# CHAMELEON 32
# MTOS-UX MANUAL

Version 3.2

TEKELEC
26580 Agoura Road
Calabasas, California
91302

Part Number 910-3315

August 1, 1990

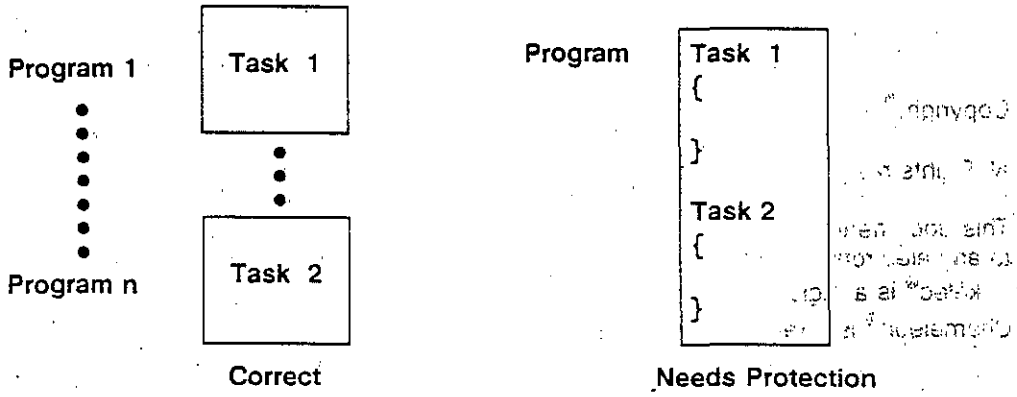
## WARNING

This manual contains documentation for those MTOS-UX Operating System functions that are relevant to the Chameleon 32 user. This documentation is being provided to you for your information only. Tekelec does not warrant that this product will meet Customer's requirements or that the operation or use of this product will be uninterrupted or error-free or that errors in programming by the Customer will be corrected as a result of the use of this product by Customer. There are no other warranties express or implied, including, but not limited to, any implied warranties of merchantability or fitness for a particular purpose.

**Note** **The functions described in this manual are recommended for experienced users who want to take advantage of the multi-tasking capability of the operating system. You need to have extensive knowledge of the C programming language and the Chameleon 32 architecture to use these functions effectively.**

Memory management functions and driver service interface functions are provided through the C library and are not included in this document. Information about these functions can be found in the *Chameleon 32 C Manual*.

**Note:** I/O C libraries (except the window interface library) are not re-entrant, unless they are protected by semaphores or other means of synchronization. Therefore, they can only be used by one task in one program. This is illustrated below.



# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# TABLE OF CONTENTS

# CHAPTER 1:
# INTRODUCTION

# CHAPTER ONE: INTRODUCTION

## Introduction

MTOS-UX is the Chameleon real-time operating system. It includes low level facilities for task creation and synchronization. The Chameleon C Development system offers higher layers of services. Access to the lower layers of the operating system compliments the C library and allows more power and control over the applications.

## Manual Overview

Subsequent chapters of this manual concentrate on the system commands that can be invoked from within a task. No prior knowledge of real-time operating systems is assumed or required. Many examples are given to show how to use the various services to solve practical problems that arise in real-time applications.

## Summary Of Features

While the details are reserved for subsequent chapters, a summary of MTOS-UX features may provide a useful guide.

*Tasks*

The basic organization of an MTOS-UX task is that of a standard C function.

*Task Control:*

A task may create another task. This involves converting an executable code module into a runable program. The code may have to be loaded from a mass storage device, or may already be in memory.

Overall memory provides the only limit to the number of tasks that may be created. This allows the application to be divided into as many concurrent component programs as the user wishes. At every instant these task programs are either:

- Executing on a processor
- Waiting their turn to execute
- Blocked until a requested service is completed
- Sleeping until needed

Each task has a priority that determines which task gets a resource (such as a processor) and which must wait. There are 256 priority levels, with limit to the number of tasks at each level.

MTOS-UX distinguishes two general classes of tasks: "durable" and "transient". A durable task may be started and restarted any number of times. If the selected "target" task is already running when a start request is issued, the request can be queued until the target is available. Each start request may set the priority at which the target will start running, may pass run-time parameters, and may set the way in which the requester will coordinate with the target. A durable task remains within the system until a specific request is issued to delete it.

In contrast, a transient task is created, runs until it terminates and then is automatically deleted. This cycle may be repeated as needed. Normally, transient tasks perform special functions that are not part of the routine operation of the system. A task may change the priority of another task. A task may change its own priority, terminate and delete itself.

*Task Coordination via Start*

A task, "A", that starts another task, "B", may coordinate with the start or the termination of "B". If "A" waits for "B" to terminate, "B" may pass a return argument back to "A".

*Task Coordination via Event Flags*

MTOS-UX contains groups of binary variables called "event flags". The application programmer assigns a logical meaning to each bit, such as "the machine is up to speed", or "the data is ready". Any task may set, reset, or wait for the flags within a group. The wait can be for AND or OR combinations of individual flags. This provides "wait for all" or "wait for any" coordination. When an event flag is set, all tasks that are waiting for the flag continue in parallel. To allow event flags to be used as alarm clocks, a task may request that event flags be set after a given time interval.

*Task Coordination via Semaphores*

Through counting semaphores, a task may gain and relinquish exclusive control over shared data or code. This permits several tasks to work with the same alterable data, or to execute the same non-reentrant code without fear of interference. Only one task at a time can gain access to the data or code.

*Task Coordination via Controlled Shared Variables*

Controlled shared variables (CSVs) are an extension of the semaphore concept. CSVs permit a task to wait until a given condition among certain variables is true and then to continue with exclusive control over those variables.

Event flags, semaphores and controlled shared variables may be created when needed and deleted when not needed.

*Signals*

A signal is sent to a task when the task causes a fault exception (such as a reference to non-existent memory or the execution of an unimplemented operation code). The default response to the signal, if the optional MTOS-UX Debugger is in place, is to halt the task as if a breakpoint had been reached. If the Debugger has not been installed, the default is to terminate the task. There is a separate signal for each major class of fault.

A task may change its response to an individual signal. It may ignore the signal or execute a given subprogram.

One task may send a signal to another. A task can pause until a signal is received. This provides a direct means of synchronizing two tasks, and of sending private, binary messages between tasks. A task can request that it be sent a singla after a given time interval.

*Communication Among Tasks*

Tasks can send and receive messages via message exchanges (mailboxes and message buffers). Any number of tasks can use an exchange as either senders or receivers. Mailboxes and message buffers can be created and deleted.

*Logical Input and Output*

Tasks may use the MTOS file system to save and retrieve information. Services include open, close, read, write, create, rename and delete a file or directory.

*Physical Input and Output*

Peripheral units and memory may also be accessed directly at the physical level. The functions generally available are: reserve the unit, read (input), write (output) and release the unit. Special functions, such as format and page eject, are provided as needed.

The standard UNIX-like C functions, such as *printf*, *getchar* and *putchar*, may be used directly to obtain formatted and character-oriented I/O, with the system console as the target device. Both the portable C and UNIX-like file functions are provided to MTOS users.

*Time Management*

A task may pause for a specified interval or "forever". The specified interval can be as short as 1 ms or as long as 255 days. Another task may cancel the pause. Pause/cancel pause may be used as a very specific way for two tasks to coordinate. A durable task may terminate with automatic restart after a given interval.

*Time-of-Day Management*

MTOS-UX maintains a time-of-day clock/calendar string. Any task may set and read this ASCII string. A task may wait until a given time-of-day string is matched, with "don't care" as a possible element. This permits a task to pause until 30 minutes after the next hour or until the beginning of the next day.

*Uniform Coordination Modes*

Some MTOS-UX services, such as peripheral I/O, may not be finished immediately. With such services a task is given four choices:

- Wait for the service to be completed
- Continue without direct knowledge of when completion occurs
- Continue but set event flag as a completion indicator
- Continue but send signal sent as a completion indicator

The maximum wait for the service to be completed can be limited to a given time interval. That interval can be zero to provide a "fail unless immediately available" restriction.

## C Integers

The exact meaning of certain terms in the C language depend upon the target processor. As used in this manual, "long integer" means a variable large enough to contain an address (a pointer), a 32-bit quantity. The term "short integer" means a 16-bit word. An unqualified "integer" is a variable at least 16 bits long, possibly longer.

## Case Sensitivity

For clarity and readability, examples in this guide have been written primarily in lower case for the C language. Include files are described by upper case code. Many compilers and assemblers, are case sensitive. If this is the situation, be sure to convert to the proper case when using examples derived from this manual.

# CHAPTER 2:

# TASKS AND MULTI-TASKING

# CHAPTER TWO:
# TASKS AND MULTI-TASKING

## Introduction

This chapter introduces the basic concepts of an operating system, a task and a multi-tasking operating system. Task states and task priority are described. The overall structure of a task as a computer program is given. Finally, the general rules for dividing a real-time application into individual tasks are set out.

## Basic Concepts

An operating system (OS) is a program that resides in memory along with the user-written application code. The OS makes the entire system: (1) more efficient in time by permitting several activities to proceed concurently, (2) more efficient in memory space by providing centralized common services, such as timekeeping, input and output, and (3) more capable by providing services that are beyond the limited scope of individual user programs.

A multi-tasking OS, in particular, enables the user to divide an application into separate, individual programs called tasks. A task is a program that can be run as an independent entity. It has its own set of register values, including program counter and stack pointer.

Finally, a real-time executive or kernel is a particular kind of multi-tasking OS which has been specifically designed to fulfill the special needs of real-time applications. These needs include very fast response to external interrupts and a rich set of facilities for intertask coordination, communication and synchronization. A real-time executive need not support editors, compilers, assemblers, linkers and similar programs since they are rarely if ever part of a real-time application. MTOS-UX is a real-time executive.

## Task States

The ability of a task to run as a separate program does not imply that it is executing at all times. Often it is not. At any given instant, each task in the system will be in one of four states: DORMANT, BLOCKED, READY or RUNNING.

A DORMANT task is totally inactive because it never started, or it has run and terminated. A DORMANT task may be started by another task, or by the OS in response to some external event, such as an unrequested peripheral interrupt.

A BLOCKED task is currently active, but is temporarily unable to continue executing. There are several BLOCKED states, depending upon the type of blockage. Generally, the task is either waiting for some shared facility to become available, waiting for a requested service to be completed, or waiting for some internal event, such as the receipt of a coordination (go ahead) indication from another task. While a task is merely waiting for its turn on the Central Processing Unit (CPU) it is not considered BLOCKED. A BLOCKED task does not compete for use of the CPU since it is not ready to proceed.

A RUNNING task is presently using a CPU. There is only one RUNNING task for each CPU in the system.

The READY tasks are those which could use the CPU if it were available.

## Priority

Tasks have various properties which the OS uses to control task activities and to allocate limited resources, such as CPU time, shared memory and access to peripherals. The state of a task is one of its properties. Another is its priority.

The priority is a number which measures the relative importance of the task. MTOS-UX uses 256 levels. A task at level 255 is most urgent, while a task at level 0 is least urgent. Priorities are dynamic; a task may request MTOS-UX to change its priority or that of another task. MTOS-UX itself never spontaneously changes a task's priority, even if the task has been waiting to execute for a long time.

The current priority is used to resolve all disputes among tasks. When tasks are queued internally waiting for a shared facility, the queue is always in descending priority order. For tasks of equal priority, it is first-come-first-served.

Access to the CPU is no exception; the highest priority task that is ready to use a CPU gets to use it (becomes the RUNNING task). If there are two or more READY tasks at the same highest level, they share execution time in round-robin fashion. However, the round-robin sharing is an internal processing. There is no guarantee that tasks of equal priority will get equal execution time.

## A Task As A C Function

A task is written in C, with the general form of a function subprogram. For example, if the task name is *typtsk* the program may be defined as:

```
typtsk (arg)
    long int  arg;
        {
            /* task code here */
        }
```

The variable *arg* is called the *run-time argument*. When one task starts another using the MTOS-UX *start* function, the requester supplies the value of this argument. There are no requirements as to the meaning of *arg*. Often, it is the address of a structure containing a variety of parameters. MTOS-UX makes no assumptions about the nature of the argument, or about the form of any structure to which it might point. The value of the argument is simply assigned to *arg* when the target task starts.

If the task does not make use of the run-time argument, arg may be omitted entirely:

```
typtsk ()
    {
        /* task code here */
    }
```

## Tasking

Multi-tasking is mandatory for a real-time application in which external events, such as the asynchronous arrival of new information, must interrupt ongoing activities, such as the analysis of old information. As a result, one of the most essential--and most difficult--aspects of application design is the actual division of the functional work load into tasks. Proper tasking is essential since a poor division can be difficult to code and debug initially, inefficient in time and memory when it does run, and awkward to maintain and modify thereafter. Tasking is also difficult since it is based on judgement gained primarily through experience.

To illustrate the design process, consider the following set of functions for a prototype control application:

*Main Functions*

* Detect random, transient data signals--
  If Type A, analyze and store attributes on disk
  If Type B, analyze and save attributes in memory
* generate routine control actions
* prepare and print an hourly summary report
* maintain a visual panel display showing the status of key signals
* respond to various inquiries entered on the system console

*Emergency Functions*

* safeguard system upon detection of a power failure
* generate special control actions upon detection of a safety violation

This illustrative system must scan a set of inputs for certain data. Since the inputs appear at random and may be of short duration, scanning must be frequent and quick.

There are two types of data, A and B. When detected, both must be analyzed further. The analysis of Type A involves access to disk files, but does not require any further data inputs; the analysis of Type B requires additional data inputs to be read, but does not involve the disk. Analyzing either type may take longer than the duration of the data signals or the interval between them.

Certain data configurations lead to routine control actions (outputs). In addition, the status of critical data must appear on a visual display panel. Once an hour a standard report (log) must be prepared and output to a printer. However, the display or report can be delayed if other processing becomes very active.

The system must also respond to requests from a system console. While most requests can be executed immediately, some require extensive analysis and many minutes of processing.

Translating a particular set of overall functional needs (such as those described above) into individual tasks is a specific problem requiring a specific solution. Nevertheless, there are some general principles:

**Each main (functionally distinct) activity should be assigned to at least one separate task.** Do not complicate a task by including several separate and functionally independent jobs. Here good judgement is indispensable since most tasks are interdependent.

Separating main functions into different tasks is especially important because changes inevitably become necessary. You do not want to rewrite every task just because one function has to be modified.

Similarly, for ease of system development and maintenance, closely related functions should be kept in the same task. For instance, scanning and detecting both Type A and B data should be in one task.

**Subfunctions that require different response or processing times should be divided into separate tasks.** In the example, the initial detection and capture of the transient data must be started promptly and finished quickly or the data could be lost. In contrast, the subsequent extensive analysis of the data can be delayed somewhat. Thus, the overall function "capture and analyze data" is partitioned into separate tasks: "capture the data and store them in a buffer" and "analyze the stored data".

Subfunctions of different importance should be put into separate tasks. This lets more important activities interrupt less important ones. For example, the analysis of Type A data (more important) should be separated from the analysis of Type B data (less important).

Functions started by different mechanisms (such as different interrupts) must be in separate tasks. This ensures rapid response to interrupts. As an example, suppose the system had just one task for all emergency processing. Suppose further that this task already had been started by a safety violation when a power failure occurs. Since the task is busy, it cannot be restarted immediately to respond to the power failure. Unless you complicate the processing of safety violations by frequent checks for power failure, the system may shut down before the power failure interrupt is ever serviced.

A task arrangement for the illustrative application is shown below.

| Task | Priority | Started by | Purpose |
|---|---|---|---|
| INITSK | 140 | startup | Perform initialization |
| CAPDAT | 100 | INITSK | Capture transient external data |
| ANALDA | 90 | CAPDAT | Analyze Type A data |
| ANALDB | 80 | CAPDAT | Analyze Type B data |
| GENCTL | 110 | CAPDAT | Generate routine control actions |
| PRPLOG | 40 | INITSK | Prepare hourly report (log) |
| PRPPAN | 20 | INITSK | Maintain panel display |
| CNSINP | 60 | console | Input and execute console requests input |
| REQAUX | 80 | CNSINP | Request auxiliary data on console |
| PWRFLR | 240 | power fail | Perform power failure shutdown interrupt |
| EMRCTL | 200 | safety | Generate emergency control actions violation interrupt |

## Example Task Program

The following example demonstrates a procedure for creating and using tasks in MTOS-UX. It creates two tasks that transmit and received Q.921/Q.931 messages over the Basic Rate Interface. The example includes four program files:

- mainsym.h — Contains the global definitions used in the other programs, such as keyboard values, screen attributes, and procotol-specific parameters, timers, and message types.
- tcd.c — Allocates memory for two tasks.
- main.c — Contains the function main() which creates the two tasks, allocates memory for message buffers, and manages the use of the message buffers.
- task.c — Tests the multi-tasking capability of MTOS-UX by transmitting messages two and from the two tasks.

The remaining pages in this chapter contains the code of the four files described above.

```
/*****************************************************************************
 * File name:     main.c
 * Description:   This file contains the function main() which creates the two tasks,
  allocates memory for message buffers, and manages the use of the message buffers.
 *****************************************************************************/
#include "MTOSUX.h"
#include "mainsym.h"

long int stbuf1,stbuf2;

typedef struct
   {
    byte row;
    byte col;
    long int recKey;
    long int sendKey;
    long int endKey;
   } TASKINIT;

TASKINIT t1, t2;

extern int testTask1(),testTask2();
extern struct tcd taskTcd1,taskTcd2;

main()
    {
    long int id1,id2,rec,msbres1,msbres2,msbend;
    int  sresult,r,c;

    /*  Format main window.     */
    disablecur(_stdvt);
    printf(CLEAR);
    printf(YELLOW);

    printf(POS_CUR,3,12);
    printf("******      TEST   MTOS-UX    MULTI-TASKING      ******");

    printf(POS_CUR,14,40);
    printf("-------------------------------------");

    printf(POS_CUR,10,40);
    printf("TASK1:");

    printf(POS_CUR,12,40);
    printf("REC: ");
    printf(POS_CUR,13,40);
    printf("SND: ");

    printf(POS_CUR,15,40);
    printf("TASK2:");

    printf(POS_CUR,17,40);
    printf("REC: ");
    printf(POS_CUR,18,40);
```

```
printf("SND: ");

r = 10;
c = 5;

/*  Init task1 window position.     */
t1.row  = 11;
t1.col  = 40;


/*  Init task2 window position.    */
t2.row  = 16;
t2.col  = 40;

/*   Specify task entry points.     */
taskTcd1.ep = (long int) testTask1;
taskTcd2.ep = (long int) testTask2;


/* Create message buffers      */
msbend  = crmsb('ENDB',MSBGBL+ 50L);
if ( msbend == BADPRM ) printf("END Buffer: BADPRM");
if ( msbend == QUEFUL ) printf("END Buffer: QUEFUL");

msbres1 = crmsb('BUF1',MSBGBL+2L);
if ( msbres1 == BADPRM ) printf("Message buffer1: BADPRM");
if ( msbres1 == QUEFUL ) printf("Message buffer1: QUEFUL");

msbres2 = crmsb('BUF2',MSBGBL+2L);
if ( msbres2 == BADPRM ) printf("Message buffer2: BADPRM");
if ( msbres2 == QUEFUL ) printf("Message buffer2: QUEFUL");

t1.sendKey = 'BUF1';
t1.recKey  = 'BUF2';
t1.endKey  = 'ENDB';

t2.sendKey = 'BUF2';
t2.recKey  = 'BUF1';
t2.endKey  = 'ENDB';


/*   Create task1.     */
id1 = crtsk(&taskTcd1);
printf(RED);
printf(POS_CUR,r++,c);
if ( (id1 & 0xFFFF0000) == 0xFFFF0000)
      printf("Task1 create error");
else printf("Task1 creation successful");

/*   Create task2.    */
id2 = crtsk(&taskTcd2);
printf(RED);
printf(POS_CUR,r++,c);
if ( ( id2 & 0xFFFF0000 ) == 0xFFFF0000)
```

```
        printf("Task2 create error");
    else printf("Task2 creation successful");

    /*   Start task1.     */
    sresult = start(id1,INHPTY,&t1,&stbuf1,CTUNOC);
    printf(RED);
    printf(POS_CUR,r++,c);
    if ( sresult == BADPRM ) printf("Task1 not started = BADPRM\n");
    if ( sresult == QUEFUL ) printf("Task1 not started = QUEFUL\n");
    if ( sresult == TIMOUT ) printf("Task1 not started = TIMOUT\n");

    / *   Start task2.     */
    sresult = start(id2,INHPTY,&t2,&stbuf2,CTUNOC);
    printf(RED);
    printf(POS_CUR,r++,c);
    if ( sresult == BADPRM ) printf("Task2 not started = BADPRM\n");
    if ( sresult == QUEFUL ) printf("Task2 not started = QUEFUL\n");
    if ( sresult == TIMOUT ) printf("Task2 not started = TIMOUT\n");

    /*    Display exit and reset screen.     */
    getmsw(msbend,&rec);
    printf(POS_CUR,r++,c);
    printf("%c[31mTASK 2 dying ...\n",0x1b);
    getmsw(msbend,&rec);
    printf(POS_CUR,r++,c);
    printf("%c[31mTASK 1 dying...\n",0x1b);
    printf(POS_CUR,r,c);
    printf("%c[31mMain dying...\n",0x1b);
    while( getch(_stdvt) != F10 );
    dlmsb(msbres1);
    dlmsb(msbres2);
    dlmsb(msbend);
    printf(CLEAR);
    printf(RESET);
    return(0L);
    } /*  end main  */
```

```
/*****************************************************************************
 *  File name:      task.c
 *  Description:    This file tests the multi-tasking capability of MTOS-UX by
                    transmitting messages two and from the two tasks.
 *****************************************************************************/
#include <MTOSUX.h>
#include "mainsym.h"

extern char *malloc();

typedef struct
  {
    byte        row;
    byte        col;
    long int    recKey;
    long int    sendKey;
    long int    endKey;
  } TASKINIT;

typedef struct
  {
    byte color[6];
    byte pos[8];
    byte text[80];
  } DISPLAY;

char *clr = ".[3.m ";
char *pos = ".[%d;%df";

testTask1(t)
TASKINIT *t;
 {
 byte       *send;
 long int    rec;
 int         result;
 DISPLAY     dsp;

 /*  Init task1 window. */
 strcpy(dsp.color,clr);
 strcpy(dsp.pos,pos);
 dsp.color[0] = 0x1b;
 dsp.color[3] = '2';
 dsp.pos[0]   = 0x1b;

 strcpy(dsp.text,"TEST TASK 1 IS RUNNING.");

 printf((byte *) &dsp,t->row++,t->col);
 t->col = t->col + 6;

/*  Open message buffers.  */
 t->recKey = crmsb(t->recKey, MSBGBL + 2L);
 t->sendKey = crmsb(t->sendKey, MSBGBL + 2L);
 t->endKey  = crmsb(t->endKey, MSBGBL + 50L);
```

```
/*  Send message to task2.  */
send = (byte *) malloc(20);
strcpy( send, "From 1 to 2 ");
result = putmse( t->sendKey, (long int ) send );
switch( result )
  {
  case NOERR: strcpy(dsp.text, "From 1 to 2");
              break;
  case QUEFUL: strcpy(dsp.text, "QUEFUL on TASK1 to TASK2");
               break;
  case BADPRM: strcpy(dsp.text, "BADPRM on TASK1 to TASK2");
               break;
  }

printf( (byte *) &dsp,t->row+1,t->col);

/*  Receive message from task2.  */
result = getmsw( t->recKey, &rec);
switch(result)
  {
  case NOERR: strcpy(dsp.text,(byte *) rec);
              break;
  case BADPRM:strcpy(dsp.text,"BADPRM on TASK1 from TASK2");
              break;
  }

printf((byte *) &dsp,t->row,t->col);
free( ( byte *) rec );
send = (byte *) malloc(4);
putmse(t->endKey, (long int) send);
}

testTask2(t)
TASKINIT *t;
{
byte      *send;
long int  rec;
int       result;
DISPLAY   dsp;

/*  Init task window.  */
strcpy(dsp.color,clr);
strcpy(dsp.pos,pos);
dsp.color[0] = 0x1b;
dsp.color[3] = '6';
dsp.pos[0]   = 0x1b;
strcpy(dsp.text,"TEST TASK 2 IS RUNNING");
printf( (byte *) &dsp,t->row++,t->col);
t->col = t->col + 6;

/* Open message buffers.  */
t->recKey = crmsb( t->recKey, MSBGBL+2L);
t->sendKey = crmsb( t->sendKey, MSBGBL+2L);
t->endKey  = crmsb( t->endKey, MSBGBL + 50L);
```

```
/*  Receive message from task1.  */
result = getmsw(t->recKey, &rec);
switch(result)
  {
  case NOERR: strcpy(dsp.text,(byte *) rec);
              break;
  case BADPRM:strcpy(dsp.text,"BADPRM on TASK2 from TASK1");
              break;
  }

printf( (byte *) &dsp,t->row,t->col);
free( (byte *) rec);

/*  Send message to task1.  */
send = (byte *) malloc(20);
strcpy(send,"From 2 to 1");
result = putmse(t->sendKey, (long int) send);
switch(result)
  {
  case NOERR: strcpy(dsp.text,"From 2 to 1");
              break;
  case QUEFUL:strcpy(dsp.text,"QUEFUL on TASK2 to TASK1");
              break;
  case BADPRM:strcpy(dsp.text,"BADPRM on TASK2 to TASK1");
              break;
  }

printf( (byte *) &dsp,t->row+1,t->col);
send = (byte *) malloc(4);
putmse(t->endKey, (byte *) send);
}
```

```
/*********************************************************************
 *  File name:      mainSym.h
 *  Description:    This file contains the global symbols.
 *********************************************************************/
/*      NT POWER SETTING. */
#define     NT_POWER          3

/*      TERMINAL TYPES */
#define AUTOMATIC             0
#define NON_AUTOMATIC         2

/*      ENVIRONMENT SPECIFIC TEI VALUES  */
#define     PHONE_A           64
#define     PHONE_B           65
#define     TA_A              1
#define     TA_B              1

/*     TIMERS AND CONSTANTS SPECIFIC TO LAPD SETUP    */
#define     T200              10
#define     T203              20
#define     N201              260
#define     N200              10
#define     WINDOW_SIZE       3
#define     MODULUS           1
#define     CONFIG            0x02

/*   PARAMETERS FOR THE INITIATION OF THE COMMUNICATION PROCESSOR. */
#define     INTERFACE         2
#define     STATION           0
#define     ENCODE            0
#define     BITRATE           (unsigned long)16000


/*   SAPI VALUES */
#define     CONTROL           0
#define     PACKET            16
#define     MNGMT             63

/*     TEI VALUES */
#define     BC                127


/*  GENERAL USEFUL SYMBOLS */
extern long    _stdvt,getch();
#define     byte              unsigned char
#define          STOP          0
#define          CONT          1
#define          TRUE          -1
#define          FALSE         0
#define          YES           1
#define          NO            0
#define          MOD(x,y)      ( x % y )
#define          AND           &&
```

```
#define         OR          ||
#define         NONE        -1

/* TYPEDEF DEFINITIONS*/
typedef struct
    {
        int type;
        int tei;
        int sapi;
    } LINK;

extern LINK linksA[];
extern LINK linksB[];


/*   KEYBOARD CODE DEFINITIONS */
#define     F1  0x81
#define     F2  0x82
#define     F3  0x83
#define     F4  0x84
#define     F5  0x85
#define     F6  0x86
#define     F7  0x87
#define     F8  0x88
#define     F9  0x89
#define     F10 0x8a
#define     key0    0x30
#define     key1    0x31
#define     key2    0x32
#define     key3    0x33
#define     key4    0x34
#define     key5    0x35
#define     key6    0x36
#define     key7    0x37
#define     key8    0x38
#define     key9    0x39
#define     UP  0x0b
#define     DOWN    0x0a
#define     RIGHT   0x0c
#define     LEFT    0x08
#define     RTN 0x0d
#define     DELETE  0x7f

/*  SCREEN COMMAND MACRO */
#define     setScr(x)   printf(x);fflush(stdout);

/* COLOR COMMANDS */
#define     BLACK       "%c[30m",0x1b
#define     RED     "%c[31m",0x1b
#define     GREEN       "%c[32m",0x1b
#define     YELLOW      "%c[33m",0x1b
#define     BLUE        "%c[34m",0x1b
#define     MAGENTA     "%c[35m",0x1b
#define     CYAN        "%c[36m",0x1b
```

```
#define     WHITE       "%c[37m",0x1b
#define     BBLACK      "%c[40m",0x1b
#define     BRED        "%c[41m",0x1b
#define     BGREEN      "%c[42m",0x1b
#define     BYELLOW     "%c[43m",0x1b
#define     BBLUE       "%c[44m",0x1b
#define     BMAGENTA    "%c[45m",0x1b
#define     BCYAN       "%c[46m",0x1b
#define     BWHITE      "%c[47m",0x1b

/*  SCREEN ATTRIBUTES */
#define     RESET       "%c[0m",0x1b
#define     HIGHLIGHT   "%c[1m",0x1b
#define     UNDERLINE   "%c[4m",0x1b
#define     BLINK       "%c[5m",0x1b
#define     REVERSE     "%c[7m",0x1b

/*  SCREEN COMMANDS */
#define     POS_CUR     "%c[%d;%df",0x1b
#define     DEL_EOL     "%c[0K",0x1b
#define     DEL_EOS     "%c[0J",0x1b
#define     CLEAR       "%c[2J",0x1b

/*      PORT DEFINITIONS */
#define     PORTA       0
#define     PORTB       1

/* CHAMELEON FUNCTION MODE */
#define     MONITOR     1
#define     SIMNT       2
#define     SIMTE       3

/*   LINK LAYER STATE. */
#define FR_DISC    0
#define LINK_REQ   1
#define REJECT     2
#define LINK_DISC  3
#define INFO_TRANSF 4
#define LOCAL_BUSY 5
#define REMOTE_BUSY    6
#define L_R_BUSY   7
#define R_NOT_RESP 8

/*   CHANNEL ATTRIBUTE */
#define     SYSTEM      1
#define     MILLIWAT    2
#define     CODEC       3
#define     EXTERNAL    4
#define     IDLE        5

/*    DEFINITION FOR UI FRAME */
#define     UI      0x03
#define     MEI     0x0f
#define     IDREQ       0x01
```

```
#define     IDASS       0x02
#define     IDDENY      0x03
#define     IDCHK       0x04
#define     IDCHKACK    0x05
#define     IDREL       0x06
#define     IDCONF      0x07

/*  DEFINITION FOR LAYER 3 MESSAGE TYPE */
#define     PD      0x08
#define     ALERT       0X01
#define     CALLPROC    0X02
#define     CONN        0X07
#define     CONNACK     0X0f
#define     PROG        0X03
#define     SETUP       0X05
#define     SETUPACK    0X0d
#define     RESUME      0X26
#define     RESUMEACK   0X2e
#define     RESREJ      0X22
#define     SUSP        0X25
#define     SUSPACK     0X2d
#define     SUSPREJ     0X21
#define     USERINFO    0X20
#define     DISC        0X45
#define     RELEA       0X4d
#define     RELCOMP     0X5a
#define     REST        0X46
#define     RESTACK     0X4e
#define     CONGCON     0X79
#define     INFO        0X7b
#define     NOTIFY      0X6e
#define     STAT        0X7d
#define     STATENQ     0X75

/*   DEFINITION FOR LAYER 3 TIMER AND OTHERS  */
#define     T302        10
#define     T303        10
#define     T305        10
#define     T306        10
#define     T308        10
#define     T310        10
#define     T312        10
```

```
/*********************************************************************
 *  File name:      tcd.c
 *  Description:    This file contains task declarations.
 *********************************************************************/
#include <MTOSUX.h>
#include "mainsym.h"

#define TASK1   'TAS1'
#define TASK2   'TAS2'

struct tcd taskTcd1 =
 { TASK1,
   ABS+TRN+APP,
   -1,
    C,
   150,
   0,
   0L,
   1000L,
   0L,
   0L,
   0,
   '\0',
   '\0',
   0L
        };


struct tcd taskTcd2 =
 { TASK2,
   ABS+TRN+APP,
   -1,
    C,
   100,
   0,
   0L,
   1000L,
   0L,
   0L,
   0,
   '\0',
   '\0',
   0L
        };
```

# CHAPTER 3:

# INVOKING TASK SERVICES

# CHAPTER THREE
# INVOKING TASK SERVICES

## Introduction

One of the main purposes of an operating system is to provide services to the tasks that run under it. For a task written in C, a service is requested by calling a specific function. This may be illustrated with some simple calls for peripheral input and output, and for a pause.

## Standard I/O Functions

Many tasks must communicate with the external world, for example, to write a message on a console, to read data from a keyboard, to generate a report on a printer. This involves input from or output to a peripheral unit. Other examples of peripheral I/O are the storage and retrieval of disk data, and the manipulation of analog data via A/D and D/A converters.

Since C is a convenient language in which to write tasks, the standard input and output library functions, such as getchar, *putchar* and *printf*, are provided for use with MTOS-UX. (Refer to the Chameleon 32 C Manual for more information about these functions.)

A simple MTOS-UX task to output a fixed message could be:

```
smptsk()
    {
        printf("\n\rWelcome to MTOS-UX\n\r");
        return(0L);
    }
```

In this case, the C compiler stores the message text within the program code region and generates a pointer to it as the value of the argument.

The return statement is not mandatory; the compiler automatically creates the equivalent. However, using an explicit return is considered good practice.

As another example, a text echo task could be:

```
echo()
    {
        #include "mtosux.h"
        char  ch;
        while ((ch = getchar()) != EOF)
            putchar(ch);
        return(0L);
    }
```

## Value Returned By Service Functions

MTOS-UX service functions normally return a success/failure indicator. For example, all standard I/O functions return the error value BADPRM if there is no system console. (Literals, such as BADPRM, are defined in the header file mtosux.h.) With this in mind, the echo task becomes:

```
echo()
{
    #include "mtosux.h"
    char ch;
    while (((ch = getchar()) != BADPRM) && (ch != EOF))
        putchar(ch);
    return(0L);
}
```

Since many service functions also return a value under normal circumstances, the exception values have been chosen to be easily distinguished. For the current case, the normal range for *getchar* is 0x0000 to 0x007F, while BADPRM is 0xFFFF. All error values are (sign-extended) negative integers.

## Idle Time Monitor Tasks

As another simple demonstration, we offer the following pair of tasks, tlytsk and rpttsk:

```
#include "mtosux.h"

external int    scalar;
external long inttally;
    tlytsk()
        {   /* This tally task must be the only one to run at priority 0
            */; for (;;)
        {
        for (scalar = 0; scalar < 60; ++scalar);
         ++tally   /* Value of tally shows how long tlytsk has run */;
        }
        }
rpttsk()
    {   /* This report task must run at a fairly high priority so that
        pause is a proper 1 minute */;
        for (;;)
    {
        scalar = tally = 0   /* reset counters */;
        pause (1 + MIN)      /* pause 1 minute */;
        printf("\n\rSystem idle tally for last minute was d",tally);
    }
    }
```

The reporting task resets the counters and then pauses for one minute, using the MTOS-UX pause service. Since the tally task has the lowest priority, it runs only when there are no other tasks ready to use the CPU. As a result, the value that tally reaches is a measure of the overall idle time. This is reported when rpttsk wakes up at the end of each minute.

## Get System Identification

MTOS-UX contains an ASCII string of system identification data. It may be copied into a user buffer via:

```
int getidn (idnbuf)
char *idnbuf;
```

The function returns with NOERR unless there is a problem writing into the buffer. For write errors, the return value is BADPRM and the error signal, 26, is sent to the task.

The buffer receives a 33-byte, null-terminated string of the form:

```
"\r\nMTOS-UX/68K MP V1.4 [171086]\r\n"
```

The data in square brackets is the last edit date: DD/MM/YY. The field *MP* is replaced by *SP*.

# CHAPTER 4:

# PAUSE AND CANCEL PAUSE

# CHAPTER FOUR:
# PAUSE AND CANCEL PAUSE

## Introduction

MTOS-UX maintains an internal millisecond clock that is used to support time-dependent services, such as pause and terminate-with-future-restart. The basic clock period is installation-dependent, with 5 ms as the normal value. Periods as low as 1 ms are easily provided, but with an increase in overhead.

The period sets the "granularity" of the internal clock. For example, with a value of 5 ms, the millisecond counter remains constant for 5 ms and then increases by 5. As a result, although a service request will accept an interval of 3 ms, it could take as long as 5 ms to recognize that the time has elapsed.

## Pause For A Given Interval

A pause request is used to efficiently delay task processing for a specified interval. During the pause the CPU is automatically used for other work. In C, the form of the request is:

```
int pause (interval)
long int interval;
```

In *pause*, as in all other requests that require an interval specification, the low-order 11 bits of the argument are used. The information is further subdivided into two fields:

```
interval = iunits[10-8] + inum[7-0]
```

Values in square brackets indicate bit positions, numbered right to left. Separate groups of bits are used for each field so that the components can be independently summed. The field *iunits* selects the time units, (1 to 7), and *inum* gives the number of such units (0 to 255). The literals to be used for *iunits* and their equivalent numerical values are:

| | | |
|---|---|---|
| MS | ms | 1 * 256 |
| TMS | ten ms | 2 * 256 |
| HMS | hundred ms | 3 * 256 |
| SEC | seconds | 4 * 256 |
| MIN | minutes | 5 * 256 |
| HRS | hours | 6 * 256 |
| DAY | days | 7 * 256 |

No scaling is needed for *inum*.

When the interval is zero, there is no limit to the wait, so that the pause could last forever. However, a pause can be cancelled by another task. Thus, an interval of zero really means "pause until cancelled". The literal NOEND may be used to make this case explicit.

If *inum* is not zero when *iunits* is zero, a parameter error results.

For success, the pause function returns either:

| | |
|---|---|
| NOERR | The specified interval ran to completion |
| TIMCAN | The pause was cancelled (by canpau) |

The failure values are either:

| | |
|---|---|
| BADPRM | Bits 11 to 15 are not 0 |
| QUEFUL | The service could not be completed for lack of internal resources |

Some examples of the pause function as called in C are:

```
status = pause(250 + MS);

status = pause(SEC + 1);

status = pause(NOEND) /* pause until cancelled */;
```

Some intervals can be composed in two ways, for example, 250 + MS or 25 + TMS. The result is the same. Nevertheless, there is slightly less internal processing whenever MS is used as the units code.

## Pause For Minimum Time Interval

In some real-time systems, there is a task which must run on every clock tick. That task often has the job of sampling input data for changes.

A common structure for such a task is as an initialization section (which is entered just once) followed by a cyclic section. The cyclic section ends with a pause for a minimum interval and a branch back to itself:

```
samptsk ()
   {

      /* initialization section */

      for (;;)
         {
            /* cyclic section */
            pause(NXTICK);
         }
   }
```

The literal NXTICK produces MS + 1. Because of the granularity of the real-time clock, a 1 ms pause is always cancelled at the next clock tick (for any value of the clock period). The value MS + 0 does not work, however, since for an interval of zero there is no pause at all.

## Synchronization For Exact Time Intervals

It is sometimes necessary to separate two events, such as the generation of two outputs ("A" and "B"), by a given interval, say 250 ms. A straightforward approach would be:

```
output "A", pause(250 + MS), output "B"
```

However, because of the granularity of the clock, the pause interval is usually shorter than expected. (On the average half the current clock period is already over when a pause is issued. Thus, the average pause is half a clock period too short.)

When accurate intervals are required, it is best to first synchronize to the start of a clock period by issuing a pause for 1 ms. The sequence would then be:

```
pause(NXTICK), output "A", pause(250 + MS), output "B"
```

When a pause ends, the task status changes from blocked to ready. If there are tasks of higher priority, the actual resumption of task execution may be further delayed. Consequently, if a task needs an exact interval, it also needs a very high priority.

## Cancel Pause

Pause/cancel-pause can be used to coordinate task activity using the following approach:

- One task, *P*, issues a pause for an indefinite time using pause(NOEND)
- Another task, *M*, cancels that pause when it wants *P* to continue

By using a definite interval for the pause, you can provide a limit in case the expected event that the monitor task *M* is seeking never occurs. Task *P* can use the value returned by pause to determine if *M* cancelled the pause (value TIMCAN) or the maximum wait time was reached (value zero).

The format of the cancel-pause request in C is:

```
int canpau (tid)
long int tid;
```

*tid* is the identifier of the target task. For an invalid *tid*, the function returns a failure value BADPRM. For success, canpau returns NOERR if the specified task was paused, or NOTOUT if the task was not paused.

The target of a cancel-pause is expected to be paused, that is, to have invoked pause. A task that uses *pausig* to wait for a signal or *trmrst* to pause and then restart, is placed in a different type of blocked state, and may not be resumed by cancel-pause.

# CHAPTER 5:

# TIME OF DAY CLOCK/CALENDAR

# CHAPTER FIVE:
# TIME OF DAY CLOCK/CALENDAR

## Introduction

In addition to the internal millisecond counter described in Chapter Four, MTOS-UX maintains a time of day (TOD) clock/calendar. The information is available as a string, with the following 21 characters of fixed-field, ASCII-encoded data:

```
DD MMM YYYY HH:MM:SS\0
```

where:

| | | |
|---|---|---|
| DD | = | day in month, starting at 01 |
| MMM | = | abbreviated month name: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC |
| YYYY | = | year |
| HH | = | hour, 00 to 23 |
| MM | = | minute, 00 to 59 |
| SS | = | second, 00 to 59 |

A sample string is:

```
11 NOV 1918 11:00:00
```

## Set Clock/ Calendar

The values within the string may be set from a task coded in C by issuing:

```
int settod (todstg)
char *todstg;
```

*todstg* points to a buffer containing a null-terminated string of the form shown above. The characters are taken as 7-level ASCII; the high-order bit is discarded prior to use.

If the format of the string is not valid (for example, the month name does not exactly match one of the three-character abbreviations), the function fails and *settod* returns a value of BADPRM. A successful invocation returns avalue of NOERR.

The valid field delimiters are blank, colon, or any character greater than 0x1F (hexadecimal 1F).

The clock/calendar string is typically composed in read-write memory by the initialization task after determining the required data from a user. Using *ipltod* as the name of the string, the call would then be:

```
settod(ipltod);
```

Once set, the string is automatically advanced each second. It is assumed that the *settod* is issued at the beginning of the given second.

The TOD string may be set and reset at will by any task. This has no effect upon outstanding pauses, timed restarts and other interval-based time processing. (Such processing involves the millisecond counter, not the TOD string.)

## Get (Read) Clock/Calendar

The current clock/calendar string may be read by issuing:

```
int gettod (todbfr)
char *todbfr;
```

The entire string (including the terminal null) is copied into the read-write buffer whose address is given by *todbfr*. The string is guaranteed to be consistent; the clock/calendar is not permitted to change during the copy.

The following C task outputs the clock/calendar every second:

```
cctask ()
  {
   char ccstg[21]              /* clock/calendar string */;
   while (pause(SEC + 1) = = 0)   /* pause 1 second */
     {
      gettod(ccstg)            /* get time */;
      printf("\n\rs",ccstg)    /* output to std console */;
     }
   return(0);
  }
```

## Synchronization With TOD

Certain tasks, typically those which produce periodic reports and summaries, must be synchronized with the clock portion of the TOD clock/calendar string. MTOS-UX has a straight-forward mechanism to perform this type of synchronization:

```
int syntod (synstg)
char *synstg;
```

*synstg* is the address of a null-terminated string of the form *HHMMSS*. Each character is either 0 - 9, or ? (match any). where HH does not exceed 23, and MM or SS do not exceed 59.

After invoking the service, the task is blocked until the given time string matches the TOD clock/calendar. This is a simple pattern match. Thus, if a wait for "103000" is issued at "103001", the task will wait until the next day. The string "??1500" waits for 15 minutes after the hour, while the string "????00" waits for the beginning of the next minute. The function returns a NOERR upon a successful call.

After the wait ends, the task becomes Ready. Actual execution does not begin until the task is the highest priority Ready task.

The function *syntod* is often invoked at the beginning of the repeated section of a cyclic task.

## Get System Time

MTOS-UX maintains a tally of the number of ms since the system was started. This 6-byte field may be copied into a given user buffer via:

```
int getime (msbuf)
struct timer
 {
  short int   u2  /* upper 2 bytes of time interval */;
  long  int   l4  /* lower 4 bytes of time interval */;
 } msbuf;
```

The function returns with NOERR unless there is a problem writing into the buffer. For write errors, the return value is BADPRM and the error signal, 26, is sent to the task.

The 6-byte value is guaranteed to be consistent, even if a clock interrupt occurs while the copy is being made.

# CHAPTER 6:

# TASK CONTROL DATA

# CHAPTER SIX:
# TASK CONTROL DATA

## Introduction

A task is a dynamic object that is created as needed in order to perform part of an application. The creation, in turn, is based on certain static data, which is known collectively as the Task Control Data (TCD). The formal definition of the TCD as a C structure is:

```
struct tcd  {
  long int    key       /* key */;
  char        attr      /* attributes */;
  char        lclgbl    /* local/global task specifier */;
  char        lang      /* language code */;
  char        ipr       /* inherent priority */;
  short int   copr      /* coprocessor use flags */;
  long int    (*ep)()   /* entry point */;
  long int    stklen    /* length of stack */;
  long int    udalen    /* len of uninit data, if abs */;
  long int    ida       /* addr of initialized data, if abs */;
  short int   apct      /* auto priority change:time interval */;
  char        apci      /*   increment */;
  char        apcl      /*   limit */;
  char        *pgmfil   /* ptr to name of pgm file, if rel */;
        };
```

The creation of tasks from the TCD is discussed in the next chapter. The remainder of this chapter is devoted to the components of a TCD.

## Task Names: Key And Identifier

Every task has two names:

- **Task Identifier** (TID) is a long word pointer that is the effective internal name by which a task is referenced in task-related services. TIDs make references efficient since MTOS-UX does not have to search a table in order to find tasks. The TID is assigned by MTOS-UX when the task is created and then remains fixed until the task is deleted. When a task goes dormant, it does not lose its TID.

- **Key** is a 4-byte field that is the external name by which a task can determine the TID (using the *gettid* service call). Normally, the key is 4 printable ASCII characters, such as 'INIT'. Nevertheless, MTOS-UX treats the key as an arbitrary binary string. The key is a fixed property that is assigned by the user. Duplication of keys is not permitted. The Key for a system task always begins with .SY; therefore, appplication tasks should not use these initial characters.

## Attributes

Every task has the following static binary attributes that influence how MTOS-UX handles service requests:

- Transient/Durable Flag
- System/Application Task Flag
- Relocatable/Absolute Program Flag
- Subpart Flag

These attributes are described on the following page.

## Transient/ Durable Flag

MTOS-UX makes a distinction between tasks that are created to be run once and then discarded, and tasks that are to be created once and then run repeatedly. The first are usually utilities that are requested via specific operator commands entered through a console. They perform special services that are not normally part of the main real-time application. To emphasize the transitory nature of such tasks, they are called *transient*.

The second type of task forms the mainstay of a system that is dedicated to a single real-time application. These are the tasks that are likely to be created during an initialization phase of the application, and then remain forever. These tasks are referred to as *durable* since then are not truly permanent, and can be deleted.

The distinction between transient and durable is a permanent attribute of a task known as the Transient/Durable Flag. The T/D Flag changes how certain requests are treated. For example, a transient task is automatically deleted after it terminates. In contrast, the function *dltsk* must be invoked in order to delete a durable task.

## System/Application Task Flag

These system tasks are available to run under MTOS-UX:

- Error Reporter ('.SYE')
- Debugger ('.SYD')
- Loader ('.SYL')

The Error Reporter must be included; the others are optional. The system tasks are very general in that they do not depend upon the application being performed. In contrast, the user-written application tasks carry out the specific requirements for which the equipment is intended.

System tasks perform functions that require privileges which might be undesirable to grant to application tasks. For example, system tasks have access to all available memory, even when there is a Memory Management Unit to shield MTOS-UX code and data from the tasks. The Debugger has a private set of services that enable it to set Breakpoints and to alter the way tasks are scheduled for execution.

The System/Application Flag indicates the class into which task falls.

## Relocatable/ Absolute Program Flag

Task code can be run either from a region allocated within a Transient Program Area (TPA) or from some fixed region outside of the TPA. (We are considering here only the code and any fixed initialized data portions of the task; the stack and any uninitialized data are always allocated within a TPA.) The Relocatable/Absolute Program Flag indicates which is the case, with relocatable programs always going into a TPA.

It is possible to use *crtsk* to create several independent tasks, each of which executes the same program code. (Each task would have a unique key.) Normally, such code would be relocatable, but it need not be. All that is required to form two or more tasks from an absolute program module is to make the code re-entrant and to have each task use a separate data area. Since the uninitialized data area is allocated for each task, it is automatically separate.

Commonly, transient programs are relocatable, but there is no requirement for this.

## Subpart Flag

Often, a program module contains the code and any initial data for a single task. In this case, the Subpart Flag is set to 0 to indicate that the task being created is not a subpart of its creator.

MTOS-UX also permits a group of tasks to be linked together so they can directly share data and subprogram code. The result is a single module with several entry points, one per task. Such a module is loaded once on behalf of all its tasks. A similar multiple-task arrangement can occur with absolute programs that are burned into ROM.

When creating multiple tasks, one of them is arbitrarily selected to be the parent; it is created first. If it is relocatable, then the entire module (containing its code and that of the other mutually-linked tasks) is loaded into an allocated code segment. In any case, the shared uninitialized data segment is allocated on behalf of all the linked tasks. The Subpart Flag must be 0 in the TCD that describes the parent.

The parent then creates the remaining tasks (the children). The Subpart Flag must be 1 in each child's TCD. The Relocatable/Absolute Program Flag and the Local/Global Task Specifier of the parent are propagated to the child; the other TCD values are taken as given. The pointer to the program file is not used for the child and may be 0.

The effect of setting the Subpart Flag is to link the parent and the children internally. When one of the tasks is deleted, its code and data segments are not deallocated until both the parent and all of the children are also deleted.

## Local/Global Task Specifier

For multiprocessor installations, a task is designated as either global or local. A global task can execute on any available CPU; a local task is restricted to run exclusively on a given CPU.

The Local/Global Task Specifier (LGTS) shows if the task is local or global. The value -1 is used for global tasks and for non-multiprocessor installations.

For a local task, the processor number (0 to N - 1) is stored in the low-order 4 bits of the LGTS. (N is the number of processors available in the system.) Normally, local tasks store their uninitialized data in an area allocated from the TPA of the same local processor. This is specified by a 0 in Bit 5 of the LGTS. If Bit 5 is a 1, the uninitialized data will be stored in the Global TPA. This permits such data to be shared with other tasks, and to be accessed directly by peripheral read/write or mailbox send/receive requests.

Similarly, if the task is relocatable, bit 6 determines if the initialized data is to be stored in the Local (0) or Global (1) TPA. For an absolute file, the initialized data is assumed to be already stored, so that bit 6 is not examined. bit 7 must be 0 for a local task.

The code and any (fixed) initialized data for a global task can be either in a global memory or in the local memory of each processor. The first uses less memory. The latter gives faster performance since the task does not have to contend with other users of the main memory (bus contention is reduced). If the task is created by MTOS-UX from a relocatable file, the code and initialized data are always placed within the TPA. No matter where the code and initialized data reside, the stack and uninitialized data are always in the Global TPA.

The code and initialized data for a local task can also be in either global memory or the local memory of the designated processor. The latter gives faster performance and is normally used. If the task is created by MTOS-UX from a relocatable file, the code and initialized data are always placed within the local TPA. No matter where the code and initialized data reside, the stack and uninitialized data are always in the local TPA.

## Language Code

Tasks are initialized in accordance with the requirements of their source language. The Language Code parameter for Chameleon 32 C indicates if the task should be initialized as an assembler or C program.

The language code refers only to register and stack initialization. Declaring a task to be written in a high-level language does not preclude its calling an assembler subprogram.

## Inherent Priority

The Inherent Priority (IPR) generally indicates the relative importance of the task. The range is from 255 for the most urgent tasks to 0 for the least urgent. The IPR may be selected as a default priority when starting the task.

## Co-processor Use Flags

Tasks may use various co-processors that are often available in particular hardware systems. The most common are the arithmetic co-processors:

- 68881 Floating-point chip for the 68020
- 32081 Floating Point Unit for the 32xxx
- 80x87 Numeric Data Processor for the 80x86

The parameter *copr* is a set of 16 bits, each of which indicates whether or not a particular co-processor is ever used by the task. The most-significant bit (left-most) refers to the arithmetic co-processor; the meaning of the other bits is installation-dependent. A value of 1 means that the co-processor may be in use by the task, while a 0 means that the co-processor is not installed, or is never used by the task.

## Entry Point

The Entry Point (EP) is the starting address of the task. For a relocatable task, the EP is an offset relative to the beginning of the code section. For an absolute task, the EP is an absolute address.

For a relocatable task, EP may be set to -1 to indicate that the entry point is to be taken from a special record within the program file.

## Length of Stack

The user specifies the length of the stack (in bytes) via *stklen*. The actual stack is rounded up to the next multiple of the TPA block size.

MTOS-UX does not use the task stack for either interrupt processing or context saving. Thus, it is not necessary to increase the estimated stack length in order to account for MTOS-UX. However, the CPU may store information on the task stack when an interrupt occurs.

The maximum number of bytes stored is:

- 68000/08 = 6
- 68010/20 = 58

These values must be added to the normal task requirements for subprogram calls and data.

For a relocatable task, *stklen* may be set to -1 to indicate that the stack length is to be taken from a special record within the program file.

## Length of Uninitialized Data

For an absolute program, the user must specify the length (in bytes) of any uninitialized data (*udalen*). If *udalen* is not 0, MTOS-UX allocates an area within the TPA for the data. The allocated area may be larger since the given length is rounded up to the next multiple of the TPA block size.

For a relocatable program, the required length is taken from the program file. Parameter *udalen* is expected to be 0.

No corresponding register load is made; however, the address is used by the Loader to resolve references to labels within the uninitialized data segment.

## Address Of Initialized Data

Some tasks make use of data with given initial values, or tables of information with fixed values. These would constitute the Initialized Data Segment of the task.

For an absolute file, the address of the data must be given in parameter ida. A value of 0 means that there is no initialized data.

For a relocatable module, the length and text of this segment is given in the program file and no further specification is needed. Thus, *ida* is expected to be 0. No corresponding register load is made; however, the address is used by the Loader to resolve references to labels within the initialized data segment.

## Automatic Priority Change Parameters

There are four parameters which jointly specify the Automatic-Priority-Change feature:

- Time units code
- Number of units
- Change increment
- Priority limit

If the feature is selected (by specifying a non-zero time interval) then at the end of each interval, the increment is algebraically added to the current priority, provided the priority is not already at or past the given limit. The increment may be positive to raise the priority of a task.

This can be used to help a low-priority task that has been waiting a long time for services. The increment may also be negative to lower priority. This might be used to penalize a task that has been executing for a long time.

## Pointer To Program File

For a relocatable program, the final parameter (*pgmfil*) is a pointer to a null-terminated string containing the name of the program file. The string has either a file name or an accessible directory path leading to the file.

When the Relocatable/Absolute Program Flag is set to relocatable, *pgmfil* must not be zero. Furthermore, within the program file, all unresolved address references must be relative to at most three origins: one for the Code Segment, one for the Initialized Data Segment and one for the Uninitialized Data Segment.

An absolute program may also reside in a program file. In this case, all code and data addresses have already been resolved to absolute values (outside of the TPA). If the code is to be loaded into RAM, the file pointer must not be zero.

Finally, an absolute program may be permanently burned into ROM or loaded by some mechanism external to MTOS-UX. This is indicated by setting the file pointer to zero.

As noted earlier, when the Subpart Flag is set in the attributes byte, *pgmfil* is not used and may be zero.

# CHAPTER 7:

# TASK MANAGEMENT

# CHAPTER SEVEN:
# TASK MANAGEMENT

## Introduction

Task management encompasses creating, starting and terminating tasks, as well as determining or changing certain task properties, such as current priority.

## Create Task

Tasks must be created before they can be run. In C, the call to create a task is:

```
long int crtsk (tcdptr)
struct tcd  *tcdptr;
```

The argument, *tcdptr*, is a pointer to the TCD, which has the same content and form as that used within USEOSI. The definition of *tcd* as a C structure can be incorporated into the requesting task by including the file mtosux.h.

As an example, the following section of the application Initialization task creates two data scanning tasks:

```
#include "mtosux.h"
#define SCN1 0x53434E31
#define SCN2 0x53434E32

long int sc1ent(), sc2ent()  /* entry point of tasks */;

struct tcd scntcd[2] = {SCN1,ABS + APL,-1,C,230,0x8000,
    sc1ent,512,0,0,SEC + 1,10,250,0
    SCN2,ABS + APL,-1,ASM,210,0,
    sc2ent,100,0,0,0,0,0,0};
```

The function checks that there is not already a task with the given key. If there is, the warning value DUPTSK is returned by the function. The parameter-error signal (26) is not sent for a duplicate task.

When there is no duplication of keys and a file name is given, MTOS-UX attempts to locate the program module in the file system and then perform the load. If the file cannot be found, or does not have the form of an object module, the function returns with an error value. Failure also occurs if the relocatability of the object code is not consistent with the Relocation/Absolute Code Flag of the TCD, or the space needed to hold the relocatable module cannot be allocated.

Once loading is completed, MTOS-UX checks the TCD parameters. A *crtsk* request will fail if any of the parameters is improper. Because of the large number of parameters involved, different error codes are used to help find the specific failure. The literals are:

- BADLNG (bad language code)
- BADTIM (bad time interval unit code)
- BADPRC (bad processor specified in local/global task parameter)

The first error detected sets the code. Each error code has 0xFFFF as the upper word; a non-error result never has that value as the upper word. Thus, a general test for any error is:

```
if (scntid[0] = = DUPTSK)
  {  /* task already exists */
  }
if ((scntid[0] & 0xFFFF0000) = = 0xFFFF0000)
  {  /* other error */
  }
```

The parameter-error signal is sent if any of these errors is detected.

For a valid TCD, MTOS-UX seeks to allocate two areas within the TPA: one for the stack and another for the uninitialized data. A block is also allocated (from a separate, internal pool) for the dynamic data required to control each task. The address of that block (which is called the Task Control Block or TCB) becomes the identifier of the task. The identifier of the task is returned as the value of the function.

At times, there will not be enough room in the TPA for the load or stack. The create function then fails. No request queuing has been provided in order to reduce the possibility of system deadlock. It is the responsibility of the calling task to resubmit the request.

If the function fails for any reason after a code module has been loaded, the load is abandoned.

## Get Task Identifier

A successful invocation of *crtsk* provides the identifier of the task. If another task wishes to obtain the identifier, it may invoke:

```
long int gettid (key);
long int  key;
```

If *key* is non-zero, it specifies the task whose identifier is sought. A zero value returns the identifier of the calling task. If there is no task matching the given non-zero key, the function returns BADPRM and signal 26 is sent.

As an example:

```
scntid[1] = gettid(SCN2);
```

## Start Task

A task is set dormant as soon as it is created. A separate request is needed to start it running. In C, this function is:

```
int start (tid,pty,arg,stabfr,qual)
long int  tid,pty,arg,*stabfr,qual;
```

The task to be started is specified by *tid*. The value must be the identifier returned by a previous call of *crtsk* or *gettid*. The task referenced by *tid* is known as the *target* of the start call. Any task that knows the identifier can start a task; it need not be the task that created the target.

The argument *pty* determines the priority with which the target task will start running when it begins because of this request. The word is composed of two component fields:

```
pty = pbasis[9-8] + pvalue[7-0]
```

If *pbasis* is set to the literal INHPTY, then the inherent priority of the target is used. For CURPTY, MTOS-UX uses the current priority of the requesting task. The larger of the requester's current priority and the target's inherent priority is selected for LRGPTY. Finally, for GVNPTY, the value given in the *pvalue* field is taken. The default (for *pty* equal zero) is LRGPTY.

The target task becomes READY immediately if it is dormant when start is invoked. Otherwise, the start request is queued until the target terminates and can be restarted. Available memory provides the only limitation to the number of requests that can be queued to a task.

If the internal facilities for queuing have already been exhausted, then the request fails immediately, returning the value QUEFUL. The priority with which the task will start also sets the order of the queue: high-priority restarts go ahead of low-priority ones. For equal priority it's first-come-first-served. Once the task starts, however, it is allowed to run to completion; a restart request (no matter how high its priority) never interferes with the execution of a task (no matter how low its current priority).

The parameter *arg* is a long-word value to be passed to the target task as a run-time argument. There is no structure imposed upon *arg*.

The next argument, *stabfr*, is the address of a status buffer. The values that may be stored in the buffer depend upon the coordination mode selected via qual.

As with other requests, the qualifier, *qual*, determines how the function should behave if the service cannot be completed immediately. It also establishes the mode of coordination between the calling task and the completion of the service. The long-word *qual* is composed of the standard three coordination components plus an end-of-service basis selector:

qual = cbasis[17] + cmode[16-11] + lunits[10-8] + lnum[7-0]

If *cbasis* is set to the literal CSTART, then the service is considered completed for the purposes of coordination when the target task starts because of the current request. If the literal CTERM is used instead, the service is not completed until the target starts because of the current request, and then terminates. CSTART has the value 0 and thus is the default if no value is given.

All four of the standard alternatives for *cmode* are supported. For WAIFIN the calling task is blocked until the service is finished. When the caller continues, the value of the function is NOERR for success. For failure, the return value may be BADPRM (tid or lunits is improper) or TIMOUT (the wait is limited and the target cannot even be started within the selected interval). For failure, the status buffer receives the error code, sign extended to a long word. For success, the buffer is cleared when cbasis is CSTART.

When cbasis is CTERM, the buffer receives the long status word that the target presented when it requested termination (Section 8.7.1), or the equivalent value created when a signal forced termination. With the other coordination modes (CTUNOC, CLEFn, CSIGn) the function always returns without delay. Possible function values are BADPRM, TIMOUT and QUEFUL (target task is busy and internal queuing facilities are exhausted). TIMOUT occurs only if the wait-limit provision has been invoked. A value of NOERR means either the request has been successfully completed immediately, or the request has been accepted and is queued.

As usual, for CLEFn the selected local event flag is first reset to 0 and then set to 1 when the service is completed. For CSIGn the selected signal is sent upon completion.

The wait-limit feature available within *start* works in the usual manner when coordination is based on the start of the target task. When the basis is task termination, the limit applies only to starting the task. If the task has not even been started when the given interval elapses, the request is cancelled and a timeout is reported. However, if the task has started but not terminated at the end of the interval, the target is permitted to proceed and the timeout is ignored.

An example of a start-task request in C is:

```
start(scntid[0],GVNPTY + 100,&data,&stabuf,WAIFIN + 250 + MS + CTERM);
```

When a task starts execution, most of its properties are left as they were when the task was first created, or as they were after the last termination. The only changes are that the stack pointer is set to the top of the stack, the program counter is set to the entry point of the program, the last start time is set to the current time, and the current priority is set to the value given in the start-task request. The local event flag group values and the actions to be taken when signals arrive are not changed.

## Get Address Of Data

The addresses of the initialized and uninitialized data for the requesting task may be determined by issuing the C function:

```
long int getdad (buf)
long int *buf;
```

The addresses of the data sections are returned within the given buffers, with the address of the initialized data section going into buf[0].

## Set Task Priority

When a start-task request is issued, it specifies the priority at which the target task is to begin. For MTOS-UX, priorities are unsigned and range from 255 (most urgent) to 0 (least urgent). Once the task is started, it can change its own priority (or that of another task) through the C system call:

```
unsigned short int setpty (tid,basis,value)
long int  tid,basis,value;
```

The argument *tid* is the identifier of the task whose priority is to be changed, or zero to change the priority of the requesting task. If basis is USEVAL then the priority is set to the low-order byte of value. For ADDVAL, the signed number value is added to the current priority and the result limited to the range 0 to 255.

Upon success, *setpty* returns with the new value of the priority. (Values above 127 are not sign-extended.) If *tid* is neither 0 nor a valid task identifier, the function returns an error value BADPRM (0xFFFF).

Some examples are:

```
newp12 = setpty(tsk12,ADDVAL,-10L)  /* decr pr of "task 12" by 10 */;

setpty(0L,USEVAL,120L)              /* set own priority to 120 */;
```

If the target task is blocked, a priority change will not become effective until the task becomes Ready.

## Terminate Task

A task can terminate, as follows:

- It can implicitly call exit by performing a return statement
- It can be killed by receiving a signal
- It can issue a terminate with automatic restart request, *trmrst*

The following describes what happens when the task *T* terminates:

If *T* was started by start using basis CTERM, first:

- The starter is continued (for WAIFIN)
- The local event flag is set (for CLEFn)
- The signal is sent (for CSIGn)

Next, MTOS-UX determines if any requests have been queued to restart *T*, and if so, the one with highest priority is now honored. If there are no restart requests and *T* is a transient task or it was marked for deletion, the deletion is performed. Otherwise, *T* goes DORMANT.

When a task terminates (for whatever reason):

- All memory allocated from a memory pool remains allocated
- All reserved semaphores and peripheral units remain reserved
- All open files remain open
- All outstanding requests for memory allocations, semaphores, peripheral I/O and mailbox message transfer remain queued
- All created semaphores, mailboxes, event flag groups and tasks remain in existence

MTOS-UX does not attempt to find all of the facilities taken by a task and give them back. A task must clean up for itself. Failure to release reserved semaphores and units would prevent any other tasks from ever using these resources and could permanently disable some tasks.

## Terminating With Automatic Restart After Given Interval

Certain tasks are inherently cyclic in nature. For example, one task might be called every five milliseconds to scan a set of inputs for changes and another task might be called every eight hours to prepare and print a shift summary report.

There are several ways to implement a cyclic task. One way is to issue a pause or a wait-for-given-time-of-day request, and then jump to the beginning of the program. Another is to have the task request that it be restarted after a given time interval when after termination. This latter facility is available through *trmrst*, using the format:

```
trmrst (retarg,intrvl)
long int  retarg,intrvl;
```

*intrvl* is composed of three non-overlapping fields:

```
intrvl = rbasis[15] + runits[10-8] + rnum[7-0]
```

*runits* and *rnum* determine the restart time interval: *runits* is the units selector (MS to DAY) and *rnum* is the number of such units (0 to 255).

The *rbasis* field indicates if the restart interval is to be added to the last scheduled time for the task (=STRTIM), or to the current, i.e. termination, time (=TRMTIM). The starting time is normally used for cyclic tasks and is the default.

A call to *trmrst* never returns. Instead, the task pauses for the specified interval, and then begins at the initial entry point. Thus, the use of *trmrst* is roughly equivalent to a pause followed by a jump to the entry point.

The two major differences are that *trmrst* completes any coordination with the task that started the terminating task, and that *trmrst* resets certain dynamic variables, such as the stack pointer. The current priority is not changed; the task restarts at the same level of importance. If *intrvl* is invalid, the task restarts immediately.

Note that the last start time is the time the task first became ready, not the time it actually began processing. If there were higher priority ready tasks, the actual start may have been delayed. Furthermore, suppose a task is to be restarted every 5 minutes based on start time. Assume further that the task started on schedule at 11:00 but did not issue the *trmrst* until 11:07. Upon termination, the task immediately restarts (since 11:07 is past the next restart time of 11:05). If the task terminates before 11:10, it will wait until then to restart; if it terminates after 11:10, it will again restart without wait, trying to catch up.

While a task is waiting to restart itself, it is blocked and not dormant. Such a task cannot be restarted by another task (although other tasks may still queue restarts). This point is normally moot, however, since it would be unusual for a cyclic task to have restart requests posted to it.

The pause generated by calling *trmrst* may not be cancelled by another task via *canpau*. As a result, it is not valid to specify a time interval code of zero, as would be the case for *trmrst*(FOREVER). In such cases, the task will restart immediately. The special call *trmrst*(NXTICK) is valid and provides a restart at the next time tick.

## Terminating Via A Signal

Signals may be sent to a task when a special condition is detected (such as attempting to execute an illegal instruction), or when a valid request for service is finished. One response to a signal is to terminate the task.

When the signal does terminate the task, it is treated as though the task had issued an exit with retarg set to:

-16*(signal number + 1)

Thus, the equivalent *retarg* is 0xFFFFFFF0 for Signal 0, and 0xFFFFFE00 for Signal 31. None of the standard error values use this range.

## Delete Task

When a durable task is no longer needed, it may delete itself by issuing the call:

```
dltsk (retarg)
long int  retarg;
```

The *retarg* is the analog of the argument presented to exit.

A request for deletion is considered to have an implicit call of *exit(retarg)* preceeding it. Thus, the coordination aspects of *exit* are performed ahead of the deletion. If there are no unsatisfied restart requests pending, then the task is deleted: the space occuppied by the code, data and stack are released and the task identifier is made available for reuse. If there are restarts already queued, then the task is marked internally as deletion-requested and the highest priority restart request is honored. Deletion is finally performed when the task completes the last restart request and would normally go dormant. A task marked for deletion may restart itself via *trmrst*. There is never a return from *dltsk*.

Only a durable task would normally call *dltsk*. If a transient task requests deletion it is treated as a call of *exit(retarg)*. This latter case is not considered an error.

A task can delete itself. There are no provisions for one task to delete another.

When a task is deleted (by an explicit call of *dltsk* for a durable task, or by a termination for a transient task), its stack and internally allocated data regions are released for reuse by other tasks.

A problem could occur with service requests that are still queued at the time of the deletion. If the stack and data regions are released and a queued request finishes, a value could be stored in a buffer whose space now belongs to some other task. To prevent this, MTOS-UX delays the reassignment of task space until all outstanding service requests are completed.

# CHAPTER 8:

# EVENT FLAGS

# CHAPTER EIGHT:
# EVENT FLAGS

## Introduction

In order to simplify the design, implementation and maintenance of application code, each task should be an independent program that is responsible for a specific job. In practice, however, tasks usually become highly intertwined since they share a common overall objective, use the same hardware, respond to the same set of inputs and control the same set of outputs. Thus, coordination and synchronization of tasks is essential.

MTOS-UX is a rich system in that it provides many different methods for task coordination. Some of the major mechanisms are via:

- Coordination options of the start-task request
- Event flags
- Semaphores
- Controlled shared variables
- Signals
- Pause/cancel pause

## Description

Depending upon context, some methods are equivalent and can be used interchangably; others are unique and must be used to achieve the desired type of coordination.

Within MTOS-UX there are discrete, binary variables called *event flags*. Each flag may be independently set to 1 or reset to 0. The event flags are arranged in groups of 16. Group 0 is local to a task; each task has its own set of 16 flags that ordinarily are not referenced by other tasks. The remaining groups are global and are available to all tasks.

Throughout MTOS-UX, all 16-bit arrays (such as the event flags) are stored in successive bits of a word, from left to right. Thus, event flag 0 is stored in the leftmost bit of the high-order byte, event flag 1 is stored in the next bit. This continues through event flag 7, which is stored in the right-most bit of the high-order byte. Event flag 8 is stored in the left-most bit of the low-order byte and event flag 15 is stored in the right-most bit of the low-order byte.

In other words, event flag 0 is masked by 0x8000 while event flag 15 is masked by 0x0001. The literals EF0 to EF15 have been equated to the corresponding masks. Masks, being 16-bit quantities, are denoted as a C "short integer".

Task coordination is achieved through six service functions:

- crefg create global event flag group
- srsefg immediately set/reset event flags
- waiefg wait for event flags to be set
- sgiefg set event flags after given interval
- srslef immediately set/reset local EFs of given task
- dlefg delete global event flag group

To give a simple example of the use of event flags, a task might first reset one of the event flags, and then issue a wait until that flag is set. Some time in the future, a coordinating task can set the event flag to continue the waiting task. The assignment of individual event flags for the purpose of coordination is left completely to the user.

## Create Global Event Flag Group

Every task comes with its own local event flag group (EFG) which can neither be created nor deleted. In contrast, global EFGs must be created. The C function to do this is:

```
long int crefg (key)
long int  key;
```

*key* is the key associated with the group. Often the key is four printable ASCII characters, but it is taken as an arbitrary, binary pattern. If an event flag group with the given key already exists, then the group identifier is returned as the value of the function. For a new key, MTOS-UX attempts to create the group. If successful, the identifier is again returned. In the unlikely case that not enough internal facilities are currently available to do the creation, the function returns the error code QUEFUL. Note that queuing the request is not possible, since there are no facilities to create the group.

The key permits tasks to share a group even if there is no direct communication between the sharers. Each task issues *crefg* with the same key. The first request creates the group and supplies the identifier to the task. Subsequent requests do not create a new group, but just inform the task of the identifier.

The group is created with all flags reset.

The following example is typical:

```
#define PMP1 0x504D5031

if ((pm1gid = crefg(PMP1)) = = QUEFUL) ...
```

## Immediately Set/ Reset Event Flags

The C function to immediately set or reset (clear) the EFs within a global group (or the local group of the task issuing the request) is:

```
long int srsefg (gid,opmask)
long int gid,opmask;
```

*gid* is the identifier of an existing global EFG or is zero to change the caller's local values.

*opmask* is the sum of an operation selector and a mask. Literals for the operation are:

```
EFSET ( = set to 1)
EFRST ( = reset to 0)
```

The latter is the default. The mask selects the particular flags to be altered. The literals for the mask are EF0 to EF15 and EFALL, which means "all of them".

Some examples are:

```
srsefg(0L,EFSET + EF1 + EF5);

srsefg(0L,EFRST + EFALL);

srsefg(0L,EFALL-EF15)  /* = reset all but flag 15 */;

srsefg(pm1gid,EFSET + EF8 + EF9);
```

If *gid* is valid, the function returns the final value of the group in the lower-order 16-bits of a long integer. The high-order 16-bits contain NOERR.

If *gid* is invalid, the function returns BADPRM as the high-order word, and 0xFFFF as the low-order word. Thus, a typical test for success is:

```
long int   result  /* result returned by function */;
short int  curval  /* current value of group */;

if ((result = srsefg(pm1gid,0L)) > = 0L)
  { /* success */
    curval = result;
    ...
  }
else
  { /* failure */
    ...
  }
```

## Wait For Event Flags

In format, the C function to wait for event flags mirrors that of the immediate set/reset:

```
long int waiefg (gid,opmask,interval)
long int  gid,opmask,interval;
```

The *gid* has the usual meaning and carries the usual penalty for errors. The mask portion of *opmask* is also the same. There are two possible operations:

EFAND is used when all specified flags must be set in order to resume the caller (AND-test). This is the default.

EFOR is used when any of the flags may be set in order to resume (OR-test).

If the selected condition is valid already, the task continues without wait. Otherwise, it waits until the condition is true or the interval is exhausted. The interval may be NOEND, a given number of time units or IMONLY (0 ms).

When the function resumes, the high-order 16 bits of the return value indicates the overall results: NOERR, BADPRM, TIMOUT or QUEFUL. The low-order 16 bits contain the image of the group, as in *srsefg*.

A typical call is:

```
pmival = waiefg(pm1gid,EFAND + EF2 + EF5,20 + SEC);
```

Note that for a mask of 0, the function always returns immediately for either type of test. Thus, one way to determine the current value of a group is:

```
curval = waiefg(pm1gid,0L,0L);
```

## Set Event Flags After Given Interval

The C function *sgiefg* permits a task to set the event flags within a global group (or its own local group) after a given time interval. This provides an alarm clock facility. At some point within a task, a timer is started by invoking *sgiefg*. The task then continues with processing that may take a variable amount of time. When that processing is completed, the task issues *waiefg* to wait until the alarm clock event flag is set.

MTOS-UX blocks the task until the remainder of the alarm clock interval runs out. This mechanism permits a task to initiate action after a predetermined time interval, and yet to use part of the wait for further work.

Since *waiefg* can be used to wait for an OR-ed combination of flags, the alarm clock can be also used to limit the wait. Say, a task wishes to wait for an event that is indicated by setting EF 1. It would like to continue even if the event does not occur after 100 seconds. EF 0 within that group is available. The task issues *sgiefg* to set EF 0 after the 100 seconds. It then wait for EF 1 or 0. The value returned by the wait can be used to determine which EF caused the task to resume.

The format of the set-EF-after-given-interval function is:

```
int sgiefg (gid,mask,interval)
long int  gid,mask,interval;
```

As in the other event flag services, *gid* is either the identifier of a global event flag group, or 0 to select the caller's local group. The flags involved are indicated by mask. The structure of interval is exactly the same as that used for a pause. An example is:

```
sgiefg(pm1gid,EF0,100 + SEC)
```

Setting another timer with exactly the same *gid* and *mask* cancels the previous one. If the new interval is 0 (with any valid time units) no new timer is started. Otherwise, the timing begins afresh with the new interval.

The function returns NOERR when it successfully starts a new timer, TIMCAN when it resets or cancels an old one, BADPRM for a bad *gid*, or time units code and QUEFUL if a new timer cannot be started for lack of internal facilities.

## Delete Global Event Flag Group

When a global event flag group is no longer needed, it may be deleted by invoking:

```
int dlefg (gid)
long int  gid;
```

If gid is not the identifier of a global event flag, the function returns a failure value of BADPRM. The function returns NOERR for success.

A problem arises in deleting an EFG that is being used by several tasks: how to know when the last task is finished with the group so that it can be removed. While there may not be a completely general solution, the following is a method that should handle most common cases.

In order to make the method work, *dlefg* does not immediately delete the group if there are either any tasks waiting for the EFG, or any outstanding timers. If there are, the EFG is internally marked "deletion-requested". Actual removal does not occur until there are no more tasks waiting or timers active. In the interim, all EF functions may be applied normally.

It is recommended that the following pair of requests be used to delete an EFG that has been created for temporary use:

```
sgiefg (gid,0L,1 + HR)    /* start a bogus timer */;
dlefg (gid)               /* delete EFG when the timer is done */;
```

The one hour would be replaced by some (over) estimate of how long it might be until the group is no longer needed. The group remains "alive" for that interval and then vanishes.

Note that this section applies only to global groups. A local group is an integral part of a task, and cannot be deleted.

## Immediately Set/ Reset Local EFs Of Given Task

The local event flags are normally the private domain of a task and are not disturbed by other tasks. However, to provide the greatest possible flexibility, MTOS-UX does permit a task to access and change even the local flags of another task. Presumably, tasks are being designed to cooperate on a common goal, rather than to confuse each other. The C function to immediately set or reset the event flagss within the local group of the selected task is:

```
long int srslef (tid,opmask)
long int  tid,opmask;
```

As always, *tid* is either the identifier of an existing task or 0 to select the caller. The treatment of a bad *tid* is the same as that for a bad *gid*. The use of *opmask* is the same as in *srsefg*. The value returned by the function is also the same, i.e. a results flag and a copy of the group after the selected operation. A simple example is:

```
tsk3lef = srslef(tsk3id,0L);
```

## Summary Of Values Returned By EFG Functions

The following shows the values that can be returned by each EFG service request:

| | |
|---|---|
| crefg | gid value; QUEFUL (long) |
| srsefg}<br>srslef} | NOERR (bits 31-16), value of group (15-0)<br>BADPRM (bits 31-16), 0xFFFF (15-0) |
| waiefg | NOERR (bits 31-16), value of group (15-0)<br>TIMOUT (bits 31-16), value of group (15-0)<br>QUEFUL (bits 31-16), value of group (15-0)<br>BADPRM (bits 31-16), 0xFFFF (15-0) |
| sgiefg | NOERR; TIMCAN; BADPRM (int) |
| dlefg | NOERR; BADPRM (int) |

# CHAPTER 9:

# SEMAPHORES AND CONTROLLED SHARED VARIABLES

# CHAPTER NINE:
# SEMAPHORES AND CONTROLLED SHARED VARIABLES

## Introduction

In many applications tasks must share a common set of data or section of code. Typical of shared data is a table that is read by one task and updated by another. While the data is being read, the update task must be blocked; while the data is being updated the reader task must be blocked. Examples of shared code are any non-reentrant subprograms that could be called by different tasks.

Such shared data and code are sometimes called "critical" regions. Not every shared resource is critical, however. A fixed data table would not be critical. Nor would a reentrant subprogram. A region is critical if it is shared and is (or can be) altered as part of its normal use.

Critical regions must be protected by guaranteeing one-task-at-a-time access. MTOS-UX provides two different facilities to achieve such mutual exclusion: the semaphore (SF) and the controlled shared variable (CSV). Since the SF is the simpler, its use will be described first.

## Semaphore

A semaphore (SF) is an internal long-word counter. If the counter is zero, the SF is free or available; otherwise, it is busy or in use. When a SF is created, it is initialized to zero.

A separate SF is assigned to each critical region. It is possible to use one SF to protect several regions, but this usually leads to excessive contention delays. Prior to using the critical region, every task must invoke *waisem* to wait for the corresponding SF to be free. If another task already has taken that SF, the new task waits. Queuing is based on the current priority of the waiting tasks. Upon exit, the current user must relinquish control to the next waiting task.

A simple way to protect critical subprograms is to make the first statement a wait-for-SF and the last statement before returning a release-SF. This places all control aspects completely within the subprogram. As a result, a caller does not have to know that the subprogram is critical.

## Create Semaphore

The C function to create a SF is:

```
long int crsem (key)
long int  key;
```

*key* is the external name associated with the semaphore. Although it is usually four printable ASCII characters, the key is taken as an arbitrary, binary pattern. If a semaphore with the given key already exists, the function returns the SF identifier. Otherwise, the SF is created and its identifier is returned. Only in the unlikely case of not having any internal resources remaining does *crsem* fail. The value QUEFUL is then returned.

A typical call is:

```
#define SF34 0x53463334

if ((s34id = crsem(SF34)) = = QUEFUL) ...
```

## Wait For Semaphore

The wait-for-semaphore function is:

```
int waisem (sid,stabfr,qual)
long int  sid,*stabfr,qual;
```

*sid* must be the identifier of an existing SF, as returned by *crsem*. Otherwise, the function returns a failure value of BADPRM. Failure values are also stored within the status buffer addressed by *stabfr*.

The qualifier, *qual*, is the sum of the standard three coordination fields. Thus:

```
waisem(s34id,&stabf,WAIFIN + 10 + SEC)
```

waits up to 10 seconds for the SF pointed to by s34id, while:

```
waisem(s34id,&stabf,CLEF2 + 1 + MIN)
```

waits no more than 1 minute for that SF and sets local event flag 2 as a completion indicator.

The task waits only if another task already has taken the SF. If the SF is already taken by the calling task, then the count within the SF is incremented by one and the task continues without wait. The count is zero when the SF is first created and when the SF is free. The count is stored internally as a long integer.

The status buffer receives the status value: NOERR, BADPRM, TIMOUT or QUEFUL.

There is value in providing counting semaphores, as opposed to just binary 1s. (A binary SF has two states: in-use and free.) In a complex application, it may be necessary to protect the same critical region in several parts of a task, say, in the main body of the code, and in some utility sub-programs. Each part of the task needs the protection of the SF. With a binary SF, it would be necessary for the task to know if it already has reserved the SF, and when it is safe to release it. With a counting SF, each part that uses the critical region is bracketed by *waisem* and *rlsem* (release semaphore). Upon exiting from the last of these nested brackets, the SF count returns to 0, and the SF is automatically freed.

There is an essential difference between event flags and semaphores, even though both are used to achieve coordination between concurrent task. If several tasks are waiting for the same EF and it is set, then ALL those tasks continue simultaneously. If several tasks are waiting for the same SF and it is released, only *one* task (the one with the highest priority) continues; the others continue to wait.

The semaphore provided by MTOS-UX is similar to, but not exactly the same as the semaphore proposed by Dijkstra ["Cooperating Sequential Processes", Technological University, Eindhoven, Netherlands, 1965 reprinted in F. Genuys (ed.), "Programming Languages", Academic Press, New York, 1968].

The MTOS-UX *waisem* is close to Dijkstra's P or wait; MTOS's *rlssem* is close to Dijkstra's V or signal. The difference is that when a Dijkstra semaphore is created, a non-negative number *s* is assigned to the SF. Thereafter, s can increase or decrease (down to 0) via P and V.

The action of P is:

if (s > 0) then --s else block task on SF

while the action of V is:

if (any tasks are blocked on SF) then release 1 task else + + s

A Dijkstra SF maintains only the tally, *s*, and the list of blocked tasks; it does not record the current "owner" of the SF (as does MTOS-UX). Thus, a Dijkstra SF permits *s* tasks to proceed into a critical region. These could be *s* different tasks, the same task *s* times, or any other combination that sums to *s*. MTOS-UX permits the same task to proceed any number of times, but blocks all other tasks.

## Deadly Embrace

There is no limit on the number of semaphores that a task can reserve and the number of wait-for-SF requests it can have outstanding. Thus, a task may wait for one SF while it has reserved another. But beware the "deadly embrace". To illustrate this phenomenon suppose that task "D" has reserved SF "SFN1" and seeks SF "SFN2". Task "E" already has SF "SFN2" and seeks SF "SFN1". This results in a deadlock, or deadly embrace.

In principle, the solution is easy: have all tasks which seek multiple semaphores always seek them in the same order. In complex cases, this may not be easy to arrange.

Deadly embraces can also arise from other combinations of limited resources. A task which has a SF and is seeking a memory pool allocation can deadlock with a task that has a large portion of the memory and is waiting for that SF.

In order to reduce the effects of a deadly embrace, try to avoid unlimited waits. And when you fail to obtain one of the needed resources, relinquish other limited resources and then try again.

## Release Semaphore

The function:

```
int rlssem (sid)
long int  sid;
```

decreases the use count of the semaphore whose descriptor is *sid* and releases it to the next user if the count becomes zero. If sid is not a SF currently held by the calling task, then the function returns a failure value of BADPRM. Two values represent success: 0 (NOERR) means that the SF was released and is now free while 1 (NOTFRE) means that the SF is still held since the count has not been reduced to 0.

For best overall performance a SF should be released as soon as possible. When a task terminates, the semaphores that it still has reserved are NOT automatically released.

## Delete Semaphore

In a dedicated real-time application it is likely that control structures, such as semaphores, would be created by an initialization task and then remain forever. In contrast, the semaphores created by a transient program are not likely to be permanent. Thus, MTOS-UX provides a mechanism to delete a semaphore once it is no longer needed:

```
int dlsem (sid)
long int  sid;
```

If *sid* is not the identifier of a SF, the function returns a failure value of BADPRM. The function returns NOERR for success.

Usually, the SF is not in use when it is deleted. If it is in use, the delete request is discarded, and the function returns the warning value NOTFRE. Each task that uses a non-permanent SF should delete the SF before it exits. Cor- respondingly, the wait request for a non-permanent SF should be coded as:

```
while (waisem(sid,&stabf,WAIFIN) ! = NOERR)
sid  = crsem(key);
```

With the wait call coded in this way, there is no harm in issuing superfluous deletes, since the SF will be re-created if it doesn't exist.

Note that what works for semaphores cannot be a general solution to the problem of deleting temporary structures. A SF is special in that it does not store any information when not in use. Event flag groups, in contrast, cannot be so casually deleted and re-created since they each contain 16 bits of information, even when no task is actively waiting for them.

## Controlled Shared Variables

In most real-time applications, a set of tasks must share a group of alterable variables, without interference from each other. As has already been seen, a semaphore can be used to grant each task exclusive access to the group. Between the time a task continues after (successfully) waiting for the SF and the time that task releases the SF, the task is said to be within the critical region associated with the SF. MTOS-UX guarantees that only one task at a time is within the critical region.

In some cases, a task must also be blocked until a certain relation exists among the group variables. For example, if the group contains binary variables (akin to event flags), the task might have to be blocked until a given set of variables are all set. Thus, the task might have to leave the critical region so that other tasks can use and change the variables, but then re-enter the critical region when the desired relation is true (see conditional critical regions in Per Brinch Hanson's "Operating System Principles", Prentice-Hall, Englewood Cliffs, N. J., 1973).

MTOS-UX includes five service calls that make it efficient to handle this type of coordination:

| | |
|---|---|
| crcsv | Create a group of controlled shared variables |
| usecsv | Wait for exclusive access to a group of controlled shared variables |
| waicsv | Wait for given function of controlled shared variables to be TRUE |
| rlscsv | Release exclusive access to a group of controlled shared variables |
| dlcsv | Delete a group of controlled shared variables |

## Create A Group Of Controlled Shared Variables

A request to create a set of controlled shared variables takes the form:

```
long int crcsv (key,len)
long int  key,len;
```

*key* is the key associated with the group of variables. Often the key is four printable ASCII characters, but it is taken as an arbitrary binary pattern. *len* is the overall length of the variables, in bytes. An example of the create call is:

```
struct meas
{
int temp[40]        /* temperature, deg F */;
int pres[40]        /* pressure, psia */;
};

static struct meas  *tpgid  /*identifier of group = addr of first variable */;

#define TPDA 0x54504441   /* key  =  'TPDA' */

tpgid  =  (struct meas *) crcsv(TPDA,(long) sizeof(struct meas));
```

If a group with the given key already exists, then the group identifier is returned as the value of the function when the current and original lengths match. When they do not match, BADPRM is returned and the SVC Parameter Error signal is sent. For a new key, MTOS-UX attempts to create the group. If successful, the identifier is again returned. If there is not enough internal memory currently available to do the creation, the function returns the error code QUEFUL.

The group identifier is also the address of the first variable. The group is created with all variables initialized to 0. The required space is taken from the Global TPA.

## Wait For Exclusive Access To Controlled Shared Variables

The function:

```
int usecsv (gid,interval)
long int  gid,interval;
```

indicates that the requesting task wishes exclusive access to the controlled variables within group *gid*. If the group does not exist, *usecsv* returns BADPRM and sends the SVC Parameter Error signal. If the task already has exclusive access to the group, the return value is DUPTSK (duplicate task request). The specified interval, the return value is NOERR. Finally, if the group remains unavailable during the given interval, the request is cancelled and the return value is TIMOUT.

Once exclusive access is granted, the task may freely and safely use and change the group variables. Recall that the group identifier is also the address of the first of the variables.

Continuing the example introduced on the previous page:

```
if (usecsv(tpgid,2 + SEC) > = 0)
  {
  tempchg = tpgid->temp[12] - tpgid->temp[9];
  tpgid->pres[12] =  40;
    .
    .
    .
  }
```

## Release Controlled Shared Variables

When a task no longer needs its exclusive access to a group of CSVs, it must issue a release request, using the format:

```
int rlscsv (gid)
long int  gid;
```

*gid* identifies the group, with the usual consequences for giving a bad parameter. Once released, the task must not alter any of the variables, even though MTOS-UX does not have the ability to enforce this rule.

Calls to *waicsv* and *rlscsv* mark the entry into and exit from the critical region for the group, in a way analogous to *waisem* and *rlssem*. When a task terminates, there is no automatic release. Failure to release a group of CSVs is an application error which cannot be detected by MTOS-UX.

## Wait For Function Of Controlled Shared Variables To Be True

The C function to wait for a certain relation among CSVs to be true is:

```
int waicsv (gid,bfun,interval)
long int  gid,interval;
int       (*bfun) ();
```

*gid* identifies the variables group. *bfun* supplies the address of the evaluation function. *interval* is the maximum time to wait before returning from *waicsv*. If the interval is NOEND, the service can never timeout. Possible return values are:

| | | |
|---|---|---|
| | NOERR | Success |
| • | TIMOUT | *bfun* is never TRUE during the specified interval |
| • | QUEFUL | The timer cannot be allocated |
| • | BADPRM | The group does not exist |

As always, the Error signal is sent when there is no such group.

When called, *bfun* is presented with a pointer to the group variables as its only argument. The function must return an integer value of zero if the task is to continue or non-zero if the task is to be blocked. No task-level service calls (SVCs) are permitted within *bfun*.

As a convenience for the user, the wait service call may be made even if the task is not currently in the critical region for the group. In this case, there is an implicit *usecsv* call with the same maximum time interval. Any time spent satisfying the implicit *usecsv* is taken away from the *waicsv*.

The evaluation function is called first when *waicsv* is invoked, or after the implicit *usecsv* returns succesfully. If the task is to be blocked, *bfun* will be called again each time a task leaves the critical region via *rlscsv* or *waicsv*. If more than one task could be unblocked because its evaluation function is true, only the highest-priority task will be continued at that point. The others will have to wait until the region is available again.

Whether or not a task had exclusive access to the group originally, it loses this privilege while it is blocked and regains it when the task becomes unblocked when the evaluation function was satisfied. However, it does not have access upon a timeout or other unsuccessful return. Thus, the application must have the following overall structure:

```
[usecsv();]
  .
  .
  .
if (waicsv() = = NOERR)
  {
    .
    .
    .
  rlscsv();
  }
```

## Delete A Group Of Controlled Shared Variables

When a group of controlled shared variables is no longer needed by any task, it may be deleted by invoking:

```
int dlcsv (gid)
long int  gid;
```

If *gid* is not the identifier of a group of controlled shared variables, the function returns a failure value of BADPRM and receives the Error signal. The function returns NOERR for success.

The same problem arises in deleting a group of CSVs as in deleting a group of Global Event Flags: how to know when the last task is finished with the group so that it can be removed. The same solution is used. Thus, *dlcsv* does not immediately delete the group if there are any tasks waiting because of *waicsv*. If there are, the group is internally marked "deletion-requested". Actual removal does not occur until there are no more tasks waiting. In the interim, all CSV functions may be applied normally.

# CHAPTER 10:

# SIGNALS

# CHAPTER 10:
# SIGNALS

## Introduction

A signal is a software interrupt that may be handled at the task level. There are four modes of use:

- **Error recovery**
  MTOS-UX automatically sends a signal to a task when the task generates an error exception, such as an arithmetic overflow.

- **Debugging**
  A signal is sent after the execution of a breakpoint. A different signal is sent after the execution of any instruction for which the trace flag is set in the task's status register. These signals may be used to invoke the Debugger, or may be handled by the task itself.

- **Coordination**
  A task may elect to have a signal sent as the completion indicator for a requested service.

- **Communication**
  A task can send a signal to another task, or to a group of tasks, as a means of communication.

There are 32 signals, numbered 0 to 31. As shown in Figure 10-1 on page 10-9. Signal 31 forces termination (kills the task). Signals 16 - 30 are reserved for debugging and error recovery. Within this group, the meaning of certain error signals is hardware-dependent.

Signals 2 - 15 are available for end-of-service indicators. An attempt to use a signal above 15 for coordination is rejected. Any task can send Signals 0 to 15 (or 31) to any task, including itself.

Signals 0 - 1 are reserved for the Chameleon 32 system.

There are four possible responses that a task can make when it receives a signal:

- It can ignore it
- It can terminate
- It can perform a subprogram
- It can become blocked

In the last case, the Debugger is started to handle the signal and unblock the task.

Most often, the task will perform a subprogram, which is both asynchronous and isolated. Asynchronous is mentioned since the signal is often sent at random with respect to the main execution path of the task. (The task may not even be running when the signal is sent.) The subprogram is isolated in the sense that the main execution path is interrupted, the subprogram is performed, and then (usually) the main path is continued.

In MTOS-UX the response to a signal does not occur until the task becomes Ready. Normally, signals arriving while a task is Blocked or Dormant are held. However, a termination signal (31) arriving while a task is Dormant is discarded. The only exception occurs when the task is blocked waiting for a signal.

## Set Response To Signal

When a task is first created, the default response is to ignore Signals 0 to 15 and terminate for Signal 31. If the optional Debugger task has been installed, then the default for the error signals (15 to 30) is to become blocked and start the Debugger to unblock it. If the Debugger is not present, the default is to print an error message on the System Console via the Error Logger task (if present), and then terminate the errant task.

The function *setsig* can be used to alter (or reset) the response to a particular set of signals. The C definition of the function is:

```
int setsig (sigmsk,resp)
long int  sigmsk;
int       (*resp) ();
```

The signals of interest are selected by *sigmsk*, using one bit per signal, left to right. A value of 0x80000000 selects only Signal 0. The literals SIG0 to SIG31 may be combined to select the appropriate bit or bits. The coordination mode literals, CSIG0 to CSIG15, may not be used.

The desired response is indicated by *resp*. Four literals are recognized:

- SIGIGN (ignore)
- SIGBLK (become blocked if the Debugger is present; terminate if not)
- SIGTRM (terminate)
- SIGDFL (reinstate the default)

Any other value is assumed to be the address of a function to be executed upon receipt of the signal.

The value returned by a successful call of *setsig* is NOERR. BADPRM indicates that the change was rejected because the function address was on an odd boundary or not accessible by the task.

Some examples of the call are:

```
setsig(SIGALL,SIGDFL);

int dbgtrc(),dbgbrk();
setsig(SIG16,dbgtrc);
setsig(SIG27,dbgbrk);
```

## Get Response To Signal

It is convenient at times to determine the response to a utility. For example, a utility may wish to process arithmetic faults itself and then restore the original response. The function *getsig* can be used to capture the current response to a particular signal. The C definition of the function is:

```
(*getsig (sig)) ()
long int sig;
```

The signal of interest is given by *sig*. For any value out of the range 0 to 31, the function returns BADPRM. With a valid signal index, the current response is returned as the value of the function.

A sample call is:

```
respf3 = setsig(3L);
```

Note that sig for Signal 3 is 3, not SIG3 (which is 0x10000000).

## Send Signal

A task can send a signal by invoking the C function:

```
int sndsig (tid,sig)
long int  tid;
long int  sig;
```

Normally, *tid* is the identifier of the task to receive the signal. A task can send a signal to itself. For the special value 0, the signal is sent to all other application tasks in the system. This might be used to terminate all tasks prior to shutting down the computer. Similarly, for the special value -1, the signal is sent to any other tasks which are sharing code or data). It is not deemed an error if there are no such tasks.

The signal to be sent is designated by *sig*. Proper values are 0 to 15, and 31.

Examples are:

```
sndsig(-1L,3L);

sndsig(tsk4id,15L);
```

## Send Signal After Given Interval

A task can have a specified signal sent to itself after a given interval via the C function:

```
int sgisig (sig,interval)
long int sig,interval;
```

The signal to be sent is specified by *sig*. Proper values are 0 to 15, and 31. *interval* is given in the usual way.

Examples are:

```
sgisig(3L,2 + MIN);

sgisig(15L,50 + MS);
```

Often, the response to a signal sent by *sgisig* is the execution of an asynchronous subprogram. In this way, a single task can carry out a primary activity, and periodically perform some auxiliary work (via the signal-invoked subprogram).

## Pause For Signal

A task can pause until a signal arrives by calling the C function:

```
int pausig (interval)
long int interval;
```

The pause is limited to the interval specified by *interval*. NOEND is valid. The first signal to arrive cancels the pause. The function returns the signal number (0 to 31), TIMOUT or BADPRM. The signal pause cannot be cancelled by *canpau*.

Note that the pause is cancelled even if the response is to ignore the signal.

Commonly, *pausig* would be used prior to termination if a task still has not received all of its expected coordination signals. If five different signals are expected, *pausig* would have to be issued five times.

## Structure Of A Signal Subprogram

When coded in C, a signal response subprogram has the general structure:

```
sigsub(sigblk)
        struct scf  *sigblk  /* ptr to signal data block */;
        {
                .
                .
                .
                [return();]
        }
```

*sigblk* is a pointer to a signal context frame that is used to transmit parameters to signal subprograms, debuggers and error loggers.

When the signal is generated during the processing of a service request issued by the target task, the exception type is 1 and the SVC return value indicates the value to be returned from the request. Otherwise, these values reflect the context at the last time the task was interrupted.

Usually, the data within the frame is for information only; it is not altered by the task receiving the frame. However, this need not be so. A task (or more commonly the Debugger) may alter frame data, such as register values. Be careful if the data is altered: when the task returns from the signal subprogram it will continue using the values within the frame. The following fields may not be changed: fsn, tti, et, pi, lsn and lsl.

At the entry point of a signal subprogram, all registers (except the stack pointer) have been saved, but still retain the values of the interrupted task code. Upon exit, the values are restored by MTOS-UX; the subprogram need not be concerned with preservation of registers. The stack may be used, but should be restored by the task prior to exit.

A signal subprogram is an extension of the main task; it has the same privileges and restrictions as that task. Any task service available to the main task code, may be called by the subprogram. In particular, Signals can be sent and setsig can be invoked.

The subprogram may exit by executing an explicit return statement, or by reaching the end of the subprogram. (In the latter case, the compiler supplies the return.) If an exit or trmrst is called, the task terminates without completing the interrupted main task code.

## Detailed Handling Of Single And Multiple Signals

A signal starts at signal level 0. When a signal arrives, its level is compared with the task's signal level. If the new signal is of the same or lower level, its arrival is noted, but no action is taken. In this case, if the same signal is already pending, the new one is effectively ignored, no matter what the response is supposed to be. Furthermore, the response action is not yet examined, so that the signal remains pending even if the response is currently to ignore the signal.

If a signal arrives with a level greater than the current signal level of the task, then the new signal is processed immediately. The status of the task is examined. If Dormant, the signal is ignored. If the task is paused waiting for a signal, the pause is cancelled making the task Ready. If Blocked, the signal is noted in the task's pending signals variable, and further action is deferred until the task becomes Ready.

For a Ready task, the response for the signal is examined. For SIGIGN, no further action is taken. For SIGTRM, the task is terminated. For SIGBLK, if the Debugger is present, the task is blocked and the Debugger is started (passing to it the same block that a signal subprogram would receive). For SIGBLK without a Debugger, the error/signal block is passed to the Error Logger task (if present) and then the errant task is terminated. Finally, for the remaining possibility, execute a signal subprogram, the task context (status register, program counter, general registers and signal level) are saved on the task stack, the task signal level is raised to that of the new signal, and the subprogram entry point becomes the new program counter.

When the signal processing is postponed because the level of the incoming signal is too low, the above steps are carried out when the processing unnests to the required level. Similarly, when a task which was Blocked becomes Ready,any pending signal processing is continued.

When a task terminates, with or without a timed restart, all pending signals are automatically cancelled.

## Cancel Pending Signals

Under certain conditions, the execution of a single task instruction can generate more than one signal. In such cases, the signal with the higher level is processed first, and the other remains pending. It may be useful in these cases to avoid processing the second signal, using the service call:

```
long int cansig (mask)
long int  mask;
```

This discards the pending signals selected by *mask*, and returns the image of the signals that still remain pending. The call *cansig*(0) returns the image of the task's pending signals without cancelling any of them. The call *cansig*(SIGALL) cancels all signals and returns 0.

## Application Notes

Both local event flags and signals can be used to achieve delayed coordination with the end of a requested service. Event flags are normally used when there will be a point within the program at which the task must wait for the service to be completed. At that point, the task issues a *waiefg*. Furthermore, the task can wait for an AND or OR combination of up to 16 different flags (within one group).

Signals are normally used when the end-of-service processing is independent of the remainder of the task program. A typical sequence is: the task allocates an area from a memory pool, builds a message and then outputs the message to a mailbox or pipe. When the message is transferred, the response must be to deallocate the block, no matter what else the task is doing. For this, signal coordination is ideal.

In many ways, the coordination pairs *pause/canpau* and *pausig/sndsig* are equivalent. However, since different signals can be sent to the paused task, the second pair can have an additional element of information (0 to 16).

# Table 10-1: Signal Usage

| No. | Level | Use Within MTOS-UX | Default* |
|---|---|---|---|
| 0-5 | 1 | Reserved | SIGIGN |
| 6-15 | 1 | Available for coordination | SIGIGN |
| 16 | 2 | Trace (single step) | SIGBLK |
| 17 | 3 | (unassigned) | SIGBLK |
| 18 | 3 | Coprocessor fault | SIGBLK |
| 19 | 3 | Arithmetic exception, such as divide by 0 | SIGBLK |
| 20 | 3 | (unassigned) | SIGBLK |
| 21 | 3 | (unassigned) | SIGBLK |
| 22 | 3 | (unassigned) | SIGBLK |
| 23 | 3 | Unimplemented software interrupt | SIGBLK |
| 24 | 3 | Illegal instruction | SIGBLK |
| 25 | 3 | Privilege violation | SIGBLK |
| 26 | 3 | Bad parameter in service call | SIGBLK |
| 27 | 4 | Breakpoint reached | SIGBLK |
| 28 | 5 | Memory access error, such as writing to ROM | SIGBLK |
| 29 | 5 | Boundary error | SIGBLK |
| 30 | 6 | Stack overflow | SIGBLK** |
| 31 | 7 | Terminate ("kill task") | SIGTRM |

(*) SIGIGN = Ignore
SIGBLK = Become blocked if the Debugger is present; terminate if not
SIGTRM = Terminate

(**) Cannot be changed to run subprogram or ignore.

# CHAPTER 11:

# MESSAGE BUFFERS AND MAILBOXES

# CHAPTER ELEVEN:
# MESSAGE BUFFERS AND MAILBOXES

## Introduction

MTOS-UX permits tasks great freedom to communicate and coordinate by passing messages to each other. The task sending the message does not specify the receiving task directly. Instead, the sender posts the message to an independent object, a message buffer or a mailbox, and the receiver obtains the message from that object.

The message buffer is the quickest and easiest mechanism for passing messages. It will be described first. The mailbox provides additional facilities and will be covered later in this chapter.

A message buffer (MSB) is a place to which a message may be sent and from which a message may be received. The number of MSBs and the kinds of messages transferred are completely up to the user; MTOS-UX imposes no restrictions of its own.

A MSB message is always a single long integer (four bytes). Often the message is the address of a structure containing the parameters of some work to be done. However, the content of the message is not significant to MTOS-UX; the value is transferred without regard to its possible meaning.

A message buffer is a storage device; when there is no receiver immediately available, the 4-byte message is copied into the buffer. The maximum number of messages is specified when the buffer is created. A task that attempts to post a message to a full buffer is given a failure return value of QUEFUL. Similarly, when a task receives a message the four bytes are removed from storage.

After a task posts a message, it always continues without coordination. The only option at the send end is whether the message should be placed at the end of the buffer (FIFO) or at the beginning of the buffer (LIFO) in case there is no task already waiting to receive the message. A MSB message does not have a priority.

A task seeking a message at an empty MSB can either wait for the next message to arrive, or continue and be notified that no message is currently available. These wait options enable tasks to coordinate their activities.

## Typical Use Of Message Buffer

To illustrate the use of message buffers, consider an application in which there are four tasks that produce blocks of parameters. Each block must be expanded into a formal report that is to be output to one of two identical printers. It is not important to specify the printer to be used for a given report. It is important that a printer not be idle while a block of parameters is available.

Each producer task allocates a work area from a memory pool, builds a parameter block in the area and then sends the address of the area as a message to a certain MSB. The producer does not wait for the message to be received and thus is immediately available to prepare the next block.

Two tasks are used to do the report generation and printing. Each task executes the same re-entrant code, but has its own dedicated printer. A printer task seeks the address of a parameter block as a message from the same MSB. (If there is no message queued, it waits.) When the printing is completed, the printer task returns the work area to its pool and then seeks the next message in an endless loop.

Thus the MSB provides an orderly way to coordinate the producers and printers. Specifically, it allows the producer to send work to the next free printer, without knowing which printer it is.

The parameter blocks need not all be of the same size or come from the same pool. Usually, they are not. Part of the parameter block can specify the length of the work area and the identity of the pool.

## Create Message Buffer

A MSB must be created before any task can use it. The C function to do this is:

```
long int crmsb (key,attr)
long int  key,attr;
```

*key* is the key associated with the message buffer. A pattern unique among MSBs is required. The attributes parameter, *attr*, is the sum of two components: a global/local specifier (gls) and a size or capacity specifier. The choices for the gls are:

- MSBGBL for a global buffer
- MSBLC0 for a buffer local to processor 0 (default)

The size specifier indicates the maximum number of messages that can be stored. The low-order 13 bits are used so that the highest value is 8191. Zero defaults to 64.

Some typical calls are:

```
#define MSB0 0x4D534230
#define MSB1 0x4D534231
#define MSB2 0x4D534232

msbid0 = crmsb(MSB0,50L);

msbid1 = crmsb(MSB1,MSBGBL + 200);

if ((msbid2 = crmsb(MSB2,MSBLC2 + 500)) = = QUEFUL) ...
```

As with an event flag group, if an MSB with the given key does not already exist, it is created by this request. The only return values are the MSB identifier for success and QUEFUL or BADPRM for failure. The successful return value does not distinguish an MSB that already existed from one that was just created.

The buffer is created within a TPA. The global TPA is used for single CPU systems and for multiple CPU systems when MSBGBL is given. If a local buffer is specified, the corresponding local TPA must already exist. An MSB is created empty.

A local message buffer must be created, used and deleted on the processor specified in its attributes field. The advantage of a local buffer is reduced traffic over the backplane. The advantage of a global buffer is universal access by all tasks.

## Get Identifier Of Message Buffer

Any C task can determine the identifier of a message buffer from the key via:

```
long int getmsb (key)
long int  key;
```

## Post Message To Buffer

The C functions to post a message to a buffer are:

```
int putmse (msbid,msg)
long int  msbid,msg;

int putmsb (msbid,msg)
long int  msbid,msg;
```

The buffer is selected by *msbid*. If *msbid* is invalid the call returns with an error value of BADPRM and sends the error signal. The 4-byte message is given by *msg*.

If there is no receiver immediately available, the message is stored in the buffer. The first function places the message after any others that are already queued; the second places it before any others. The "normal" case is *putmse*, which provides FIFO buffering. If there is no room left in the buffer, the value QUEFUL is returned, but no signal is sent.

Note that the message buffer facility has been designed primarily for speed. Thus, there are no provisions for message priority or coordination at the send end. Applications needing these features can find them in the mailbox services that are described later in this chapter.

## Get Message From Buffer

A corresponding pair of C functions is used to get a message from a buffer:

```
int getmsw (msbid,dstadr)
long int  msbid,*dstadr;

int getmsn (msbid,dstadr)
long int  msbid,*dstadr;
```

Parameter *msbid* must be the identifier of an MSB; otherwise an error is generated. The address of the buffer to receive the message is given by *dstadr*. With the first function the task will be blocked until a message is available. With the second, if no message is already queued, the return value is MBEOF. Thus, *getmsw* is "get a message with wait" and *getmsn* is "get a message with no wait".

For getmsw, tasks waiting for a message are queued first-come-first-served. There is no limit to the number of tasks that may be queued waiting for messages.

## Delete Message Buffer

A MSB may be deleted by invoking:

```
int dlmsb (msbid)
long int  msbid;
```

If *msbid* is not the identifier of a buffer, the function returns a failure value of BADPRM. The value for success is NOERR.

Usually, the MSB is not being used when it is deleted. However, if there are any queued messages or pending receive requests, then the buffer is marked *deletion pending*, but it is not removed until activity ceases. New requests will still be honored while the buffer is awaiting deletion.

## Using A Message Buffer To Grant Exclusive Access

A message buffer can be used to achieve exclusive access to a critical region, as an alternative to the semaphores described in the previous chapter. A buffer is created and then primed by sending it one dummy message. Thereafter, whenever a task needs access to the variables, it requests to receive that message with wait. Since there is only one message, only one task at a time could proceed; all others queue up at the buffer. Sending the message back to the buffer enables the next task to use the variables.

The idea is easily generalized for cases that can permit access by more than one task at a time. To give a concrete example, suppose there are four independent and equivalent channels on a certain piece of equipment. Several tasks wish to use a channel, but do not care which one is provided. A buffer is created to handle the assignment of channels. The creating task initially fills the MB with four messages, each containing a channel number (say, 0 to 3). Now a task waits for a message granting it permission to use one of the channels, and eventually returns the message to release the channel to the next user.

## Mailboxes vs. Message Buffers

While message buffers are versatile enough to solve many problems that arise in real-time applications, there are often cases in which a stronger facility is required. MTOS-UX mailboxes provide full coordination at both the send and receive ends, arbitrary message length, unlimited queuing and 256 levels of message priority.

In some situations, it is possible to fabricate the features of the MTOS-UX mailbox by linking several simple buffers through a complex arrangement of calls. Nevertheless, there are two major disadvantages for creating strong facilities at the task level from simple operating system services:

- Such creations often have to be very complex in order to avoid subtle bugs that arise if a task is interrupted by a higher priority task in the middle of a sequence of calls.
- The use of many simple calls usually results in far greater overhead than making one complex call, even though the complex call, taken by itself, is longer than any one simple call.

As with a message buffer, a mailbox (MBX) is a place to which a message may be sent and from which a message may be received. The number of MBXs and the kinds of messages transferred are completely up to the user.

A MBX message can be a record containing any number of characters. The content of the record is not significant; the bytes are transferred as an unstructured string. Thus, a record may be a block of text to be processed, a set of data to be reduced, or even the address and length of the real text or data, as stored in a memory pool.

A task receiving a MBX message may specify an input buffer shorter than the incoming message. This is considered normal. The message is truncated, with the excess text discarded. In any case, receiving a message always consumes it; that is, removes it completely from the MBX.

After a task sends a message, it has the option of continuing, or waiting until the message is received. Similarly, a task seeking a message at a MBX that presently has no messages can either continue, or wait for the next message to arrive. These wait options enable tasks to coordinate their activities.

MBX messages have a priority. If there is no receiver waiting, more important (higher priority) messages are stored in a queue ahead of less important ones. For messages of equal priority it's first-in-first-out.

There is no corresponding priority for receivers. When a task waits for a MBX message, it's strictly first-come-first-served. It is assumed that all receivers are identical so that there is no need for priority ordering of the wait queue.

A mailbox is also a buffer: a storage device with a send end, a receive end and storage in between. However, the storage is used to hold only the parameters of unfulfilled send or receive requests. The content of a message is not copied until a receiver is available and then it is transferred directly into the receiver's buffer. The sender may choose to send a message and then continue without waiting for a task to receive it. Nevertheless, because there is no internal storage of text, the sender cannot alter the area containing the message until it is transfered to the receiver.

The differences between a message buffer and a mailbox are summarized below.

| FEATURE | MESSAGE BUFFER | MAILBOX |
|---|---|---|
| Length of message | 4 bytes | any |
| Store message? | yes | no |
| Maximum number? | yes | no |
| Message priority? | no | yes |
| Coordination for send? | none | general |
| Coordination for receive? | WAIFIN | general |
| Speed? | faster | slower |

## Open/Create Mailbox

A MBX must be opened before it can be used. The C function to do this is:

```
long int opnmbx (key,mode)
long int  key,mode;
```

*key* is the external name (4 bytes). *mode* is the intended manner of use, as follows:

- MBRCV ( = 0) for receiving
- MBSND ( = 1) for sending

A task may make both types of open (without any requirement for an intervening close) if it intends to both send and receive messages with the same target MBX. A typical call is:

```
#define MB03 0x4D423033

if ((mb3id = opnmbx(MB03,MBRCV)) = = QUEFUL) ...
```

If a MBX with the given key does not already exist, it is created by this request. The only return values are the MBX identifier for success, BADPRM for an incorrect mode and QUEFUL for failure. The return value does not distinguish a MBX that already existed from one that was just created. A MBX is created empty (no senders or receivers waiting).

Each time a MBX is opened, a tally within the control data for the MBX is incremented, and each time the MBX is closed the tally is decremented.

There are separate tallies for send and receive opens. However, the identity of the task making the request is not saved. As a result, it is not necessary for each task that uses a MBX to have opened it. All that is required is that the current tally of opens minus closes be greater than zero for the mode of use.

Furthermore, it is unusual, but not wrong, for a task to issue an open for a MBX which it has already opened. When a task is terminated or deleted, its open MBXs are not automatically closed.

## Send Message To Mailbox

The C function to send a message to a mailbox is:

```
int sndmbx (mbid,srcadr,prty,stabfr,qual)
long int  mbid,prty,*stabfr,qual;
char      *srcadr;
```

The mailbox is selected by *mbid*, which must be the identifier of a MBX that has been opened for sending (and not subsequently closed). If *mbid* is invalid, the call returns with an error value of BADPRM.

*srcadr* is a pointer to the message. The length of the text (in bytes) is supplied in the first four bytes, followed directly by the text. A length of zero is accepted. Placement of messages on word boundaries considerably improves the speed of this service call.

If there is no receiver immediately available, and IMONLY was not specified, the parameters of the request are queued. If *prty* is non-zero, the message is queued in priority order based on *prty*. If prty is 0, the message goes immediately to the end of the queue. As a result, if priority is not an issue, all messages should use a value of 0 for fastest processing.

The parameter *stabfr* is a pointer to a long integer buffer into which the transfer status can be stored. *qual* is the coordination/service limit qualifier, specifying the *cmode*, *lunits* and *lnum* fields.

If the transfer is made, the status buffer contains NOERR. For failure, the buffer has BADPRM, TIMOUT or QUEFUL. Some examples are:

- To send the 7-byte text "MTOS-UX" with unlimited wait and priority 100:

  ```
  sndmbx (mb3id,"\0\0\0\7MTOS-UX",100L,&status,WAIFIN);
  ```

- To send the same text with a wait limited to 1 hour; force message to end of queue (priority 0):

  ```
  sndmbx (mb3id,"\0\0\0\7MTOS-UX",0L,&status,1 + HRS);
  ```

- To send the text within string msgstg with 255 priority, continue, and set LEF 0 when done:

  ```
  sndmbx (mb3id,&msgstg,255L,&status,CLEF0);
  ```

When sending a message to a MBX it is not necessary that any task currently have the box opened for receiving.

## Receive Message From Mailbox

A similar C function is used to receive a message from a mailbox:

```
int rcvmbx (mbid,dstadr,stabfr,qual)
long int  mbid,*stabfr,qual;
char      *dstadr;
```

*mbid* is the identifier of a MBX open for receiving. *dstadr* is the address of the buffer to receive the message. The maximum size of the text (in bytes) is specified in the first four bytes of the buffer. The actual buffer should be four bytes longer than the maximum text size.

The transfer is limited to the smaller of the length of the message content and length of the receiving area. The actual number of bytes transferred is stored in the first four bytes of the area. However, the area need not be on a word boundary.

If the area is longer than the message, the unused portion of the area is not cleared. If the message is longer than the area, the unused portion is discarded. Neither case is considered an error. A text length of zero is valid and provides coordination without text transfer.

*qual* is the coordination/service limit qualifier. *stabfr* is a pointer to a buffer into which the status can be stored.

Coordination for *rcvmbx* is similar to that used for *sndmbx*. The only difference is that all receivers are assumed to have equal priority so that the wait queue is strictly FIFO. Two examples are:

To receive up to 125 bytes into rcvstr, with unlimited wait:

```
struct MB125 {long int  tsiz;
char      txt[125]} rcvstr;
.
.
.
rcvstr.tsiz  =  125          /* set max len */;
rcvmbx (mb2id,rcvstr,&status,WAIFIN);
```

To fill rcvstr, continue, and set LEF 15 when done:

```
rcvmbx (mb2id,&rcvstr,&status,CLEF15);
```

If there is a message available when the receive is issued then the function returns immediately with status NOERR. Otherwise, the task is expected to wait or not, as specified in the coordination qualifier. However, when the MBX is being used as a private conduit (pipe) between tasks, it is important to be able to distinguish a MBX that is temporarily empty from one that is permanently in that state. In the first case, it makes sense to wait for a message; in the second it does not. Towards this end, if the MBX was once opened for sending and is currently not opened in that mode, then an unsatisfied receive request is not queued. Instead, it returns immediately with "at end of file" (MBEOF) status. This applies for all values of the coordination qualifier.

## Close Mailbox

A MBX that is currently open in a given mode can be closed via the C command:

```
int clsmbx (mbid,mode)
long int  mbid,mode;
```

A valid close decrements the opens-remaining tally for the given mode. If the new tally is still one or more, the function returns NOTFRE to indicate that there are other opens still outstanding.

If there are no more opens left for the given mode, the function returns 0 (=NOERR). When that happens for the send mode and there are receive requests queued, then the receive queue is purged. The unsatisfied receive requests are given the status MBEOF. It is assumed in this case that there will not be any more messages posted to the MBX. Purging a request constitutes completion for purposes of coordination. The corresponding actions are not taken for send requests when there are no receivers.

For a close in either mode, if the MBX has already been opened for both sending and receiving, and is now not opened in any mode, and there are no requests queued, then the MBX is deleted. The rationale for delaying the deletion until the MBX is opened at least once for receiving is simple: all senders may produce and post all of their messages and close the MBX before the receivers start execution.

When a task terminates or is deleted, its open MBXs are not automatically closed. As a result, MBXs can remain in existence (using limited internal resources) if they are not specifically closed before a task terminates.

## Delete Mailbox

A MBX may be deleted by invoking:

```
int dlmbx (mbid)
long int  mbid;
```

If *mbid* is not the identifier of a mailbox, the function returns a failure value of BADPRM. The value for success is NOERR.

To understand the main purpose of *dlmbx* consider the Command Line Interpreter (CLI). Often the CLI starts sets of tasks that communicate via MBXs. If one of these tasks were to terminate before it had a chance to close a MBX, the MBX would remain forever. The CLI can issue *dlmbx* to purge such MBXs.

Usually, the MBX is not being used when it is deleted. If there are any send or receive requests pending, then the requests are purged, with error status MBDLT.

## Using A Mailbox As A Pipe

A pipe is a connection between two tasks, arranged so that the output of one task becomes the input to the other. Under UNIX a pipe is implemented via the file system; under MTOS-UX a pipe can be achieved using a MBX.

The following suggests one method to create a mailbox pipe. Many variations are possible.

A sender task ("S") issues an *opnmbx* with a key, say 'PS/R', and mode MBSND. When "S" wishes to output some text, it uses alloc to obtain a pool area large enough to house the text. (Typically the area is larger than needed because of the granularity of a pool allocation.) The text is stored. "S" then posts a message containing the address and length of the allocated area to the pipe MBX. The priority is 0 so that messages proceed FIFO. No coordination is used; "S" continues. When there is no more output, "S" closes the MBX.

A receiver task ("R") issues a corresponding opnmbx with identical key and mode MBRCV. "R" seeks a message from the pipe MBX with unlimited wait. When "R" continues, it has either the address of the pool area or the MBEOF status. In the former case, it uses the text, deallocates the pool area and then repeats the loop. In the latter case, it also closes the MBX to delete it.

## Activating Service Tasks

Two methods are available for organizing a service task ("S") that does work specified by a block of parameters. In the first, the task is arranged as a loop: it seeks the block of parameters at a mailbox, does the required work and then cycles back to seek another block at the MBX. For this case, a task requests the service provided by "S" by sending a message to the MBX.

In the alternate method, the service is requested by starting "S" with the address of the parameters as the restart argument. "S" becomes active, does the service and then terminates to be available for the next request.

Both methods are similar in that several tasks can invoke the services of "S" and assign a priority to the units of work (block of parameters). Furthermore, there can be as many copies of "S" as are needed to handle the work. However, in several ways these methods are significantly different.

With the mailbox approach, when there are multiple copies of "S", load leveling occurs automatically, since as soon as an "S" finishes it seeks the next available unit of work. With the start-task method, the user must assign a unit of work to a specific copy of "S". Thus, one copy might have many unsatisfied requests queued while the other copies are idle.

The use of a mailbox is also more general since the transfer of the parameters message can proceed via a network. When only the address of the block is supplied (via the restart argument) the parameters must be directly accessible to "S". On the other hand, if the parameters block is very long and happens to be directly accessible, then communicating the address is more efficient than transferring the entire block.

The main advantage of using start-task is that it is easy to coordinate with the termination of "S" and thus to know when the service has been completed. With MBX messages, the coordination is limited to the transfer of the parameters block and thus to the start of the service. Indirect methods (such as having "S" set a given global event flag) must be used to determine when the service is finished.

# APPENDIX A:

# SUPERVISOR SERVICES SUMMARY

# APPENDIX A: SUPERVISORY SERVICES SUMMARY

Introduction

This appendix provides a summary of the Supverisory Services. The available functions are listed below with a brief description. Each function is described in more detail following the brief listing.

# CANCEL PAUSE

**DESCRIPTION** Continue given task if it is paused for time interval.

**REFERENCE** Page 4-4

**C FUNCTION**

```
int canpau(tid)
long inttid;
```

**PARAMETERS** tid = task identifier.

**RETURNS**

| | |
|---|---|
| NOERR | Task was paused. |
| NOTOUT | Task was not paused. |
| BADPRM | tid is invalid. |

**EXAMPLES**

```
result = canpau(scntid);
if (canpau(logtid) = = NOTOUT) ...
```

# CANCEL PENDING SIGNALS

**DESCRIPTION** Cancel pending signals of requesting task.

**REFERENCE** Page 10-8

**C FUNCTION**

```
long int cansig(mask)
long intmask;
```

**PARAMETERS** mask = signals to be cancelled (SIGALL for all).

**RETURNS** Image of signals that remain still pending.

**EXAMPLES**

```
cansig(SIGALL);
siglft = cansig(SIG16 + SIG17);
pendsg = cansig(0L);
```

# CLOSE MAILBOX

**DESCRIPTION** Close mailbox.

**REFERENCE** Page 11-11

**C FUNCTION**
```
int clsmbx(mbid,mode)
long intmbid,mode;
```

**PARAMETERS** mbid = mailbox identifier.

mode = type of close (MBRCV, MBSND).

**RETURNS**

| | |
|---|---|
| NOERRM | B has no more opens left for given mode. |
| NOTFREM | B has 1 or for more opens left for given mode. |
| BADPRM | mbid or mode is invalid. |

**EXAMPLES**
```
if (clsmbx(mb0id,MBSND) = = NOTFRE) ...
```

# CREATE CONTROLLED SHARED VARIABLES

**DESCRIPTION** Create group of controlled shared variables.

**REFERENCE** Page 9-7

**C FUNCTION**

```
long int crcsv(key,len)
long intkey,len;
```

**PARAMETERS**

key = external name of group.

len = length of group (bytes).

**RETURNS** Identifier if group is successfully created or already exists.

| | |
|---|---|
| BADPRM | Parameter is invalid. |
| QUEFUL | Insufficient internal resources to create group. |

**EXAMPLES**

```
#define TPDA 0x54504441

struct meas
    {
        int temp[40];
        int pres[40];
    };

struct meas *tpgid    /* identifier of group = addr of first variable */;

tpgid = (struct meas *) crcsv(TPDA,(long) sizeof(struc meas));
```

# CREATE EVENT FLAGS

**DESCRIPTION** Create group of (global) event flags.

**REFERENCE** Page 8-2

**C FUNCTION**
```
long int crefg(key)
long intkey;
```

**PARAMETERS** key = external name of group.

**RETURNS** Identifier if pool is successfully created or already exists.

QUEFUL Insufficient internal resources to create group.

**EXAMPLES**
```
#define PMP1 0x504D5031

pmp1id = crefg(PMP1);
```

# CREATE MESSAGE BUFFER

**DESCRIPTION** Create message buffer.

**REFERENCE** Page 11-2

**C FUNCTION**

```
long int crmsb(key,attr)
long intkey,attr;
```

**PARAMETERS**

key = external name of buffer.

attr = gls + maxmsg

gls global/local specifier (MSBGBL, MSGLC0, ... ,MSGLCF)

maxmsg maximum number of messages to be stored (1 to 8191)

**RETURNS**

Identifier if buffer is successfully created or already exists.

| | |
|---|---|
| BADPRM | Invalid parameter found. |
| QUEFUL | Insufficient internal resources to create buffer. |

**EXAMPLES**

```
#define MSB0 0x4D534230
#define MSB1 0x4D534231
#define MSB2 0x4D534232

msbid0 = crmsb(MSB0,50L)/* single CPU system */;
msbid1 = crmsb(MSB1,MSBGBL + 200);
if ((msbid2 = crmsb(MSB2,MSBLC2 + 500)) = = QUEFUL) ...
```

# CREATE SEMAPHORE

**DESCRIPTION** Create (counting) semaphore.

**REFERENCE** Page 9-2

**C FUNCTION**

```
long int crsem(key)
long intkey;
```

**PARAMETERS** key = external name of semaphore.

**RETURNS** Identifier if semaphore is successfully created or already exists.

| | |
|---|---|
| BADPRM | Parameter error found. |
| QUEFUL | Insufficient internal resources to create semaphore. |

**EXAMPLES**

```
#define SF34 0x53463334

if ((s34id = crsem(SF34)) = = QUEFUL) ...
```

# CREATE TASK

**DESCRIPTION** Create task.

**REFERENCE** Page 7-3

**C FUNCTION**
```
long int crtsk(tcdptr)
struct tcd*tcdptr;
```

**PARAMETERS** tcdptr = address of task parameters (tcd).

**RETURNS** Identifier if task is successfully created.

| | |
|---|---|
| DUPTSK | Task already exists. |
| BADLNG | Bad language code found in tcd. |
| BADTIM | Bad time interval code found in tcd. |
| BADPRC | Bad processor index found in tcd. |
| QUEFUL | Insufficient internal resources to create task. |

**EXAMPLES**
```
scntid = crtsk(&scntcd);

if ((scntid & 0xFFFF0000) = = 0xFFFF0000)
 {/* tcd error found *//
.
.
.
 }
```

# DELETE CONTROLLED SHARED VARIABLES

**DESCRIPTION** Delete group of controlled shared variables.

**REFERENCE** Page 9-11

**C FUNCTION**
```
int dlcsv(csvid)
long intcsvid;
```

**PARAMETERS** csvid = group identifier.

**RETURNS**

| | |
|---|---|
| NOERR | Group successfully deleted. |
| BADPRM | csvid is invalid. |

**EXAMPLES**
```
result = dlcsv(tpgid);
```

# DELETE EVENT FLAGS

**DESCRIPTION** Delete a group of event flags.

**REFERENCE** Page 8-6

**C FUNCTION**
```
int dlefg(gid)
long intgid;
```

**PARAMETERS** gid = group identifier.

**RETURNS**

| | |
|---|---|
| NOERR | Group successfully deleted. |
| BADPRM | gid is invalid. |

**EXAMPLES**
```
result = dlefg(pmp1id);
```

# DELETE MAILBOX

**DESCRIPTION** Delete mailbox.

**REFERENCE** Page 11-12

**C FUNCTION**
```
int dlmbx(mbid)
long intmbid;
```

**PARAMETERS** mbid = mailbox identifier.

**RETURNS**

| | |
|---|---|
| NOERR | Mailbox successfully deleted. |
| BADPRM | mbid is invalid. |

**EXAMPLES**
```
result = dlmbx(mb3id);
```

# DELETE MESSAGE BUFFER

**DESCRIPTION** Delete message buffer.

**REFERENCE** Page 11-5

**C FUNCTION**
```
int dlmsb(msbid)
long intmsbid;
```

**PARAMETERS** msbid = buffer identifier.

**RETURNS**

| | |
|---|---|
| NOERR | Buffer successfully deleted. |
| BADPRM | msbid is invalid. |

**EXAMPLES**
```
result = dlmsb(msbid1);
```

# DELETE SEMAPHORE

**DESCRIPTION** Delete semaphore.

**REFERENCE** Page 9-5

**C FUNCTION**

```
int dlsem(sid)
long intsid;
```

**PARAMETERS** sid = semaphore identifier.

**RETURNS**

| | |
|---|---|
| NOERR | Semaphore successfully deleted. |
| BADPRM | sid is invalid. |

**EXAMPLES**

```
result = dlsem(s34id);
```

# DELETE TASK

**DESCRIPTION** Delete requesting task.

**REFERENCE** Page 7-12

**C FUNCTION**
```
int dltsk(retarg)
long intretarg;
```

**PARAMETERS** retarg = argument (value) to be returned to task that started task being deleted.

**RETURNS** dltsk does not return.

**EXAMPLES**
```
dltsk(0L);

dltsk(result);
```

# GET ADDRESS OF DATA SEGMENTS

**DESCRIPTION** Get address of initialized and uninitialized data segments of requesting task.

**REFERENCE** Page 7-8

**C FUNCTION**
```
long int getdad(buf)
long int*buf;
```

**PARAMETERS** buf = address of buffer to receive segment addresses.

**RETURNS**

| | |
|---|---|
| NOERR | Data successfully copied. |
| BADPRM | Unable to write into buf. |

EXAMPLES
```
long int *buf[2];

getdad(buf);
```

# GET SYSTEM IDENTIFICATION

**DESCRIPTION** Get MTOS-UX identification data.

**REFERENCE** Page 3-4

**C FUNCTION**

```
int getidn(idnbuf)
char*idnbuf;
```

**PARAMETERS** idnbuf = address of 33-byte buffer to receive null-terminated string of form:

"\r\nMTOS-UX/68K MP V1.3 [200586]\r\n"

**RETURNS**

| | |
|---|---|
| NOERR | String successfully copied. |
| BADPRM | Unable to write into idnbuf. |

**EXAMPLES**

```
char mtosid[33];

getidn(mtosid);
```

# GET SYSTEM TIME

**DESCRIPTION** Get number of ms since system was started.

**REFERENCE** Page 5-3

**C FUNCTION**

```
int getime(msbfr)
char*msbfr;
```

**PARAMETERS** msbfr = address of 6-byte buffer to receive time value.

**RETURNS**

| | |
|---|---|
| NOERR | Value successfully copied. |
| BADPRM | Unable to write into msbfr (on clock master processor). |

**EXAMPLES**

```
char mstime[6];

getime(mstime);
```

# GET TASK KEY

**DESCRIPTION** Get key of given task.

**REFERENCE** Page 3-3

**C FUNCTION**
```
long int getkey(tid)
long inttid;
```

**RETURNS** key (external name) if task exists.

| | |
|---|---|
| BADPRM | Task does not exist. |

**EXAMPLES**
```
t3key = getkey(t3id);
```

# GET IDENTIFIER OF MESSAGE BUFFER

**DESCRIPTION** Get identifier of message buffer.

REFERENCE Page 11-3

**C FUNCTION**

```
long int getmsb(key)
long intkey;
```

**PARAMETERS** key = external name of message buffer.

**RETURNS** Identifier if buffer exists.

BADPRM Buffer does not exist.

**EXAMPLES**

```
#define MSB2 0x4D534232

msbid2 = getmsb(MSB2);
```

# GET MESSAGE FROM BUFFER WITHOUT WAIT

**DESCRIPTION** Get 4-byte message from buffer. Return immediately, if message is not available.

**REFERENCE** Page 11-4

**C FUNCTION**

```
int getmsn(msbid,dstadr)
long intmsbid,*dstadr;
```

**PARAMETERS** msbid = buffer identifier.

dstadr = address of 4-byte area to receive message.

**RETURNS**

| | |
|---|---|
| NOERR | Message successfully copied. |
| MBEOF | No message available. |
| BADPRM | Parameter error found. |

**EXAMPLES**

```
if (getmsn(msbid1,&nework) = = MBEOF) ...
```

# GET MESSAGE FROM BUFFER WITH WAIT

**DESCRIPTION** Get 4-byte message from buffer. Wait if message is not immediately available.

**REFERENCE** Page 11-4

**C FUNCTION**
```
int getmsw(msbid,dstadr)
long intmsbid,*dstadr;
```

**PARAMETERS** msbid = buffer identifier.

dstadr = address of 4-byte area to receive message.

**RETURNS**

| | |
|---|---|
| NOERR | Message successfully copied. |
| BADPRM | Buffer does not exist. |

**EXAMPLES**
```
getmsw(msbid1,&nework);
```

# GET SIGNAL RESPONSE

**DESCRIPTION** Get response to given signal.

**REFERENCE** Page 10-3

**C FUNCTION**
```
(*getsig()) (sig)
long intsig;
```

**PARAMETERS** sig = signal number (0 to 31)

**RETURNS** address of response procedure, or

SIGBLK (if task is to become blocked)

SIGIGNif (task is to ignore signal)

SIGTRMif (task is to terminate)

**EXAMPLES**
```
sigres = getsig(18L);
```

# GET TASK IDENTIFIER

**DESCRIPTION** Get identifier of task with given key.

**REFERENCE** Page 7-5

**C FUNCTION**
```
long int gettid(key)
long intkey;
```

**PARAMETERS** key = external name of task.

**RETURNS** Identifier of task if it exists.

BADPRM Task does not exist.

**EXAMPLES**
```
#define SCN2 0x53434E32

sc2id = gettid(SCN2);
```

# GET TIME OF DAY

**DESCRIPTION** Get time of day clock/calendar string.

**REFERENCE** Page 5-2

**C FUNCTION**

```
int gettod(todbfr)
char *todbfr;
```

**PARAMETERS** *todbfr* is the address of a 21-byte area to receive the TOD string. See settod for format of string.)

**RETURNS**

| | |
|---|---|
| NOERR | String successfully copied. |
| BADPRM | Unable to copy string. |

**EXAMPLES**

```
char ccstg[21];

gettod(ccstg);
```

# GET UNIT IDENTIFIER

**DESCRIPTION** Get identifier of unit with given key.

**C FUNCTION**
```
long int getuid(key)
long int key;
```

**PARAMETERS** *key* is the external name of the unit.

**RETURNS** Identifier of unit if it exists.

BADPRM Unit does not exist.

**EXAMPLES**
```
#define SYSC 0x53595343

scnuid = getuid(SYSC);
```

# OPEN MAILBOX

**DESCRIPTION** Open mailbox, creating it if it does not already exist.

**REFERENCE** Page 11-8

**C FUNCTION**
```
long int opnmbx(key,mode)
long int key,mode;
```

**PARAMETERS** *key* is the external name of the mailbox

*mode* = MBRRCV (for receiving)
MBRSND (for sending)

**RETURNS** Identifier if mailbox is successfully created or already exists.

BADPRM if *mode* is incorrect.

QUEFUL if insufficient internal resources to process request.

**EXAMPLES**
```
#define MB03 0x4D423033

if ((mb3id = opnmbx(MB03,MBRCV)) = = QUEFUL)...
```

# PAUSE

**DESCRIPTION** Pause for given time interval.

**REFERENCE** Page 4-1

**C FUNCTION**

```
int pause(interval)
long int interval;
```

**PARAMETERS**

interval = iunits + inum

iunits:time units (MS, TMS, HMS, SEC, MIN, HRS, DAY)

inum: number of such units (0 to 255)

interval can also be NOEND for "pause until cancelled" or NXTICK for "pause until next clock tick".

**RETURNS**

| | |
|---|---|
| NOERR | Specified interval ran to completion. |
| TIMCAN | Pause cancelled by canpau. |
| BADPRM | iunits field not correct. |
| QUEFUL | Pause could not be performed for lack of internal resourses. |

**EXAMPLES**

```
result = pause(250 + MS);

result = pause(25 + TMS);

result = pause(SEC + 1);

pause(NOEND);
```

# PAUSE UNTIL SIGNAL ARRIVES

**DESCRIPTION** Pause until signal arrives.

**REFERENCE** Page 10-5

**C FUNCTION**

```
int pausig(interval)
long int interval;
```

**PARAMETERS** interval = iunits + inum

iunits:time units (MS, TMS, HMS, SEC, MIN, HRS,DAY)

inum: number of such units (0 to 255)

interval can also be NOEND for "pause until cancelled".

**RETURNS** signal number (0 to 31).

| | |
|---|---|
| BADPRM | iunits field not correct. |
| TIMOUT | Signal did not arrive within given interval. |
| QUEFUL | Pause could not be performed for lack of internal resourses. |

**EXAMPLES**

```
result = pausig(250 + MS);
signum = pausig(NOEND);
```

# POST MESSAGE TO BEGINNING OF BUFFER

**DESCRIPTION** Post 4-byte message to the beginning of buffer.

**REFERENCE** Page 11-4

**C FUNCTION**

```
int putmsb(msbid,msg)
long int  msbid,msg;
```

**PARAMETERS** msbid = buffer identifier.

msg = 4-byte message.

**RETURNS**

| | |
|---|---|
| NOERR | Message transfered to waiting task or stored in buffer. |
| QUEFUL | No task waiting and no room left in buffer. |
| BADPRM | Buffer does not exist. |

**EXAMPLES**

```
result = putmsb(msb03,0x1234L);
result = putmsb(msbid1,newmsg);
```

# POST MESSAGE TO END OF BUFFER

**DESCRIPTION** Post 4-byte message to the end of buffer.

**REFERENCE** Page 11-4

**C FUNCTION**

```
int putmse(msbid,msg)
long int  msbid,msg;
```

**PARAMETERS** msbid = buffer identifier.

msg = 4-byte message.

**RETURNS**

| | |
|---|---|
| NOERR | Message transfered to waiting task or stored in buffer. |
| QUEFUL | No task waiting and no room left in buffer. |
| BADPRM | Buffer does not exist. |

**EXAMPLES**

```
result = putmse(msbid1,0x5678L);
result = putmse(msbid1,nextmsg);
```

# RECEIVE MESSAGE FROM MAILBOX

**DESCRIPTION** Receive first available message from mailbox.

**REFERENCE** Page 11-10

**C FUNCTION**

```
int rcvmbx(mbid,dstadr,stabfr,qual)
long int  mbid,stabfr,qual;
char *dstadr;
```

**PARAMETERS**

mbid = mailbox identifier.

dstadr = address of area to receive message.

stabfr = address of long word buffer to receive status of request.

qual = cmode + lunits + lnum

cmode: coordination mode (WAIFIN, CLEFn, CSIGn; n = 0 to 15)

lunits: time units for wait limit (MS, TMS, HMS, SEC, MIN, HRS, DAY)

lnum: number of such units (0 to 255)

lunits + lnum may be replaced by IMONLY (to return TIMOUT if message not available immediately).

if lunits + lnum is zero then there is no wait limit.

**RETURNS**

| | |
|---|---|
| NOERR | Request successfully queued or completed. |
| BADPRM | Parameter error found. |
| MBEOF | Mailbox not open at send end. |
| QUEFUL | Insufficient internal resources to process request. |
| TIMOUT | Message did not become available within maximum specified interval. |

**EXAMPLES**

```
rcvmbx(mb3id,nxtmsg,&result,WAIFIN + 2 + SEC);

rcvmbx(mb3id,newmsg,&result,CLEF1);
```

# RELEASE CONTROLLED SHARED VARIABLES

**DESCRIPTION** Release group of controlled shared variables.

**REFERENCE** Page 9-9

**C FUNCTION**
```
int rlscsv(csvid)
long int  csvid;
```

**PARAMETERS** csvid = group identifier.

**RETURNS**

| | |
|---|---|
| NOERR | Group has been released. |
| BADPRM | Group does not exist or is not reserved to requesting task. |

**EXAMPLES**
```
result = rlscsv(tpgid);
```

# RELEASE SEMAPHORE

**DESCRIPTION** Release semaphore.

**REFERENCE** Page 9-5

**C FUNCTION**

```
int rlssem(sfid)
long int  sfid;
```

**PARAMETERS** sfid = semaphore identifier.

**RETURNS**

| | |
|---|---|
| NOERR | Semaphore in-use tally has been decremented, and if now 0, semaphore has been released. |
| BADPRM | Semaphore does not exist or is not reserved to requesting task. |

**EXAMPLES**

```
result = rlssem(s34id);
```

# SET TASK PRIORITY

**DESCRIPTION** Set current priority of given task.

**REFERENCE** Page 7-9

**C FUNCTION**

```
unsigned short int setpty(tid,basis,value)
long int  tid, basis, value;
```

**PARAMETERS** tid = task identifier (0 = requesting task).

basis = USEVAL (use given value)
ADDVAL (add given value).

value = value to be used or added.

**RETURNS** new value of priority (0 to 255).
BADPRM Parameter error found.

**EXAMPLES**

```
result = setpty(tsk12,USEVAL,120L);

curpty = setpty(0L,ADDVAL,0L);
```

# SET RESPONSE TO SIGNAL

**DESCRIPTION** Set response to one or more signals.

**REFERENCE** Page 10-2

**C FUNCTION**

```
int setsig(sigmsk,resp)
long intsigmsk;
int (*resp) ();
```

**PARAMETERS** sigmsk = mask to select signals can form from SIG0 to SIG31 or SIGALL.

resp = address of response subprogram
SIGIGN (ignore signal)
SIGBLK (become blocked)
SIGTRM (terminate)
SIGDFL (reinstate default)

**RETURNS**

| | |
|---|---|
| NOERR | Response successfully set. |
| BADPRM | Parameter error found. |

**EXAMPLES**

```
setsig(SIGALL,SIGDFL);

setsig(SIG16,&dbgtrc);
```

# SET STANDARD CONSOLE

**DESCRIPTION** Install given unit as standard console of requesting a task.

**C FUNCTION**

```
int setstc(uid)
long int  uid;
```

**PARAMETERS** uid = unit identifier.

**RETURNS**

| | |
|---|---|
| NOERR | Standard console successfully set. |
| BADPRM | Unit does not exist. |

**EXAMPLES**

```
#define SYSC 0x53595343

setstc(getuid(SYSC));
```

# SET TIME OF DAY

**DESCRIPTION** Set time of day clock/calendar.

**REFERENCE** Page 5-1

**C FUNCTION**

```
int settod(todstg)
char*todstg;
```

**PARAMETERS** todstg = address of null-terminated TOD string of the form

"DD MMM YYYY HH:MM:SS"

DD = day of month, starting at 01
MMM = abbreviated month name (JAN, FEB, ... ,DEC)
YYYY = year
HH = hour (00 to 23)
MM = minute (00 to 59)
SS = second (00 to 59)

**RETURNS**

| | |
|---|---|
| NOERR | No errors found in string. |
| BADPRM | Format of string is not valid. |

**EXAMPLES**

```
settod("11 NOV 1918 11:00:00");

result = settod(date);
```

# SET EVENT FLAGS AFTER GIVEN INTERVAL

**DESCRIPTION** Set event flags after given interval of time.

**REFERENCE** Page 8-5

**C FUNCTION**

```
long int sgiefg(gid,mask,interval)
long intgid,mask,interval;
```

**PARAMETERS**

gid = identifier of group.

mask = mask to select flags can form from EF0 to EF31 or EFALL.

interval = iunits + inum

iunit: (MS, TMS, HMS, SEC, MIN, HRS, DAY)

inum: number of such units (0 to 255)

**RETURNS**

| | |
|---|---|
| NOERR | No parameter errors found. |
| TIMCAN | Previous timer was reset or cancelled. |
| BADPRM | Parameter error found. |
| QUEFUL | Request could not be performed for lack of internal resourses. |

**EXAMPLES**

```
result = sgiefg(pmp1id,EF0 + EF6,100 + SEC);
```

# SEND SIGNAL AFTER GIVEN INTERVAL

**DESCRIPTION** Send specified signal after given interval of time.

**REFERENCE** Page 10-5

**C FUNCTION**

```
int sgisig(sig,interval)
int  sig,interval;
```

**PARAMETERS**

sig = signal number (0 to 15 or 31).

interval = iunits + inum

iunits: time units (MS, TMS, HMS, SEC, MIN, HRS, DAY)

inum: number of such units (0 to 255)

**RETURNS**

| | |
|---|---|
| NOERR | No parameter errors found. |
| TIMCAN | Previous timer was reset or cancelled. |
| BADPRM | Parameter error found. |
| QUEFUL | Request could not be performed for lack of internal resourses. |

**EXAMPLES**

```
result = sgisig(3L,2 + MIN);
```

# SEND MESSAGE TO MAILBOX

**DESCRIPTION** Send message to mailbox.

**REFERENCE** Page 11-9

**C FUNCTION**

```
int sndmbx(mbid,srcadr,prty,stabfr,qual)
long intmbid,prty,stabfr,qual;
char*srcadr;
```

**PARAMETERS**

mbid = mailbox identifier.

srcadr = address of message (with length in first 4 bytes).

prty = message priority

stabfr = address of long word buffer to receive status of request.

qual = cmode + lunits + lnum

cmode coordination mode (CTUNOC, WAIFIN, CLEFn,

lunits: time units for wait limit (MS, TMS, HMS, SEC,MIN, HRS, DAY)

lnum: number of such units (0 to 255)

lunits + lnum may be replaced by IMONLY (to return TIMOUT if message not available immediately).

if lunits + lnum is zero then there is no wait limit.

**RETURNS**

| | |
|---|---|
| NOERR | Request successfully queued or completed. |
| BADPRM | Parameter error found. |
| QUEFUL | No receiver available and no internal resources available to queue request. |
| TIMOUT | Receiver did not become available within maximum specified interval. |

**EXAMPLES**

```
sndmbx(mb0id,newmsg,0,&result,WAIFIN + 2 + SEC);
```

# SEND SIGNAL

**DESCRIPTION** Send signal to one task or group of tasks.

**REFERENCE** Page 10-4

**C FUNCTION**

```
int sndsig(tid,sig)
long int  tid,sig;
```

**PARAMETERS**

tid = identifier of task to receive signal
- 0 to send signal to all other application tasks
- -1 to send signal to any other tasks which are sharing code or data.

sid = signal number (0 to 15, or 31).

**RETURNS**

| | |
|---|---|
| NOERR | Signal successfully sent. |
| BADPRM | Parameter error found. |

**EXAMPLES**

```
sndsig(-1L,31L);

sndsig(tsk4id,15L);
```

# SET/RESET EVENT FLAGS

**DESCRIPTION** Immediately set or reset event flags.

**REFERENCE** Page 8-3

**C FUNCTION**

```
long int srsefg(gid,opmask)
intgid,opmask;
```

**PARAMETERS** gid = group identifier (0 = local group of requesting task).

opmask = op + mask

op: EFSET (set) or EFRST (reset)

mask: flag selection composed from EF0 to EF15 or EFALL

**RETURNS** final value of group (with high-order bits cleared).

BADPRM Parameter error found.

**EXAMPLES**

```
srsefg(0L,EFRST + EFALL);
value = srsefg(0L,EFRST + EF1 + EF5)
curlef = srsefg(0L,EFSET + 0L);
value = srsefg(pmp1id,EFSET + EF8);
```

# SET/RESET LOCAL EVENT FLAGS OF GIVEN TASK

**DESCRIPTION** Immediately set or reset local event flags of given task.

**REFERENCE** Page 8-7

**C FUNCTION**

```
long int srslef(tid,opmask)
long int  tid,opmask;
```

**PARAMETERS** tid = task identifier (0 = requesting task).

opmask = op + mask:

op: EFSET (set) or EFRST (reset)

mask: flag selection composed from EF0 to EF15 or EFALL.

**RETURNS** final value of group (with high-order bits cleared).

BADPRM Parameter error found.

**EXAMPLES**

```
srslef(0L,EFRST + EFALL);
value = srslef(scntsk,EFSET + EF1 + EF5)
curlef = srslef(0L,EFSET + 0L);
```

# START TASK

**DESCRIPTION** Start given task.

**REFERENCE** Page 7-5

**C FUNCTION**

```
int start(tid,pty,arg,stabfr,qual)
long int  tid,pty,arg,*stabfr,qual;
```

**PARAMETERS**

tid = task identifier.

pty = pbasis + pvalue

pbasis: basis of computing priority at which task starts:

INHPTY (use inherent priority of target task)
CURPTY (use current priority of requesting task)
LRGPTY (use larger of inherent priority of requesting task & current priority of requesting task)
GVNPTY (use priority given in pvalue).

arg = argument to be presented to task when it starts.

stabfr = address of long word buffer to receive status of request.

qual = cbasis + cmode + lunits + lnum

cbasis (coordination basis):

CSTART (end request when target task starts)
CTERM (end request when target task terminates).

cmode: coordination mode (CTUNOC, WAIFIN, CLEFn, CSIGn; n = 0 to 15)

lunits:time units for wait limit (MS, TMS, HMS, SEC, MIN, HRS, DAY)

lnum: number of such units (0 to 255)

lunits + lnum may be replaced by IMONLY (to return TIMOUT if task not currently dormant).

if lunits + lnum is zero then there is no wait limit.

**RETURNS**

| | |
|---|---|
| NOERR | Request successfully queued or completed. |
| BADPRM | Parameter error found. |
| QUEFUL | Insufficient internal resources to process request. |
| TIMOUT | Task could not be started within maximum specified interval. |

**EXAMPLES**

```
start(scntid,GVNPTY + 100,&data,&stabuf,
 WAIFIN + 2 + SEC + CTERM);
```

# SYNCHRONIZE WITH TIME OF DAY

**DESCRIPTION** Wait for given time of day.

**REFERENCE** Page 5-3

**C FUNCTION**

```
int syntod(synstg)
char *synstg;
```

**PARAMETERS** synstg = address of null-terminated match string of the"HHMMSS"

HH = hour (00 to 23, or ?? to match any hour)
MM = minute (00 to 59, or ?? to match any minute)
SS = second (00 to 59, or ?? to match any second)

**RETURNS**

| | |
|---|---|
| NOERR | No errors detected in string. |
| BADPRM | Format of string is not valid. |

**EXAMPLES**

```
result = syntod("??1500");

syntod(nxthr);
```

## TERMINATE TASK WITH RESTART AFTER GIVEN INTERVAL

**DESCRIPTION** Terminate requesting task with automatic restart after given interval of time.

**REFERENCE** Page 7-11

**C FUNCTION**
```
long int trmrst(retarg,intrvl)
long int  retarg,intrvl;
```

**PARAMETERS** retarg = argument (value) to be returned to task that started task being terminated.

intrvl = rbasis + iunits + inum

rbasis: STRTIM (add interval to last start time)
TRMTIM (add interval to termination time).

iunits: time units (MS, TMS, HMS, SEC, MIN, HRS, DAY)

inum: number of such units (0 to 255)

interval can also be NXTICK for "until next clock tick".

**RETURNS** trmrst does not return.

**EXAMPLES**
```
trmrst(0L,STRTIM + 1 + HOUR);
```

# TRANSFER START OF TASK

**DESCRIPTION** Start given task and transfer coordination to new task.

**C FUNCTION**
```
int tstart(tid)
long int  tid;
```

**PARAMETERS** tid = task identifier.

**RETURNS** For success, tstart does not return.

BADPRM tid is invalid.

**EXAMPLES**
```
tstart(fstask);
```

# USE CONTROLLED SHARED VARIABLES

**DESCRIPTION** Wait for exclusive control over group of controlled shared variables.

**REFERENCE** Page 9-6

**C FUNCTION**

```
int usecsv(csvid,interval)
long intcsvid,interval;
```

**PARAMETERS**

csvid = group identifier.

interval = iunits + inum

iunits: time units (MS, TMS, HMS, SEC, MIN, HRS, DAY)

inum: number of such units (0 to 255)

interval can also be NOEND for "forever".

**RETURNS**

| | |
|---|---|
| NOERR | Group is available. |
| DUPTSK | Duplicate task request (task already has group). |
| BADPRM | Parameter error found. |
| QUEFUL | Service could not be performed for lack of internal resources. |
| TIMOUT | Group did not become available within given interval. |

**EXAMPLES**

```
result = rlscsv(tpgid,10 + SEC);

rlscsv(tpgid,NOEND);
```

# WAIT FOR EVENT FLAGS

**DESCRIPTION** Wait until event flags are set.

**REFERENCE** Page 8-4

**C FUNCTION**

```
long int waiefg(gid,opmask,interval)
intgid,opmask,interval;
```

**PARAMETERS** gid = group identifier (0 = local group of requesting task).

opmask = op + mask:

op EFAND (wait for all bits to be set)
EFOR (wait for any bits to be set)

mask = mask to select flags can form from EF0 to EF31 or EFALL.

interval = iunits + inum

iunits: time units (MS, TMS, HMS, SEC, MIN, HRS, DAY)

inum: number of such units (0 to 255)

**RETURNS** final value of group (with high-order bits cleared).

| | |
|---|---|
| BADPRM | Parameter error found. |
| QUEFUL | Service could not be performed for lack of internal resources. |
| TIMOUT | Group did not become available within given interval. |

**EXAMPLES**

```
pmival = waiefg(pmlgid,EFAND + EF2 + EF5,20 + SEC);

waiefg(pmp1id,EF1)/* can use default for 1 flag */;

curlef = waiefg(0L,0L);
```

## WAIT FOR FUNCTION OF CONTROLLED SHARED VARIABLES TO BE TRUE

**DESCRIPTION**

Wait for function of controlled shared variables to be true and then continue with exclusive use of the CSV.

**REFERENCE**

Page 9-9

**C FUNCTION**

```
int waicsv(csvid,bfun,interval)
long intcsvid,interval;
 int (*bfun) ();
```

**PARAMETERS**

csvid = group identifier.

bfun = function that returns TRUE (non-zero) only when wait is to end.

interval = iunits + inum

iunits: time units (MS, TMS, HMS, SEC, MIN, HRS, DAY)

inum: number of such units (0 to 255)

**RETURNS**

| | |
|---|---|
| NOERR | Function was true, or became true within specified interval. |
| BADPRM | Group does not exist. |
| TIMOUT | Function did not become true within given interval. |
| QUEFUL | Service could not be performed for lack of internal resourses. |

**EXAMPLES**

```
int tstfn() /* declare test function */;

result = waicsv(tpgid,&tstfn,200 + MS);

tstfn(csvars)
struct meas*csvars;
        {
            if (csvars->pres[20] < 200)
                return(0)                   /* keep waiting */;
            else
                return(1)                   /* end wait */;
        }
```

# WAIT FOR SEMAPHORE

**DESCRIPTION** Wait for given (counting) semaphore to be free.

**REFERENCE** Page 9-2

**C FUNCTION**

```
int waisem(sid,stabfr,qual)
long intsid,*stabfr,qual;
```

**PARAMETERS**

sid = semaphore identifier.

stabfr = address of long word buffer to receive status of request.

qual = cmode + lunits + lnum

cmode coordination mode (WAIFIN, CLEFn, CSIGn; n = 0 to 15)

lunits: time units for wait limit (MS, TMS, HMS,SEC, MIN, HRS, DAY)

lnum: number of such units (0 to 255)

lunits + lnum may be replaced by IMONLY (to return

TIMOUT if semaphore not available immediately).

if lunits + lnum is zero then there is no wait limit.

**RETURNS**

| | |
|---|---|
| NOERR | Request successfully queued or completed. |
| BADPRM | Parameter error found. |
| QUEFUL | Insufficient internal resources to process request. |
| TIMOUT | Semaphore did not become available within maximum specified interval. |

**EXAMPLES**

```
waisem(s34id,&result,CLEF5 + 200 + MS);
```

# APPENDIX B:

# ERROR CODES

# APPENDIX B:
# ERROR CODES FROM DIAGNOSTIC PROGRAM

| File: cmptst.c | Code | Meaning |
|---|---|---|
| | 1001 | crcmp with block size > pool did not return BADPRM |
| | 1002 | BADPRM on crcmp did not send Signal 26 |
| | 1003 | crcmp without room for 2 blocks did not return BADPRM |
| | 1004 | BADPRM on crcmp did not send Signal 26 |
| | 1005 | crcmp with block size 0 did not return BADPRM |
| | 1006 | BADPRM on crcmp did not send Signal 26 |
| | 1007 | crcmp with block size 2**64 did not return BADPRM |
| | 1008 | BADPRM on crcmp did not send Signal 26 |
| | 1009 | crcmp with nonexistant memory did not return BADPRM |
| | 1010 | BADPRM on crcmp did not send Signal 26 |
| | 1011 | crcmp 1 with valid parameters returned an error |
| | 1012 | crcmp 2 with valid parameters returned an error |
| | 1013 | crcmp with different keys returned same identifier |
| | 1014 | getcmp with invalid key did not return BADPRM |
| | 1015 | BADPRM on getcmp did not send Signal 26 |
| | 1016 | getcmp got different identifier than crcmp |
| | 1017 | getcmp got different identifier than crcmp |
| | 1018 | alloc with number of bytes > size of pool did not return BADPRM |
| | 1019 | BADPRM on alloc did not send Signal 26 |
| | 1020 | alloc with invalid identifier did not return BADPRM |
| | 1021 | BADPRM on alloc did not send Signal 26 |
| | 1022 | alloc returned incorrect value |
| | 1023 | alloc did not return TIMOUT when unsatisfied |
| | 1024 | dalloc with invalid identifier did not return BADPRM |
| | 1025 | BADPRM on dalloc did not send Signal 26 |
| | 1026 | dalloc of bit map block did not return BADPRM |
| | 1027 | BADPRM on dalloc did not send Signal 26 |
| | 1028 | dlcmp with valid identifier did not return NOERR |
| | 1029 | dalloc of deleted pool did not return BADPRM |
| | 1030 | BADPRM on dlcmp did not send Signal 26 |
| | 1031 | dlcmp with global TPA did not return BADPRM |
| | 1032 | BADPRM on dlcmp did not send Signal 26 |
| | 1033 | dlcmp with invalid identifier did not return BADPRM |
| | 1034 | BADPRM on dlcmp did not send Signal 26 |
| | 1035 | dlcmp with valid identifier did not return NOERR |

| File: csvtst.c | Code | Meaning |
|---|---|---|
| | 1101 | QUEFUL reported by crcsv |
| | 1102 | Duplicate CSV creates reported different values |
| | 1103 | Different CSV creates reported same value |
| | 1104 | CSV3 create caused error |
| | 1105 | Create with different lengths did not return BADPRM |
| | 1106 | BADPRM on crcsv did not send Signal 26 |
| | 1107 | dlcsv (id1) did not report NOERR |
| | 1108 | dlcsv (id2) did not report NOERR |
| | 1109 | dlcsv (id3) did not report NOERR |
| | 1110 | CSV create with odd length returns error |
| | 1111 | CSV create with length one returns error |
| | 1112 | CSV create with length 0 returns value |
| | 1113 | BADPRM on crcsv length 0 did not send Signal 26 |
| | 1114 | CSV create with length FF0000 returns value |
| | 1115 | dlcsv (id1) did not report NOERR |
| | 1116 | dlcsv (id2) did not report NOERR |
| | 1117 | dlcsv(100L) did not report BADPRM |
| | 1118 | BADPRM on dlcsv(100L) did not send Signal 26 |
| | 1119 | CSV1 create caused error |
| | 1120 | VTST1 was not created properly |
| | 1121 | Different tasks returned different identifier |
| | 1122 | usecsv(100L) did not report BADPRM |
| | 1123 | BADPRM on usecsv did not send Signal 26 |
| | 1124 | usecsv(id1) did not report NOERR |
| | 1125 | usecsv(id1) again did not report DUPTSK |
| | 1126 | rlscsv(100L) did not report BADPRM |
| | 1127 | BADPRM on rlscsv(100L) did not send Signal 26 |
| | 1128 | rlscsv(id1) did not report NOERR |
| | 1129 | VTST1 was not created properly |
| | 1130 | usecsv(id1) did not report TIMOUT |
| | 1131 | rlscsv used by other task did not return BADPRM |
| | 1132 | BADPRM on rlscsv other task did not send Signal 26 |
| | 1133 | waicsv with bad identifier did not report BADPRM |
| | 1134 | BADPRM on waicsv did not send Signal 26 |
| | 1135 | waicsv with false function did not TIMOUT |
| | 1136 | waicsv with true function did not NOERR |
| | 1137 | rlscsv(id1) did not report NOERR |
| | 1138 | dlcsv (id1) did not report NOERR |
| | 1139 | CSEF Event Flag Create caused error |
| | 1140 | VTST3 was not created properly |
| | 1141 | VTST4 was not created properly |
| | 1142 | VTST5 was not created properly |
| | 1143 | The three tasks did not function correctly |
| | 1144 | CSV create of varsid returns error |
| | 1145 | dlcsv (varsid) did not report NOERR |
| | 1146 | dlefg (csefid) did not report NOERR |
| | 1147 | CSV1 create caused error |
| | 1148 | usecsv(idtest) did not report NOERR |
| | 1149 | rlscsv(idtest) did not report NOERR |
| | 1150 | CSV create of varsid returns error |

| | | |
|---|---|---|
| | 1150 | rlscsv(varsid) did not report NOERR |
| | 1151 | CSV create of varsid returns error |
| | 1152 | CSV create of varsid returns error |
| | 1153 | rlscsv(varsid) did not report NOERR |

| File: efgtst.c | Code | Meaning |
|---|---|---|
| | 1201 | QUEFUL reported by crefg |
| | 1202 | Duplicate EFG creates reported different values |
| | 1203 | Different EFG creates reported same value |
| | 1204 | EFG3 create caused QUEFUL |
| | 1205 | dlefg did not report NOERR |
| | 1206 | waiefg did not report error on bad identifier |
| | 1207 | srsefg returned an error |
| | 1208 | waiefg after timeout did not return TIMOUT |
| | 1209 | ETST1 was not created properly |
| | 1210 | Setting global EF did not restart waiting task |
| | 1211 | Create failed |
| | 1212 | dlefg did not report NOERR |
| | 1213 | dlefg did not report NOERR |
| | 1214 | waiefg after dlefg returned an error |
| | 1215 | dlefg on deleted EF did not report BADPRM |
| | 1216 | QUEFUL reported by crefg |
| | 1217 | Error in srsefg setting mask |
| | 1218 | Error in srsefg resetting mask |
| | 1219 | dlefg reported BADPRM |
| | 1220 | QUEFUL reported by crefg |
| | 1221 | Error in srsefg setting mask |
| | 1222 | Error in srsefg resetting mask |
| | 1223 | dlefg reported BADPRM |
| | 1224 | QUEFUL reported by crefg |
| | 1225 | Error in srsefg setting mask |
| | 1226 | dlefg reported BADPRM |
| | 1227 | QUEFUL reported by crefg |
| | 1228 | Error in srsefg resetting mask |
| | 1229 | dlefg reported BADPRM |
| | 1230 | Error in srsefg setting mask |
| | 1231 | Error in srsefg resetting mask |
| | 1232 | dlefg reported BADPRM |
| | 1233 | ETST2 was not created properly |
| | 1234 | Setting local EF did not restart waiting task |
| | 1235 | ETST3 was not created properly |
| | 1236 | Resetting the local EF of another task did not work |

| File: fixtst.c | Code | Meaning |
|---|---|---|
| | 1301 | crfbp with block size 0 did not return BADPRM |
| | 1302 | BADPRM on crfbp did not send Signal 26 |
| | 1303 | crfbp with nonexistant memory did not return BADPRM |
| | 1304 | BADPRM on crfbp did not send Signal 26 |
| | 1305 | crfbp 1 with valid parameters returned error |
| | 1306 | crfbp 2 with valid parameters returned error |
| | 1307 | crfbp with different keys returned same identifier |
| | 1308 | getfbp with invalid key did not return BADPRM |
| | 1309 | getfbp with invalid key did not send Signal 26 |
| | 1310 | getfbp got different identifier than crfbp |
| | 1311 | getfbp got different identifier than crfbp |
| | 1312 | alofbp with TPA specified did not return BADPRM |
| | 1313 | BADPRM on alofbp did not send Signal 26 |
| | 1314 | alofbp with invalid identifier did not return BADPRM |
| | 1315 | BADPRM on alofbp did not send Signal 26 |
| | 1316 | alofbp returned incorrect value |
| | 1317 | dalfbp with invalid identifier did not return BADPRM |
| | 1318 | BADPRM on dalfbp did not send Signal 26 |
| | 1319 | dalfbp of TPA did not return BADPRM |
| | 1320 | BADPRM on dalfbp did not send Signal 26 |
| | 1321 | dlfbp with valid identifier did not return NOERR |
| | 1322 | dalfbp of deleted pool did not return BADPRM |
| | 1323 | BADPRM on dlfbp did not send Signal 26 |
| | 1324 | dlfbp with invalid identifier did not return BADPRM |
| | 1325 | BADPRM on dlfbp did not send Signal 26 |
| | 1326 | dlfbp with valid identifier did not return NOERR |

| File: mbxtst.c | Code | Meaning |
|---|---|---|
| | 1401 | QUEFUL reported by crmsb |
| | 1402 | Different MSB creates reported same value |
| | 1403 | putmse with bad identifier did not return BADPRM |
| | 1404 | BADPRM on putmse did not send Signal 26 |
| | 1405 | putmsb with bad identifier did not return BADPRM |
| | 1406 | BADPRM on putmsb did not send Signal 26 |
| | 1407 | getmsw with bad identifier did not return BADPRM |
| | 1408 | BADPRM on getmsw did not send Signal 26 |
| | 1409 | getmsn with bad identifier did not return BADPRM |
| | 1410 | BADPRM on getmsn did not send Signal 26 |
| | 1411 | dlmsb with bad identifier did not return BADPRM |
| | 1412 | BADPRM on dlmsb did not send Signal 26 |
| | 1413 | putmsb 10L did not return NOERR |
| | 1414 | putmse 20L did not return NOERR |
| | 1415 | putmsb 30L did not return NOERR |
| | 1416 | getmsw did not return NOERR |
| | 1417 | message 30L was not read |
| | 1418 | getmsw did not return NOERR |
| | 1419 | message 10L was not read |

| | |
|---|---|
| 1420 | getmsw did not return NOERR |
| 1421 | message 20L was not read |
| 1422 | getmsn did not return MBEOF |
| 1423 | QUEFUL not reported on putmsb 100 messages |
| 1424 | dlmsb MSB1 did not return NOERR |
| 1425 | getmsw did not return NOERR |
| 1426 | message was not read |
| 1427 | Getting last message did not delete message buffer |
| 1428 | QUEFUL reported by crmsb (2nd time) |
| 1429 | QUEFUL not reported after 100 messages queued |
| 1430 | dlmsb MSB1 did not return NOERR |
| 1431 | getmsw did not return NOERR |
| 1432 | message was not read |
| 1433 | Getting last message did not delete message buffer |
| 1434 | QUEFUL reported by crmsb (3rd time) |
| 1435 | QUEFUL not reported on putmsb 100 messages |
| 1436 | dlmsb MSB1 did not return NOERR |
| 1437 | getmsn did not return NOERR |
| 1438 | message was not read |
| 1439 | Getting last message did not delete message buffer |
| 1440 | QUEFUL reported by crmsb (4th time) |
| 1441 | QUEFUL not reported after 100 messages queued |
| 1442 | dlmsb MSB1 did not return NOERR |
| 1443 | getmsn did not return NOERR |
| 1444 | Message was not read |
| 1445 | Getting last message did not delete |
| 1446 | dlmsb MSB2 did not return NOERR |
| 1447 | QUEFUL reported by crmsb (5th time) |
| 1448 | QUEFUL not expected |
| 1449 | getmsn did not return NOERR |
| 1450 | QUEFUL not expected |
| 1450 | getmsn did not return NOERR |
| 1451 | Not enough TPA to run this test |
| 1452 | QUEFUL reported by crmsb (6th time) |
| 1453 | dlmsb MSB1 did not return NOERR |
| 1454 | QUEFUL reported by opnmbx |
| 1455 | Different MB creates reported same value |
| 1456 | QUEFUL reported by opnmbx MB03 |
| 1457 | Bad transfer of message 2 |
| 1458 | Bad transfer of message 1 |
| 1459 | MBEOF did not set EF |
| 1460 | MBDLT did not set EF |
| 1461 | QUEFUL reported by opnmbx MB04 send |
| 1462 | Mailbox not sent correctly |
| 1463 | Mailbox not sent correctly |
| 1464 | Mailbox not sent correctly |
| 1465 | Mailbox not sent correctly |
| 1466 | Mailbox not sent correctly |
| 1467 | Message at odd address not sent correctly |
| 1468 | Message with length 0 not sent correctly |
| 1469 | missed QUEFUL |
| 1470 | missed QUEFUL |
| 1471 | QUEFUL not detected |
| 1472 | Wrong data detected |

| | |
|---|---|
| 1473 | Bad signal number |
| 1474 | Bad processor index |
| 1475 | Bad last signal number |
| 1476 | Bad last signal level |
| 1477 | Bad signal number |
| 1478 | Bad processor index |
| 1479 | Bad last signal number |
| 1480 | Bad last signal level |
| 1481 | QUEFUL reported by opnmbx MB04 send |
| 1482 | waiefg error |
| 1483 | Message incorrect |
| 1484 | waiefg error status |
| 1485 | Bad transfer of odd addr |
| 1486 | Bad transfer of length 0 |
| 1487 | Bad transfer of length 0 |
| 1488 | dlmsb did not return NOERR |

**File: pautst.c**

| Code | Meaning |
|---|---|
| 1501 | 'NOEND + 5' does not return BADPRM |
| 1502 | BADPRM on pause did not send Signal 26 |
| 1503 | canpau with bad identifier does not return BADPRM |
| 1504 | BADPRM on canpau did not send Signal 26 |
| 1505 | PTST1 was not created properly |
| 1506 | canpau did not return NOERR |
| 1507 | PTST2 was not created properly |
| 1508 | canpau did not detect the task was not paused |
| 1509 | PTST3 was not created properly |
| 1510 | canpau did not return NOERR |
| 1511 | TIMCAN was not returned on NOEND pause |
| 1512 | TIMCAN was not returned on 4 + SEC pause |

**File: piotst.c**

| Code | Meaning |
|---|---|
| 1601 | Can not read with default prompt |
| 1602 | Bad status from read with default prompt |
| 1603 | Bad tally from read with default prompt |
| 1604 | Can not read with given prompt |
| 1605 | Bad status from read with given prompt |
| 1606 | Bad tally from read with given prompt |
| 1607 | Can not write |
| 1608 | Bad status from write |
| 1609 | Bad tally from write |
| 1610 | Bad status from PIORSV |
| 1611 | Bad status from PIORSV expected RSVERR |
| 1612 | Bad status from PIORLS expected 0 |
| 1613 | Bad status from PIORLS expected RSVERR |
| 1614 | Did not get BADPRM on PIORSV + PREEMP |
| 1615 | Did not get BADPRM on PIORLS + PREEMP |
| 1616 | Can not read single character |
| 1617 | Can not write ms3prm |
| 1618 | Can not write ms4prm |

| | |
|---|---|
| 1619 | Can not write ms2prm |
| 1620 | Can not write ms1prm |
| 1621 | TIMOUT status not returned on PIORE1 |
| 1622 | Bad status on 1 PIOWRI expected NOERR |
| 1623 | Bad status on 2 PIOWRI expected NOERR |
| 1624 | Bad status on 3 PIOWRI expected NOERR |
| 1625 | Bad status on 4 PIOWRI expected NOERR |
| 1626 | Can not read single character |
| 1627 | Either PIORE1 or PIOWRI did not work |
| 1628 | Bad status on PIOWR1 expected NOERR |
| 1629 | Can not write to system printer |
| 1630 | Bad status on PIOWRI to printer |

| File: semtst.c | Code | Meaning |
|---|---|---|
| | 1701 | QUEFUL reported by crsem |
| | 1702 | Different SF creates reported the same value |
| | 1703 | waisem did not report error on bad identifier |
| | 1704 | waisem had error on good call #1 |
| | 1705 | waisem reported error on good call #2 |
| | 1706 | rlssem did not report NOTFRE |
| | 1707 | dlsem did not report NOTFRE |
| | 1708 | rlssem did not report NOERR |
| | 1709 | dlsem did not report NOERR |

| File: sigtst.c | Code | Meaning |
|---|---|---|
| | 1801 | Bad signal number |
| | 1802 | Bad last signal number |
| | 1803 | Bad last signal level |
| | 1804 | All signals did not arrive |
| | 1805 | Wrong number of signals |

| File: todtst.c | Code | Meaning |
|---|---|---|
| | 1901 | Undetected error in syntod (240000L) |
| | 1902 | Undetected error in syntod (236000L) |
| | 1903 | Undetected error in syntod (????6?) |
| | 1904 | Undetected error in settod 20 FEM 1985 |
| | 1905 | Error in syntod with ("??0000") |
| | 1906 | Error in syntod with ("120000") |
| | 1907 | Error in syntod with ("?????1") |
| | 1908 | Error in syntod with ("000000") |
| | 1909 | Error in syntod with ("2?????") |
| | 1910 | Error in syntod with ("??????") |
| | 1911 | Error in syntod with ("????4?") |
| | 1912 | Bad parameter did not send Signal 26 |

| File: tsktst.c | Code | Meaning |
|---|---|---|

| | |
|---|---|
| 2001 | DUPTSK status on first creation of a task |
| 2002 | Error creating a task |
| 2003 | gettid could not find a valid task |
| 2004 | crtsk and gettid returned different identifiers |
| 2005 | DUPTSK status was not returned |
| 2006 | Bad Language error did not send Signal 26 |
| 2007 | Bad Language error did not return BADLNG |
| 2008 | Bad Local/Global flag error did not send Signal 26 |
| 2009 | Bad Local/Global flag error did not return BADPRC |
| 2010 | Bad Time Unit error did not send Signal 26 |
| 2011 | Bad Time Unit error did not return BADTIM |
| 2012 | gettid on non-existant task did not return BADRPM |
| 2013 | gettid on non-existant task did not send Signal 26 |
| 2014 | gettid(0L) does not return correct tid |
| 2015 | start on deleted task did not return BADRPM |
| 2016 | start on deleted task did not send Signal 26 |
| 2017 | gettid on deleted task did not return BADRPM |
| 2018 | gettid on deleted task did not send Signal 26 |
| 2019 | Error creating a task |
| 2020 | contsk with bad identifier did not return BADRPM |
| 2021 | contsk with bad identifier did not send Signal 26 |
| 2022 | Valid contsk did not return NOERR |
| 2023 | Trying to clear contsk did not return NOERR |
| 2024 | contsk did not start the task |
| 2025 | contsk did not call turn off int routine |
| 2026 | contsk did not start itself with task identifier of 0 |
| 2027 | setpty with bad identifier did not return BADRPM |
| 2028 | setpty with bad identifier did not send Signal 26 |
| 2029 | setpty USEVAL 213 did not work on current task |
| 2030 | setpty ADDVAL 25 did not work on current task |
| 2031 | setpty ADDVAL -50 did not work on current task |
| 2032 | setpty ADDVAL 100 did not work on current task |
| 2033 | setpty ADDVAL -125 did not work on current task |
| 2034 | setpty ADDVAL -140 did not work on current task |
| 2035 | setpty USEVAL 200 did not work on current task |
| 2036 | setpty USEVAL 200 did not work on another task |
| 2037 | setpty ADDVAL 25 did not work on another task |
| 2038 | setpty ADDVAL -80 did not work on another task |
| 2039 | setpty ADDVAL -150 did not work on another task |
| 2040 | setpty ADDVAL 195 did not work on another task |
| 2041 | setpty ADDVAL 66 did not work on another task |
| 2042 | incorrect return arg expected 1234 |
| 2043 | Initialized data address was not passed correctly |
| 2044 | Run-time argument was not passed correctly |
| 2045 | Run-time argument not passed correctly (contsk) |

# APPENDIX C:

# MTOS-UX DEMONSTRATION

# APPENDIX C: MTOS-UX DEMONSTRATION USAGE

Introduction

The included software package is an example of how a multi-tasking environment is implemented under MTOS-UX.

In this example there are 5 tasks. These, along with a brief description of each task, are as follows:

1) CO: This task is responsible for the initiation of the i/o ports, creation and start-up of the other tasks, creation of all message buffers and clean up at shut down. ( CO stands for coordinator )

2) PORT: This task polls all the i/o ports initiated in this system: AUX.2, Basic Rate port A, Basic Rate port B and the keyboard. When it detects something at one of these ports it notifies the specified task.

3) DSP: This task displays the strings other tasks request to have displayed. In this example each string displayed on the screen is also sent out on AUX.2 by DSP.

4) MA1: This task is the analysis task for basic rate port A. MA1 receives the buffer received on port A from 'PORT' it then does what it wants with this buffer and sends the display request ( or send on AUX.2 ) of what it chooses to 'DSP'. In this example MA1 only notifies 'DSP' of a message receipt. This can however easily be expanded to suit another need.

5) MA2: This task does the same as the above only it interacts with port B.

There are 14 files included in this package. They are as follows:

| | |
|---|---|
| co.c | the body of the 'CO' task ( main() ) |
| port.c | the body of the 'PORT' task |
| dsp.c | the body of the 'DSP' task |
| ma.c | the bodies of 'MA1' and 'MA2' |
| util.c | send and receive functions used by all tasks |
| linkcom.c | initiation functions for basic rate setup |
| links.c | configuration of link connections needed |
| tcd.c | initiation structures for each task2 |
| mainsym.h | global and useful symbols |
| tos.h | events and corresponding structures task initiation structures |
| paval.h | symbols used to initiate AUX drivers |
| err.o | error handling functions |
| makefile | |
| sw.doc | the documentation you are now reading |

It should be noted that each task started by 'CO' receives a pointer to a structure where it finds its configuration information.

Each task also starts by initiating itself (the initiation function to be found at the end of each task file) and exchanging START_REQ and START_CONF messages with 'CO'.

The following configuration information will help you to start-up in your environment.

## Configuration

Two files must be edited to match your environment before running this demonstration program. These are:

1. co.c

   The co.c file handles the initiation of the basic rate ports by calling a function named setup().

   ```
   setup(PORTA,...,...,...,nt_power);

   setup(PORTB,...,...,...,nt_power);
   ```

   The parameter nt_power should be set to the value required by the TE's used. The Chameleon Basic Rate Library explains the meaning of the different values.

2. links.c

   The links.c file specifies which links are to be handled in this particular case. The structure *linksA* defines the links on port A, *linksB* defines the links on port B. The structure must be terminated with the link NONE,NONE,NONE. The structure (defined in mainsym.h) has the following appearance:

   ```
   typedef struct
     {
     int type; /*specifies whether the TEI value has to be set
                 */
     int tei; /* TEI value */
     int sapi; /* SAPI value */
     } LINK;
   ```