有空闲、并且可用的内存块，如果存在这样的内存块，那么函数将把这个内存块的起始地址返回给指针。当得到内存块起始地址时，用户必须知道只有 cxbfcre 函数创建内存池时定义的内存大小范围内的内存块才是可用的。

同时用户必须确保在指定内存池中对内存块的使用，不会超过创建内存池时指定的内存块的尺寸，否则会导致内存被破坏。如果这个指定内存池中所有的内存块都已被使用，并且没有内存块将被释放，用户会得到这个内存池中没有空闲内存块的通知，并且传递给指针的地址是无效的。

假如有一个内存块是可以使用的，那么这些内存块所占的内存将是连续的，而不会是由一些碎片段组成，但是其中可能包含一些先前被使用时所留下的垃圾。当用户确定对某内存池中的内存块的使用已经完成时，用户也可以将其释放回内存池中，这些内容将在下一小节中描述。

以下是一个关于 cxbfget 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                  /* 头文件 */
byte  cxbfget ( void * , byte *) ;    /* 函数声明 */

struct{/* 在 cxbfcre 函数之前建立 */
          unsigned char   *dummy_ptr;
          unsigned char   pool_bytes[BLK_SIZE  *  NUM_BLOCKS];
```

```
    } MEM_POOL1;              /* 这是内存池 1 */
unsigned char   *BLOCK_ADDR; /* 内存块指针，可以是局部变量也可以是全局变量。*/
unsigned char   STATUS;           /* 声明局部变量 */

STATUS  =  cxbfget(&MEM_POOL1 , &BLOCK_ADDR);
```

参数传递

&MEM_POOL1：是这个内存池驻留在内存中的起始地址。
&BLOCK_ADDR：是无符号字符的指针地址，用于存放得到的内存块起始地址。

```
#define   NBLOCKS   20   /* 内存池中的内存块的数量 */
#define   BSIZE        10   /* 若 CPU 对指针的存放有限制，那么该值应该是两个相邻的存储边界之
                             间尺寸的倍数。*/
struct   {
           unsigned char   *head_ptr;
           unsigned char    body[BSIZE * NBLOCKS];
         } MEM_POOL1;

void  task2(void)
{
   unsigned char   status;
   unsigned char   *block_user_ptr;       /* 根据用户的意愿创建一个指针 */
   unsigned char   *block_ptr;         /* 创建另一个指针用于存放返回的内存地址到 block_ptr 变量中,
                                          该地址也用于 cxbfrel 函数中。 */
   status  =  cxbfget(&MEM_POOL1 , &block_ptr);
   /* 用空闲内存块的内存地址位置装载指针 */
   if (status = = AOK)
      {
        block_user_ptr  =  block_ptr;   /* 保存内存块的起始地址，因为 block_ptr 中的值可能会被破
                                           坏。*/
      }
   else
      {
        /* 出错处理 */
      }
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERR = Error：在这个内存池中没有空闲块。
如果 STATUS 的值为 AOK, 那么 BLOCK_ADDR 中将包含这个内存块的起始地址。为了避免该指针中的起始地址被破坏而将这个起始地址拷贝到另一个指针中，但是当释放这个内存块时又需要把这

个起始地址重新取回。

6.3 cxbfrel 内存块释放函数

cxbfrel 函数将一个固定大小的内存块释放回指定的内存池中。使用该函数需要提供曾经从中取出这个内存块的内存池起始地址，以及调用 cxbfget 函数时得到的内存块自身的起始地址。这个地址必须与取出这个内存块时得到的起始地址的相一致。

当释放一个内存块时，该内存块将成为空闲块、并加入到曾经从中检索出空闲内存块的特定内存池中的空闲块组中。当一个内存块被释放时，该内存块的存储区也会被再次释放，并且内存块的内容将被破坏，变成无效。

提供给上述函数的地址不会被检测并且始终假定它们都是正确的。如果所提供的地址发生错误，内存就会被破坏并会导致严重的后果。

以下是一个关于 cxbfrel 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                    /* 头文件 */
void  cxbfrel ( void  * , byte  *);     /* 函数声明 */
struct   {  /* 在 cxbfcre 函数之前建立 */
            unsigned char   *dummy_ptr;
            unsigned char   pool_bytes[BLK_SIZE  *  NUM_BLOCKS];
         } MEM_POOL1;  /* 内存池 1 */
unsigned char   *BLOCK_ADDR; /* 内存块指针，可以是局部变量也可以是全局变量。*/

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

参数传递

&MEM_POOL1：是这个内存池驻留在内存中的起始地址。

BLOCK_ADDR：是 BLOCK_ADDR 地址中的内容，它包括由 cxbfget 函数检索得到的内存块起始地址。

```
#define  NBLOCKS   20  /* 内存池中的内存块的数量 */
#define  BSIZE       10   /* 若 CPU 对指针的存放有限制，那么该值应该是两个相邻的存储边界之
                             间尺寸的倍数。 */

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

void   task2(void)
{
   unsigned char   *block_user_ptr;       /* 根据用户的意愿创建一个指针 */
```

```
    unsigned char   *block_ptr;         /* 创建另一个指针用于存放返回的内存地址到 block_ptr 变量中,
                                           该地址也用于 cxbfrel 函数中。 */
    /* 以下是处理 block_ptr 的用户代码。 */
    cxbfrel(&MEM_POOL1 , block_ptr);   /* 完成对该内存块的操作，将其释放回内存池 。*/
}
```

返回值

该函数没有任何返回值。

注意：确保传递给这个函数的地址与调用 cxbfget 函数所检索到的地址相一致，函数不检测这个地址的正确性。

7、消息管理函数

消息管理函数是 CMX 库中的一部分，它允许在任务间传递消息。消息管理函数如下：

cxmssend、cxmssenw、cxmsget、cxmswatm、cxmsack、cxmsbxev

传递给邮箱的仅仅是消息的地址而不是消息本身的内容。这使得当用 cxmssend 函数或 cxmssenw 函数发送消息、或者用 cxmsget 函数或 cxmswatm 函数从邮箱中检索消息时，消息传递变得非常快，因为真正的消息数据并没有被拷贝到邮箱中。在 RTOS 配置模块中可以配置所有邮箱的消息总数。用户需要为存储消息地址而准备一定的内存，但这些内存实际上不属于任何邮箱。

当把消息传递给邮箱时，消息内存(一个消息块)就会被给予指定的邮箱(建立关联)。CMX 在启动过程中早已创建所有的消息块并已把它们结构化，这使得消息发送和接收函数能够迅速地得到一个消息块来存储消息地址以及其它一些 CMX 数据项。同时传递给某指定邮箱的消息将排列成基于先进先出的队列。当所有的消息块都已经被分配给邮箱时，那么再有额外的消息时，就不能进行消息发送了。记住，邮箱中可以有一个以上的消息。

用户需要在 RTOS 配置模块中设置邮箱的最大数量。只有任务才能够拥有邮箱的所有权，但每个任务可能拥有不止一个邮箱，然而对于从邮箱中检索消息的任务却只能拥有一个邮箱。任务和中断能够自由地把消息发送给任何邮箱。

当一个任务确定它不再需要某特定邮箱时，其它的任务可以声明对那个邮箱的所有权。只有当使用了 cxmsbxev 函数后，这种邮箱所有权转换才能成立。当这个邮箱中有消息时，函数(cxmsbxev)会通过设置一个特定任务(指接收任务)的事先已经定义的事件来通知邮箱(以启动一个指定的接收任务)。如果这个函数没有被使用或者所有权发生了变动、使得这个任务的事件再也不会被设置，那么只要没有其它任务在等待这个邮箱，在任何时候任何任务都可以使用任何邮箱。

7.1 cxmssend 消息发送函数

这个函数允许任务和中断向一个邮箱发送消息。注意这个消息本身并没有被传送，只是消息的地址被传送给了邮箱。只要传送者和接收者遵循相同的格式，那么消息的内容可以是任何事物。这一点是极其有用的，比如中断可以发送一个用于识别端口引脚的状态消息。一个任务可能拥有多个邮箱，在这种情况下需要设置一个优先级配置方案以使高优先级的消息发送到 1 号邮箱，较低优先级的消息被发送到 2 号邮箱，最低优先级的消息被发送到 3 号邮箱。

当 cxmssend 函数被一个任务或中断调用时，调用者必须提供将把消息发送到的邮箱号。除此之外，调用者还必须提供驻留在内存中的消息内容的地址。

由于邮箱收到的仅仅是消息的地址，因此消息的长度可以是任意长。如果在调用这个函数的时候至少有一个消息块是空闲的，那么邮箱还可以接收其它消息。每个邮箱可以接收的消息数并没有限

制，唯一的限制是必须有空闲的消息块可以给予该邮箱。一旦某个消息块被给予了给一个邮箱，则这个消息块对于所有其它的邮箱来说是不可用的，直到此消息被传送给了一个任务。

调用发送消息函数的任务或中断将被立即返回(要测试是否有比发送任务更高优先级的，正在等待该邮箱中消息的任务，否则不等待发送任务而将继续执行)。然而如果一个正在等待邮箱中消息的任务的优先级高于消息发送任务的优先级，那么调度程序将会被通知进行一次迅速的任务切换。在这种情况下，消息发送任务或者是先于中断运行的任务将会被挂起，而等待消息的任务将成为新的当前运行任务。

如果 cxmsbxev 函数已被激活用来设置占有这个邮箱的任务的事件，并且当消息到来时邮箱是空的(先前的消息都已发送完毕，已被其它任务取走)，那么 cxmssend 函数将自动设置拥有这个邮箱的任务的特定的事件。

以下是一个关于 cxmssend 函数的例子：

调用

任务，中断。

中断可以调用这个函数，但只能是间接调用。有关具体细节请参阅特定处理器的章节。

```
#include <cxfuncs.h>                        /* 头文件 */
byte  cxmssend( byte , void * );            /* 函数声明 */
#define  MBOX1   ???                        /* 定义用于识别特定邮箱的数字标识 */
unsigned char   SOURCE_BYTES[] =  ??? /* 既可以是全局变量，也可以是局部变量。 */
unsigned char   STATUS;                     /* 对任务来说该变量必须是局部变量 */

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

参数传递

MBOX1：是任务将把消息发送到的邮箱号。这个值的范围从 0 到邮箱配置中的最大数目减 1。

SOURCE_BYTES ：是消息内容驻留在内存中的地址，该地址将被拷贝到邮箱中消息指针处。

```
#define   MBOX1    1
void  task2(void)
{
   unsigned char   status;
   status  =  cxmssend(MBOX1 , "This message for mailbox 1\n");
   /* 把消息发送到 1 号邮箱，记住被传递的是消息的地址而不是其内容。*/
   if (status != AOK)      /* 检测返回状态值以确认调用是否成功 */
      {
       /* 检查出错原因，并且采取适当的措施。 */
      }
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERR = Error：没有可用的消息块或者邮箱号非法。

如果 STATUS 值等于 AOK, 那么该消息已经被放入邮箱的消息块中。注意每个邮箱都是以先进先出的队列方式来工作的。

7.2 cxmssenw 消息发送函数

这个函数与 cxmssend 函数基本上是一样的，除了一点就是调用 cxmssenw 函数的任务还将传递另外一个参数，该参数表明此任务愿意为来自接收任务 cxmsack 函数的响应而愿意等待的时间。如果该时间值是 0，该任务将进行无限期的等待，并且该值最大可达 65535。调用该函数的任务将会被挂起，直到此消息被某个任务从邮箱中取走并调用 cxmsack 函数给以应答，或者直到该任务等待的指定时间耗尽。

这就好似对调用任务的应答。因为当消息接收任务调用 cxmsack 函数时， 因调用 cxmssenw 函数而挂起的任务将自动变为就绪态。如果任务等待的指定时间耗尽，那么消息发送任务会自动被唤醒并准备恢复运行。当这个任务变为当前运行任务时，它就会知道它是因为等待时间耗尽被唤醒的而不是被调用 cxmsack 函数的接收任务唤醒的。

只有任务才可以调用这个函数。如果接收这个消息的任务没有运行或者在其检索下一个消息前未曾调用 cxmsack 函数，那么这个消息的发送任务将不会运行除非任务指定了一个非零的等待时间值并且该时间已经耗尽。cxtwakf 函数也可以用来强制唤醒这种任务。

以下是一个关于 cxmssenw 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                    /* 头文件 */
byte  cxmssenw(byte , word16 , void *);         /* 函数声明 */

#define   MBOX1      ???                /* 定义用于识别特定邮箱的数字标识 */
#define   TIME_CNT  ???                 /* 需要等待的时间或者是 0 代表进行无限期等待 */
unsigned char   SOURCE_BYTES[]  =  ???  /* 既可以是全局变量，也可以是局部变量。*/

unsigned char   STATUS;                 /* 对任务来说该变量必须是局部变量 */

STATUS  =  cxmssenw(MBOX1 , TIME_CNT , SOURCE_BYTES);
```

参数传递

MBOX1：是任务把消息发送到的邮箱号。此数值的范围从 0 到邮箱配置中的最大数目减 1。

TIME_PERIOD：是这个任务愿意为 cxmsack 函数来唤醒它而等待的系统滴答数。此值为 0 表示任务将进行无限期的等待。该值可以从 0 到 65535。

SOURCE_BYTES 是消息内容驻留在内存中的地址，该地址将被拷贝到邮箱中消息指针处。

```
#define   MBOX1    1
unsigned char   mesg1[]  =  {"hello task 1\n"};
void   task2(void)
{
```

```
    unsigned char   status;
    status  =  cxmssenw(MBOX1 , 100 , mesg1);
    /* 将这个消息发送到 1 号邮箱，注意传递给邮箱的是消息的地址而不是消息的内容。同时为
       cxmsack 函数的应答而愿意等待长达 100 个系统滴答。 */
    if (status = = AERR) /* 检测返回状态值以确认调用是否成功*/
       {
         /* 检查出错原因，并且采取适当的措施 。*/
       }

    if (status = = AERTIME)
       {
         /* 指定的等待时间在 cxmsack 函数发回应答前已经耗尽，并采取适当的操作 。*/
       }
    ……/* 消息接收任务发回正确的应答在指定的等待时间耗尽之前 */
}
```

返回参数

STATUS ：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERR = Error：没有可用的消息块或者邮箱号非法。

AERTIME = Warning：在 cxmsack 函数被用来唤醒这个任务前时间值耗尽。

如果 STATUS 值为 AOK，那么一个消息已被放入邮箱中并且某任务接收了这个消息同时调用了 cxmsack 函数通知发送任务消息已经收到。

7.3 cxmsget 消息接收函数

这个函数允许调用它的任务从邮箱中检索一条消息，只要这个邮箱中有任何消息的话。只有任务可以调用 cxmsget 函数。

传递给这个函数的唯一参数是邮箱标识号。如果函数调用时，邮箱中没有任何消息，那么函数将返回一个空指针(Null)以表明没有消息。如果邮箱中至少有一条消息，那么首条消息的地址将被返回给调用者，同时包含这条消息地址的消息块将自动返回到空闲消息块队列中并被标记为空闲状态。

一个任务可以拥有多个邮箱，同时它能够从其中任何一个邮箱中接收消息。如果想对消息进行按优先次序的分类，这种方式是非常有用的。

当一个邮箱将消息地址传递给任务时，消息接收任务将自动获知消息发送任务(因为消息发送任务使用了 cxmssenw 函数)。随后消息接收任务可以调用 cxmsack 函数通知消息发送任务以表明它已收到消息。消息接收任务可以在任何时候调用 cxmsack 函数发出确认信息，只要在从任何邮箱中检索另一个消息之前调用即可。

如果某任务在调用 cxmsget 函数时指定了一个与其它任务在调用 cxmswatm 函数时使用的标识号相同的邮箱，并且此时那个任务正在等待那个邮箱中的消息，那么该任务将获得一个空的返回值。换句话说，在任何时候决不允许多个任务同时等待一个邮箱。

不管邮箱中是否有消息，cxmsget 函数都会迅速的返回。消息将按照它们被邮箱接收的先后顺序，即按照先进先出的方式被检索出。

以下是一个关于 cxmsget 函数的例子：

调用

任务。

```
#include <cxfuncs.h>            /* 头文件。 */
void   * cxmsget(byte);         /* 函数声明 */
#define    MBOX1    ???         /* 定义用于识别特定邮箱的数字标识 */
unsigned char   *RECV_PTR;    /* 既可以是全局变量，也可以是局部变量。*/
RECV_PTR  =  cxmsget(MBOX1);
```

参数传递

MBOX1 是任务把消息发送到的邮箱号。此数值的范围从 0 到邮箱配置中的最大数目减 1。

```
#define   MBOX1   1
void   task1(void)
{
   unsigned char   *recv_ptr;                  /* 创建一个用于接收消息地址的指针 */
   recv_ptr =  cxmsget(MBOX1);                /* 如果邮箱中有消息，则直接将其检索回*/
   if (recv_ptr != (unsigned char *)NULL)     /* 检测返回的指针变量值是否为 NULL */
      {
        /* 处理接收到的消息 */
      }
}
```

返回值

RECV_PTR ：是用于存放指向消息的地址指针。

如果 RECV_PTR 为 NULL (0), 说明当任务调用这个函数时时，邮箱中没有消息。如果 RECV_PTR 是一个非零值，那么这个值就是存放消息的地址。

7.4 cxmswatm 等待接收消息函数

cxmswatm 函数完成的操作与 cxmsget 函数基本相同，但 cxmswatm 函数还可以使得任务在邮箱中没有消息时等待一段指定的时间或者进行无限期的等待，以希望在这期间获得消息。

函数有两个参数，第一个参数是希望从中检索到消息的邮箱标识号，第二个参数是当邮箱中没有消息时任务愿意等待的时间。

cxmswatm 函数使得任务可以在调用该函数时邮箱中没有消息的情况下允许将自身挂起，并且该任务会因为某些情况而自动返回到就绪态。这种情况有两种，其一是某消息到达该邮箱，其二是任务指定的等待时间耗尽。

第二个参数指定的时间是任务用于等待消息的系统滴答数。该值的范围可以从 0 到 65535。如果该值取为 0 则表明这个任务将会无限期地等待消息的到达。

以下是一个关于 cxmswatm 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                /* 头文件 */
void  * cxmswatm(byte , word16);  /* 函数声明 */
#define  MBOX1        ???          /* 定义用于识别特定邮箱的数字标识 */
#define  TIME_CNT   ???
unsigned char   *RECV_PTR;       /* 既可以是全局变量，也可以是局部变量。*/

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

参数传递

MBOX1：是任务把消息发送到的邮箱号。此数值的范围从 0 到邮箱配置中的最大数目减 1。

TIME_CNT ：是用于等待消息的系统滴答数。该值的范围从 0 到 65535。如果该值取为 0 则表明这个任务将会无限期地等待消息的到达。

```
#define    MBOX1    1

void   task1(void)
{
   unsigned char   *recv_ptr;                  /* 创建一个用于接收消息地址的指针 */
   recv_ptr  =  cxmswatm(MBOX1 , 100); /* 如果邮箱中有消息，则直接将其检索回，否则将为消息
                                                到达邮箱而最多等待 100 个系统滴答的时间。*/
    if (recv_ptr != (unsigned char *)NULL)  /* 检测返回的指针变量值是否为 NULL */
       {
          /* 处理接收到的消息 */
       }
 }
```

返回值

RECV_PTR：是用于存放指向消息的地址指针。

如果 RECV_PTR 为 NULL (0), 说明在消息到达邮箱之前任务的等待时间已经耗尽或者是邮箱标识号参数非法。如果 RECV_PTR 是一个非零值，那么这个值就是存放消息的地址。

如果 cxmswatm 函数的返回值为空，则可能出现以下的情况之一：邮箱中没有消息、邮箱标识号超出范围而非法或者已经有其它任务正在这个邮箱中等待消息的到来。

7.5 cxmsack 消息接收应答函数

在使用 cxmsget 或者 cxmswatm 函数检索回消息以后，任务需要调用 cxmsack 函数来给消息发送任务发一个确认信号并同时唤醒消息发送任务。只有任务才可以调用这个函数。

如果消息发送任务使用的消息发送函数是 cxmssend 函数，那么消息接收任务没有必要发回一个确认信号因为消息发送任务对此并不关心，虽然此任务可以调用这个函数。从 cxmsack 函数的返回值可以得知消息发送任务并没有在等待确认。

消息接收任务必须在检索下一个消息或者结束运行之前调用这个函数以唤醒被挂起的消息发送任务(因为调用了 cxmssenw 函数)。否则消息发送任务将会一直被挂起直到此任务的等待时间耗尽(如果指定的等待时间为非 0 值)或者用户使用了 cxtwakf 函数来强制唤醒它。

只要任务接收到消息，消息接收任务就应该调用 cxmsack 函数以确保消息发送任务不会被挂起。

如果能够确定消息发送任务并没有使用 cxmssenw 函数来发送消息，那么消息接收任务也可以不调用 cxmsack 函数。

以下是一个关于 cxmsack 函数的例子：

调用

任务。

```
#include <cxfuncs.h>       /* 头文件 */
byte  cxmsack(void);       /* 函数声明 */
unsigned char   STATUS; /* 该变量必须是局部变量。*/
STATUS  =  cxmsack();
```

参数传递

该函数不需要任何参数。

```
#define  MBOX1    1

 void   task1(void)
 {
    unsigned char   *recv_ptr;          /* 创建一个用于接收消息地址的指针 */
    recv_ptr  =  cxmsget(MBOX1);   /* 如果邮箱中有消息，则直接将其检索回。 */
    if (recv_ptr != (unsigned char *)NULL) /* 检测返回的指针变量值是否为 NULL */
       {
          /* 处理接收到的消息 */
          cxmsack();  /* 因为消息发送任务使用 cxmssenw 函数发送消息，所以需要调用该函数来唤
                         醒发送消息的任务 */
       }
 }
```

返回值

STATUS ：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERNOWAIT = Warning：消息发送任务并不在等待来自消息接收任务的确认。

如果 STATUS 值为 AOK，那么一个消息已被放入邮箱中并且某任务接收了这个消息同时调用了 cxmsack 函数通知发送任务消息已经收到。

7.6 cxmsbxev 消息事件设置函数

在调用了这个函数之后，消息接收任务便会与邮箱之间建立了一定的相关关系，当指定的邮箱中有消息时，邮箱会自动通知此任务。当邮箱中存在或者接收到一个消息时，邮箱将自动使用 cxesig 函数来设置消息接收任务的特定的事件位。这使得任务可以用等待事件的方式来等待消息。当一个消息到达邮箱时，邮箱便设置消息接收任务的某事件位以表明邮箱中已有消息。

调用这个函数需要传递三个参数。第一个参数就是邮箱标识号，该值的范围将从 0 到在配置文件中设置的最大的邮箱数减 1。

第二个参数是某任务的槽号，该任务可以是 cxtcre 函数建立的任何任务只要这个任务没有被

cxtrmv 函数移去。如果该任务槽号的参数取值为 0，那么邮箱不会设置任何任务的事件位。

第三个参数是当邮箱中出现消息时，将被设置的指定任务的事件位。cxmsbxev 函数是这样工作的。当消息到达空邮箱时，则邮箱将自动调用 cxesig 函数。cxesig 函数将使用传递给 cxmsbxev 函数的事件位参数。当然 cxesig 函数必须使用模式 0，以保证只有指定任务的事件位被设置。

当任务使用 cxmsget 函数或者 cxmswatm 函数检索回消息以后，如果邮箱中还有消息那么邮箱还会设置该任务的事件位。

这样使得任务可以等待各种各样的事件和邮箱。当某任务使用该函数等待一个或者多个邮箱时，任务便会知道有消息到达邮箱或者邮箱中有许多的消息。最好将任务用于等待事件的 cxewatm 函数设置成自动清除方式，以便事件发生后函数会自动清除这些事件位。当任务恢复运行时，它将检索到所需要的消息。当任务检索回消息以后，如果邮箱中还有消息，那么邮箱还会再次自动地设置这些事件位。

邮箱的参数可以在任何时候被修改。如果希望其它任务占有这个邮箱或者修改当消息出现时邮箱需要设置的事件位，那么这种功能将是非常有用的。当然也可以指定任务的槽号参数为 0，这会使消息到来时邮箱不设置任何事件位。

注意，只要没有任务在等候这个邮箱种的消息，任何任务都可以从该邮箱检索可用的消息。CMX 规定一个任务只可以等待一个邮箱的消息，这就是为什么当邮箱中有消息时 cxmsbxew 函数只设置一个任务的事件位。

以下是一个关于 cxmsbxev 函数的例子：

调用

进入 RTOS 之前，任务。

```
#include <cxfuncs.h>                    /* 头文件 */
byte  cxmsbxev(byte , byte , word16);   /* 函数声明 */

#define    MBOX1 ???            /* 定义用于识别特定邮箱的数字标识 */
unsigned char  TASK_SLOT;   /* 该变量必须是全局变量 */
#define   EVENT   ???           /* 需要设置的事件位 */
unsigned char  STATUS;        /* 函数的返回状态值 */

STATUS  =  cxmsbxev(MBOX1 , TASK_SLOT , EVENT);
```

参数传递

MBOX1：是任务把消息发送到的邮箱号。此数值的范围从 0 到邮箱配置中的最大数目减 1。
TASK_SLOT： 是当邮箱中有消息时需要设置事件位的任务的槽号。
EVENT：是指定任务的将被设置的事件位，它是一个无符号的 16 位的变量或常量。

```
unsigned char  task1_slot;   /* 包括 cxtcre 函数所返回的任务 1 的槽号 */
#define  MBOX1      1
#define  MBOX2      2
/* 定义当消息出现在邮箱 1 中时，cxmsbxev 函数将设置的事件位。 */
#define  TSK1_MB1_FLAG   0x0080
/* 定义当消息出现在邮箱 2 中时，cxmsbxev 函数将设置的事件位。 */
#define  TSK1_MB2_FLAG   0x0040
```

```
void  task1(void)
{
   unsigned char  status;
   unsigned short  event_bits;       /* 存放 cxewatm 函数的返回值，表明已被设置的事件位 */
   unsigned char  *recv_ptr;         /* 创建一个用于接收消息地址的指针 */
   status  =  cxmsbxev(MBOX1 , task1_slot , TSK1_MB1_FLAG);
   /* 当邮箱 1 中有消息时，邮箱 1 将设置任务 1 的事件位为 TSK1_MB1_FLAG (位 7)*/
   status  =  cxmsbxev(MBOX2 , task1_slot , TSK1_MB2_FLAG);
   /* 当邮箱 2 中有消息时，邮箱 2 将设置任务 1 的事件位为 TSK1_MB2_FLAG (位 6)*/
   while(1)
        {
          event_bits  =  cxewatm(TSK1_MB1_FLAG  || TSK1_MB2_FLAG , 0 , 2);
          /* 这将强制任务 1 进行无限期地等待直到邮箱 1 或者邮箱 2 中出现消息。当出现消息而
             使任务恢复时，cxewatm 函数会自动清除已所设置的事件位。如果任务的邮箱中还有
             消息或者又有消息到达时，那么任务检索回消息后，任务的事件位将再次被设置。*/
          if (event_bits  &  TSK1_MB1_FLAG)
             {
               recv_ptr  =  cxmsget(MBOX1); /* 从邮箱 1 中直接检索消息 */
               /* 注意，当任务的邮箱中出现消息的时侯，该邮箱将会自动设置此任务的事件位
                  (TSK1_MB1_FLAG)。*/
                  ……/* 消息处理 */
               /* 如果消息发送任务使用了 cxmssenw 函数，则需要做如下的工作。*/
               cxmsack();
               /* 唤醒消息发送任务因为它使用了 cxmssenw 函数 */
             }
          if (event_bits  &  TSK1_MB2_FLAG)
             {
               recv_ptr  =  cxmsget(MBOX2); /* 从邮箱 2 中直接检索消息 */
               /* 注意，当任务的邮箱中出现消息的时侯，该邮箱将会自动设置此任务的事件位
                  (TSK1_MB2_FLAG)。*/
               ……/* 消息处理 */
               /* 如果消息发送任务使用了 cxmssenw 函数，则需要做如下的工作。*/
               cxmsack();
               /* 唤醒消息发送任务因为它使用了 cxmssenw 函数。 */
             }
        }/* 不断循环处理来自邮箱 1 和邮箱 2 的消息。 */
}
```

返回值

STATUS ：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERR = Error：因为邮箱标识号超出范围而非法。

注意，当第一个消息到达邮箱时，邮箱将自动设置任务的事件位，如果这个邮箱中有许多的消息，那么在任务每次检索回消息时其事件位也会被设置。这个函数可以多次调用以指定多个不同的事件和(或)任务。

8、资源管理函数

资源管理函数是 CMX 库中的一部分，它包括的函数如下所列：

cxrsget、cxrsrsv、cxrsrel

在任何时候这些函数仅仅允许一个任务可以访问某个特定的资源。如果当任务访问某资源时发现该资源已被别的任务占有，这个任务将会挂起等待一段指定的时间或者无限期地等待该资源。当然任务发现资源不可访问时也可以不挂起自身。

CMX 在资源上具有优先级的继承性。当某资源被其所有者释放时最高优先级的任务将占有该资源。这一点将在以后详细介绍。用户在编写和调试应用程序时需要记住这个属性。

优先级继承性是指拥有资源的当前任务的优先级将会暂时提升到与处于正在等待该资源的优先级最高的任务的优先级相同。当该任务释放了该资源以后，它的优先级又会恢复为原先在占有该资源之前的级别。

当某资源被当前的资源占有者释放后，等待该资源的优先级最高的任务将获得该资源。这个过程根本不考虑该任务与其它在等待该资源的任务间的申请资源先后顺序。这意味着 CMX 的资源管理并不象其它 RTOS 那样建立在先申请先获得的机制上，而是依靠等待该资源的任务间的优先级关系来确定哪个任务将占有该资源。

8.1 cxrsget 获取资源函数

任务可以使用 cxrsget 函数来申请某个特定的资源。调用这个函数时，任务需要提供所申请资源的标识号。该数值的范围从 0 到配置模块中设置的资源最大数量值减 1。

如果某资源空闲并且没有被任何其它任务所占用，那么任务在调用了这个函数之后会立即得到这个资源。如果该资源已被其它任务所占用，那么调用这个函数的任务将会立即得到一个返回值表明资源已被占用，但该任务并不挂起。

如果所申请的资源已被其它任务占用，那么调用 cxrsget 函数的任务既不会被阻塞，也不会挂起等待该资源再次成为空闲态。当然如果一个任务通过调用 cxrsget 函数得到了某资源，那么当它使用完该资源后必须调用 cxrsrel 函数以释放该资源使得其它任务能够获得这个资源。

以下是一个关于 cxrsget 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                /* 头文件 */
byte  cxrsget(byte);                /* 函数声明 */
#define   RESOURCE_NUM   ???  /* 用于识别资源的标识号 */
unsigned char  STATUS;              /* 该变量必须是局部变量 */

STATUS  =  cxrsget(RESOURCE_NUM);
```

参数传递

RESOURCE_NUM ：是任务希望获得的资源的标识号。

```
#define   RESOURCE1    1
void  task2(void)
{
   unsigned char   status;
   status  =  cxrsget(RESOURCE1);
   /* 任务 2 将资源 1 的标识号作为参数调用该函数来申请该资源。如果现在资源 1 空闲，那么任
      务 2 将立即获得资源 1，这一点通过检查返回值可以得知。如果现在资源 1 已经被其它任务
      所占用，那么任务 2 可通过返回值知道这个情况但任务不会被挂起。*/
   if (status = = AOK) /* 确认函数调用是否成功 */
      {
        /*此时任务 2 占用资源 1，当它完成必要的操作后不再需要该资源时必须通过调用 cxrsrel
           函数来释放资源 1。*/
      }
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERRESOWNED = Error：所申请的资源已被其它任务所占有而出错返回。

AERR = Error：资源标识号超出范围而出错返回。

如果返回值 STATUS 为 AOK，那么调用该函数的任务已经获得了该资源。如果 STATUS 的值是 AERRESOWNED 或是 AERR，那么任务对该资源的申请失败。

用户需要检查返回值才能得知任务是否已经获得了该资源。如果任务发现已经获得了该资源，那么任务就可以访问这个特定的资源。否则，任务就不应该访问这个资源，因为它已经被其它任务所占有。如果出现两个任务同时访问同一个资源的现象，那么也就会出现资源竞争的现象甚至会使整个系统崩溃。

8.2 cxrsrsv 等待获取资源函数

这个函数完成的操作与 cxrsget 函数基本相同。不同之处在于当任务调用 cxrsrsv 函数申请资源时，如果所申请的资源已经被其它任务所占用，那么这个任务将会被挂起直至资源再次空闲。

资源申请等待队列以先进先出的方式工作。这意味着调用这个函数的任务如果发现资源已被占用那么该任务将被放入资源的等待队列中。当资源变为空闲时，队列中的第一个任务将获得这个资源。

被挂起并被放入资源等待队列中的任务在资源变为空闲并且允许该任务获得此资源时，任务将自动成为恢复态，准备恢复运行。

如果在任务所等待的资源变为空闲之前，该任务指定的等待时间已经耗尽，那么任务将自行醒来并恢复运行，通过检查函数的返回值任务可以得知它是因为等待时间耗尽且没有得到所申请的资源而醒来的。

任务通过调用这个函数而获得了某资源，在任务使用完该资源之后必须要使用 cxrsrel 函数来释放这个资源，以使其它任务能够获得这个资源的使用权。

以下是一个关于 cxrsrsv 函数的例子：

调用

任务。

```
#include <cxfuncs.h> /* 头文件*/
byte  cxrsrsv(byte , word16);          /* 函数声明 */
#define   RESOURCE_NUM    ???    /* 用于识别资源的标识号 */
#define   TIME_PERIOD       ???    /* 任务等待资源变为空闲的时间 */
unsigned char   STATUS;              /* 该变量必须是局部变量 */

STATUS  =  cxrsrsv(RESOURCE_NUM , TIME_PERIOD);
```

参数传递

RESOURCE_NUM：是任务希望获得的资源的标识号。

TIME_PERIOD：是任务等待指定资源的系统滴答数，其范围从 0 到 65535。如果此值取为 0，那么任务将无限期地等待该资源。

```
#define   RESOURCE1    1
void  task2(void)
{
   unsigned char  status;
   status  =  cxrsrsv(RESOURCE1 , 100);
   /*任务 2 将资源 1 的标识号作为参数调用该函数来申请该资源。如果资源 1 现在空闲，那么任
      务 2 将获得这个资源，该函数的返回值将表明这一点。如果资源已经被其它任务占用，那么
      任务 2 将会被挂起等待 100 个系统滴答的时间。如果在等待时间耗尽前资源变为空闲并且该
      任务是资源等待队列中的第一个任务，那么任务将获得这个资源。如果任务的等待时间耗尽
      了，那么该任务将会被从资源等待队列中移去，并且任务可以从其返回值得知它是因为等待
      时间耗尽而被唤醒的，这说明该任务对此资源的申请已失败。*/
   if (status = = AOK) /* 确认函数调用是否成功 */
      {
         /*此时任务 2 占用资源 1，当它完成必要的操作后不再需要该资源时必须通过调用 cxrsrel
         函数来释放资源 1。*/
       }
}
```

返回参数

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERTIME = Error：因任务的等待时间耗尽而使函数返回。

AERR = Error：资源标识号超出范围而出错返回。

如果返回值 STATUS 为 AOK，那么调用该函数的任务已经获得了该资源。如果 STATUS 的值是 AERTIME 或是 AERR，那么任务对该资源的申请失败。

用户需要检查返回值才能得知任务是否已经获得了该资源。只有确认该任务是唯一可以访问某资源的任务，占有资源的任务才可以使用这个资源。任务也必须保证在使用完资源后要使用 cxrsrel 函数

来释放该资源。

8.3 cxrsrel 释放资源函数

任务可以使用 cxrsrel 函数来释放其已经占有的资源。只有已经占有一个资源的任务才能够释放其所占有的资源。因为一个任务可能拥有不止一个的资源，所以任务在调用这个函数时需要指定被释放的资源的标识号。

用户必须确保一个任务在没有释放完其占用的所有资源之前该任务不可以结束。当任务调用其它会把自身挂起很长时间的函数时一定要考虑仔细，因为还有其它任务在等着访问此资源。

以下是一个关于 cxrsrel 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                /* 头文件  */
byte   cxrsrel(byte);               /* 函数声明 */
#define   RESOURCE_NUM    ???  /* 用于识别资源的标识号 */
unsigned char   STATUS;             /* 该变量必须是局部变量 */

STATUS  =  cxrsrel(RESOURCE_NUM);
```

参数传递

RESOURCE_NUM：是任务希望获得的资源的标识号。

```
#define   RESOURCE1   1
void  task2(void)
{
   unsigned char   status;
   /* 任务 2 占用资源 1 并且已经不再需要这个资源，它将释放资源 1。*/
   status  =  cxrsrel(RESOURCE1); /* 任务 2 正在释放资源 1 */
   if (status != AOK) /* 确认函数调用是否成功 */
      {
         /*如果该函数调用失败，任务需要采取适当的补救措施。一般不会在这个地方出错，除非
         在调用 cxrsget 函数或 cxrsrsv 函数申请资源后没有检查它们的返回值以确认任务是否真的
         申请到了资源。*/
      }
}
```

返回值

STATUS ：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERRNOWN = Error：任务并没有占有该资源而出错返回。
AERR = Error：资源标识号超出范围而出错返回。

如果 STATUS 值为 AOK，那么任务已经释放了指定的资源。

占有资源的任务必须要释放其所有的资源之后才可以结束运行。如果任务准备调用 cxtrmv(任务删除)函数，那么任务必须释放其占有的资源，否则在等待这个资源的等待队列中的所有任务将会被永远地挂起直至它们的等待时间耗尽。

任务必须要检测函数的 STATUS 返回值字节，以确定资源是否已经被释放。如果返回值 STATUS 是一个错误值，那么表明在应用程序代码中存在编程上的错误。如果某任务成功地释放了其所占用的资源并且在这个资源的等待队列中的下一个任务的优先级比该任务的优先极要高，那么会立即发生重新调度，这个等待的任务将变为当前运行任务。

9、信号量管理函数

信号量管理函数是 CMX 库的一部分，包括以下函数：

cxsminit、cxsempd、cxsempdw、cxsempst、cxsemfsh

这些函数允许一个或多个任务在任何时候访问一个特定的信号量。一个任务或中断都可以增加某信号量的计数。信号量的计数增加后，如果有任务正在等待这个信号量，那么这个任务将被恢复，同时信号量计数值将减少。

一个任务可以与一个信号量相关联，即任务使用相关函数调用申请某信号量。如果此时关联操作中指定的信号量的计数值为 0，那么函数可能立即返回并指出信号量计数值为 0，或者函数调用任务被无限期地挂起等待该信号量或指定一段时间来等待该信号量。如果某信号量的计数值大于 0，那么该信号量的计数值将被减少，并且返回一个状态表明任务现在已经申请到该信号量。

信号量有很多用途。第一个用途是单个任务可以把一个信号量作为计数器使用。每次当一个信号量被释放时，信号量计数器将加一。一个任务可以在某信号量的等待队列中等待该信号量。一旦该信号量被任务或中断所释放，这个正在等待的任务将会进入恢复态准备运行。这个任务恢复运行后可以执行一些操作和循环直到该信号量计数值减少到 0。

信号量的另一个用途就是可以使用信号量的数量来代替其所依附的特定实体的数量。假设某信号量计数器的初始值为 2，即有两个任务可以成功地获得这个信号量。如果第三个任务试图获得这个关联操作中指定的信号量，那么函数可能立即返回并指出信号量不可用，或者任务被无限期地挂起直到有一个任务释放了该信号量，或者任务将一直等待直到等待时间耗尽。

9.1 cxsminit 信号量初始化函数

这个函数用于初始化一个信号量。cxsminit 函数需要两个参数，第一个参数是信号量的标识号，该参数的取值范围是 0 到配置模块中定义的信号量的最大数量。第二个参数是信号量计数器的初始值。这个计数器是一个无符号的短整型变量，即它的取值范围可以从 0 到 65535。

以下是 cxsminit 函数的一个示例：

调用

进入 RTOS 之前，任务。

```
#include <cxfuncs.h>              /* 头文件 */
byte   cxsminit(byte , word16);  /* 函数声明*/
#define   SEM_NUM      ???    /* 用于识别信号量的标识号 */
#define   SEM_COUNT   ???   /* 定义信号量计数器的初始值 */
unsigned char  STATUS;         /* 该变量必须是局部变量 */
STATUS  =  cxsminit(SEM_NUM , SEM_COUNT);
```

参数传递

SEM_NUM：是该函数需要初始化的信号量标识号。该值的取值范围是从 0 到配置模块中定义的信号量的最大数量减 1。

SEM_COUNT：是信号量计数器的初始值。该值的取值范围在 0 到 65535 之间。

```
#define   SEM1    1            /* 信号量 1 */
#define   SEM1_CNT    0     /* 定义信号量计数器的初始值为 0 */
void  task1(void)
{
   unsigned char   status;
   status  =  cxsminit(SEM1 , SEM1_CNT);
   /* 使用初始信号量计数值 0 来设置信号量 1 */
   if (status != AOK) /* 检查函数调用是否出错，正常情况下不应该出错 */
      {
        /* 如果发生了错误则需要采取正确的处理。一般情况下该函数调用不会出错，除非调用该
           函数时使用了超过信号量最大数量的信号量标识号，该最大值由用户在配置模块中定
           义。
      }
}
```

返回值

STATUS ：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERR = Error：信号量标识号超出范围而出错返回。

9.2 cxsempd 信号量申请函数

任务可以使用 cxsempd 函数来申请使用某个特定的信号量。任务需要提供信号量的标识号。该值的范围是从 0 到配置模块中设置的信号量最大数量减 1。

如果信号量计数器大于 0，那么任务将得到该信号量。如果信号量计数器值为 0，调用该函数的任务将会立即得到一个表示信号量不可用的返回值。调用该函数的任务不会被挂起等待该信号量。如果任务通过调用该函数申请到了该信号量，当它使用完信号量后需要调用 cxsempst 来释放这个信号量。

以下是一个关于 cxsempd 函数的例子：

调用

任务。

```
#include <cxfuncs.h>          /* 头文件 */
byte  cxsempd(byte);          /* 函数声明*/
#define   SEM_NUM    ???   /* 用于识别信号量的标识号 */
unsigned char    STATUS;     /* 该变量必须是局部变量 */
STATUS  =  cxsempd(SEM_NUM);
```

参数传递

SEM_NUM：是该函数需要初始化的信号量标识号。该值的取值范围是从 0 到配置模块中定义的信号量的最大数量减 1。

```
#define   SEM1   1
void  task2(void)
{
   unsigned char   status;
   status  =  cxsempd(SEM1);
   /* 任务 2 把信号量 1 的标识号作为参数调用 cxsempd 函数来申请信号量 1。如果信号量 1 没有
      被其它任务所占有，那么任务 2 将获得信号量 1 并可由函数返回值得知这一点。如果所申请
      的信号量已经被其它任务所占有，那么任务 2 将得到一个说明这种情况的返回值，但任务 2
      不会被挂起。*/
   if (status = = AOK) /* 检查函数调用是否成功 */
      {
        /* 任务 2 现在占用信号量 1，当任务不再需要该信号量时需要调用 cxsempst 函数来释放该
           信号量。*/
      }
}
```

返回值

STATUS ：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERRNOSEMA = Error：信号量已经被其它任务所占有而出错返回。
AERR = Error：信号量标识号超出范围而出错返回。
如果 STATUS 的值为 AOK，那么任务已经申请到并占有了该信号量。如果 STATUS 的值为 AERRNOSEMA 或 AERR，那么任务对该信号量的申请失败。

用户必须要检查函数的返回值以确认对该信号量的申请成功与否。如果函数调用成功，那么该任务就可以访问这个信号量。如果函数调用失败，该任务就不可以访问这个信号量，因为其它任务早已占有该信号量。多个任务同时操作一个信号量会使它们之间发生冲突甚至导致系统的崩溃。

9.3 cxsempdw 信号量等待申请函数

这个函数完成的操作与 cxsempd 函数完成的操作基本相同。唯一的不同之处在于调用这个函数的任务可以指定一段时间来等待这个信号量。当这段时间耗尽后，任务会自动恢复并被告知等待时间已经耗尽时信号量还是不可用。

信号量等待队列以先进先出的方式工作。调用这个函数的任务如果发现信号量不可用，那么任务会被放入到该等待队列中。当信号量变为空闲状态时，此队列中的第一个任务将获得该信号量。

当信号量变为空闲状态时，如果某被挂起的任务被允许占有该信号量，那么该任务将会自动进入恢复态准备运行。

如果一个任务调用了该函数后获得了该信号量，那么当它使用完信号量后需要调用 cxsempst 来释

放这个信号量。

以下是一个关于 cxsempdw 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                    /* 头文件 */
byte  cxsempdw(byte , word16);    /* 函数声明*/
#define   SEM_NUM        ???      /* 用于识别信号量的标识号 */
#define   TIME_PERIOD   ???   /* 定义任务用于等待该信号量的时间 */
unsigned char    STATUS;          /* 该变量必须是局部变量 */
STATUS  =  cxsempdw(SEM_NUM , TIME_PERIOD);
```

参数传递

SEM_NUM：是该函数需要初始化的信号量标识号。该值的取值范围是从 0 到配置模块中定义的信号量的最大数量减 1。

TIME_PERIOD：任务用于等待该信号量的系统滴答的数量，取值范围为 0 到 65535。如果该值取为 0，那么任务将为该信号量进行无限期的等待。注意使用等待时间为 0 的参数调用 cxsempdw 函数与调用 cxsempd 函数并不相同。

```
#define    SEM1   1
void   task2(void)
{
   unsigned char   status;
   status  =  cxsempdw(SEM1 , 100);
   /* 任务 2 把信号量 1 的标识号作为参数调用 cxsempd 函数来申请信号量 1。如果信号量 1 没有
      被其它任务所占有，那么任务 2 将获得信号量 1 并可由函数返回值得知这一点。如果所申请
      的信号量已经被其它任务所占有，那么任务 2 将被挂起 100 个系统滴答时间。如果信号量在
      定时时间耗尽前变为空闲状态并且此任务是等待队列中的第一个任务，那么该任务将自动获
      得这个信号量。如果任务等待的定时时间耗尽了，那么此任务会被从该信号量的等待队列中
      移去并得到一个说明等待时间耗尽时信号量还是不可用的返回值，该任务对此信号量的申请
      失败。*/
   if (status = = AOK) /* 检查函数调用是否成功 */
      {
        /* 任务 2 现在占用信号量 1，当任务不再需要该信号量时需要调用 cxsempst 函数来释放该
           信号量。*/
      }
}
```

返回值

STATUS ：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERTIME = Error：任务的等待时间耗尽而出错返回。

AERR = Error：信号量标识号超出范围而出错返回。

如果 STATUS 的值为 AOK，那么任务已经申请到并占有了该信号量。如果 STATUS 的值是其它值，那么任务对该信号量的申请失败。

用户必须要检查函数的返回值以确认对该信号量的申请成功与否。如果函数调用成功，那么该任务就可以访问这个信号量。如果函数调用失败，该任务就不可以访问这个信号量，因为其它任务早已占有该信号量。多个任务同时操作一个信号量会使它们之间发生冲突甚至导致系统的崩溃。

9.4 cxsempst 信号量释放函数

任务可以使用 cxsempst 函数来释放已经占有的信号量。任务在调用这个函数的时候需要指定希望释放的信号量的标识号，因为一个任务可以同时拥有多个信号量。

任务和中断可以把 cxsempst 函数当作计数器使用。例如，每次当一个数据包到达串口的时候，中断可以增加信号量的计数。另一个正在等待该信号量的任务就会醒来并且处理这个数据包。

必须保证任务在结束之前要释放其拥有的所有信号量。还有当调用其它一些会使任务挂起一段很长时间的函数时需要慎思，因为可能有其它任务正在急着等待访问该信号量。

以下是一个关于 cxsempst 函数的例子：

调用

任务，中断。

```
#include <cxfuncs.h>            /* 头文件 */
byte   cxsempst(byte);          /* 函数声明*/
#define   SEM_NUM   ???   /* 用于识别信号量的标识号 */
unsigned char   STATUS;     /* 该变量必须是局部变量 */
STATUS  = cxsempst(SEM_NUM);
```

参数传递

SEM_NUM 是该函数需要初始化的信号量标识号。该值的取值范围是从 0 到配置模块中定义的信号量的最大数量减 1。

```
#define   SEM1   1
void  task2(void)
{
   unsigned char  status;
   status  =  cxsempd(SEM1); /* 申请信号量 1 */
   /* 其它应用程序代码 */
   /* 任务 2 占用信号量 1 并且它已经使用完该信号量，现在需要释放该信号量。*/
   status  =  cxsempst(SEM1); /* 任务 2 正在释放信号量 1 */
   if (status != AOK) /* 检查函数调用是否出错，正常情况下不应该出错 */
      {
         /* 如果发生了错误，则需要采取某些正确的操作。一般情况下该函数调用不会出错，除非
            在调用 cxsemget 或 cxsempdw 函数时没有测试函数的返回值以确定任务是否已经真正申
            请到了该信号量。*/
```

```
    }
}
```

返回值

STATUS ：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERR = Error：信号量标识号超出范围而出错返回。
如果 STATUS 的值为 AOK，那么任务已经释放了该信号量。

拥有信号量的任务必须保证在结束运行之前调用这个函数来释放它所拥有的信号量。任务在调用 cxtrmv 函数之前首先要释放其所有的信号量。否则，在该信号量等待队列中的任务将会被一直挂起知道它们的等待时间耗尽为止。

用户必须要检查函数的返回值以确认对该信号量的释放操作成功与否。如果返回的 STATUS 中是一个错误代码，那么说明应用程序执行过程中出现了错误。如果任务成功地释放了该信号量，并且在信号量等待队列中的下一个任务的优先级比当前任务的优先级要高，那么将会立即发生任务的重新调度使得该任务成为运行态的任务。

9.5 cxsemfsh 信号量重置函数

这个函数可以用于重置一个指定的信号量。调用了这个函数以后，信号量等待队列将被清空并且信号量计数器将被重置为 cxsminit 函数在初始化信号量时所指定的计数值。调用 cxsemfsh 函数需要两个参数。

第一个参数是信号量的数目。该值的取值范围是从零到一个小于在配置模块中所指定的信号量的最大数目的值。

第二个参数是信号量的重置模式。重置模式为 0 意味着信号量在被任务占用的情况下，信号量将不会被重置。重置模式值大于 0 意味着无论信号量是否被任务所占用，信号量都将被重置。

以下是一个关于 cxsemfsh 函数的例子：

调用

任务。

```
#include <cxfuncs.h>                /* 头文件 */
byte   cxsemfsh(byte , byte);       /* 函数声明*/
#define  SEM_NUM          ???   /* 用于识别信号量的标识号 */
#define   FLUSH_MODE   ???   /* 指明在信号量被占用的情况下是否要重置信号量 */
unsigned char   STATUS;          /* 该变量必须是局部变量 */
STATUS  =  cxsemfsh(SEM_NUM , FLUSH_MODE);
```

参数传递

SEM_NUM：是该函数需要初始化的信号量标识号。该值的取值范围是从 0 到配置模块中定义的信号量的最大数量减 1。

FLUSH_MODE ：决定在信号量被任务占用的情况下，信号量是否需要被重置。该模式值为 0 则表明只有当信号量没有被任务所占用的情况下才会被重置。其余的模式值表明无论在什么情况下信号

量都将被重置。

```
#define   SEM1    1
void  task2(void)
{
   unsigned char   status;
   status  =  cxsemfsh(SEM1 , 0);
   /* 任务 2 使用模式值 0 来重置信号量 1。如果没有任务占有这个信号量，那么它将被重置并函
      数返回值为 AOK。如果有某个任务占有信号量 1，那么函数将返回 AERR 的返回值同时该信
      号量未被重置。*/
   if (status = = AOK) /*检查函数调用是否成功*/
      {
        /* 信号量 1 的等待队列已被清空，并且信号量计数器将重置为 cxsminit 函数所指定的值。
        */
      }
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：
AOK = Good：该函数调用成功正常返回。
AERR = Error： 信号量的标识号超出范围，或当重置模式值为 0 时，信号量被某任务所占有。
如果 STATUS 值为 AOK，那么信号量已经被重置。其它任何返回值将说明信号量没有被重置。

10、循环定时器管理函数

循环定时器管理函数是 CMX 库中的一部分，它包括下列函数：
cxctcre、cxctstt、cxctrsti、cxctrstc、cxctrstt、cxctstp
循环定时器管理函数可以在指定数量的系统滴答时间后使用定时器来自动执行事件触发函数。当然也可以把定时器设置为以指定的时间只工作一次或者循环不停地执行。
所有的循环定时器都基于 CMX 时间任务之上运行。CMX 时间任务是优先级最高的任务。循环定时器的执行顺序为从 0 号定时器到标识号最大的定时器只要该循环定时器处于运行状态，这个最大的标识号在配置文件中由用户设置。
使用循环定时器就可以按指定的时间间隔不断地设置某事件。任务创建定时器时需要告诉定时器要触发的事件以及 cxesig 函数执行的方式。任何一个任务都可以在任何时候修改该事件和函数执行方式。当然任务在启动一个循环定时器时需要指定定时器的首次定时时间和循环定时时间。任务和中断都可以改变一个循环定时器的首次定时时间和循环定时时间，或者重新开始一个已经停止运行的循环定时器，或者停止一个循环定时器。

10.1 cxctcre 循环定时器创建函数

这个函数用于创建一个循环定时器。调用这个函数必须要有四个参数。
第一个参数是循环定时器的标识号。这个标识号的取值范围从 0 到配置模块中设置的循环定时器的最大数目减 1。
第二个参数是当循环定时器的定时时间耗尽时 cxesig 函数的执行方式(MODE)。这个参数的取值

范围从 0 到 6，表明循环定时器调用 cxesig 函数的方式。

第三个参数可以是任务的槽号或者是任务的优先级，这个参数将提供给 cxesig 函数使用，当然这个参数具体是什么还需要由提供给 cxesig 函数的执行方式参数决定。由第二个参数所选择的函数执行方式将决定这个参数是任务的槽号，还是任务的优先级，还是一个无效的参数。

第四个参数是循环定时器调用 cxesig 函数执行时需要设置的事件位。这也与所选择的 cxesig 函数的工作方式有关。

用户需要仔细阅读事件管理函数中详细描述 cxesig 函数的那一节内容。因为当循环定时器的定时时间耗尽时，它就会自动调用 cxesig 函数，所以用户应该完全理解 cxesig 函数的使用以便能正确地设置 cxctcre 函数的四个参数。

循环定时器的用途很多。它可以使任务之间以固定的时间间隔进行同步，或者触发一个任务通知它需要改变某端口的引脚状态。一个中断也可以不断地重新开始某个循环定时器的首次定时时间。如果在这个指定的时间内中断没有发生，那么循环定时器将会发生超时并被告知在指定的时间内中断没有发生。

以下是一个关于 cxctcre 函数的例子：

调用

进入 RTOS 之前、任务。

```
#include <cxfuncs.h>                        /* 头文件 */
byte  cxctcre(byte , byte , byte , word16); /* 函数声明 */
#define   CYCLIC_NUM   ???          /* 用于识别循环定时器的标识号 */
#define   MODE         ???          /* 定义 cxesig 函数的工作方式 */
#define   EVENT        ???          /* 定义需要设置的事件位 */

unsigned char   TASK_PRI;       /*与所选择的 cxesig 函数的工作方式有关，可以是指定的任务，
                                  或者是任务的优先级，还可以是一个无效的值。*/
unsigned char   STATUS;         /* 该变量必须是局部变量 */

STATUS  =  cxctcre(CYCLIC_NUM , MODE , TASK_PRI , EVENT);
```

参数传递

CYCLIC_NUM ：是这个函数需要创建的循环定时器的标识号，取值范围从 0 到事先定义的循环定时器数目最大值减 1，该最大值由用户在配置模块中设置。

MODE：是 cxesig 函数的工作方式。

TASK_PRI ：是一个由所选择的 cxesig 函数的工作方式决定的值，它可以是任务的槽号、任务的优先级还可以是一个无效的参数值。

EVENT ：是在循环定时器运行过程中调用 cxesig 函数时需要设置的事件位。

```
#define   TMR1              1       /* 循环定时器 1 */
#define   TMR1_MODE         0       /* 定义 cxesig 函数的工作方式。这个值取为 0 表明 cxesig 函数
                                       将要操作某个特定的任务。*/
#define   TSK2_TM1_EVT   0x0002 /* 定义当循环定时器的定时时间耗尽时需要设置的事件位。 */
unsigned char   task2_slot;         /* 该变量中包含了任务 2 的任务槽号 */
```

```
void  task1(void)
{
   unsigned char   status;
   status  =  cxctcre(TMR1 , TMR1_MODE , task2_slot , TSK2_TM1_EVT);
   /* 创建循环定时器 1，并当它的定时时间耗尽时将自动设置任务 2 事件位中的位 1。*/
   if (status != AOK) /* 检查函数调用是否成功 */
      {
        /* 如果函数调用失败，那么任务需要采取合适的操作。一般任务调用该函数不会失败，除
           非调用者使用的定时器的标识号超过了在配置模块中所声明的最大的定时器数。 */
      }
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERR = Error：循环定时器的标识号超出范围而出错返回。

如果 cxctcre 函数的返回值为 AOK 表明函数调用成功，如果为 AERR 则是因为循环定时器的标识号超过范围而出错返回。注意 cxctcre 函数不会检测由调用者提供的任务槽号是否有效，只有当循环定时器调用 cxesig 函数时，cxesig 函数才会测试这个参数有效性。

10.2 cxctstt 循环定时器初始化 / 启动函数

这个函数用来初始化并启动一个已经由 cxctcre 函数创建的已设置好的循环定时器。只有任务才可以调用这个函数。调用这个函数时需要提供所想要启动的循环定时器的识别号。这个标识号的取值范围从 0 到配置模块中设置的循环定时器的最大数目减 1。

同时还需要提供在循环定时器开始循环定时之前执行的首次定时时间，该值以系统滴答的数目方式给出。这个值的取值范围在 0 到 65535 之间，如果把首次定时时间值取为 0 则系统实际上默认这个值为 65536。

每次发生系统滴答，循环定时器非 0 的定时时间都会减少一个单位。当这个定时时间减少到 0 时，则循环定时器调用 cxesig 函数。用户提供的定时器的首次定时时间只能在循环定时器的第一次执行中被使用，之后循环定时器将以指定的循环定时时间自动执行定时，除非这个循环定时时间的值为 0，表明这个循环定时器只执行一次定时，即只执行首次定时时间。

提供给 cxctstt 函数的最后一个参数是定时器将使用的循环定时时间。在首次定时时间耗尽后，循环定时时间将自动被载入循环定时器中。这个值的取值范围在 0 到 65535 之间。如果这个值取值为 0，则循环定时器将不会载入循环定时时间，循环定时器将停止运行。在这种方式下，循环定时器为只执行一次定时的定时器，这种方式也称之为*一箭*方式。如果这个循环定时时间不为 0，则这个值将会被循环定时器自动载入并不断被系统滴答减少。当这个值减少到 0 时循环定时器将使用由 cxctcre 函数所指定的循环定时器事件作为参数来调用 cxesig 函数。

用户可以在任何时候调用 cxctstt 函数来重新覆盖循环定时器中剩余的时间并重新开始循环定时。当然新的循环定时时间只有在新的首次定时时间耗尽后才能生效。

以下是一个关于 cxctstt 函数的例子：

调用

任务、中断、进入 RTOS 之前。

中断也可以调用这个函数，但只能是间接调用。有关间接函数调用的内容请参阅处理器详细说明

章节。

```
#include <cxfuncs.h>                    /* 头文件 */
byte  cxctstt(byte , word16 , word16);  /* 函数声明 */
#define   CYCLIC_NUM       ???        /* 用于识别循环定时器的标识号 */
#define   INITIAL_PERIOD   ???        /* 指定首次定时时间 */
#define   CYCLIC_PERIOD    ???        /* 指定循环定时时间 */
unsigned char   STATUS;               /* 该变量必须是局部变量 */
STATUS  =  cxctstt(CYCLIC_NUM , INITIAL_PERIOD , CYCLIC_PERIOD);
```

参数传递

CYCLIC_NUM：是任务需要启动的循环定时器的标识号。这个标识号的取值范围从 0 到配置模块中设置的循环定时器的最大数目减 1。

INITIAL_PERIOD ：是会被循环定时器立即载入其时间计数器中的系统滴答数。该值随每次系统滴答的发生而不断减少，当该值减少到 0 时循环定时器将执行 cxesig 函数。该值的取值范围在 0 到 65535 之间。

CYCLIC_PERIOD：是循环定时器使用的用于循环定时的系统滴答数。该数的取值范围从 0 到 65535。

```
#define    TIMER_0             0     /* 循环定时器 0 */
#define    T0_INIT_TIME        100   /* 定义首次定时时间为 100 个系统滴答 */
#define    T0_CYCLE_TIME   50    /* 定义循环定时时间为 50 个系统滴答 */

void task2(void)
{
   unsigned char   status;
   status  =  cxctstt(TIMER_0 , T0_INIT_TIME , T0_CYCLE_TIME);
   /* 启动 0 号循环定时器。这个循环定时器将以 100 个系统滴答的时间执行首次定时时间，之后
      循环定时器将不断地执行 50 个系统滴答为定时时间的循环定时。因此循环定时器执行第一次
      定时需要花费 100 个系统滴答时间，之后每次定时只需 50 个系统滴答时间。*/
   if (status != AOK) /* 检查函数调用是否成功 */
      {
        /* 函数调用失败，采取合适的操作 */
      }
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERR = Error：循环定时器的标识号超出范围而出错返回。

如果 返回值 STATUS 为 AOK，则表示指定的循环定时器已经启动。循环定时器将使用指定的首次定时时间和循环定时时间。

对一个特定的循环定时器，可以多次调用这个函数以改变循环定时器的首次定时时间和循环定时时间。

10.3 cxctrsti 修改循环定时器初始时间函数

cxctrsti 函数可以用来改变一个循环定时器的首次定时时间。函数调用完成后这个首次定时时间会立即生效直接覆盖循环定时器中用于定时的剩余定时时间。

任务和中断都可以调用这个函数，但循环定时器的循环定时时间却不会被修改。在调用这个函数时用户需要提供给 cxctrsti 函数两个参数以确定要操作哪个循环定时器以及新的首次定时时间。该值的取值范围在 0 到 65535 之间。

调用该函数时指定的首次定时时间将会立即生效。例如，如果某循环定时器的定时时间仅剩下 2 个系统滴答，并且任务 1 用一个长达 30 个系统滴答时间作为首次定时时间来重新调用这个函数，那么函数调用后这个值将替代原先的 2 个系统滴答的定时时间而使循环定时器执行一个 30 个系统滴答的定时时间。

该函数有一个非常有用的特性。用户可以使用循环定时器的“一箭”(one-shot，既“一次定时器”)方式来建立一个软件看门狗程序。例如，由 cxctstt 函数设置的某循环定时器的首次定时时间为 200 个系统滴答，而且任务 1 将大约在每 80 个系统滴答就执行一次 cxctrsti 函数(用于指定一个新的长达 100 个系统滴答的首次定时时间)来确定任务已经完成了一定的指定工作。假如由于某种原因任务 1 没能完成指定的工作，那么循环定时器将会发生超时并触发一个事件。

中断也可以调用这个函数，用来在未及时收到某个特定的中断信号的情况下通知任务，例如定时器已经被停止运行。因为当中断发生时，它就可以使用 cxctrsti 函数来重新设置这个循环定时器的首次定时时间 。

另一个例子是使用中断来定时接收某端口引脚上的数据。每当一个数据位到达时，它就会引发一个中断。这个中断将接收这个数据位并将其填入到一个位循环缓冲中，修改位指针，调用 cxctrsti 函数更新循环定时器的首次定时时间，然后退出。当数据位停止输入时，循环定时器将发生超时，并通过调用 cxesig 函数设置任务所等待的事件位的方式通知任务数据已经接收到可以开始处理了。

以下是一个关于 cxctrsti 函数的例子：

调用

任务、中断。

中断也可以调用这个函数，但只能是间接调用。有关间接函数调用的内容请参阅处理器详细说明章节。

```
#include <cxfuncs.h>                        /* 头文件 */
byte  cxctrsti(byte , word16);              /* 函数声明 */
#define    CYCLIC_NUM              ???   /* 用于识别循环定时器的标识号 */
#define    NEW_INITIAL_PERIOD      ???   /* 定义新的首次定时时间 */
unsigned char    STATUS;                    /* 该变量必须是局部变量 */

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

参数传递

CYCLIC_NUM 是任务需要启动的循环定时器的标识号。这个标识号的取值范围从 0 到配置模块中设置的循环定时器的最大数目减 1。

NEW_INITIAL_PERIOD 是会被循环定时器立即载入其时间计数器中的系统滴答数。该值随每次系统滴答的发生而不断减少，当该值减少到 0 时循环定时器将执行 cxesig 函数。该值的取值范围在 0

到 65535 之间。

```
#define TIMER_0            0      /* 循环定时器 0 */
#define T0_INIT_TIME     100    /* 定义首次定时时间为 100 个系统滴答 */
void   task2(void)
{
   unsigned char   status;
   while(1)
        {
          ……/* 先等待然后处理每 60 个系统滴答就出现一次的数据 */
          status  =  cxctrsti(TIMER_0 , T0_INIT_TIME);
          /* 重新开始循环定时器 0 的定时，这个循环定时器将执行 100 个系统滴答时间的定时，
             除非任务在这段时间内任务 2 再次重新启动循环定时器 0。另外循环定时器还可以通
             过设置一个事件来通知看门狗任务，任务 2 并没有象想象中的那样执行。*/
          if (status != AOK) /* 检查函数调用是否成功 */
             {
             /* 函数调用失败，采取合适的操作 */
             }
        }
}
```

返回值

如果返回给调用任务的状态为 AOK 则表示函数调用成功，指定的循环定时器的首次定时时间已经被改变并且循环定时器已经开始执行定时。如果返回值为 AERR 则表示循环定时器的标识号超出其范围而出错返回。

通过一个看门狗程序不断地调用这个函数来重新执行定时，这一点是非常有用的。如果看门狗程序没有执行或者检测到了错误，循环定时器的定时将最终因其时间计数器的值减少到 0 而使循环定时器通知某任务来完成正常的关闭操作。

10.4 cxctrstc 修改循环定时时间函数

cxctrstc 函数完成的功能与 cxctrsti 函数完成的功能很相似，只是它改变的是循环定时器的循环定时时间而不是首次定时时间。这个函数将会重新修改当前的循环定时时间，当循环定时器中当前的剩余定时时间减少到 0 时，这个循环定时器就会载入这个新的循环定时时间。

任务和中断都可以调用这个函数。这个函数调用不会修改当前循环定时器中剩余的定时时间。调用者需要为调用 cxctrstc 函数指定的循环定时器应该使用的新的循环定时时间。这个值的取值范围在 0 到 65535 之间。如果指定新的循环定时时间为 0，则循环定时器将成为*一箭*方式工作的定时器，即仅仅完成对剩余定时时间的定时就停止定时。

当 cxctrstc 函数被调用时，如果指定的循环定时器处于停止状态，那么这个循环定时器将被启动且它的首次定时时间默认为 65536。只有当该首次定时时间耗尽以后，循环定时器才会使用传递给这个函数的循环定时时间来进行定时。

以下是一个关于 cxctrstc 函数的例子：

调用

任务、中断。

中断也可以调用这个函数，但只能是间接调用。有关间接函数调用的内容请参阅处理器详细说明章节。

```
#include <cxfuncs.h>                    /* 头文件 */
byte  cxctrstc(byte , word16);          /* 函数声明 */
#define   CYCLIC_NUM              ???   /* 用于识别循环定时器的标识号 */
#define   NEW_CYCLIC_PERIOD       ???   /* 定义新的循环定时时间 */
unsigned char   STATUS;                 /* 该变量必须是局部变量 */
STATUS  =  cxctrstc(CYCLIC_NUM , NEW_CYCLIC_PERIOD);
```

参数传递

CYCLIC_NUM：是任务需要启动的循环定时器的标识号。这个标识号的取值范围从 0 到配置模块中设置的循环定时器的最大数目减 1。

NEW_CYCLIC_PERIOD ：是循环定时器用于进行循环定时的新的系统滴答数，即新的循环定时时间。该值的取值范围在 0 到 65535 之间。

```
#define  TIMER_0                  0    /* 循环定时器 0 */
#define  T0_NEW_CYCLE_TIME    100  /* 定义新的循环定时时间为 100 个系统滴答时间 */
void  task2(void)
{
   unsigned char  status;
   status  =  cxctrstc(TIMER_0 , T0_NEW_CYCLE_TIME);
   /* 改变循环定时器 0 的循环定时时间。只有在循环定时器中的当前剩余定时时间减少到 0 后，
      新的循环定时时间才会被循环定时器所使用。*/
   if (status != AOK) /* 检查函数调用是否成功 */
      {
        /* 函数调用失败，采取合适的操作 */
      }
}
```

返回参数

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERR = Error：循环定时器的标识号超出范围而出错返回。

如果 返回值 STATUS 为 AOK，则表示指定的循环定时器中的循环定时时间已经被修改，定时器已经开始执行定时，即使该定时器在函数调用之前处于停止状态。

10.5 cxctrstt 重新启动循环定时器函数

cxctrstt 函数用于重新启动一个循环定时器。这个函数不会改动当前循环定时器中的剩余定时时间以及循环定时时间。

这个函数可以用于重新启动一个曾经被 cxctstp 函数停止的循环定时器。所有的定时时间值都不会被改动。如果循环定时器是因为*一箭*方式的定时而停止的，那么当它被重新启动的时侯，它的首次定时时间为 65535 个系统滴答时间。

以下是一个关于 cxctrstt 函数的例子：

调用

任务、中断。

中断也可以调用这个函数，但只能是间接调用。有关间接函数调用的内容请参阅处理器详细说明章节。

```
#include <cxfuncs.h>                    /* 头文件 */
byte  cxctrstt(byte);                   /* 函数声明 */
#define   CYCLIC_NUM     ???      /* 用于识别循环定时器的标识号 */
unsigned char   STATUS;                 /* 该变量必须是局部变量 */
STATUS  =  cxctrstt(CYCLIC_NUM);
```

参数传递

CYCLIC_NUM：是任务需要启动的循环定时器的标识号。这个标识号的取值范围从 0 到配置模块中设置的循环定时器的最大数目减 1。

```
#define  TIMER_0    0     /* 循环定时器 0 */
void  task2(void)
{
   unsigned char  status;
   status  =  cxctrstt(TIMER_0);   /* 重新重新启动循环定时器 0，循环定时器 0 中所有的定时时间值
                                      都不会被改动。*/
   if (status != AOK) /* 检查函数调用是否成功 */
      {
        /* 函数调用失败，采取合适的操作 */
      }
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERR = Error：循环定时器的标识号超出范围而出错返回。

如果 返回值 STATUS 为 AOK，则表示指定的处于停止状态中的循环定时器已经启动定时。如果这个循环定时器早就已经启动，那么这个函数将不起任何作用。

10.6 cxctstp 暂停循环定时器函数

cxctstp 函数可用于停止一个已经启动的循环定时器。这个函数不会改变循环定时器中的任何时间数据值(在它被调用前所剩下的定时时间以及定时器的循环定时时间)。

调用者必须指定需要停止的循环定时器的标识号。这个标识号的取值范围从 0 到配置模块中设置的循环定时器的最大数目减 1。

当这个函数被调用后，指定的循环定时器将被置于停止状态同时从循环定时器启动列表中移去。

在 cxctstp 函数被调用时，循环定时器中所剩下的定时时间将会被保留，同样这个定时器的循环定时时间也不会被改动。因为这样当调用 cxctstt 函数重新开始这个循环定时器时，这个循环定时器中保留的时间值将会被继续使用。

以下是一个关于 cxctstp 函数的例子：

调用

任务、中断。

中断也可以调用这个函数，但只能是间接调用。有关间接函数调用的内容请参阅处理器详细说明章节。

```
#include <cxfuncs.h>                    /* 头文件 */
byte  cxctstp(byte);                    /* 函数声明 */
#define  CYCLIC_NUM     ???     /* 用于识别循环定时器的标识号 */
unsigned char       STATUS;          /* 该变量必须是局部变量 */
STATUS  =  cxctstp(CYCLIC_NUM);
```

参数传递

CYCLIC_NUM：是任务需要启动的循环定时器的标识号。这个标识号的取值范围从 0 到配置模块中设置的循环定时器的最大数目减 1。

```
#define  TIMER_0     0     /*循环定时器 0 */

void  task2(void)
{
   unsigned char  status;
   status  =  cxctstp(TIMER_0);
   /* 调用函数停止循环定时器 0，循环定时器 0 将立即停止同时从循环定时器的启动队列中移
      去。记住这个循环定时器中剩下的定时时间以及循环定时时间都不会被改动。*/
   if (status != AOK) /* 检查函数调用是否成功 */
      {
         /* 函数调用失败，采取合适的操作 */
      }
}
```

返回值

STATUS：是该函数的返回状态参数，有如下情况：

AOK = Good：该函数调用成功正常返回。

AERR = Error：循环定时器的标识号超出范围而出错返回。

如果 返回值 STATUS 为 AOK，则表示指定的循环定时器已经停止。如果在调用这个函数的时候循环定时器已经停止，那么这个函数对那个定时器不起任何作用。

11、操作系统函数

这些函数是 cmx 操作系统的一个组成部分。
cmx_init、cmx_go、cxint_in、cxint_ex、enable_slice、
disable_slice 、cmx_tic、cmx_power、cxtslot、cmxticks

11.1 cmx_init CMX 初始化函数

这个函数用来初始化 CMX 变量、参数及可配置的系统对象的最大值。cmx_init 函数必须在其它所有 CMX 函数被调用之前被调用。这个函数的调用被安排在用户的启动代码中。每当用户改变“cxconfig.h”文件时，包含这个函数的文件就必须要重新编译，“cxconfig.h”文件声明了应用程序的中的一些对象最大值。

以下是一个有关于 cmx_init 函数的一个例子

调用

在使用任何其它 CMX 函数之前。

```
#include <cxfuncs.h>  /* 头文件 */
void  cmx_init(void);  /* 函数声明 */
cmx_init();
```

参数传递

该函数不需要任何参数。

返回值

该函数没有任何返回值。

记住这个函数必须在使用任何其它 CMX 函数之前被调用，否则会产生灾难性的后果。

cmx_init 函数举例如下：

```
void   main(void)
{
   /* 定义任何需要用到的局部变量 */
   cmx_init(); /* 初始化 CMX */
   ……
   /* 现在用户可以访问任何在进入 CMX 操作系统环境之前可以使用的 CMX 函数。*/
   ……
}
```

11.2 cmx_go 启动 RTOS 函数

cmx_go 函数用来启动 CMX 操作系统。一旦这个函数被调用，CMX 操作系统就能接过 CPU 的控制权并决定什么时候执行任务及循环定时器什么时候启动。用户必须在调用 cmx_go 函数之前用 cxttrig 函数使得至少有一个任务是处于就绪状态或即将进入就绪状态。

cxttrig 函数可能在启动代码中被调用或是在进入 CMX 操作系统后被一个中断调用。如果没有任务处于就绪状态，CMX 操作系统将拥有所有 CPU 时间，除了中断和循环定时器所占用的时间。

以下是一个有关于 cmx_go 函数的一个例子：

调用

当用户想要进入 CMX 操作系统的时候。

```
#include <cxfuncs.h>  /* 头文件 */
void  cmx_go(void);   /* 函数声明 */
cmx_go();
```

参数传递

该函数不需要任何参数。

返回值

永远不会从 CMX 操作系统中返回。

cmx_go 函数举例如下：

```
void   main(void)
{
   /* 定义任何需要用到的局部变量 */
   cmx_init(); /* 初始化 CMX */
   ……
   /* 现在用户可以访问任何在进入 CMX 操作系统环境之前可以使用的 CMX 函数。*/
   ……
   /* 通过创建任务，循环定时器等等来设置 CMX。还有可能需要创建一些队列，设置邮箱等
      等。另外至少需要启动一个任务。*/
   ……
   cmx_go(); /* 进入 CMX 操作系统 */
   /* 注意：该函数不可能返回。*/
}
```

11.3 cxint_in 中断预处理函数

cxint_in 函数被大多数中断所用。中断的第一条指令就是调用 cxint_in 函数。这便告知 CMX 系统当中断出现时应该保存 CPU 寄存器的上下文并切换到中断的堆栈。如果要了解细节，就需要具体根据

所用的 CPU 来参阅“处理器详细信息”章节的有关内容。

一个中断并不是非得调用 cxint_in 函数不可。如果中断不调用此函数，就得由用户来正确保存和恢复中断所要用的寄存器的上下文。这样这个中断就不能调用任何 CMX 函数。这样的中断代码一定不可以在结束时调用 cxint_ex 函数。

“处理器详细信息”章节详尽描述了中断调用这个函数的内容。

以下是一个有关于 cxint_in 函数的一个例子：

调用

中断。

```
#include <cxfuncs.h>     /* 头文件 */
void  cxint_in(void);     /* 函数声明 */
```

必须根据指定的 CPU 指令使用相应的汇编语言来调用这个函数。

```
call cxint_in   ；在某些 CPU 的环境下
jsr cxint_in    ；在其它 CPU 的环境下
```

参数传递

该函数不需要任何参数。

返回值

该函数没有任何返回值。

当函数返回的时候中断已被禁止。是否重新开启中断由用户决定。当然不可屏蔽中断是不可能被禁止的。当该函数调用完成后，任务的上下文已经被保存。如果这是第一层中断(因为 CMX 运行中断嵌套)，那么任务堆栈已经切换到中断堆栈。

cxint_in 函数举例如下：

注：需要以汇编语言编写。

```
INTERRUPT_X_HANDLER：
；CPU 识别某中断后控制将转向的该中断处理程序的首地址。
；中断预处理代码，如果需要的话。根据 CPU 的情况来决定。
；
call cxint_in   ；在某些 CPU 中的汇编代码。
jsr cxint_in    ；日立(Hitachi)H8/300 系列。
；
```

现在程序可以进行中断处理。在此也可以调用一些 CMX 函数。中断代码也可以用 C 语言编写，但 CMX 建议最好不要这样，因为中断总是希望执行得尽可能快。在此也可以重新开启中断屏蔽。如果要重新开启中断屏蔽，那么必须要保证屏蔽当前这个中断(不能再次被识别)或者 CPU 硬件不会再转向当前这个中断直到该中断处理结束为止(因为中断还没处理完毕)。

```
；
；中断代码……。
call cxint_ex   ；如果中断进入时调用了 cxint_in 函数，那么这个中断后处理也必须存在。
；
```

如果中断使用了 cxint_in 函数调用，那么在此程序中就不再需要用于实现中断返回的指令(如 8051 中的 RETI 指令，Hitachi H8/300 系列 CPU 的 RTE 指令)。CMX 的中断将在 cxint_ex 的代码中自动返回。

11.4 cxint_ex 中断后处理函数

大多数中断都要用到 cxint_ex 函数。这个函数是中断代码所要调用的最后一条指令。调用了 cxint_in 函数的中断不可以使用表示“中断返回”的指令。当 cxint_ex 函数被调用时，CMX 将自动执行“中断返回”的指令，告诉 CPU 中断已结束。

同时 CMX 操作系统也将决定是否恢复任务或中断的寄存器上下文，或者是执行重新调度，让处于就绪状态的具有最高优先权的任务开始运行。

对大多数处理器来说，cxint_in 和 cxint_ex 一定得在一个汇编程序中调用。如果有必要，在 cxint_in 被调用之后可以调用一个 C 函数来进行大部分的中断处理。

*处理器详细信息*章节详尽描述了中断调用这个函数的内容。

以下是一个有关于 cxint_ex 函数的一个例子

调用

中断。

```
#include <cxfuncs.h>          /* 头文件 */
void  cxint_ex(void);         /* 函数声明 */
```

注：必须根据指定的 CPU 指令使用相应的汇编语言来调用这个函数。

```
call cxint_ex  ；在某些 CPU 中的汇编代码
jsr cxint_ex   ；在其它 CPU 的环境下的代码
```

参数传递

该函数不需要任何参数。

返回值

该函数不会返回。

注：如果发生中断嵌套(意味着这个中断至少是第二层中断)，那么这个函数调用将返回到前一个中断处理程序中。如果这是第一层中断，那么这个函数调用将会触发任务调度程序。调度程序将决定是否因为中断过程中的某些 CMX 调用而需要进行任务的切换，或者继续运行中断前运行的任务。

cxint_ex 函数举例如下：

注：需要以汇编语言编写。

```
INTERRUPT_X_HANDLER：
；CPU 识别某中断后控制将转向的该中断处理程序的首地址。
；中断预处理代码，如果需要的话。根据 CPU 的情况来决定。
```

；

call cxint_in ；在某些 CPU 中的汇编代码。

jsr cxint_in ；日立(Hitachi)H8/300 系列。

；

现在程序可以进行中断处理。在此也可以调用一些 CMX 函数。中断代码也可以用 C 语言编写，但 CMX 建议最好不要这样，因为中断总是希望执行得尽可能快。在此也可以重新开启中断屏蔽。如果要重新开启中断屏蔽，那么必须要保证屏蔽当前这个中断(不能再次被识别)或者 CPU 硬件不会再转向当前这个中断直到该中断处理结束为止。

；

；中断代码……

call cxint_ex ；如果中断进入时调用了 cxint_in 函数，那么这个中断后处理也必须存在。

；

如果中断使用了 cxint_in 函数调用，那么在此程序中就不再需要用于实现中断返回的指令(如 8051 中的 RETI 指令，Hitachi H8/300 系列 CPU 的 RTE 指令)。CMX 的中断将在 cxint_ex 的代码中自动返回。

11.5 enable_slice 分时调度函数

调用 enable_slice 函数将启动分时调度机制。这就可以让具有相同优先级的任务根据在“CXCONFIG.H”中声明的 TSCALE_SCALE 时间片值来进行分时运行。这个函数没有任何参数，只有任务才可以调用它。

“分时”章节中有 CMX 操作系统分时调度机制工作原理的完整说明。

以下是一个有关于 enable_slice 函数的一个例子：

调用

任务。

```
#include <cxfuncs.h>            /* 头文件 */
void  enable_slice(void);       /* 函数声明 */
enable_slice();
```

参数传递

该函数不需要任何参数。

返回值

该函数没有任何返回值。

用户需要仔细学习“分时”章节以充分理解分时调度的工作原理。

```
void  task1(void)
{
   unsigned char   status;       /* 声明局部变量 */
   enable_slice();           /* 启动分时调度 */
```

```
    ……                    /* 应用程序代码 */
    disable_slice();

    /*如果需要的话，关闭分时调度。*/
    ……
}
```

注：如果需要的话，用户可自由地启动或关闭分时调度。大多数情况下不需要分时调度。

11.6 disable_slice 禁止分时调度函数

调用 disable_slice 函数将关闭分时调度机制，即关闭被 enable_slice 启动的分时调度机制。这个函数不需要任何参数，只有任务才可以调用它。

“分时”章节中有 CMX 操作系统分时调度机制工作原理的完整说明。

以下是一个有关于 disable_slice 函数的一个例子：

调用

任务。

```
#include <cxfuncs.h>          /* 头文件 */
void  disable_slice(void);          /* 函数声明 */
disable_slice();
```

参数传递

该函数不需要任何参数。

返回值

该函数没有任何返回值。

用户需要仔细学习“分时”章节以充分理解分时调度的工作原理。

disable_slice 函数举例如下：

```
void  task1(void)
{
    unsigned char  status;        /* 声明局部变量 */
    enable_slice();               /* 启动分时调度 */
    ……                            /* 应用程序代码 */
    disable_slice();
    /* 如果需要的话，关闭分时调度。 */
    ……
}
```

注：当需要的时候用户可以自由地允许任务启动或禁止分时调度。大多数情况下是不需要分时调度的。

11.7 cmx_tic 系统滴哒函数

这个函数通常被定时器(一般为 T_0)中断调用。CMX 需要一个定时器中断作为时钟来为使用了定时等待的任务以及循环定时器执行调度操作。

cmx_tic 函数必须被一个中断调用。这个中断必须能够以一个固定的时间周期产生。中断的频率应该是固定的，应为所有与时间有关的活动都基于这个固定频率的中断。这中断应首先调用 cxint_in 函数，然后再调用 cmx_tic 函数。cmx_tic 函数将在调度标志被设置之前递减一个计数器，这个计数器的值被预置为调用此函数所需的中断次数。此计数器所载入的值是用户在"CXCONFIG.H"文件中用 C 语言的形式定义的 C_RTC_SCALE 常量值，正如在"RTOS 配置文件"章节中所描述的一样。

当由 C_RTC_SCALE 指定的计数值减到 0 时，C_RTC_SCALE 的值会重新赋给计数器，并且如果有任何任务定时器或循环定时器需要服务的话 CMX 的时间标志将会被设置。当中断调用了 cxint_ex 函数后，调度程序将决定是执行 CMX 定时器任务还是恢复执行被中断的当前运行任务。如果调度程序决定需要执行定时器任务时，那么在中断发生之前执行的当前运行任务将被置于恢复态同时将它的上下文保存，然后调度程序将让 CMX 定时器任务运行。(有关调度程序详细的工作原理请参考 "调度程序"章节。)

如果在中断之前执行的当前运行任务调用了 CMX 的 cxprvr 函数，那么 CMX 时间标志将会被设置并且计数器将重新载入由 C_RTC_SCALE 指定的值。这样使得在该中断返回时不会触发调度程序。这是因为特权标志已被 cxprvr 函数设置。特权标志不允许任何任务切换直到特权标志被清除，而且只有设置该特权标志的任务才有权清除它。要清除特权标志，任务就必须调用 CMX 的 cxprvl 函数。

$$f_{tic} = f_{osc} / (N \times (65536 - Timer\_Reload\_Value))$$

其中：f_{tic} 为系统滴哒频率(滴哒时间的倒数)；

f_{osc} 为系统振荡频率；

N 设置为 4；

Timer_Reload_Value 为定时器 T_0 重装载寄存器 RTH_0 和 RTL_0 的值。

11.8 cmx_power 进入低功耗模式函数

cmx_power 函数用汇编语言编写并且由 CMX 调度程序模块调用。这个函数的代码需要由用户来编写以启动 CPU 低功耗模式。

用户很可能会使用某些处理器指令来降低 CPU 的功耗，而用中断来唤醒处理器。这是因为调用 cmx_tic 函数的中断还是照常发生。这就允许中断唤醒处理器并执行 CMX 定时器任务来不断减少所有循环定时器的定时时间值和挂起任务的等待时间值。

编写这个函数时要注意，根据具体的 CPU 及所选择的低功耗模式，此函数需要完成的操作可能不尽相同。CMX 的汇编模块认为此函数一定会返回到调用它的那条指令处。必须保证低功模式的正确退出，并返回到下一条指令处而且恢复 cmx_power 函数被调用之前的寄存器和堆栈状态。

可以使用汇编语言编写这个函数和通过操纵寄存器和/或堆栈来保证处理器返回到调用 cmx_power 函数的指令的下一条指令。

CMX 提供了一个空的 cmx_power 函数，这样 CMX 一般不会进入低功耗模式除非用户编写了该函数的代码。最好在应用程序编写完毕并经过调试和测试之后再编写这个函数。

以下是一个有关于 cmx_power 函数的一个例子：

调用

CMX 调度程序。

```
#include <cxfuncs.h>          /* 头文件 */
void  cmx_power(void);        /* 函数声明 */
……
“在汇编编写的调度程序模块中”
call  cmx_power
```

参数传递

该函数不需要任何参数。

返回值

该函数没有任何返回值。

注：注意用户必须根据需要来编写这个函数以启动 CPU 的低功耗模式并且需要保证处理器能够返回到调用 cmx_power 函数之后的指令。

cmx_power 函数举例如下：

调度程序需要决定在因为所有的任务都处于空闲态或被挂起而使没有任务可以运行的时候是否要进入低功耗模式。

call cmx_power ; 在 CMX 调度程序的汇编模块中，并且需要用合适的汇编代码编写。

```
void  cmx_power(void)
{
   ……/* 编写用户程序代码 */
   /* 注意：对于某些特定的 CPU 和/或 C 语言的环境，该函数可能需要用汇编语言编写。*/
}
```

11.9 cxtslot 获取任务号函数

使用这个函数可以获得当前运行任务的槽号值。该值由 CMX 系统在使用 cxtcre 函数创建任务时赋予任务。

以下是一个有关于 cxtslot 函数的一个例子：

调用

任务。

```
#include <cxfuncs.h>              /* 头文件 */
byte   cxtslot(void);                   /* 函数声明 */
unsigned char   TASK_SLOT;    /* 该变量可以是局部变量也可以是全局变量 */
TASK_SLOT  =  cxtslot();
```

参数传递

该函数不需要任何参数。

```
Void   task2(void)
{
   unsigned char   task_slot;

   task_slot  =  cxtslot();
   /* 返回当前处于运行态的任务的槽号*/
}
```

返回值

TASK_SLOT 是当前处于运行态的任务的槽号

11.10 cmxticks 函数

定时器中断每次调用 CMX 的 cmx_tic 函数都将减少由 C_RTC_SCALE 指定的计数值(直到减少到 0)，同时增加一个称为 cmx_tick_count 的全局变量。这个变量包含了运行中系统滴答的总计数。cmxticks 函数将返回 cmx_tick_count 的当前值。

在 CMX 操作系统中有关 cmx_tic 函数的工作原理请参阅“cmx_tic 函数”部分。

以下是一个有关于 cmxticks 函数的一个例子：

调用

任务。

```
#include <cxfuncs.h>              /* 头文件 */
word32   cmxticks(void);          /* 函数声明 */
unsigned long   TICK_COUNT;       /* 该变量可以是局部变量也可以是全局变量 */
TICK_COUNT  =  cmxticks();
```

参数传递

该函数不需要任何参数。

```
Void   task2(void)
{
   unsigned long   tick_count;
   tick_count  =  cmxticks();
   /* 返回由 C_RTC_SCALE 决定的系统滴答总计数 */
}
```

返回值

TICK_COUNT 是每个 C_RTC_SCALE 所决定的当前系统滴答总计数。如果 C_RTC_SCALE 是 1，这个值是 cmx_tic 函数被调用的总次数。如果 C_RTC_SCALE 的值大于 1，那么这个值就是

cmx_tic 函数被调用次数的倍数。

12、CMX 定时器任务

CMX 定时器任务是由 CMX 创建的。它必须第一个被创建，这个过程是由 CMX 的 cmx_init 函数自动完成的。(对任何需要任务号的 CMX 函数调用禁止使用 0 值，否则被调用的函数将不执行它的代码并返回一个错误。)

当 do_timer_tsk 标志被设置时，定时器任务将被调度程序调用。当需要定时器任务执行时，do_timer_tsk 标志就会被 cmx_tic 函数设置。当任务在等待超时和/或循环定时器在运行时，就会有这种需要。

定时器任务确定是否有特定的任务定时器(该任务使用了 CMX 函数调用来等待一个指定的时间)需要减少其定时时间值，若有则减少该值。在等待了一段指定的时间之后，当该定时时间值减少到 0 时，定时器任务将自动接管被挂起的任务，将此任务置于就绪态。定时器任务将通知任务是何时重新恢复运行的以及何时指定的定时时间被耗尽。

定时器任务的另一个工作是：如果有循环定时器已启动且当它们的定时时间耗尽的时候，定时器任务就执行有关循环定时器的内容。当一个循环定时器被启动并且指定的定时时间减少到零，此循环定时器将自动执行，其具体操作就是调用带事件参数的 cxesig 函数。

CMX 定时器任务将按照你指定的调度频率来执行。如果当前运行任务调用了 CMX 的 cxprvr 函数，那么定时器任务将被延迟(阻塞)直到该任务调用了 cxprvl 函数来恢复定时器任务的运行。

CMX 定时器任务补充说明：

每一个 CPU 中都使用一个定时器来作为产生系统滴答时钟的物理基础，每当该定时器里的值减为 0 时(计时到)，定时器产生中断，该中断调用 cmx_tic()函数，使 C_RTC_SCALE 的计数值减 1，当把 C_RTC_SCALE 的值减到 0 时，cmx_tic()函数置位 do_timer_tsk 标志来启动定时器任务，这样就可以处理循环定时器和任务等待时间等。

13、堆栈

这节将讨论任务和中断的堆栈。因为所有的 C 编译器会把参数放在堆栈中，而函数(任务可作为函数)用堆栈来分配局部数据空间和存放返回地址，所以对这节内容的理解很重要。

对于大多数微处理器和微型计算机而言，堆栈空间驻留在外存中，虽然有一些处理器具有内部堆栈。不同的处理器会略有不同。处理器使用堆栈的具体操作将在后面的*处理器详细信息*一节中讨论。

所有的处理器都有一个堆栈指针寄存器(一些是专有的，其余的处理器是使用根据 C 编译器制造商所指定的特定的寄存器)。一些 C 编译器制造商也创建一种框架指针用来存取在一个函数中产生的局部数据，以及存取传递给函数的参数。同样框架指针也是一种专有寄存器或由 C 编译器制造厂方所指定的寄存器。

CMX 能让用户设置存放所有任务堆栈的外存的大小，并且设置中断堆栈的大小。每个任务都将有它自己的堆栈。这就能让 CMX 快速的保存任务的上下文。当一个任务的上下文被保存时，仅需将寄存器内容压入堆栈。之后，堆栈指针将指向另一内存地址(切换过来的内存地址将会是中断堆栈或另一个任务的堆栈)。

对于用户来说保证每一个任务堆栈和中断堆栈有足够大的空间是至关重要的。任务堆栈的大小必须要能够容纳下特定的任务的局部数据区、每个函数被调用时被压入堆栈的所有返回地址、以及被调用函数的局部数据区。每一个任务堆栈在出现中断时必须可以保存下所有的 CPU 寄存器中的内容。如果 CPU 有一种*不可屏蔽的中断*，那么任务堆栈必须能够容纳下更多字节的内容。这就是为什么 CPU 寄存器中的内容要在不可屏蔽中断发生时被保存的原因。

我们将展示如何计算堆栈所需空间的大小并提供两个例子。建议用户最好把自己所估计到的所有任务堆栈和中断堆栈的大小再翻倍，至少在刚开始时应这样做。当用户变得更加熟练并且代码测试表明这堆栈的大小可以再小些时，那时堆栈大小可以被缩减。

当一个任务堆栈或者是中断堆栈的内容被压入不属于该堆栈的存储区域时，将会出现不可预测的结果。

关于例子有一点值得注意的是，所有的 C 编译器中函数调用的具体方式是不尽相同的(一些使用堆栈结构，一些则不是)。CMX 极力建议用户看一看 C 编译器的手册以便更好的了解特定的 C 编译器堆栈的使用，手册描述了堆栈的使用、局部数据和参数的传递以及 C 编译器产生的代码(汇编格式)。

以下仅仅是例子而已，CMX 并不保证这是任务堆栈和中断堆栈所需的确切字节数。在这些例子中假定整形数(ints)占两个字节长，而实际上在一些处理器上允许整形数占四个字节长。

例 1：(涉及大量局部变量，没有嵌套函数调用)

```
void   Task1(void)     /*任务不接受也不返回参数*/
{
   /*注意有些 C 编译器会创建一种堆栈框架,它将使用这个任务堆栈的一些空间来存放堆框架指针
     和其它一些可能的寄存器*/
   int     a;      /* 占两字节的局部变量空间*/
   char   b;      /*占一字节的局部变量空间*/
   char  *ptr;    /*两字节的局部变量,针对于寻址能达到 65535 字节且没有存储区域转换、也不需要
                    存储边界调整(对齐)的处理器*/
   一些处理器需要把整数、指针等放在处理器中适当的内存地址边界上。在一些情况下这些整
   数，指针处于内存中一个偶数地址，否则就处于奇数地址。如果局部变量不能根据处理器的对
   齐规则正确的放置，那么局部空间将多消耗一个字节。
   func_A(char , int);   /* func_A 是一个接受一个字符和一个整数的函数，这将用去 3 字节的堆栈空
                            间，再加上 2 字节用作保存返回地址*/
   func_B(int , int);     /* func_B 是一个接受两个整型数的函数。这将用去 4 字节的堆栈空间，再加
                            上 2 字节用作保存返回地址*/
}

void   func_A(char , int)
{
   char   a;       /*此局部变量在实际中占用了任务 1 的堆栈空间中的 1 个字节*/
   int     b;      /*此局部变量在实际中占用了任务 1 的堆栈空间中的 2 个字节*/
   /* func_A 的代码*/
}
void  func_B(int , int)
{
   int   a; /*此局部变量在实际中占用了任务 1 的堆栈空间中的 2 个字节*/
   int   b; /*此局部变量在实际中占用了任务 1 的堆栈空间中的 2 个字节*/
   /* func_B 的代码*/
}
```

一些处理器需要把整数、指针等放在处理器中适当的内存地址边界上。在一些情况下这些整数，指针处于内存中一个偶数地址，否则就处于奇数地址。如果局部变量不能根据处理器的对齐规则正确

的放置，那么局部空间将多消耗一个字节。

以上的计算不会包括保存堆栈框架指针的空间，同时也不包括 CPU 内存地址对齐方式所花费的空间。

例 2：(涉及大量局部变量，存在嵌套函数调用)

```
void   Task1(void) /*任务不接受也不返回参数*/
{
  /*注意有些 C 编译器会创建一种堆栈框架,它将使用这个任务堆栈的一些空间来存放堆框架指针
    和其它一些可能的寄存器*/
  int     a;      /* 占两字节的局部变量空间*/
  char   b;       /*占一字节的局部变量空间*/
  char   *ptr;    /*两字节的局部变量,针对于寻址能达到 65535 字节且没有边界转换、也没有队列边
                   界的处理器*/

  一些处理器需要正确排列整数、指针等来对齐处理器的队列边界。在一些情况下这处于一个内
  存偶数地址，不然就处于奇数地址。如够局部变量不能根据处理器的队列规则正确的排列，那
  么局部空间将多消耗一个字节。
  func_A(char , int);    /* func_A 是一个接受一个字符和一个整数的函数，这将用去 3 个字节的堆
                            栈空间，再加上 2 个字节用作保存返回地址*/
  func_B(int  int);      /* func_B 是一个接受两个整型数的函数。这将用去 4 个字节的堆栈空间，
                            再加上 2 个字节用作保存返回地址*/
}

void   func_A(char , int)
{
  char   a;    /*此局部变量在实际中占用了任务 1 的堆栈空间中的 1 个字节*/
  int   b;      /*此局部变量在实际中占用了任务 1 的堆栈空间中的 2 个字节*/

  /* func_A 的代码*/

  func_C(int , int);    /* 嵌套调用 func_C 函数。func_C 是一个接受两个整数的函数。这将用去任
                           务 1 的堆栈空间的 4 个字节，再加上 2 字节用作保存返回地址空间*/
}

void   func_B(int , int)
{
  int   a; /*此局部变量在实际中占用了任务 1 的堆栈空间中的 2 个字节*/
  int   b; /*此局部变量在实际中占用了任务 1 的堆栈空间中的 2 个字节*/
  /* func_B 的代码*/
}

void   func_C(int,int)
{
  int a;   /*此局部变量在实际中占用了任务 1 的堆栈空间中的 2 个字节*/
  int b;   /*此局部变量在实际中占用了任务 1 的堆栈空间中的 2 个字节*/
```

```
    /* func_C 的代码*/
  }
```

在以上的例子中，在任务 1 的堆栈空间中用来保存任务 1 的局部变量的空间将占 5 个字节，用来保存 func_A 函数的参数的为 3 个字节，用来保存 func_A 函数的返回地址的为 2 个字节，保存 func_A 函数的局部变量的空间为 3 字节；保存 func_C 函数的参数的为 4 个字节，保存 func_C 函数返回地址的空间为 2 个字节，保存 func_C 函数的局部变量的空间为 4 字节。注意在此没计算 func_B 是因为调用了 func_C 的 func_A 比 func_B 用了更多的内存空间。

以上的计算不会包括保存堆栈框架指针的空间，同时也不包括 CPU 内存地址对齐方式所花费的空间。

正如用户所看到的，任务 1 的堆栈用于存放任务 1 的局部变量和任务 1 所调用的所有函数。这其中包括了任务 1 所调用的所有函数的局部变量和它们的返回地址。

CMX 建议用户将所有任务堆栈空间用某个特定的值填满，例如十六进制值 AA，然后再运行应用程序代码。这样你可以察看指定任务堆栈空间实际上使用了多少堆栈。如果堆栈的内容超过了分配给它的堆栈空间的大小，那么将会出现严重后果，如系统崩溃和内存被完全破坏。

14、RTOS 配置文件

用户必须为 CMX 的应用程序预先配置 RTOS(实时多任务操作系统)环境。这样允许 CMX 可以事先分配必需的内存，并且知道一些特定对象的最大值。这样做只是为了提高程序的运行速度，减少了在 CMX 函数执行时申请所需要的内存而必需的代码(当程序在运行时)。CMX 特别建议在嵌入式实时系统中必须这样做。

预先配置 CMX RTOS 并不是一件困难的事，并且所设置的值也可以随需要而改变。在开始运行新的应用程序前，下面的符号常量必须事先已经确定。CMX 建议在开始时设置的值要略超过原先的估计值，当你的应用程序完成时再减小这些值。

下述符号常量放在 CMX 的 *CXCONFIG.H* 文件里，该文件是 CMX 的配置文件。每一个符号常量之后都带有该值如何使用的详细说明。

(1). C_MAX_TASKS

```
＃define    C_MAX_TASKS      (用户值)
```

C_MAX_TASKS ：是应用程序所允许的任务的最大数量，因为对于所有的任务都需要分配一定的内存。每一个任务都有它自己的*任务控制块*。此常量最大可被设置为 255。

(2). C_MAX_RESOURCES

```
#define    C_MAX_RESOURCES    (用户值)
```

C_MAX_RESOURCES：是应用程序中资源的最大数量。CMX 允许的资源的最大数量为 255，此常量由 CMX 资源管理函数使用。如果程序不使用任何资源，那么应该把此常量设为 0。

(3). C_MAX_CYCLIC_TIMERS

```
# define    C_MAX_CYCLIC_TIMERS     (用户值)
```

C_MAX_CYCLIC_TIMERS：是应用程序中循环定时器的最大数量。CMX 允许的循环定时器的最大数量为 255，此常量被 CMX 循环定时器函数所使用。

(4). C_MAX _MESSAGES

define C_MAX_MESSAGES (用户值)

C_MAX_MESSAGES：是被任何一个或所有的邮箱所使用的消息的最大数量。该常量的设置仅仅是为了分配消息所占用的内存。此常量最大可被设置为 65535。如果应用程序中不需要任何消息，此时用户可以将此常量设置为 0。

(5). C_MAX_QUEUES

define C_MAX_QUEUES (用户值)

C_MAX_QUEUES：是应用程序中用户所希望具有的队列的最大数量。CMX 允许的队列的最大数量数为 255。如果应用程序未使用队列，那么此常量可以设置为 0。

(6). C_MAX_MAILBOXES

define C_MAX_MAILBOXES (用户值)

C_MAX_MAILBOXES：是应用程序所使用的邮箱的最大数目。CMX 系统允许的邮箱最大数目为 255，该常量为 CMX 邮箱/消息函数所使用。如果应用程序未使用邮箱，则此常量可设定为 0。

(7). C_MAX_SEMAPHORES

#define C_MAX_SEMAPHORES (用户值)

C_MAX_SEMAPHORES：是应用程序中信号量的最大数量。CMX 系统允许该值最大为 255，该常量被 CMX 的信号量管理函数所使用。如果应用程序未使用任何信号量，那么该值可以被设置为 0。

(8). C_INTERRUPT_SIZE

define C_INTERRUPT_SIZE (用户值)

它可应用于大多数的处理器，但不是所有的。

C_INTERRUPT_SIZE：是用来定义中断堆栈的大小，这个堆栈被 CPU 所识别的第一个以及随后的若干个中断所使用。同时，此堆栈还被 CMX 调度程序、CMX 定时器任务以及正在执行其内容的 CMX*中断管道*所使用。另外要注意，中断管道在执行 CMX 函数时就好象一个任务在调用它们，因此需要足够的堆栈空间来满足中断管道可能调用的最深层的 CMX 函数的需求。另外还需要把发生嵌套的中断堆栈考虑在内。计算这个参数的方法是将可能发生的中断嵌套的最大数量乘以特定处理器中的寄存器的数量及大小。另外，还要加上每个中断能够压入此堆栈的字节数，再加上专门的 CMX 函数被中断管道调用时能够使用的字节数。这个计算是相当困难的，CMX 建议用户开始时先将计算出的堆栈大小加倍。CMX 认为最小的堆栈空间是 128 个字节，用户可以将此作为起始点。

下面的#define 设定了 CMX*中断管道*的大小并且同时声明那些可被任何一个中断所间接调用的 CMX 函数。如果所有的可以被中断间接调用的 CMX 函数都未被使用，那么 CMX 代码长度将会缩短。另外也要注意：如果某个特定的 CMX 函数并未被任何一个中断所使用，任务仍可以调用这个 CMX 函数而不必考虑以下的设置。

(9). C_PIPE_SIZE

define C_PIPE_SIZE (用户值)

C_PIPE_SIZE：该参数值定义了 CMX“ 中断管道 ”的大小。这个参数必须是 2 的幂，并且不能小于 16 或大于 256，有效的参数值可以是 16，32，64，128，256。设置该值的目的就是 CMX 希望“ 中断管道 ”能够尽可能快地执行。注意中断管道实际上是一个循环缓冲区并且它必须被不停的检测以确保进入指针和输出指针不会超过它们的最大值。

如果用户输入了一个不在上述所列出的范围之内的值，那么中断管道的大小将默认为 256。如果中断管道已满，则中断不会知道已经出错，因为指定的 CMX 函数并没有被放入到中断管道中。如果中断管道的内容可能被频繁地填入，那么中断管道的大小需要变得更大，或者修改使用大量 CMX 函数的中断代码以减少调用中断管道的频率。

(10). CMXBUG_ENABLE

#define CMXBUG_ENABLE (用户值：必须是 0 或 1)

CMXBUG_ENABLE：该值将决定是否初始化并设置某些函数与 CMXBUG.C 协同运行。当该值取为 1 的时候，CMX 的模块 CMX_INIT.C 中将包含 CMXBUG.C 模块的 setup_bug，bug_getchr 和 bug_putchr 函数。同时这三个函数也将随 CMX_INIT.C 一起被编译。setup_bug 函数将启动 9600 波特率且不带奇偶校验的 UART。bug_putchr 和 bug_getchr 函数主要用于字符的串行收发，对于用户所使用的特定的处理器需要经过适当的剪裁以变得更加简练才行。用户必须根据所选择的晶振来对这三个函数进行剪裁，特别是 bug_setup 函数。

(11). CMXTRACKER_ENABLE

#define CMXTRACKER_ENABLE (用户值：必须是 0 或 1)

CMXTRACKER_ENABLE：该值将决定是否初始化并设置某些函数与 CMXTRACK.C 协同运行。当该值取为 1 的时候，CMX 的模块 CMX_INIT.C 中将包含 CMXTRACK.C 模块的 setup_bug，bug_getchr 和 bug_putchr 函数。同时这三个函数也将随 CMX_INIT.C 一起被编译。setup_bug 函数将启动 9600 波特率且不带奇偶校验的 UART。bug_putchr 和 bug_getchr 函数主要用于字符的串行收发，对于用户所使用的特定的处理器需要经过适当的剪裁以变得更加简练才行。用户必须根据所选择的晶振来对这三个函数进行修改，特别是 bug_setup 函数。

(12). CMXTRACKER_SIZE

#define CMXTRACKER_SIZE (用户值)

CMXTRACKER_SIZE：该值定义了用于按时间顺序存放被捕获的 RTOS 数据流的 CMX 队列的空间。这个值的范围是从 0 到 65535。该值选得越大，所被捕获的数据也就越多。

(13). EN_CXTWAKE

define EN_CXTWAKE (用户值：必须是 0 或 1)

EN_CXTWAKE：是用来声明一个中断是否可以间接调用 CMX 的 cxtwake 函数。该值为 0 将不允许中断间接调用此函数，中断虽然仍可以将此函数放入中断管道内，但中断管道并不会接受它而并直接返回。任何其它非 0 值都将允许一个中断间接调用此函数并且允许中断管道接收此函数。

(14). EN_CXTWAKF

define EN_CXTWAKF (用户值：必须是 0 或 1)

EN_CXTWAKF：是用来声明一个中断是否可以间接调用 CMX 的 cxtwakf 函数。该值为 0 将不允许中断间接调用此函数，中断虽然仍可以将此函数放入中断管道内，但中断管道并不会接受它而并直接返回。任何其它非 0 值都将允许一个中断间接调用此函数并且允许中断管道接收此函数。

(15). EN_ CXTTRIG

define EN_CXTTRIG (用户值：必须是 0 或 1)

EN_ CXTTRIG：是用来声明一个中断是否可以间接调用 CMX 的 CMXttrig(任务触发启动)函数。该值为 0 将不允许中断间接调用此函数，中断虽然仍可以将此函数放入中断管道内，但中断管道并不会接受它而并直接返回。任何其它非 0 值都将允许一个中断间接调用此函数并且允许中断管道接收此函数。

(16). EN_CXCTSTP

define EN_CXCTSTP (用户值：必须是 0 或 1)

EN_CXCTSTP：是用来声明一个中断是否可以间接调用 CMX 的 cxctstp(暂停循环定时器)函数。该值为 0 将不允许中断间接调用此函数，中断虽然仍可以将此函数放入中断管道内，但中断管道并不会接受它而并直接返回。任何其它非 0 值都将允许一个中断间接调用此函数并且允许中断管道接收此函数。

(17). EN_CXCTRSTT

define EN_CXCTRSTT (用户值：必须是 0 或 1)

EN_CXCTRSTT：是用来声明一个中断是否可以间接调用 CMX 的 cxctrstt (重新启动循环定时器)函数。该值为 0 将不允许中断间接调用此函数，中断虽然仍可以将此函数放入中断管道内，但中断管道并不会接受它而并直接返回。任何其它非 0 值都将允许一个中断间接调用此函数并且允许中断管道接收此函数。

(18). EN_CXCTRSTI

define EN_CXCTRSTI (用户值：必须是 0 或 1)

EN_CXCTRSI：是用来声明一个中断是否可以间接调用 CMX 的 cxctrsti(修改循环定时器初始时间)函数。该值为 0 将不允许中断间接调用此函数，中断虽然仍可以将此函数放入中断管道内，但中断管道并不会接受它而并直接返回。任何其它非 0 值都将允许一个中断间接调用此函数并且允许中断管道接收此函数。

(19). EN_CXCTRSTC

define EN_CXCTRSTC (用户值：必须是 0 或 1)

EN_CXCTRSTC：是用来声明一个中断是否可以间接调用 CMX 的 cxctrsc(修改循环定时时间)函数。该值为 0 将不允许中断间接调用此函数，中断虽然仍可以将此函数放入中断管道内，但中断管道并不会接受它而并直接返回。任何其它非 0 值都将允许一个中断间接调用此函数并且允许中断管道接收此函数。

(20). EN_CXMSSEND

define EN_CXMSSEND (用户值：必须是 0 或 1)

EN_CXMSSEND：是用来声明一个中断是否可以间接调用 CMX 的 cxmssend 函数。该值为 0 将不允许中断间接调用此函数，中断虽然仍可以将此函数放入中断管道内，但中断管道并不会接受它而并直接返回。任何其它非 0 值都将允许一个中断间接调用此函数并且允许中断管道接收此函数。

(21). EN_CXESIG

define EN_CXESIG (用户值：必须是 0 或 1)

EN_CXESIG：是用来声明一个中断是否可以间接调用 CMX 的 cxesig 函数。该值为 0 将不允许中断间接调用此函数，中断虽然仍可以将此函数放入中断管道内，但中断管道并不会接受它而并直接返回。任何其它非 0 值都将允许一个中断间接调用此函数并且允许中断管道接收此函数。

(22). EN_ CXSEMPST

#define EN_CXSEMPST (用户值：必须是 0 或 1)

EN_CXSEMPST：用于决定是否允许一个中断可以间接地使用 CMX 的 cxsempst(信号量释放)函数。如果该值是 0，那么将不允许中断间接地调用该函数。在这种情况下如果中断依然把该函数利用间接调用放入中断管道之中，然而中断管道并不接受这个函数。其它任何非 0 值都将允许一个中断间接地调用该函数并允许中断管道接受这个函数。

只有哪些对应使能标志被声明为非零值的 CMX 函数才允许进入 CMX 的“中断管道”。如果对应某函数的使能标志用#define 定义为零或者中断管道已满，则中断管道将直接返回而不会不会把该函数和它的参数放入管道中。

15、分时

如果有必要用户可以启动分时调度机制。CMX 认为大多数应用程序不需要分时调度机制。但是 CMX 还是为那些需要分时的情况提供了分时调度机制(主要针对优先级相同任务)。

为了启动分时调度，需要调用 enable_slice 函数。当此函数被调用以后，C_TSLICE_SCALE 参数将被用来决定一个任务时间片的系统滴答数，所有的任务都将使用这个时间片。

当分时调度机制被激活以后，当前运行任务的优先级将决定了当前哪些任务将参与分时调度。所有与当前运行任务具有相同或较低优先级的任务将参与分时调度。如果一个较高优先级的任务实现了抢占取得运行权，那么这个任务的优先级将被用来决定那些需要分时的任务。

当分时调度机制被激活以后，一个任务将自动运行一段由时间片所决定的一定系统滴答数的时间。当一个任务的时间片耗尽的时候另一个分时任务将被激活。当调度程序决定运行下一个分时任务时，该分时任务必须处于就绪态或恢复态。

所有参与分时的任务将以轮流循环的方式运行。每一个分时任务运行一个时间片的时间。如果某分时任务因为正在等待一些实体而被阻塞，那么当 CMX 调度程序决定需要运行哪一个分时任务的时候，该任务将不会考虑在内。

如果一个分时任务因为调用了 CMX 函数而使任务被挂起，那么这个任务被认为是其时间片已经耗尽。直到此任务脱离了挂起态并且是分时任务队列中下一个即将运行的分时任务，这个任务才会重新恢复运行。

另外要注意如果一个任务调用了 cxsched 函数来实现协作式调度，那么当前运行的分时任务的时

间片将自动结束。在任务链表中下一个可运行的任务将成为运行任务，而不考虑它的优先级情况。

用户可以在任何时候调用 disable_slice 函数来关闭分时调度机制。注意只有正在运行的任务才可以调用这个函数。

不能做这样的假设，所有与当前运行任务的优先级相比具有相同或者较低优先级的任务将会成为下一个运行任务。因为较高优先级的任务会不断地在时间片结束之前抢占低优先级任务的运行时间，这将会使下一个具有相同优先级的任务接着运行(在参与分时调度机制的任务中，也是按照优先级的大小来取得时间片的)。

16、中断

这一节将介绍 CPU 中断以及如何使用 CMX 的*中断管道*。处理器详细信息一节明确地解释了中断以及如何在用户特定的 CPU 环境下使用中断。

CMX 允许中断嵌套。当 CPU 硬件识别一个中断以后，根据不同的 CPU 会进行不同的处理。

首先将中断发生时的应用程序所在地址压入堆栈。有些处理器会同时会把所有的或者是一部分的寄存器也压入堆栈。有些处理器会同时自动清除中断允许标志，以防止 CPU 识别多个中断。但这并不能阻止非可屏蔽中断的发生。

之后，处理器将跳转到中断向量地址，或者从中断向量的地址中取回代码地址。硬件将自动把该中断服务程序入口地址装入程序计数器中。

CMX 要求所有需要间接调用任何 CMX 函数(通过*中断管道*实现)或者需要调用 cmx_tic 函数的中断(此函数决定了调度的频率)，都必须调用 CMX 中断服务程序 cxint_in 和 cxint_ex。如果此中断不使用这两个 CMX 中断服务程序，那么用户必须自行将所有的会被中断破坏的寄存器保存并恢复。用户必须将 CPU 上下文环境恢复为在中断执行前的状态。这种中断不能使用任何 CMX 函数，也不能执行任何会引起任务重新调度的操作。另外，此中断也不允许其它任何会调用 CMX 函数的中断被识别(在调度上述两个函数期间，决不允许响应和处理其它中断)。

中断服务程序可以通过增减一个 CMX 变量之后再间接地使用 CMX 函数来避免出现这种情况，因为当该中断被更高优先级的中断所抢占后*中断管道*中的内容并不会立即被执行。注意不同的处理器情况会有所不同，有关更详细的细节请查阅处理器详细信息一节。

CMX 让中断间接的调用一定数量的 CMX 函数。如果一个任务正在某个 CMX 函数的*代码的临界区*中运行，那么该任务的执行过程不可以被中断，只有该任务退出该临界区以后中断才能够识别并响应。

这样使得中断的识别不需要任何的延迟，也不会在一段时间内被暂时禁止。这也会减少中断调用一个 CMX 函数所需的时间，使得中断可以用最快的方式执行完它的代码。

17、中断与 CMX 函数的接口

CMX 创建了一个“中断管道”。这个中断管道实际上是一个循环缓冲，它包含了 CMX 函数的标识符和有关 CMX 函数所需要的参数。用户可以在 CXCONFIG.H 文件中指定中断管道的大小。中断使用一定的参数来调用中断管道的具体情况如下所述：

第一个参数是被间接调用的 CMX 函数标识值。当通道内容被执行的时候，这个标识值将决定了哪一个 CMX 函数需要被调用。有关可以间接调用的函数及其被间接调用的函数标识值请参阅处理器详细信息一节。

下一个参数是当中断管道处理管道中的 CMX 函数时，调用 CMX 函数时所需要的实际参数。这种机制使得中断可以快速地传递 CMX 函数和它相关的参数，而不考虑执行实际的 CMX 函数所需花费的时间。这个中断管道允许重入，因此一个更高优先级的中断可以打断一个低优先级的中断，一旦高优先级的中断返回该低优先级的中断将重新恢复运行。

当中断处理已经完成并且调用了 cxint_ex 函数(该中断进入时必须已经调用了 cxint_in 函数)准备退出中断时，具体情况如下所述。首先，CMX 中断处理程序将判断是否存在某个中断(较低优先级的中断)需要继续处理，如果是则先处理该中断。如果此时没有其它中断需要处理(即这是第一个被识别的中断)，那么 CMX 中断处理程序将判断中断管道是否已被使用过。如果中断并没有调用中断管道也未将任何 CMX 函数放入其中，那么在中断之前执行的任务将从它被中断的地方继续运行。如果中断已经调用了中断管道，那么 CMX 将确定中断管道是否正在被执行，如果是则继续执行该中断管道中的内容，以及确定是否有任务停在一个 CMX 函数的中间(临界区)。

如果任务停在一个 CMX 函数中间，那么它将被允许先完成该函数。当该任务完成该函数之后，中断管道会使用中断管道中的参数来真正地调用由中断间接调用的 CMX 函数，而此时该任务会被阻塞。如果任务并不是在一个 CMX 函数中间被中断的，或者中断管道当前并没有开始执行，CMX 降自动开始执行中断管道中的内容。当中断管道的内容执行完成之后，被阻塞的任务有可能将恢复执行，也有可能因为 CMX 函数的使用而使某更高优先级的任务进入就绪态，使得此高优先级的任务进入运行态。

因为 CMX 中断管道的特征和结构，CMX 仅仅需要在 3 到 5 个(与处理器有关)地方需要关闭总中断，而且关闭的时间很短。这个时间与 CPU 的处理速度以及晶振频率有关。

在处理器专用信息一章中，CMX 描述了专门的处理器中断如何传递 CMX 函数和它的参数到中断管道中。这一章还将解释中断如何明确的调用 CMX cxint_in 和 cxint_ex 程序。

在处理器详细说明一节中将会具体介绍如何使用中断管道来传递 CMX 函数和它的参数，以及明确解释了对 cxint_in 和 cxint_ex 函数使用。

CMX 建议在中断中使用这两个 CMX 中断服务例程，cxint_in 和 cxint_ex。前者在进入中断后立即执行。cxint_in 会保存任务的上下文，包括 CPU 寄存器和堆栈，并且切换堆栈到中断堆栈。cxint_in 函数在完成这些工作之后又会回到中断。

根据不同的 CPU，一般这段处理时间约为几个微秒。在调用了 cxint_in 函数之后，中断就可以使用各种可以被间接调用的函数。cxint_in 函数会将中断允许标志置于合适的值，禁止 CPU 识别任何中断除了那些该中断允许标志无法屏蔽的中断。用户可以根据需要重新打开某些中断屏蔽位。

如果发生了调度(任务切换)，CMX 将会根据任务的优先级来确定下一个要运行的任务。如果没有发生调度，那么因为中断而阻塞的任务其上下文将被恢复，该任务将继续运行就象中断从没有发生过一样。

18、CMX 调度程序

这一节的内容详细解释了 CMX 调度程序的工作原理。CMX 调度程序是基于真正的抢占机制。这就意味着由于调用了某个 CMX 函数而使得有一个高优先级任务可以开始运行，任务和中断将会立即引起任务的切换。当然也有可能发生协作式调度，一个任务放弃运行而让下一个任务(具有较低或相同的优先级的任务)运行。另外还有可能发生真正的分时调度。分时调度机制允许较高优先级的任务和其它分时任务之间实现分时运行，在较高优先级的任务所分配的时间片结束后可以使其它分时任务重新获得 CPU 的控制权。这样在这种调度程序的管理下使得用户可以完全控制任务的执行。

在 CMX 中，一个任务有以下的状态：

空闲态

一个用 cxtcre 函数创建但未由 cxtrig 函数启动的任务处于空闲态。一个已经运行完毕并调用了 cxtend 函数的任务，而且在该任务的控制块中没有明显的触发调用，那么该任务也处于空闲态。处于空闲态的任务是不会运行的。

就绪态

就绪态就是通知调度程序任务随时准备运行，但还没有运行的任务所具有的状态。当发生任务的调度操作时，调度程序会根据任务之间的优先级关系来决定运行哪个任务。

运行态

正在执行的任务处于运行态并且占有所有的 CPU 时间。在任何时候只能有一个任务处于运行状态。

等待(挂起)态

一个因调用 CMX 函数而被挂起的任务处于等待(挂起)状态。有许多函数调会使任务挂起。处于挂起态的任务包括因等待时间、事件、标志、消息或应答等而被挂起的任务。

恢复态

恢复态和就绪态是基本相同的，唯一的不同点是，处于恢复态的任务会告知调度程序它的代码曾经执行过但没有完成。这说明有优先级较高的任务实现了抢占并迫使当前运行任务转变成恢复态，或者该任务因某函数调用而被挂起并且现在已经从等待(挂起)态转变为恢复态。

19、CMX 操作标志

CMX 通常用 CMX 函数和中断来设置和清除一些 CMX 操作系统特殊的标志，从而通知调度程序该执行什么操作。下面将具体描述这些特殊的标志和这些标志是如何被设置的以及调度程序是如何操作这些标志的。

被某个中断调用的 cmx_tic 函数将决定是否有正在执行定时的任务定时器以及循环定时器需要服务。该函数设置了 CMX 的 do_timer_tsk 标志表明调度程序应该执行 CMX 定时器任务。另外如果该函数发现当前运行任务是一个分时调度的任务并且它的时间片已经到时，那么它将设置 do_time_slice 标志。这说明此分时任务应该放弃控制给下一个处于就绪状态的分时任务运行。

大多数 CMX 函数都会设置 CMX 的标志。当一个任务因为调用 CMX 函数而被挂起时，CMX 的抢占标志将被设置。如果一个任务调用了某个 CMX 函数，此函数可能会唤醒一个挂起任务，那么 CMX 函数将检测以前的挂起任务(此任务现在处于恢复态)，检测它是否具有比调用此函数的当前运行任务具有更高的优先级。如果是，那么 CMX 的抢占标志将被设置。

CMX 中的 do_coop_sched 标志由 CMX 的 cxsched 函数设置。该标志的设置说明当前运行任务将让在就绪表中的下一个任务成为运行任务。这说明任务执行了协作调度。

CMX 所跟踪的最后一个标志是 do_int_pipe 标志，该标志的设置说明有中断已经调用了中断管道程序并且需要执行某 CMX 函数。因为这些 CMX 函数有可能在“代码的临界区”内，所以中断不可以直接调用这些 CMX 函数。CMX 中断管道程序会自动设置 do_int_pipe 标志说明中断管道中有内容需要执行。

另外还有两个 CMX 在大多数处理器上都使用的计数器。第一个计数器被称为 locked_out 计数器。这个计数器在函数进入或退出“代码临界区”时分别进行累加或递减。这种方法替代了对 CPU 的总中断允许标志或中断优先级屏蔽的关闭和开启。这样使得大多数 CMX 函数在返回前调用的 cx_return 函数和 CMX 中断处理程序决定调度程序是否可以执行任务切换或者因为当前运行任务正处于“代码临界区”而暂时阻塞调度程序。通常在这种情况下不会发生任务切换。然而，如果当前运行任务正处于 CMX 函数的“代码临界区”，那么当此 CMX 函数退出这个“代码临界区”时可能会发生任务的切换。

第二个计数器是 CMX 的中断计数器。CMX 用它来确定中断的嵌套层数。当存在其它被高优先级中断所打断的低优先级中断还没有完成处理时，CMX 操作系统不可能执行任务切换。

当 CMX 中断处理程序或者 cx_return 函数调用 CMX 调度程序时，将发生以下事件。CMX 的 locked_out 计数器必须是 0 以表明没有“代码临界区”的存在。CMX 中断计数器也必须是 0 以表明不存在中断嵌套现象。在上述介绍的五个标志中至少有一个必须被设置：do_coop_sched，do_timer_tsk，

preempted，do_time_slice 以及 do_int_pipe。

这就是为什么当处于 CMX 函数的“代码临界区”时，CPU 还可以使能所有的中断。在实时环境中，中断被禁止的时间应该尽可能短而不至于带来很长的中断延迟时间。举例来说，当一个 UART 以 115K 或更快的波特率来收发字符，CPU 大约每隔 87 微秒中断一次。如果中断完全被禁止，UART 就有可能接收到一个重载错误。还有许多其它的中断每隔 20 到 50 微秒就会发生。

另外还需要考虑中断的处理时间。因为中断并不直接调用 CMX 函数，所以只知道此中断完成函数的间接调用所花费的时间是表面而非实质的。每一个函数的执行时间都互相不同，因此必须分析中断所间接调用的每个函数以真正计算用于处理中断以及相关代码所花费的时间。

此外还需要注意因为中断被完全关闭所带来的最坏情况下的中断延迟时间。这个中断延迟时间是可变的量，有赖于 CMX 函数的“代码临界区”、CPU 指令集的效率以及 CPU 晶振的速度等。因为中断使用中断管道来传递函数及相关的参数，所以 CMX 有很短的中断延迟时间，而且可以很容易地估算出中断耗费的时间，一般这个值是非常确定的。

当调度程序被触发而运行时，它将根据所设置的标志来决定它需要执行的操作。调度程序首先完整地保存当前运行任务的上下文。这就使得任务轮到它运行时可以顺利地恢复运行。然后调度程序将执行由已设置的标志决定的操作。调度程序将执行 CMX 定时器任务如果对应该函数的标志已被设置的话。

然后调度程序将处理中断管道的内容，只要中断管道不是空的而且需要处理的话。之后调度程序将决定是否改变分时任务的指针，表明需要运行下一个分时任务。如果抢占标志被设置，那么调度程序将判定哪个可运行的任务具有最高的优先级并让该任务进入运行态。如果 do_coop_sched 标志被设置，那么下一个可运行的任务将进入运行态。

调度程序将决定下一个运行的任务是恢复它以前的代码执行状态还是从该任务的入口开始运行。如果任务要恢复运行，那么调度程序将恢复任务的上下文，包括 CPU 寄存器，堆栈指针等等。这就是为什么任务能够接着以前被中断的地方继续运行，通过把 CPU 寄存器恢复为原来的状态使得该任务就好象未曾被中断过一样。

调度程序还将判定是否所有的任务都处于挂起态或空闲态。如果情况的确这样，那么当前就没有一个任务可以运行。此时调度程序将调用 CMX 的进入低功耗模式的函数，让处理器进入低功耗状态只要处理器支持低功耗并且允许进入低功耗的状态。同时必须保证这种低功耗的状态可以被调用 CMX 系统滴答函数的中断所唤醒并恢复运行以退出该处理器的低功耗状态。这是因为正在执行定时的循环定时器和等待超时的任务都需要以适当的系统滴答频率来递减它们各自的时间计数值。

用户需要仔细研究根据当前所用的处理器汇编语言编写的调度程序的源代码。当然还需要仔细研究所提供的所有 CMX 函数的 C 语言源代码模块。(注意：其实所有的 CMX 函数都是单独的模块，仅仅是为了方便才把它们放入一个库。)

（完）

2005 年 5 月 9 日

飞利浦 80c51XA 系列处理器特殊说明

CMX-RTX Ver.5.0

一、 处理器特殊说明

首先，在你阅读本章内容之前，我们假设你已经阅读了《CMX 实时多任务操作系统使用手册》中的各章内容，如果你现在还没有阅读过，那么请你阅读一下。本章将详细讨论飞利浦 80C51XA 处理器。由于 CMX RTOS 能与不同的 C 编译器生产商所提供的产品一起使用，并且这些 C 编译器生产商可能会使用不同的寄存器，因此用户应该学习特定的 C 编译器使用说明以及 CMX 公司所提供的阅读文件，这种文件对用户所使用的某一特定的 C 编译器公司的产品进行了相应的说明。所使用的 C 编译器是以 Tasking C 编译器为标准建立起来的，因此你在阅读本章时，请切记这一点。任何其它公司的 C 编译器的一些变化都在其中的一个 CMX 阅读文件中进行了说明。

所有的 CMX 代码都是按照 Tasking C 编译器所支持的三种内存模式来进行编译的。这三种内存模式为：MEDIUM、LARGE、SMALL。CMX 之所以不支持 Tiny 模式，是因为在这种模式下，所有变量都被分配在直接地址空间上(1k RAM)，所以不能更好地满足 RTOS 和用户应用程序的需要。CMX 提供了一个以上的 Make 文件。使用这种文件可以制作出不同内存模式下的库代码。在使用 Tasking C 编译器的情况下，Make 文件为 Makxams.mak、Makxamm.mak、Makxaml.mak。这三种文件将能生成 Tasking C 和 CMX 所支持的三种内存模式的代码。请阅读 Tasking C 编译器使用手册，在这本手册中，你可以得到有关三种内存模式内容的相关说明，

用户可以自由地编辑 Make 文件，以满足他们调试和测试程序的需要。我们极力推荐用户对所修改的 Make 文件之处进行注释说明，以便于和原文件进行比较，这样可让用户及时掌握该文件的修改变化情况。

另外，CMX 还为用户提供了三个调度程序汇编文件来作为 Tasking 的“ 软件工具 ”。这三个文件分别是：Cxsk5ps.asm、Cxsk5nm.asm、Cxsk5nl.asm。这三种文件分别对应用 CMX 所支持的三种内存模式，因此用户可根据系统设计的需要，有选择地使用某一特定的调度文件。这三种调度文件分别适用于 SMALL、MEDIUM、LARGE 内存模式。

警告：80C51XA 处理器必须运行在“ 系统方式 ”而不能运行在“ 用户方式 ”。我们假设用户所使用的 80C51XA 处理器是运行在“ 自然 ”方式，而不是“ 兼容 ”方式。我们还假设，当用户使用 SMALL 和 LARGE 内存模式时，处理器运行在“ 非零页 ”方式。

警告：用户必须选择 0# 寄存器堆做为缺省寄存器堆。中断程序有可能使用其它寄存器堆，请进一步参阅有关这方面的内容说明。

二、CMX UART 函数

CMX 为了方便用户工作，提供了一些用 C 语言编写的 UART 函数。CMX 认为所有 UART 过程都应该采用“ 中断驱动 ”方式，而不是“ 查询 ”方式，这包括从 UART 接口中接收一个或更多的字符，以及把一个或更多的字符传送到 UART 发送器中。

CMX 的确认为所有的 UART 过程代码都应用汇编语言来编写，这样做，一方面可以让中断处理程序迅速切入和退出，从而允许 UART 能够选择更高的通迅速率(波特率)，而且也不会更多地占用 CPU 的运行时间。另一方面允许 CMX RTOS 在一个合理的时间内(尽可能地短)为其它可能发生的低优先级中断提供服务，让 CPU 把大部分时间都花在执行任务代码上。

用户可以自由地接受 CMX 所提供的 C 语言程序设计思想，并把这种思想移植到汇编语言程序设计中，以满足某些特定要求。如果用户仍然认为他们有必要使用 CMX 提供的 UART 函数的话，那么他们可以放心，大胆地使用这些函数。

用户可能在某些情况下，必须使用 C 编码的 Uart_update 函数，该函数被 UART 接收器“ 满 ”和发送器“ 就绪 ”中断所调用。由于处理器以及 UART 结构配置不尽相同，因而 CMX 不可能根据每个人的需要来编制相应的函数代码。我们认为用户可以非常容易地、精确地调整 Uart_update 函数中的某些部分代码以满足用户的特殊需要。

如果处理器和(或)用户程序需要的话，那么用户可以让相应的串行口中断处理程序设置和(或)清除任何一个特殊标志位，但两个 Uart_update 函数对一些寄存器不执行任何位操作。

三、80C51XA 中断程序以及与中断管道的接口

一个优先抢占的操作系统必须决定一个更高优先级的任务何时应该运行，从而可以抢占当前正在运行的任务。记住，一个中断可能引起一个更高优先级的任务准备成为运行任务，这会导致调度程序立即保存(在中断处理程序结束执行后)当前正在运行任务(在中断发生之前的任务)的上下文环境，并允许更高优先级的任务立即开始执行。为了允许中断嵌套，可以通过让中断程序使一个计数器的值加(和减)1 的办法来使中断处理程序知道第一层嵌套的中断是何时结束其代码执行的。注意，有些操作系统是让“ 中断操作处理程序 ”来完成对该计数器加 1 和减 1 操作的，而另外一些操作系统是让中断程序代码完成这种操作的。CMX 中断处理程序会自动地完成这项工作。另外，中断处理程序也会把所有的寄存器保存到堆栈上，并且，如果是第一个发生的中断，那么就启动中断堆栈。

下面将讲述用户应如何编写中断程序代码，以便于与 CMX 相兼容。CMX 将说明几个中断例行程序，通过这些中断例行程序可以让用户了解到中断例行程序是如何允许和禁止其它中断发生，以及是如何允许或禁止中断程序与 CMX“ 中断管道 ”和 CMX 定时器嘀哒函数接口。

另外一个需要注意的是：Cxint_in 汇编代码例程在返回到中断程序时，中断开放位(EA 位)被置位，这样就可以仍然让所有其它中断发生。该例程是直接控制 80C51XA 中断硬件系统的函数。此后，用户可以自由通过清除 EA 位禁止所有中断发生。同样，Cxint_ex 汇编代码例程也是在一个很短的时间内自动地禁止所有中断发生。

用户应仔细阅读 80C51XA 参考手册，该手册将十分详细地描述了中断以及它们是如何在处理器中工作的，这将有助于用户更好地理解本章内容，并且也会让用户更加充分地理解 CXSK5??.ASM 汇编文件。该文件包含了 cxint_in 和 cxint_ex 两个中断例行函数。

下面将说明和演示编写一个中断程序代码所使用的几种方法。

四、中断及其代码的编写

cxint_in 函数所执行的操作如下所示。对于所使用的不同内存模式，其代码可能略有差别。

下面的 cxint_in 函数源代码适用于 SMALL 模式。

堆栈结构如下所示：

；PC：(程序计数器)由硬件系统压入堆栈。注意，由于处理器是运行在“ 零页面 ”方式，所以只把 PC 的两个字节压入堆栈 。

；PSW：(程序状态字)由硬件系统压入堆栈中。

；PC：(程序计数器)在调用 cxint_in 函数之后，中断程序的返回地址。由调用指令压入堆栈中，并将被 R6 寄存器所替换。

；R5

；R4

；R3

；R2

；R1

；R0

cxint_in：

；for call (note 16bit call)

CLR IE.7；clear EA bit，so we won't be interrupted

PUSH.W R5, R4, R3, R2, R1, R0；Save all registers,except R6

MOV.W [R7+12]；get address to return to

MOV.W [R7+12]，R6；now save R6

... MORE CODE, see assembly files

PUSH.W R2；Place return address of interrupt handler that called us

RET ; return to interrup handler

正如你在上述程序中所看到的那样，所有必要的寄存器的内容都已保存到堆栈中了，因而也就不再需要中断处理程序来完成这项工作了。

警告：*在大多数情况下，中断处理程序必须用汇编语言来编写，至少中断处理程序的开头和结尾。用户可以自由地用汇编语言编写的中断处理程序调用一个 C 函数，以便让该函数完成 C 代码中断处理程序所需要的操作。另外，用户不能使用 Tasking 扩展关键词"interrupt"，因为该关健词在调用任何具体的 C 语言之前，将生成代码来保存寄存器的内容。*

在有些场合中，可能需要完全 C 语言编写中断处理程序，包括使用" interrupt"关健词，但这种中断处理程序不能调用任何的 CMX 函数(包括 cxint_in，cxint_ex，int_pipe 函数)，并且将按最高优先级(IM = 0FH)来操作。**注意：**在其它可能的情况下，中断程序是按照小于 15 的中断屏蔽等级来操作的，下面将进行有关这方面的说明。记住，任何没有调用 cxint_in 和 cxint_ex 函数的中断将导致中断程序有可能使用当前正在运行任务的堆栈来作为它自己的堆栈。这就意味着所有任务不但需要它们的堆栈尺寸要足够大以满足任务堆栈的正常使用要求，而且还要加上这个中断程序所使用的一些字节数来满足中断程序的使用需要。同样，用户也必须保证编译器保存和恢复将被使用的所有寄存器的内容。

Interrupt X routine：(该中断程序将调用 cmx_tic 函数)

Interrupt_handler：

Call_cxint_in；必须是第一条指令。

AND.B IEL, #XX；用户现在可以按照自己所喜欢的方法来操作 IE 寄存器的内　　容，记住，在调用 cxint_in 函数之前和之后，同一个中断程序的 EA 位的状态是相同的。

SWI X；用户也可以自由地使用 SWI 指令来降低中断屏蔽级或或直接使用 AND.B PSWH，#0F?H 来修改中断屏蔽码。请参阅 XA 使用手册，以保证更好地领会这方面的内容。

User code．．．；现在用户可以自由地用任何方法使用寄存器，也可以调用Ｃ代码的中断处理程序。

；切记，在函数中不可使用" interrupt 关键词。

Call _cmx_tic；

Call _cxint_ex；必须是最后一条指令。

注：在程序结束时，不能使用" RETI "指令。

Interrupt X routine：(该中断程序将调用" 中断管道 "函数)

Interrupt_handler:

Call_cxint_in；必须是第一条指令。

AND .B IEL，#XX；用户现在可以自由地利用他们所喜欢的方式来操作 IE 寄　　存器的内容。

注意：*程序执行到这一步时，IE 寄存器中的 EA 与在调用 cxint_in 函数之前的 EA 位的状态是相同的。*

SWI X；用户现在可以自由地使用 SWI 指令降低中断屏蔽等级或直接使用 AND.B PSWH，#0F？H 指令。请参阅 XA 使用手册，以确保更好地领会这方面的内容。

Jsr c_coded_interrupt_work_handler；现在用户可以调用 C 代码函数，该函数代码中使用了中断管道，这样就可直接在中断程序中调用一个 CMX 函数。请参看" 可由中断程序调用的 CMX 函数 "这一

节中的内容，它将告诉你具体的操作方法。

More. code．．．；记住：在函数中不可使用＂interrupt”关健词．

Call_cxint_ex；必须是最后的一条指令．

注：在程序结束时，不能使用“RETI”指令。

如果一个中断程序不打算调用任何 CMX 函数或 CMX_TIC 函数，那么中断程序就不必调用 CMX 中断处理函数。下面将演示几种具体的实施方法：

下面将演示一个中断程序，该中断程序将禁止其它中断的发生。注意，所使用的堆栈可能是当前任务的堆栈，也可能是中断堆栈。

Interrupt x routine：(该中断程序将不使用“ 中断管道 ”，也不会调用 cmx_tic 函数)

INTERRUPT VECTORS

ORG ???

DW 8F00，interrupt_handler；IM 中的中断屏蔽码为 0XF。

Interrupt_handler：

；把所有可能遭受破坏的寄存器中的内容保存起来。

User code．．．；用户在可能用任何方法来使用寄存器，也可以调用Ｃ代码的中断处理程序。

；恢复所有被保存的寄存器内容。

RETI；从中断程序中返回。

下面所演示的中断程序将允许其它中断发生，但这些中断程序不能调用 CMX 中断管道或 cmx_tic 函数。

注意：用户在编写这种中断代码时，要非常谨慎小心。

Interrupt x routine：(中断程序不会调用“ 中断管道 ”或 cmx_tic 函数)

INTERRUPT VECTORS

ORG ???

DW 8F00，interrupt_handler；注：IM 屏蔽码为 0XF。

Interrupt_handler：

；现在要屏蔽掉任何可能使用 CMX 中断管道或 cmx_tic 函数的中断。这可能通过清除所有可能将调用 CMX 中断管道或 cmx_tic 函数中断的相应的中断允许位来实现。

；保存所有可能遭受破坏的寄存器的内容。

；关闭所有可能调用 CMX 函数的中断允许位。

SWI X；注意：用户现在可以自由地使用 SWI 指令来降低中断屏蔽等级，或直接使用 AND.B PSWH，#0F？H 指令。请参阅 XA 使用手册，以确保你更好地理解这方面的内容。

User code．．．；用户现在可以自由地利用各种方法来使用寄存器，也可以调用Ｃ代码中断处理程序。

CLR IEL.T；再次关闭所有中断。

；重新开放各中断允许位。

；恢复所有被保存的寄存器内容。

RETI；从中断程序中返回。

下面的程序代码也可能适用于某些中断，这些中断将允许其它的，调用了 int_pipe 或 cmx_tie 函数的中断发生。如果用户要想实施这种代码，那么 cmx 希望用户理解这样一种可能情况：如果另外一个中断随之发生，并调用了 int_pipe 或 cmx_tic 函数，那么根据该中断是否是第一次出现来确定立即重新调度操作是否像通常情况下那样进行。另外，是否使用中断堆栈，还要取决于该中断发生时与其它调用了 cxint_in 和 cxint_ex 函数的中断之间的相互关系。

警告：这种中断程序不应该让没有经验的编程人员来完成，如果确实想要这么做的话，那么应该在事先要有一个很好的构思，并要在实施过程中加倍小心。

Interrupt x routine：(该中断程序不会调用“ 中断管道 ”或 cmx_tic 函数)

INTERRUPT VECTORS

ORG ???

DW 8F00，interrupt_handler；

Interrupt_handler：

ADDS.B _int_count，#1；CMX 中断计数器加 1。

；保存所有可能受到破坏的寄存器内容。

SWI X；注，用户现在可以自由地使用 SWI 指令来降低中断屏蔽级，或直接使用 AND.B PSWH，#0F?H 指令。请参阅 XA 使用手册，以确保你更好地理解这些内容。

Code．．．；用户代码。

；恢复所保存的寄存器内容。

CLR IEL.T；再次关闭所有中断。

ADDS.B _int_count，#_1；CMX 中断计数器减 1。

RETI；返回到被中断的程序代码中。

五、可被中断调用的 CMX 函数

在任何情况下，调用了 int_pipe 函数后，都将返回一个“ good ”或“ error ”状态。如果中断管道满，或者某一特定的 CMX 函数没有被用户在 Cxconfig.H 文件中选择允许，那么“ error ”状态就可能发生。

记注，为了让中断处理程序符合上述规则，对于那些调用了 int_pipe 和 cmx_tic 函数的中断程序，必须使用 cxint_in 和 cxint_ex 函数作为程序的开头和结尾。我们假设用户已经这样做了，所以我们在这里就没有必要再做进一步的提示了。

下面列举了一些可以被中断程序调用的 CMX 中断函数以及和它们等效的 CMX 函数，并带有对它们的简要功能说明。每一个 CMX 中断函数所使用的参数完全与等效的 CMX 函数一样。

CMX 中断函数	**等效的 CMX 中断函数**	**说明**
int_cxtwake	twake	唤醒一个任务
int_cxtwakf	cxtwakf	强制唤醒任务
int_cxttrig	cxttrig	触发一个任务
int_cxctstp	cxctstp	停止一个循环定时器
int_cxctrstt	cxctrstt	重新启动一个循环定时器
int_cxctrsti	cxctrsti	按新的初始时间重新启动一个循环定时器
int_cxctrstc	cxctrstc	按新的循环时间重新启动一个循环定时器
int_cxmssend	cxmssend	发送一条消息到邮箱中
int_cxesig	cxesig	设置一个或更多的事件位
int_ cxsempst	cxsempst	邮递旗语

警告：*CMX“ 中断管道 ”不能用任何方法进行测试判断某一特定的 CMX 函数参数的正确性。*

下面 C 代码编写的“ C_interrupt_send_routine ”函数，该函数将被用汇编代码编写的中断处理程序所调用。

```
#define MailBox_0 0
char int_mesg [ ] = “ This is the interrupt message ”；
Void C_interrupt_send_routine(void)
  {
    /*现在发送一条来自中断程序的消息。注意：CMX 中断函数的参数的传递方法与非中断环境
      下的，等效的 CMX 函数是完全一样的。
                  */
```

int_cxmssend(MailBox_0，int_mesg)；

六、与 C 中断处理程序的接口

用户必须像在上述例子中那样，编写中断处理程序代码。

Interrupt X：

Call_cxint_in；

Call_c_INTERRUPT_HANDLER；进入 C 代码的中断处理程序。

Call_cxint_ex；

注意：结束时，没有使用 RETI 指令。

七、CMX_RTX 汇编模块的配置

CMX 汇编模块文件可以按照用户的要求进行编辑。有几个开关选项需要设置为 1 或 0。这样可以允许汇编器有条件地汇编成正确的代码，以适用于特定的结构配置，以便于 CMX 汇编模块能够与 C 模块正确地工作(这些 C 模块有可能已被配置成与 CMX 所默认的不同的构成方式)。

CMX 汇编模块(对于 80c51XA 系列)有几个开关选项需要用户来调整。下面将解释说明这几个开关选项不同的设置以及设置到 1 或 0 的结果。

LOW_POWER_ACTIVE EQU (1 or 0)

_CMX Default setting = 0. (for test program)

该开关选项在所有任务都处于等待状态或空闲状态(意味着当前没有运行任务)时，分别允许或禁止 CMX 调度程序调用 Cmx_power 函数。如果该开关选项被设置为 1，那么 CMX 调度程序在需要时，将可以调用 cmx_power 函数。否则，CMX 调度程序即使在需要的时候，也不能调用该函数。

注意：cmx_power 函数需要由用户编写，它被定位于 cmx_init.c 模块中。

SLICE_ENABLE EQU (1 or 0)

_CMX Defanlt setting = 1. (for test program)

该开关选项具有允许或禁止“ 时间分片(TIME SLICING) ”机制是否变成有效的能力。该开关选项是与 CMX slice_enable 和 slice_disable 函数一起使用的，这两个函数将决定是否真的启动或关闭“ 时间分片 ”机制。如果该开关选项被设置为 1，那么 CMX“ 汇编 ”模块将允许调用上述两个函数，以确定是否进行“ 时间分片 ”，否则，即使在程序中调用了这两个函数，那么 CMX 也不会进行“ 时间分片 ”的操作。

附录：在中断程序中调用 CMX 函数的新方法

CMX 已修改了在中断程序中调用 CMX 函数的方法。原先的方法是通过寄存器传递参数来调用 CMX 函数的方法。之所以采用这种方法，是出于对执行速度的考虑。在分析代码和听取用户意见之后，我们现在决定用 C 语言来编写能被中断程序调用的 CMX 函数。这样可产生一个更为“ 清楚 ”的接口，尽管这种做法与汇编方法相比，中断程序的执行速度和执行效率受些影响，但我们应认为这样会更便于用户使用。

大许多情况下，中断处理程序应被要求用汇编语言编写，这个原则是不变的。在“ 处理器特殊说明 ”这一章中，描述了中断处理程序为了某一特定的目标和用户所使用的编译器而应该采取的代码编写的正确方法。现在我们将为用户详细说明和演示用新的方法来调用 CMX 函数。

原先的调用 CMX 函数的方法如下：

irq_handler：

．．．；中断程序前言，参看“ 处理器特殊说明 ”这一章中的内容。

．．．；用户的中断处理程序代码。

；用户现在希望中断把一条消息发送到０＃邮箱中。

MOVE .B #7，ROL；cxmssend 函数标识符。

MOVE .L #_int_mesg，ER5；消息的在址。

MOVE .B R5H，R4L；把 R5 的高字节放入 R4L 中。

MOVE .W E5，R3；把 E5(高 16 位)的内容传送到 R3(R3L 将具有 16-23 位)中。

mov.B R3H，R2L；把 24_31 位传送到 R2L 中。

JSR @_int_pipe；启动中断管道。

．．．；其它用户代码。

．．．；中断程序结尾，参看“处理器特殊说明”这一章中的内容。

就象你所看到的那样，原先的方法要求用户操作使用寄存器进入一个特殊的预先确定的模式。尽管这种方法我们已经做了详细的说明，但许多用户应然感觉难以理解和掌握。

新的调用 CMX 函数方法如下：

irq_handler：

．．．；中断程序前言。

．．．；用户代码。

；用户现在想利用中断程序把一条消息发送到０＃邮箱中。

JSR @interrupt_send_routine；现在将调用 C 代码的例程发送一条消息。

．．．；用户其它中断处理程序代码。

．．．；中断程序结尾。

“interrupt_send_routine”函数的 C 代码如下所示：

```
#define MailBox_0  0
char int_mesg [ ] = “ This is interrupt message ”；
Void interrupt_send_routine(Void)
  {
    /*现在从中断程序中发送一条消息。注意：CMX 中断函数所使用的参数与在正常情况下所使
      用的 CMX 函数的参数，在传递方式和使用要求是完全一致的。*/
    int_cxmssend(MailBox_0，int_mesg)；
  }
```

如果用户愿意的话，技术上讲，他们可以让编译器生成上述函数的汇编列表文件，并且还可以把所生成的主要的汇编指令拷贝到他们的汇编代码的中断程序中；删除 C 代码函数的“CALL”(CALL 是处理器的相关的汇编指令)，并删除“RET”指令。这样做可以节省几个时钟周期。

#define C_pipe_size(该定义语句被放置于 coconfig.h 文件或复本中)语句用来指明放置在“中断管道”中的 CMX 中断函数的数量。这个数量只能为 2^n(1、2、4、8、16、32、．．．)，并且范围从１到 255。

新方法中也不再使用 int_pipe 函数了，代替该函数的等效的 C 代码函数为 pipe_in(定位于 cmx_init.C 或其复本中)。“中断管道”数组现在是一个结构数组，数组中每一个结构元素的尺寸是由最大的 CMX 中断函数及其相关参数所决定的，这样的函数通常是 cxmssend 或 cxesig 函数。

结构的定义如下所示，这种结构可随处理器和编译器的内存模式而升级。

```
Typedef  struct  pipe_element {
byte  identifier；/*函数标识符，被 switch/case 语句所使用*/
byte  p1；/*参数 1 */
byte  p2；/*参数 2 */
union {
        word16  p3；/*参数 3 */
```

```
        viod  *p4；/*消息地址*/
      } pipe_u；
} PIPE_STRUC；
PIPE_STRUC  pipe [C_PIPE_SIZE]；
```

CMXTracker 使用手册

CMXTracker 代码分析程序提供了在 CMX RTOS 环境下，在应用程序运行期间，以实时方式，按时间顺序跟踪记录任务执行流程，捕获正在执行的任务、被调用的 CMX 函数及其参数、使用 CMX 函数的中断和 CMX 系统嘀哒时钟的能力，并能把由 CMXTracker 任务(通常作为最高优先级的任务)所完成的跟踪记录的内容显示出来。在大多数情况下，把某一个目标处理器的 UART 通道当作输入／输出设备来使用。带有键盘的一个简单终端或 CPU 就可以满足使用 CMXTracker 的设备要求。

当 CMXTracker 任务被启动时，它将把一个菜单传送到显示器上。如果选择了菜单上所提供的某一项功能提示，用户就可以观察按时间顺序排列的跟踪记录内容，复位运行日记，重新恢复运行应用程序，或者在一定数量的运行日记输入项或系统嘀哒时钟之后，“ 自动地唤醒 ”CMXTracker 任务。当 CMXTracker 任务正在运行时，它将阻止其它任务运行，停止任务定时器和循环定时器，并且禁止中断程序调用 CMX 函数，因此，应用程序被“ 冻结 ”在 CMX ROTS 环境中。

CMXTracker 允许用户来观察运行日记的开头和结尾处的内容，并能通过上下翻页来查看中间内容。各个任务准确的执行过程，被调用的带有参数的 CMX 函数及其返回结果(例如被发送或被接收的消息、被设置的事件位、超时等)和中断程序的执行情况都和 CMX 系统嘀哒时钟(是一个“ timeline ” stamp)一起被显示出来。

CMXTracker 任务具有“ 单步 ”一个系统嘀哒时钟的能力，也就是说允许应用程序执行一个系统嘀哒时钟的时间，并且在这个“ 单步 ”之后，CMXTracker 任务再重新恢复运行。另外，CMXTracker 所要等待的时间是可以根据用户的需要来进行修改。这是一个非常重要的和有益的特殊功能。

应该把一个使用串口来发送和接收字符的终端，或者任何一台带有串口和通讯软件(如 Procomm，Mirror 等)的 PC 与目标处理器连接起来。对于通讯软件，你也可以使用 CMXBugio.exe。所选择的通讯软件应该足够快，并能以 9600 波特率来接收和显示字符而不丢失它们。

CMX 提供了一个 CMXBugio.exe 程序，它允许 PC 利用其 com1 或 com2 口来发送和接收字符。这是一个中断驱动程序，它允许接收发送字符而不丢失所接收到的字符。该程序在一个正常的“ 轮询 (Polled) ”环境中，将以一个更高的波特率进行通讯。为了使用 CMXBugio.exe 程序，目标处理器的 UART 接口必须被配置成 9600 波特率，8 个数据位，一个停止位，无效校验(9600N81)。

该手册不打算更为详细地叙述 CMX RTOS 以及 CMX 函数的调用方法。我们假设认为用户已阅读了 CMX 手册，并能熟练地使用 CMX 函数及其参数，并对 CMX RTOS 是如何工作的有一个初步的理解和领会。

警告：请阅读该手册末尾的那一节的内容，既“ CMXTracker 处理器特殊说明部分 ”这一章节。该节将为你在使用 CMXTracker 任务时，提供一些对处理器和 C 编译器额外的特殊要求和规定。

建立和设置 CMXTracker 任务的方法：

警告：CMXTracker 任务与 CMXBug 任务不能在同一个应用程序中一起使用。

CMX 具有让 CMXTracker 任务代码包含或不包含在 CMX 函数中的能力。如果在 CMX 函数中包含了 CMXTracker 代 码，那么将会使 CMX 函数的代码尺寸增加，执行时间也会随之延长。这是因为需要 CMXTracker 任务把每一个 CMX 函数在执行过程中的状态和数据记载到运行日记中。为了让 CMXTracker 代码被包含在 CMX 函数中，必须使用 C 编译器生产商所提供的编译开关，来通知编译器把 CMXTracker 代码包含在 CMX 函数中。这个编译开关相当于一条“ define CMXTracker ”宏定义语句。在大多数情况下，这个编译开关是一个“ _D ”，在这种情况下，命令行的格式如下所示：Compiler_EXE_Name_D CMXTRACKER file.c。由于 CMX 为 CMX 函数库提供了一个 MAKE 文件，因此，所有需要来重建库文件的那些事情，都要编辑 MAKE 文件(*.mak)，通过增加适当的编译器开关，来包含这个全局型的宏定义。CMX 推荐使用 MAKE 文件来生成两种函数库，一种函数库包含 CMXTracker 代码，另一种不包含 CMXTracker 代码。

提示：即使把 CMXTracker 代码包含到 CMX 函数中，CMXTracker 代码也不会得到执行，除非通过编译开关设置，允许 CMXTracker 运行。下面将通过 一个简单的函数来说明 CMXTracker 的使用方法。

```
Cmx_function()
{
  …                              /*函数的正常代码*/
  #ifdef     CMXTRACKER  /*判断是否包含了 CMXTracker 代码*/
 if (CMXTRACKER_ON)    /*用户允许 CMXTracker 来记录 CMX 函数执行的各项数据吗?如果
                                是，那么就进行记录*/
    {
     cmxtracker_in(THIS_FUNCTION)；/*记录该项数据*/
    }
  #endif
  …/*返回到函数的正常代码*/
}
```

在 CMX 的头文件“ cxconfig.h ”中，有两个与 CMXTracker 有关的宏定义：

CMXTRACKER_ENABLE 和 CMXTRACKER_SIZE。其中 CMXTRACKER_ENABLE 是用来在 CMX_INIT.C 模块中定义允许或禁止 CMXTracker。CMX_INIT.C 模块被 CMX 所使用，并按配置文件中的要求来分配所需要的内存和变量。当 CMXTRACKER_ENABLE 被定义为 1 时，允许启动 CMXTracker，为 0 时，则禁止。CMXTRACKER_SIZE 用来在 CMX_INIT.C 模块中定义 CMXTracker 运行日记所需要的 RAM 空间的尺寸。这个尺寸应该足够大，以便于记录一定数量的跟踪数据。CMX 尽量压缩运行日记中跟踪记录的数量。在大多数情况下，进入到运行日记中的数据输入项将占用 3 到 8 个字段。CMX 推荐 CMXTRACKER_SIZE 至少也要设置到 1K(1024byte)，最好应该设置到 4K 以上。

CMXTracker 可以根据需要进行修改，以满足你的特殊要求。“ CMX_INIT.C ”文件中包含了 3 个函数，这 3 个函数和一个全局变量都可以按用户要求来进行修改。

CMXTracker 采用“ 轮询(Polled) ”方式来测试判断处理器的串口是否有字符出现。CMXTracker 任务每隔 10 个 Tick(缺省值)就被唤醒来测试判断是否有“ ＋ ”键被按下。如果该键没被按下，CMXTracker 将转入另外一个 10 个 Tick 的睡眠期。这种状态和过程被多次重复，直到“ ＋ ”键被按下为止。CMXTracker 的睡眠时间可以根据用户需要，增加或减少。这个决定 CMXTracker 睡眠时间的变量是出现在 cmx_init()函数中的 BUG_WAIT_TICKS.

BUG_WAIT_TICKS = 10；

可以通过修改 setup_bug()函数来选择所希望的串口。这个函数是通过配置波特率，数据位、奇偶校验位等来设置串口的。该函数也可以通过对任何相关的寄存器标志的清零和／或置位来使串口有效。如果使用了 CMXBugio.exe 文件，那么目标处理器的 UART 串口必须被配置成 9600 波特率，8 个数据位，一个停止位，无效验位(9600N81)，同时，该串口必须被配置成“ 轮询(Polled) ”方式，而不是中断驱动方式。

Bug_getchar()函数使用由 setup_bug()函数所设置的串口来为 CMXTracker 任务接收字符。它即可以，也可以不清除/置位任何必要的接收器标志，这取决于所使用的处理器。

Bug_putchr()函数使用由 setup_bug()函数所设置的串口来为 CMXTracker 任务发送字符。它即可以，也可以不清除/置位任何必要的发送器标志，这取决于你所使用的处理器情况。

你虽然可以自由地更改所希望的串口，但你必须保证你所使用的处理器带有这种串口，并能够对任何接收器或发送器标志进行清除和置位操作。

你不要尝试让监控程序、用户应用程序等都同时使用一个串口来发送和接收字符，否则，你就很

可能遇到一个冲突问题。

CMX 汇编文件也有一个“开关”，这个“开关”用来确定是否包含 CMXTracker 代码，这个“开关”被称为 CMXTRACKER_ENABLE。如果要想汇编 CMX 汇编模块，使其具有与 CMXTracker 代码接口的功能，那么该“开关”应被设置为 1。否则的话，CMX 汇编模块就不会与 CMXTracker 代码一起被汇编。如果这个“开关”被设置为 off，同时又链接了其它 CMXTracker 模块，那么将不会执行对一个任务进行跟踪记录操作的切换。CMX 可以编制一些汇编模块代码来检测是否允许了 CMXTracker 代码，但这样做将会增加一些额外的代码和上下文切换时间。因此，我们设计了这样一种“开关”，它必须由用户来允许/禁止，其缺省设置为禁止。

CMX 提供了把一个“user entry”放入到 cmxtracker 运行日记中。Cmxtracker_user()函数就可以完成上述功能，该函数的原型如下所示：

Void cmxtracker_user(unsigned char user_num)；

如果用户调用了这个函数，CMX 推荐在每次调用该函数时，其参变量 user_num 的值按 1 来递增。对于每一个任务，允许最多连续调用该函数 256 次。由于 CMX 能够自动地识别是哪一个任务调用了这个函数，因此，每个任务可以使用不同的用户数(use_num)来连续调用该函数 256 次。

CMX 还提供了另外一种函数，该函数可以用来声明在 CMXTracker 运行日记被“冻结”之前(不允许更多的跟踪记录项被输入)，应该接收多少个跟踪记录项，或者必须发生多少个系统嘀哒时钟数。因此该函数提供了一种在所接收的跟踪记录的数量或所发生的系统嘀哒时钟数达到规定值时，能够使 CMXTracker 任务被自动地唤醒的能力。该函数的原型如下所示：

Void cmxtracker_mode(unsigned char MODE，unsigned short COUNT)；

COUNT：是由 MODE 参数所规定的，在跟踪日记停止记录任务执行流程，所调用的 CMX 函数等之前，所必须出现的跟踪记录项的数量。

MODE：用来规定 CMXTracker 应该如何执行。该字节参数的各位所代表的功能如下：

Bit0=没被使用

Bit1=没被使用

Bit2=COUNT 参数按每一个系统嘀哒时钟，减 1。

Bit3=COUNT 参数每次在记录一个跟踪数据项时，减 1。

Bit4=没被使用

Bit5=没被使用

Bit6=没被使用

Bit7=当 COUNT 被减到 0 时，自动唤醒 CMXTracker 任务。

注：为跟踪日记所声明的内存(RAM)尺寸，一定要大于 COUNT 值，否则 COUNT 参数会被忽略。

有两种方法可以用来启动跟踪日记(LOGGING)。第一个方法是把 CMXTRACKER_ON 变量设置为 1；第二个方法是通过按“+”键使 CMXTracker 任务被唤醒，然后对 CMXTracker 任务进行“QUICK GO and RESUME”或“GO and RESUME”或“EXIT CMXTRACKER function”操作。当你把 CMXTRACKER_ON 变量设置为 0 时，这样就可以禁止对跟踪数据的记录，CMXTracker 任务不会被启动。然而，即使你把这个变量设置为 0，但又按了“+”键唤醒了 CMXTracker 任务，并且进行了上述功能操作，那么在这种情况下，CMXTRACKER_ON 变量仍会被强制地设置为 1，于是仍可允许记录跟踪数据。

使用 CMXTracker 任务：

CMXTracker 是一个任务，它必须被包含在应用程序中才能被使用。该任务的源代码文件是“CMXTracker.c”文件。CMX 函数库(包含各种 CMX 库函数)必须与“define CMXTRACKER“宏语句一起被编译构建(build)，这样才能使 CMXTracker 代码被嵌入到 CMX 库函数中，以便于允许 CMXTracker 任务正确地工作。

为了启动 CMXTracker 任务，可以按下控制台输入设备上的“+”键即可。如果使用了

CMXTracker 的“ 自动唤醒 ”功能，就不必再按“ + ”键，因为 CMXTracker 任务会被自动地触发。一旦 CMXTracker 任务被触发，它将取得对 CMX RTOS 系统的控制权。然后，CMXTracker 将提供一个如下所示的菜单：

Select the desired function

Enter 1 to DISPLAY LOG

Enter 2 RESET LOG

Enter 3 GO and RESUME CMXTracker

Enter 4 QUICK GO and RESUME CMXTracker

Enter 99 EXIT CMXTracker

Enter P/p to toggle ECHO mode

Your choice?

下面将详细地解释上述每项功能的具体含义。首先我们先要说明一下你将要看到的某些数据项提示符的含义。

由于 CMXTracker 通常会通过 UART 与终端或 CPU 交互，因此一般的数据项是受到限制的。另外，为了把 CMXTracker 任务的代码尺寸降至最少，在<Return/Enter>键被子按下之前，是不会分析所键入的数据正确与否。

下面是一些被经常频繁使用的“ 键 ”：

<PgDn>:向下翻阅跟踪日记一页，每一页中有 18 项跟踪记录内容。

注：如果达到或接近跟踪日记的末尾，那么也许可能不会显示出完整一页内容。

<PgUp>：向上翻阅跟踪日记一页，每一页中有 18 项跟踪记录内容。

<Home>：切换到跟踪日记的第一页上。

<END>：切换到跟踪日记的最后一页上。注意：最后一页也许不是完整的一页。

使用“ <return>to exit ”能够返回到上一级菜单上。为了退出当前功能，必须首先按<return>键。

注：在任何时候，只要你按下“ + ”键，就可以返回到主菜单上。

下面我们将把不同的 CMX 函数以及在所选定的功能被显示出来时，你将看到的相关“ 画面 ”展示给你。

注：根据不同的 CMXTracker 版本，这个“ 画面 ”中的内容可能略有差别。

首先被显示的是被用户代码、任务代码或中断处理程序所调用的 CMX 函数的名称，然后，我们为你显示这个函数调用所得到的三种结果之一：INFORMATIONAL，SUCCESSFUL 或 UNSUCCESSFUL。有些函数带有“ 时间参数 ”，在一般情况下，该参数的值会被显示出来的。但是，如果在另外一个可能存在的跟踪数据项出现之前，由该函数所规定的等待时间周期到，那么将会显示出“ Timed Out ”。由一个 CMX 函数而引起产生的一个以上的错误代码，这种情况是可能存在的。对于每一个 CMX 函数，我们将完全列表显示出 INFORMATIONAL，SUCCESSFULand UNSUCCESSFUL 的结果。我们不会总是把一个 CMX 函数拆分成若干个最小的，失败的结果代码，因为这样做会需要更多的代码空间，并且还会增加函数的执行时间。注：INFORMATIONAL 被显示为 INFO，SUCCESSFUL 被显示为 GOOD，UNSUCCESSFUL 被显示为 ERROR。

如果一个任务是通过使用 cxtname()函数来被命名的，那么该任务名最多只能为 12 个字符。如果没有为任务取名，则将显示“ slot #?? ”。我们所列表显示的“ TASK NAME ”是用来指示任务名或插槽号。另外，“ ? ”代表由 CMXTracker 填写的值。

CMX 推荐，在 CMXTracker 工作期间，使用 cxtname()函数来为任务命名。注：当你在调用 cmx_go()函数，进入 CMX RTOS 之前，你尽管调用了一些 CMX 函数，但不会显示出任何任务，因为当前没有可运行的任务。

下面是三个例子，第一个是在调用 cmx_go()函数，进入 RTOS 之前，调用一个函数的例子，第二个是在没有为一个任务取名的情况下的例子，第三个是在为一个任务取了名的情况下的例子。

USER CODE…

SLOT #??…

TASK NAME…

警告：如果 CMXTracker 在跟踪日记中不认识某一个特定的跟踪记录项，那么下列的一条提示信息将会出现。如果这条住处出现了，那么该项跟踪记录后面的其它所有记录项都将被看成无效。用户可以放心大胆地调用ＣＭＸ函数，因为这种情况一般不会轻易出现。

ERROR：“ Not Valid command ”

下面的这些“ 关键词 ”在任务使用它们时，将以大写字母方式被显示。

>>>CMX Tick<<<:它表示 CMX“ 系统嘀哒时钟 ”已经经发生了。它通常被当作“ time line ”来使用，用于由任务和中断所引起的系统调用。

EXECUTING：它表示所显示的任务是当前正在运行的任务。当执行调度程序时，你不会在离开调度程序之前获得另外一个正在运行的任务，除非另外一个任务变成一个“ 新 ”的运行任务。

INFO：“ TASK NAME EXECUTING ”

-- CYCLIC ?? CXESIG：表明已被创建并被启动的循环定时器在其可编程定时周期到时，已被自动地执行了。另外，还表明循环定时器在调用 cxesig()函数时，是否成功。如果调用成功了，那么它将列出所有执行的“ mode ”值。注：“ mode ”值是在循环定时器被创建时所设置的。任名、插槽号或优先级将按照所选定的“ mode ”值被显示出来。

INFO：“ -- CYCLIC ?? CXESIG，… ”

++INTERRUPT：这意味着调用一个函数的，是一个中断，而不是一个任务。它不会识别是哪种中断代码调用了该函数，只有在中断管道以外，所有执行的函数，才会知道是哪一种中断代码调用了它。

INFO：“ ++INTERRUPT… ”

下面的内容紧随在调用这些函数的用户程序，任务代码或中断程序的后面：

CXTCRE:表示调用了 cxtcre()函数。

GOOD：“ CXTCRE，successful ”

ERROR:“ CXTCRE，ERROR ”

(2)CXTTRIG:表示调用了 cxttrig()函数。

GOOD:“ CXTTRIG，successful，TASK NAME ”

ERROR:“ CXTTRIG，ERROR→TASK NAME ”

(3)CXTPRI:表示调用了 cxtpri()函数。

GOOD:“ CXTPRI，Successful， TASK NAME， MEW PRI.is?? ”

ERROR:“ CXTPRI，ERROR→TASK NAME ”

(4)CMTRMV：表示调用了 cxtrmv()函数。

GOOD:“ CXTRMV，Successful， TASK NAME ”

ERROR:“ CXTRMV，ERROR→TASK NAME ”

ERROR:“ CXTRMV，ERROR→BUSY ，TASK NAME ”

(5)CXTWAK:表示调用了 cxtwake()或 cxtwatef()函数。

GOOD:“ CXTWAK，Successful， TASK NAME ”

ERROR:“ CXTWAK，ERROR→TASK NAME ”

ERROR:“ CXTWAK，ERROR→not waiting， TASK NAME ”

(6)CXTWATM:表示调用了 cxtwatm()函数。

当调用该函数时，一直显示：“ CXTWATM TIME PERIOD=???? ”。如果定时周期满，或者调用了 cxtwake()/cxtwakf()函数，那么会显示出另外两条信息中的其中之一。

INFD：“ CXTWATM TIME PERIOD=???? ”

INFO：“ CXTWATM Time out ”

INFO：“ CXTWATM Woken by CXTWAK ”

(7)CXTEND:表示调用了 cxtend()函数。

INFO：“ CXTEND ”

(8)CXMSGET:表示调用了 cxmsget()函数。如果该函数调用成功，那么将会显示出“ MBOX #?? ”。另外，已被接收到的消息的内容超过 12 个字符，在显示出前 12 个字符以后，紧随 3 个圆点。(例如：Mesg.recv＝messaye stri…)

GOOD:“ CXMSGET，Successful， MBOX #??，Mesg.recv＝MESSAGE ”

ERROR:“ CXMSGET，ERROR→range，MBOX #?? ”

ERROR:“ CXMSGET，ERROR→MBOX owned ，MBOX #?? ”

ERROR:“ CXMSGET，ERROR→no message，MBOX #?? ”

(9)CXMSWATM:表示调用了 cxmswatm()函数，如果该函数调用成功，那么将显示出“ MBOX #?? ”。另外已被接收到的消息的前 12 个字符也会被显示出来。如果所接收到的消息内容超过 12 个字符，那么在显示前 12 个字符以后，后随 3 个圆点。(例如：Mesg.rev＝MESSAGE Stri…)

INFO:“ CXMSWATM，MBOX #?? ”

GOOD:“ CXMSWATM，Successful，MBOX #??，Mesg.recv＝MESSAGE ”

ERROR:“ CXMSWATM，ERROR→Range， MBOX #?? ”

ERROR:“ CXMSWATM，ERROR→MBOX owned，MBOX #?? ”

ERROR:“ CXMSWATM，ERROR→Timed out no message，MBOX #?? ”

(10)CXMSSEND:表示调用了 cxmssend()函数。如果该函数调用成功，那么将显示出“ MBOX #?? ”和邮号 ”。另外，正在被发送的消息的前 12 个字符也会被显示出来。并且这前 12 个字符以后的其它的字答不被显示而用圆点代替。(例如：Mesg.sent＝message stri…)。

GOOD:“ CXMSSEND，Successful，MBOX #??，Mesg.sent＝MESSAGE ”

ERROR:“ CXMSSEND，ERROR→Range/No，MSG slots，MBOX #?? ”

(11)CXMSSENW:表示调用了 cxmssenw()函数。如果该函数调用成功，那么“ MBOX #?? ”和邮箱号将被显示出来。另外，正在被发送消息的前 12 个字符也会被显示出来。并且这前 12 个字符以后的其它的字符不被显示，而用圆点代替。(例如：Mesg.sent＝message stri…)。

INFO：“ CXMSSENW MBOX #?? ”

GOOD:“ CXMSSENW，Successful，MBOX #??，Mesg.sent＝MESSAGE ”

GOOD:“ CXMSSENW，Successful，ACKED by receiving task，MBOX #?? ”

ERROR:“ CXMSSENW，ERROR→Range/No，MSG slots，MBOX#?? ”

ERROR:“ CXMSSENW，ERROR→Timed out WAITING for ACK，MBOX #?? ”

(12)CXMSACK：表示调用了 cxmsack()函数。如果调用成功，那么将显示出任务名/插槽号。

GOOD:“ CXMSACK，Successful，TASK NAME ”

(13)CXMSBXEV：表示调用了 cxmsbxev()函数。如果调用成功，那么将显示出“ MBOX #?? ”和邮箱号。

ERROR:“ CXMSBXEV，ERROR→Range，MBOX #?? ”

GOOD:“ CXMSBSEV，Successful，MBOX #?? ”

(14)CXESIG：表示调用了 cxesig()函数。如果函数调用成功，那么将列出所执行的“ mode ”值。任务名/插槽号或优先级也将按照所选择的方式值被显示出来。另外，任务的那些将被设置的事件位也将以 0X????格式显示出来。注：每个任务只能设置 16 个事件位以及这些事件位的任何组合形式。

注意：“ mode ”值的范围从 0 到 6。下面的输出是在不同的“ mode ”值下产生的。

GOOD:“ CXESIG，Successful，MODE 0，TASK NAME ”

GOOD:“ CXESIG，Successful，MODE 1，TASK NAME ”

GOOD:“ CXESIG，Successful，MODE 2，TASK NAME ”

GOOD:“ CXESIG，Successful，MODE 3，ALL TASKS ”

GOOD:“ CXESIG，Successful，MODE 4，ALL TASKS Waitting ”

GOOD:“ CXESIG，Successful，MODE 5，ALL TASKS Same PRI.”

GOOD:“ CXESIG，Successful，MODE 6，ALL TASKS Same PRI. Waitting”

ERROR：“ CXESIG，ERROR→mode#?? ”

ERROR：“ CXESIG，ERROR→TASK NAME ”

(15)CXEWATM：表示调用了 cxewatm()函数。当一个任务调用了该函数时，将显示出该函数的等待时间周期参数(该参数是用来规定调用者希望等待至少有一个事件位被设置的时间量)。如果所规定的等待时间周期满，那么将显示“ Event bits match＝0X???? ”，其中那些被设置为 1 的位(在二进制下)所对应的事件是调用者正在等待发生的事件。

INFO：“ CXEWATM Puttime() ”

INFO：“ CXEWATM，Timed Out ”

GOOD：“ CXEWATM，Successful，Event bits match＝0X???? ”

(16)CXERST：表示调用了 cxerst()函数。如果函数调用成功，那么将显示任务名/插槽号，并且那些需要被复位的事件位也将以“ 0X???? ”显示出来。注意：如果某一位为设置为 1，那么它所对应的事件发生标志位将被复位。

GOOD:“ CXERST，Successful，TASK NAME Event bits reset＝0X???? ”

ERROR：“ CXERST，ERROR→TASK NAME ”

(17)CXBFCRE：表示调用了 cxbfcre()函数。如果函数调用成功，那么内存块的地址 BASE 将以十六进制格式(0X????)显示出来。

GOOD:“ CXBFCRE，Successful，Memory Block BASE address 0X???? ”

(18)CXBFGET：表示调用了 cxbfget()函数。如果函数调用成功，那么内存块的地址将以十六进制格式(0X????)显示出来。

GOOD:“ CXBFGET，Successful，Memory address is 0X???? ”

ERROR：“ CXBFGET，ERROR→none free，Memory Block BASE address 0X???? ”

(19)CXBFREL：表示调用了 cxbfrel()函数。如果函数调用成功，将以十六进制格式(0X????)显示出所释放的内存块地址。

GOOD:“ CXBFREL，Successful，Memory address retunrned 0X???? ”

(20)CXCTCRE：表示调用了 cxctcre()函数。如果函数调用成功，那么将显示出该循环定时器号。

ERROR:“ CXCTCRE，ERROR→Range ：cyclic timer #?? ”

GOOD:“ CXCTCRE，Successful，cyclic timer #?? ”

(21)CXCTSTT：表示调用了 cxctstt()函数。如果函数调用成功，那么将显示出该循环定时器号。

GOOD:“ CXCTSTT，Successful，cyclic timer #?? ”

ERROR:“ CXCTCOM，ERROR→Range ：cyclic timer #?? ”

(22)CXCTRSTT：表示调用了 cxctrstt()函数。如果函数调用成功，那么该循环定时器号将被显示出来。

GOOD:“ CXCTRSTT，Successful，cyclic timer #?? ”

ERROR:“ CXCTCOM，ERROR→Range ：cyclic timer #?? ”

(23)CXCTRSTC：表示调用了 cxctrstc()函数。如果函数调用成功，将显示循环定时器号。

GOOD:“ CXCTRSTC，Successful，cyclic timer #?? ”

ERROR:“ CXCTCOM，ERROR→Range ：cyclic timer #?? ”

(24)CXCTRSTI：表示调用了 cxctrsti()函数。如果函数调用成功，将显示循环定时器号。

GOOD:“ CXCTRSTI，Successful，cyclic timer #?? ”

ERROR:“ CXCTSTP，ERROR→Range ：cyclic timer #?? ”

(25)CXQCRE：表示调用了 cxqcre()函数。如果函数调用成功，将显示出所创建的队列号。

ERROR:“ CXQCRE，ERROR→queue #?? ”

GOOD:“ CXQCRE，Successful queue #?? ”

(26)CXQATT: 表示调用了 cxqatt()函数。如果函数调用成功，将显示出队列号。

GOOD: “ CXQADD，Successful，queue #?? ”

ERROR: “ CXQADD，ERROR→Range ：queue #?? ”

ERROR: “ CXQADD，ERROR→full ：queue #?? ”

(26)CXQATB: 表示调用了 cxqatb()函数。如果函数调用成功，将显示出队列号。

GOOD: “ CXQADD，Successful，queue #?? ”

ERROR: “ CXQADD，ERROR→Range ：queue #?? ”

ERROR: “ CXQADD，ERROR→full ：queue #?? ”

(27)CXQRFT: 表示调用了 cxqrft()函数。如果函数调用成功，将显示出队列号。

GOOD: “ CXQRMV，Successful，queue #?? ”

ERROR: “ CXQRMV，ERROR→Range：queue #?? ”

ERROR: “ CXQRMV，ERROR→Empty:queue #??”

(28)CXQRFB: 表示调用了 cxqrfb()函数。如果函数调用成功，将显示出队列号。

GOOD: “ CXQRMV，Successful， queue #?? ”

ERROR: “ CXQRMV，ERROR→Range ：queue #?? ”

ERROR: “ CXQRMV，ERROR→Empty ：queue #?? ”

(29)CXQRST: 表示调用了 cxqrst()函数。如果函数调用成功，将显示出队列号。

GOOD: “ CXQRST，Successful， queue #?? ”

ERROR: “ CXQRST，ERROR→queue #?? ”

(30)CXRSGET: 表示调用了 cxrsget()函数。如果函数调用成功，将显示出资源号。

GOOD: “ CXRSGET，Successful， resource # ”

ERROR: “ CXRSGET，ERROR→resource #?? ”

ERROR: “ CXRSGET，ERROR→already owned， resource #?? ”

(31)CXRSRSV：表示调用了 cxrsrsv()函数，如果由该函数所指定的，用来等待一个资源的时间周期满，则将显示 “ Timed out ”。如果等待时间周期还没满，那么，任务所申请的资源号和等待时间周期将被显示出来。

INFO: “ CXRSRSV，resource #?? TIME PERIOD=?? ”

INFO: “ CXRSRSV，Timed out ”

GOOD: “ CXRSRSV，Successful， resource #?? ”

ERROR: “ CXRSRSV，ERROR→resource #?? ”

(32)CXRSREL：表示调用了 cxrsrel()函数。资源号将被显示出来。

GOOD: “ CXRSREL，Successful，resource #?? ”

ERROR: “ CXRSREL1，ERROR→resource #?? ”

ERROR: “ CXRSREL1，ERROR→not owned resource #?? ”

(33)ENABLE_SLICE：表示调用了 enable_slice()函数。

INFO：“ Enable_slice ”

(34)DISABLE_ SLICE：表示调用了 disable_slice()函数。

INFO：“ disable_slice ”

(35)CXPRVR：表示调用了 cxprvr()函数。

INFO：“ CXPRVR，Successful ”

(36)CXPRVL：表示调用了 cxprvl()函数。

INFO：“ CXPRVL，Successful ”

(37)CXSCHED：表示调用了 cxsched()函数。

INFO：“ CXSCHED ”

(38)USER ENTRY: 表示调用了 cmxtracker_user()函数。

INFO:“ USER ENTRY，entry #?? ”

(三)“ GO AND RESUME CMXTracker ”功能：

输入等待时间(系统嘀哒时钟数)，或者按<return>键，离开。

该项功能允许 CXMTracker 在规定的时间内(嘀哒时钟数)，释放对 CMX RTOS 的控制权，当所指定的时间到时，CXMTracker 将自动地恢复对 CMX RTOS 的控制权，并冻结所有其它任务。

这是该项功能最重要的特点，因为它允许用户在所希望的时间内(系统嘀哒时钟)，执行“ 多步 ”，同时还可以看到 CMX RTOS 环境变化，这对调试和校对应用程序是非常有帮助的。

注意：当执行该项功能时，通过敲击相应键，CMXTracker 任务不会被立刻唤醒。

(四)“ QUICK GO and RESUME CMXTracker ”功能：

该项功能允许 CMXTracker 任务在一个系统嘀哒时钟内释放对 CMX RTOS 的控制权。当下一个系统嘀哒时钟用完时，CMXTracker 任务恢复对 CMX RTOS 的控制权，并冻结所有其它任务。

这是该项功能的一个重要特点，因为每一次它允许用户来“ 单步 ”执行一个系统嘀哒时钟，同时还可以看到 CMX RTOS 环境的变化，这对于调试和蒋对程序是非常有帮助的。如果为上述“ GO and RESUME CMXTracker ”功能所规定的等待时间为一个系统嘀哒时钟，那么该项功能将同上述功能是相同的。

(五)“ EXIT CMXTracker ”功能：

该项功能允许 CMXTracker 任务永久地释放对 CMX RTOS 的控制芭，并运行应用任务。要想重新进入 CMXTracker 任务，按“ + ”键既可。

(六)“ Enter P/p to toggle ECHO ”方式：

该项功能允许 CMXTracker 任务允许或禁止 ECHO 方式。缺省设置为 ECHO，即把所有接收到的有效字符返送给终端或 PC 机。如果被禁止，那么就不会向终端或 PC 机返送任何字符。如果按下一个有效的键并显示出两个字符，那么 CMXTracker 的 ECHO 方式可能被禁止，或者终端或 PC 机拒绝接收返回的字符。

CMX 补充说明文件

For CMX Version version 4.0 as of 4/3/97
For use with Tasking version 1.0
CMXBug version 1.1
CMX Tratker version 1.0

注：CMX 是使用下面的 Tasking C compiler v1.0 来创建提供给用户的 CMX 代码。

警告：用户产生的最大错误是忘记在任务代码末尾的“ } ”之前调用 cxtend 函数。除非任务代码永远不会执行到“ } ”处，否则必须使用 cxtend 函数来结束任务代码的执行。不然的话，将产生不可预料的严重后果。另外，所有的 CMX 源代码必须被 Tasking compiler、 linker、 assembler 和 librarian versions 编译或汇编成目标代码。

一、 关于用户使用 Tasking C 编译器的说明：

警告：所有的中断程序代码必须用汇编语言来编写，以确保正确地保存和恢复程序的上下文环境。用户应阅读“ 8051XA 处理器详细说明 ”这一章。在该章节中，讲述了如何编写中断程序和调用中断管道。此后，用户就可以自由地在中断程序中调用 C 例行程序（函数）。

必须遵守中断程序在调用 C 函数和库函数时，Tasking 所规定的一些要求和原则。

二、 关于使用 Tasking 链接器的说明

CMX 只需要几个可直接访问的数据字节空间，这几个字节可安排在 8051XA 的 1Kbyte 可直接访问的数据空间中的任何地方。

CMX 还需要１个可位寻址的数据字节，该字节可安排在 8051XA 的可位寻址数据区域中的任何地方。

另外，给该字节所选定的存贮单元应该是被 CMX 所使用，而不能被复用。用户不可以把这一个可位寻址的字节数据单元改变到其它内存区段上。

CMX 需要使用外部扩展 RAM，其具体数量用户可在“ cxconfig “ 配置文件中，进行声明确定。

CMX 需要代码空间，其具体尺寸由用户来决定。如果程序中调用了一个函数，而该函数的目标代码不包含在 cmxlb???.a 文件中（CMX 函数库的汇编代码文件），那么必须把该程序（函数）模块与目标库文件相链接。

为了使 CMX 能够正确地执行，下面的文件也应被链接上。

1．Cmx_init.c（或其复制文件）：

该文件中包含了 cxconfig.h 文件（或其复制文件）。如果你修改了 cxconfig.h 文件中的某些内容，那么你必须重新编译 cmx_init.c 文件。

2.cxsk4sl.asm:

该文件为 CMX RTOS 汇编文件，该文件可以完成上下文切换，执行中断处理程序和 CMX 定时器任务等。

3．Cmxlb???.a:

用户应该根据所使用的内存模式来正确地指定 CMX 库文件－cmxlb???.a，该文件包含了所有的库函数的调用。

4.符合某一种正确的“ C ”内存模式的 Tasking C 库文件。

5．所有用户已创建完的程序模块。

注：用户应该创建一个 username.xcl 文件，该文件是一个命令文件，链接程序将会读和处理该文件。这样做，就使用户不必在进行链接时，指明所有的链接选项。该文件的建立过程将在 Tasking C 使用手册中进行详细地说明。

三、对于所有用户的说明：

1.Cxconfig.h 文件中，有 4 个附加的“ define ”定义，它们在 CMX RTOS 手册中没有进行解释，现在就分别对它们进行详细说明。

①#define CMXBUG_ENABLE 0(或 1)

允许或禁止启动 CMXBUG 任务。设置为 1，表示允许，若为 0，则禁止。如果用户要启动 CMXBUG 任务，则必须把 cmxbug.obj 文件链接上，以使 CMXBUG 任务代码在 CMX RTOS 中起作用。

②#define CMX_RAM_INIT 0(或 1)

允许或禁止对 CMX RAM 变量进行初始化。在大多数情况下，通常是由编译器启动代码来完成的。如果不是这样，那么必须把 CMX_RAM_INIT 设置为 1 。一般情况下，编译器代码都会把全局型变量（已被初始化）和非初始化变量设置为 0 。设置为 1 ，允许 CMX 代码来初始化它的变量，否则，禁止初始化。

注：下面 2 个 define 定义，在 CMX TRACKER 手册中进行了更为详细的说明。它们应该被设置为 0，除非用户要想使用 CMX TRACKER。

③#define CMXTRACKER_ENABLE 0(或 1)

允许或禁止链接 cmxtracker.obj 文件。“ 1 ”表示允许，“ 0 ”表示禁止。如果设置为 1 ，那么就可以把 cmxtracker.obj 链接到 CMX RTOS 目标代码文件中，使其成为 CMX RTOS 的一部分功能。

④#define CMXTRACRER_SIZE 4096

定义保持 CMX Tracker 压缩信息的缓冲区字节尺寸。

2.另外，也把 cxtname 函数加入到 cmx_init.c 模块中，以允许用户给一个任务命名，这样做可以帮助用户在使用 cmxbug 调试程序时，了解和辨别任务。该函数的原形和应用例子如下：

byte cxtname (byte task_slot , char *name);

task_slot：为任务的插槽号。

Name：一个字符型指针变量，该变量包含了任务名存放在内存中的地址。

返值：

AERR：表示所提供的任务插槽号还没有被创建，或该任务已被删除。

AOK：表示函数调用成功。

注：在给该任务命名之前，该任务必须是已被创建完的任务。

举例：

Status=cxtcre(5 , &task1_slot , task1,128)；

Status=cxtname(task1_slot , ”task1”)；

3.Cmx_tic 函数已经从 CMX 函数库中删除了，现在被放到 cmx_init.c 模块，这样做可以更好地支持 CMXBug 调试程序。下面的函数也已经被加入到 cmx_init.c 模块中，以支持使用 CMXBug 调试程序。

```
Void  setup_bug(viod);
Byte  bug_getchr(byte *ptr);
Void  bug_putchr(char);
```

4.另外，还有几个 CMX 变量已经被加入到 cmx_init.c 模块中，以支持使用 CMXBug 调试程序。

```
Long  stat_array[cc_MAX_TASKS+1];
char  *task_name[CC_MAX_TASKS+1];
Byte  CMXBUG_ACTIVE;
Byte  BUG_WAIT_TICKS;
```

5.下面的几项内容，大多数用户有可能需要对其进行修改，这几项内容仅仅影响到 CMXBug 调试程序与 UART 接口方式以及在 CMXBug 任务测试判断 “ + ” 键出现之前所需要等待的嘀哒时钟的数量。

注：下面的内容都被放置在 cmx_init.c 模块中。

```
BUG_WAIT_TICKS=10;

 /*被放置在 cmx_init 函数中*/
 #defin   CMXBUG_SERIAL    1
 /#如果把定时器 2 作为波特率发生器的话，则被设置为 1*/
 /*该定义仅仅被放置在 setup_bug 函数的外层*/
```

6.steup_bug 函数：用户可以通过该函数来修改波特率，波特率发生器以及其它用来满足调试需要的一些事项。

用户在使用 CMX RTOS 之前，应完整地阅读 CMX RTOS 手册。

用户应该学习和掌握 CMX 公司提供的实例。在最初应用时，用户应该编写一些小型的、不复杂的程序，这样才能更好地理解 CMX 是如何工作的，以及是如何把 Tasking C 编译器、链接器、汇编器和函数库连在一起配合使用。

用户应该学习和掌握 cmxsamp.c 和 cmxsampa.src 这两个文件，它们为用户显示了如何利用定时 0（或 1）中断来建立调度频率。对于激活 cmx_tic 函数和决定该函数的调度频率，用户都可以通过编制相应的代码来实现。

在 EA 位被清除之前，用户不应该允许执行调用 cmx_tic 函数和 int_pipe 函数的中断程序。在调用 cmx_go 函数之前，用户应该清除 EA 位，然后才可以开放所希望的中断，这时不需要再进行重新开放 EA 位的操作，因为 CMX 的代码会进行这项工作。

注：从技术上讲，用户可以开放 EA 位，因为 CMX 代码将测试判断是否已经进入了 CMX RTOS，如果没有进入的话，那么个别的 CMX 函数（cmx_tic 或 int_pipe）的代码就不会得到执行。

下面是由 CMX 所提供的所有文件清单，以及对每一个文件所进行的相应说明。

(1).veraion.cmx：该文件为用户显示了 CMX RTOS 的版本号。用户没有必要对其进行修改，因为它仅仅为你提供技术支持。

(2).cxconfig.h：该文件为一个配置文件，允许用户根据需要进行编辑修改（注：应该制作一个与该文件不同名的复本）。每当用户对该文件或其复本进行编辑和修改时，都必须重新编译包含该文件的 cmx_init.c 文件。注：如果用户创建或具有了该文件的别名复本，那么应把它加入到（拷贝）到 cmx_init.c 文件中，并修改#include，以便引入正确的.h 模块。

(3).bgconfig.h：cxconfig.h 文件的复本，带有对 bugsamp 演示程序的定义设置。允许 CMXBug 和 bug_init.c 模块查找该文件。

(4).tkconfig.h：cxconfig.h 文件的复本，带有对 trksamp 演示程序的定义设置。允许 CMXTracker 和 trk_init.c 模块查找该文件。

(5).cmx_init.c： C 源代码文件，每次用户修改 cxconfig.h（或其复本）时，都要对该文件进行重新编译。如果用户制作了一个该文件的复本，应对该复本文件进行编译。该文件将根据用户在 cxconfig.h 文件中所规定的参数来初始化所有的 CMX 变量。

(6).bug.init.c：是 cmx_init.c 文件的复本，用来显示 CMXBug 演示程序。

(7).trk_init.c：是 cmx_init.c 文件的复本，用来显示 CMXTracker 演示程序。

下面的 4 个文件为汇编源文件，它们包含了 CMX 调度程序，中断处理程序和中断管理。在用户使用链接器时，必须选择这 4 个文件的其中一个进行声明，以便于把 CMX RTOS 内核与应用代码链接起来，这取决于特定的应用程序所使用的内存模式。

(8).cxsk4ml.asm：用于_ML 模式(大 RAM，大 ROM)

(9).cxsk4mlt.asm：用于_ML(大 RAM，大 ROM)，带 CMX_Tracker 功能。

(10).cxsk4mm.asm：用于_Mm(小 RAM，大 ROM)

(11).cxsk4ps.asm：用于_Ms(小 RAM，大 ROM)，零页方式。

(12).cmxio3.c：这是一个 C 源代码文件，用来执行所有的 CMX UART 函数。该文件不能被拆分成单独模块使用，因为它需要由用户来编辑用于特定的需要。如果用户打算使用其中某一个 UART 函数，那么该文件必须被链接上。如果用户希望的话，可以在该文件中自由地放置#define 语句来禁止不需要的函数被编译。该文件被放置在 CMXMOD 子目录中。

(13).cxfuncs.h：该文件是被所有 CMX 函数所使用的头文件。如果要编译任何一个或所有的 CMX 函数，则必须使用该文件，因为它包含了所有 CMX 函数的原型声明。该文件被放置在 CMXMOD 子目录中。

(14).cxstruct.h：该文件是被所有 CMX 函数所使用的头文件。如果要编译任何一个或所有的 CMX 函数，则必须使用该文件，因为它包含了所有 CMX 函数结构声明。该文件被放置在 CMXMOD 子目录中。

(15).cxetern.h：该文件是被所有 CMX 函数所使用的头文件。如果要编译任何一个或所有的 CMX 函数，则必须使用该文件，因为它包含了 CMX 外部数据变量的声明。该文件被放置在 CMXMOD 子目录中。

(16).cxdefine.h：该文件是一个被所有 CMX 函数所使用的 CMX 头文件。如果要编译任何一个或所有 CMX 函数，则必须使用该文件，因为它包含了 CMX 数据定义。该文件在 CMXMOD 子目录上。

(17).cxvendor.h：该文件是一个被所有 CMX 函数所使用的 CMX 头文件。如果要编译任何一个或所有 CMX 函数，则必须使用该文件，因为它包含了“ C ”厂商的一些特殊定义和声明。该文件在 CMXMOD 子目录上。

(18).cmxtrack.h：当允许启动 CMXTracker 任务时，该文件将被所有 CMX 函数所使用。

(19).cmxsamp.c：为用户显示一个用 C 编写的典型的用户程序。

(20).bugsamp.c: 是一个 C 源码用户程序，用来为用户演示如何使用 CMXBug。

(21).trksamp.c: 是一个 C 源码用户程序，用来为用户演示如何使用 CMXTracker。

(22).cmxintl.asm：

(23).cmxintm.asm：

(24).cmxints.asm：

这三个.asm 文件是汇编文件，用来显示一对不同的用户中断处理序。这两个中断处理程序分别调用了决定定时频率的 cmx_tic 函数以及 cmx_pipe 函数，这是该文件为用户演示的唯一目的。这三个汇编文件分别适用于大、中、小内存模式。

下列文件适用于_MS SMALL_PAGE ZERO 模式(小 RAM/小 ROM)：

(25).cmxsamps.bat：是一个批处理文件。使用该文件可以完成编译 cmxsamp.c 文件、cmx_init.c 文件、cmxints.asm 文件，并同时汇 编 cxsk4ps.asm 文件，然后再使用链接器，链接上述文件，使之成为一个 CMX 可执行文件。

(26).cmxsamps.xcl：用于链接上述文件的链接器文件。

(27).bugsamps.bat：是一个批处理文件，使用该文件可以正确地编译 bug_samp.c、bug_init.c、cmxints.asm 文件,并能汇编 cxsk4ps.asm 文件，然后再把上述文件链接起来，使之成为 CMX 可执行文件。该批处理文件适用于编译链接包含 CMXBUG 任务的应用程序。

(28).bugsamps.xcl：适用于上述文件的链接文件。

下列文件适用 _Mm medinm 模式(小 RAM/大 ROM)：

(29).cmxsampm.bat：该文件是一个批处理文件，使用该文件可以正确地编译 cmxsamp.c、cmx_init.c、cmxintm.asm，并能汇编 cxsk4mm.asm 文件，然后再把上述文件链接起来，使之成为 CMX 可执行文件。

(30).cmxsampm.xcl：适用于上述文件的链接器文件。

(31).bugsampm.bat：该文件是一个批处理文件，使用该文件可以正确地编译 bugsamp.c 文件、bug_init.c 文件、cmxintm.asm 文件，并能汇编 cxsk4mm.asm 文件，然后再把上述文件链接接起来，使之成为 CMX 可执行文件。该批处理文件适用于编译链接包含 CMX BUG 任务的应用程序。

(32).bugsampm.xcl：适用于上述文件的链接器文件。

下列文件适用于_ML LARGE 模式(大 RAM/大 ROM)：

(33).cmxsampl.bat： 这是一个批处理文件，使用该文件可以正确地编译 cmxsamp.c 文件、cmx_init.c 文件、cmxint.asm 文件并能汇编 cxsk4ml.asm 文件，然后再把这些文件链接起来，使之成为 CMX 可执行文件。

(34).cmxsampl.xcl：适用于上述文件的链接器文件。

(35).bugsampl.bat：这是一个批处理文件，使用该文件可以正确地编译 bugsamp.c 文件、bug_init.c 文件、cmxintl.asm 文件，并能汇编 cxsk4ml.asm 文件，然后再把上述文件链接起来，使之成为 CMX 可执行文件。该文件适用包含 CMX BUG 任务的程序。

(36).bugsampl.xcl：用于链接上述文件的链接器文件。

(37).trksampl.bat：这是一个批处理文件，使用该文件可以正确地编译 trksamp.c 文件、trk_init.c 文件、cmxintl.asm，并能汇编 cxsk4mlt.asm 文件，然后再把上述文件链接成 CMX 可执行文件。该文件适用于包含 CMXTracker 任务的程序。

(38).trksampl.xcl：用于完成链接上述文件的链接器文件。

(39).cmxbug.c：为 CMXBUG 源代码文件。

(40).cmxtrack.c：为 CMXTracker 源代码文件。

(41).cmxbugio.exe：是一个在 PC 机上使用的通讯软件，对其详细说明请参看所提供的 CMXBUG 手册。

下面的“ MAKE ”文件是放置在 CMXMODE 子目录下。它们允许用户生成不同内存模式的库代码。如果用户修改了任何一个 CMX 函数或 CMX 头文件（包含了结构定义及函数原型），那么应使用 make.exe 文件。对所选定的某一内存模式的“ make ”文件进行维护。用户可以进入并编辑该文件来选择不同的编译开关设置。

makxaml.mak：适用于_ML 模式。
makxamlt.mak：适用于_ML 模式，并允许启动 CMXTracker 任务。
makxamm.mak：适用于_Mm 模式。
makxammt.mak：适用于_Mm 模式，并允许启动 CMXTracker 任务。
makxams.mak：适用于_Ms 模式。
makxamst.mak：适用于_Ms 模式，并允许启动 CMXTracker 任务。
make.exe：该文件由微软所提供，是用来维护 makxa?.mak。如果用户修改了任何一个 CMX 函数或 CMX 头文件，那么必须使用该文件来重新维护 makxa?.mak。另外，如果用户需要的话，可以用该文件来产生新的内存模式的库代码。该文件的使用方式为：make makxa?.mak

下列文件，允许库管理程序来生成一个适用于某一特定内存模式的 CMX 库文件。这项工作可通过所选定的“ makxa?.mak ”文件来自动地完成。

lbcaxa.xlb；适用于所有内存模式。

下列的 CMX 库文件将由库管理程序通过使用所选定的 makxa?.mak 文件被 make.exe 文件所生成的。

cmxlbml.lib
cmxlbmlt.lib
cmxlbmm.lib
cmxlbms.lib

注：下列模块被包含于 cmxlb???.a 文件中，该文件和下列模式的源代码存放于 cmxmod 子目录上。

Cxtcre，cxttrig，cxtwatm，cxtwake，cxtwakf，cxtpri，cxprvr，cxprvl，cxsched，cxtend，cxtrmv，cxerst，cxewatm，cxesig,cxrsrsv，cxrsget，cxrsrel，cxqcre，cxqrst，cxqatt，cxqatb，cxqrft，cxqrfb，cxmswatm，cxmsget，cxmssend，cxmssenw，cxmsack，cxmsbxev，cxctcre，cxctstt，cxctstp，cxctrstt，cxctrsti，cxctrstc，cxbfcre，cxbfget，cxbfrel，cmx_go，timertsk，cxtson，cxtsoff。

下列函数是用于 CMX RTOS 的内部调用：

Cxdelink， cxpri_in， cx_copy， cxgetptr， cxtwak， cxrsrsvl， cxqadd， cxqrmv， cxmswat，cxmssenl，cxreturn，cxtsttim。

\EXAMPLES 目录

在这个子目录中包含了一些例子，这些例子将为你显示如何在 CMXRTOS 环境中，编写用户应用程序。

注：切记在链接器中必须包含下列文件。

cxsk4ml.obj；CMX RTOS 内核。

cmx_init.obj；或是该文件的复本。

cmxlb???.a；CMX 库文件。

cmxbug.obj；如果需要的话，可加入链接。

cmxtrack.obj；如果需要的话，可加入链接。

用户程序。