CIRSSE-R-97

NASA-CR-192734

NAGW-1333

[illegible]
11-63-CR
153760
P-287

CIRSSE

TECHNICAL REPORTS

Center for Intelligent Robotic Systems for Space Exploration

Rensselaer Polytechnic Institute
Troy, New York 12180-3590

(NASA-CR-192734) LECTURE MATERIALS FOR THE CTOS/MCS INTRODUCTORY COURSE (Rensselaer Polytechnic Inst.) 287 p

N93-21308

Unclas

G3/63 0153760

LECTURE MATERIALS FOR THE CTOS/MCS INTRODUCTORY COURSE

by

Keith Fieldhouse, Kevin Holt, Don LeFebvre,
Steve Murphy, Dave Swift, and Jim Watson

Rensselaer Polytechnic Institute
Electrical, Computer, and Systems Engineering Department
Troy, New York 12180-3590

July 1991

CIRSSE REPORT #97

Lecture Materials for the CTOS/MCS Introductory Course

Keith Fieldhouse Kevin Holt Don Lefebvre
Steve Murphy Dave Swift Jim Watson

August 12, 1991

Abstract

On July 18 and 19, 1991 the Center for Intelligent Robotic Systems for Space Exploration presented a course on its robotic testbed support software as it then existed. The course materials are collected as a reflection of the state of those systems at that time.

1 Introduction

The CIRSSE testbed consists of two Unimation PUMA 6 degrees-of-Freedom manipulator arms mounted on a 6 degrees-of-freedom (two 3 DOF carts on a 12 foot rail system) transporter platform. The testbed hardware is controlled through several Motorola single board computers and associated VME I/O boards.

The interface to the system is managed by a software system currently under development at CIRSSE. This software has evolved into two distinct sub-systems: the CIRSSE Testbed Operating System (CTOS) and the Motion Control System (MCS). The design of CTOS/MCS is driven by several fundamental requirements:

- The system must provide a designed, convenient interface to the testbed for both of its distinct user groups:
  - Researchers who wish to work on the actual control of the testbed devices. Such researcher may wish to substitute customized controllers, trajectory generators, device interfaces etc.
  - Researchers who require motion service from the testbed as part of their research agenda, but who are more concerned with the reliability and repeatability of the motion rather than the algorithm which produced it.
- As homogeneous an interface as possible should exist between the user and the 18 degrees-of-freedom available in the testbed. Different experimental set-ups should be possible, allowing the testbed to be treated as two 9 DOF arms, one 18 DOF manipulator system, 3 6 DOF manipulators and so on. Further, it should be possible to reconfigure the testbed with new manipulator devices as they become available.
- For performance reasons, the controlling software for the testbed runs on multiple single board computers on a VME backplane. This introduces a level of complexity that the software system must encapsulate and hide as much as possible.
- The entire control system must be a part (at the execution level) of overall CIRSSE hierarchy of intelligent robotic control.

As was noted earlier, two distinct software sub-systems are being developed to achieve these goals. The first, CTOS, is a layer of utility routines that extend the base operating system, notably in the area of inter-process (and inter-process*or*) communication and synchronization. The second, MCS, establishes the control and command interface to the testbed hardware.

At the time the CTOS/MCS course was presented the following software had been developed:

- As part of **CTOS**:
  - A bootstrap system which provides for the distribution of processes across any of the Single Board Computers on a single VME chassis.

  - A message passing system which provides easy, efficient (though not "realtime") and flexible inter-process and inter-processor communication.
  - A time synchronization library that allows multiple processes across multiple processors by be synchronized at different clock rates.
  - Other utilities that provide on demand synchronization, shared memory access and protection and various other useful functions.

- As part of **MCS**:
  - The MCS State Manager, which manages communication between the devices available through the MCS.
  - "Channel Drivers" (hardware interfaces) for CIRSSE's transporter platform and the two PUMA manipulators.
  - Several different controllers (Basic PID, Gravity compensation) for the PUMAs and platform.
  - A simple trajectory generator capable of reading (from a file) and interpolating between a series of joint space set points.

The development of the software to this level represented the substantial achievement of an early CTOS/MCS milestone. Specifically, that enough of the system be in place that members of CIRSSE not a part of the core development team could make use of it. To further achieve this goal, an internal CTOS/MCS course was developed, the materials for which are collected in this report.

Divided into 3 lecture sections, a lab exercise period, a case study and a round table discussion, the CTOS/MCS course ran over a period of two days.

Due to the broad range of experience levels at CIRSSE, especially with respect to real time programming issues, the first lecture section was a review of C language programming, real-time and hardware programming issues and the VxWorks operating system (a real time OS developed by Wind River Systems of Alameda California and the software platform on which most CIRSSE real-time development is done). The intent of the first section of the course set out to insure that all course participants had at least some degree of common vocabulary and understanding of the issues on which the rest of the course was based.

Section II of the course lecture covered the CIRSSE Testbed Operating System. This section was of particular importance, as CTOS is expected to be the infrastructure on which most of the CIRSSE intelligent control hierarchy is built. Thus, most of the class participants could be expected to make use of the CTOS interface whether or not they make direct use of the manipulator testbed.

After the first two lecture sections, the class was broken into groups to work on a series of lab exercises based on the lecture material presented. These exercises served to give the participants an opportunity to familiarize themselves with both the programming environment established for testbed development and the programming techniques used to work with CTOS.

Day two of the class covered the Motion Control System itself. This portion of the class was of primary interest to those participants planning to develop custom components for the MCS and who wished to participate in the further development of the base components of the system. This lecture section was followed by a case study of a typical MCS application and the components that comprise it.

The remainder of this document contains are the lecture notes, supplementary materials, lab exercises and solutions for the first CIRSSE CTOS/MCS class. These materials are collected here solely as a reflection of the state of development of the software and are in no way intended to supplant further, more comprehensive documentation of the systems.

Acknowledgement

This work was supported by NASA Grant No. NAGW-1333.

CTOS/MCS

July 18 - July 19, 1991

CTOS/MCS

Section I: Overview

Introduction and Overview

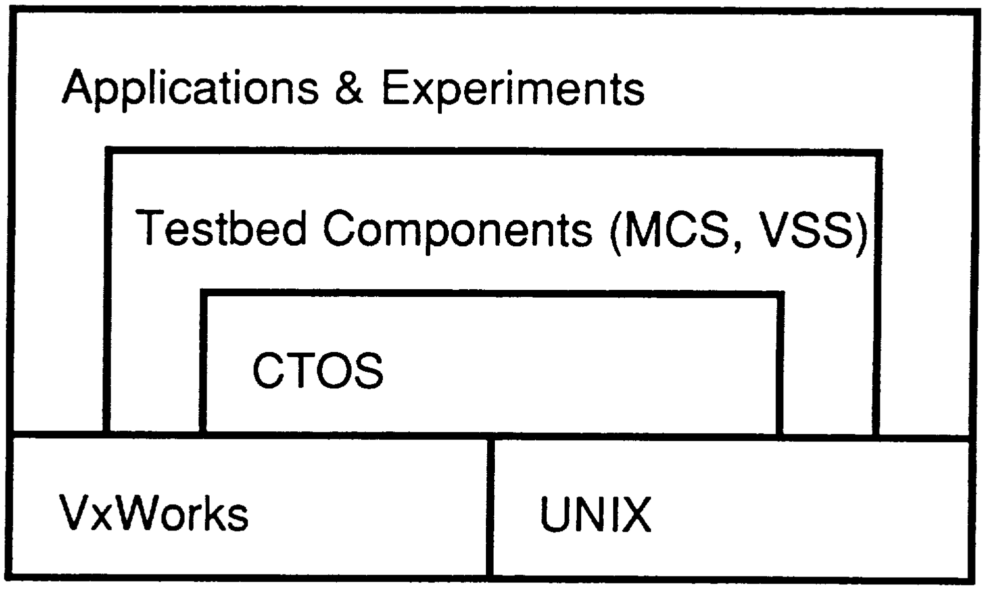
- The Context of CTOS & MCS
- C Programming
- Realtime Programming and Distributed Processing
- VxWorks
- The CIRSSE Testbed Development Environment

Context of CTOS & MCS

- **CTOS** CIRSSE Testbed Operating System

- **MCS** Motion Control System

Context of CTOS & MCS



Context of CTOS

- Developed to overcome limitations in UNIX and VxWorks with respect to interprocessor process communication, synchronization and distribution

- Provides a framework and a consistent programming interface for testbed components and applications

- Provides an infrastructure for the development of the Intelligent Machine

Context of MCS

- Major interface to the testbed manipulators

- Designed, implemented and tested with multiple manipulators

- Functional components may be replaced and reconfigured with minimal intervention

- Developed in conjunction with CTOS and the current design of the CIRSSE Intelligent Machine hierarchy

C for CTOS/MCS Programmers

- Syntax
- Pointers and Addresses
- The C Pre-processor
- Sources of Information

C Syntax – Literals

`9`	A decimal integer, with value 9_{10}
`010`	An octal integer, with value 8_{10}
`0xf`	A hexadecimal integer, with value 15_{10}
`'A'`	A single character, the letter “A”
`'\007'`	A single character, ASCII 7_{10}, the bell
`'\xb'`	A single character, ASCII 11_{10}, vertical tab
`'\t'`	A single character, a tab
`"Hello World"`	The character string “Hello World”
`"HI" "MOM"`	The character string “HIMOM”
`'\0'`	The null character

C Syntax – Functions

- All functions have a return type (which may be void)

- *All* parameters are passed by value

Function syntax:

```
return-type
function-name(parameter-list or void)
{
declarations
statements
...
}
```

C Syntax – Scoping Rules

```
/*
** File:  example.c
*/

int x;              /* Global Variable */
extern int y;       /* Also defined as IMPORT */
static int z;       /* Also defined as LOCAL */

int fun(int a, int b, int c)

{
int i;i                /* Automatic variable, local
                          to function fun */
static int count;      /* Not automatic,
                          but still local to function fun */

for (i = 0; i < 15; i++) {
    int e;             /* Automatic variable
                          local to the for loop */
    e = i + y;
    }
e = d + i;             /* Error, e is undefined */
}
```

C Syntax – switch statements

- Multi-way decision
- Each `case` must be an integer constant
- Each `case` must be unique
- A `break` must be used to end a case
- A default case is available but not required
- The `switch` expression must evaluate to an integer

C Syntax – switch statements

Switch Statement Syntax:

```
switch ( variable ) {
    case 1:
        /* Statements for case 1 */
        break;
    case 2:
        /* Statements for case 2 */
        return;
    case 3:
        /* Statements for case 3 */
    case 4:
        /* Statements for case 3 and 4 */
        break;
    default:
        /* Statements for default case */
        break;
    }
```

C Syntax – Structures, Unions and Typedefs

- Aggregates of multiple variables, possibly of different data types
- May be copied and assigned to.
- May be passed to and returned by functions
- A *structure* contains space for all of its elements while a *union* contains space for any *one* of its elements
- An individual union must be used consistently
- A *typedef* provides an alias for a previously defined type

C Syntax – Structures

Structure Syntax:

```
struct point2d {
    int x;
    int y;
    } p1, p2;

struct point2d p3;
```

To reference elements in the structure:

```
p1.x = 5;
p3.y = p2.x;
```

C Syntax – Unions and Typedefs

```
union jointInfo {
    float position[MAX_JOINTS];
    int   period[MAX_JOINTS];
    } ;
```

```
typedef union jointInfo JOINTINFO;
```

```
JOINTINFO jlist;
```

To reference elements in a union:

```
jlist.position[3] = 7.5;
jlist.period[5] = 4;
```

C Pointers – Notation

- Genuinely an address: `0xffd0` *not* $\longrightarrow$
- Use & to get the address of a variables
- Use $*$ to get the contents of an address and to declare a variable as an address

C Pointers – Function parameters

- Pointers can be used to create argument passing by reference:

```
void increment(int *p)

{
*p = *p + 1;
}


...


int x = 1;
increment(&x)
```

C Pointers – Structures and Unions

Structure and Union pointers are often used to avoid passing large data structures back and forth. The usefulness of this construct lead to a shorthand for dereferencing a structure through a pointer to it:

```
struct test {
    int a;
    double b;
    char c;
} t1 *pTest;

pTest = &t1;

(*pTest).a = 5;
pTest->c = 'a';
```

C Pointers – Pointer arithmetic

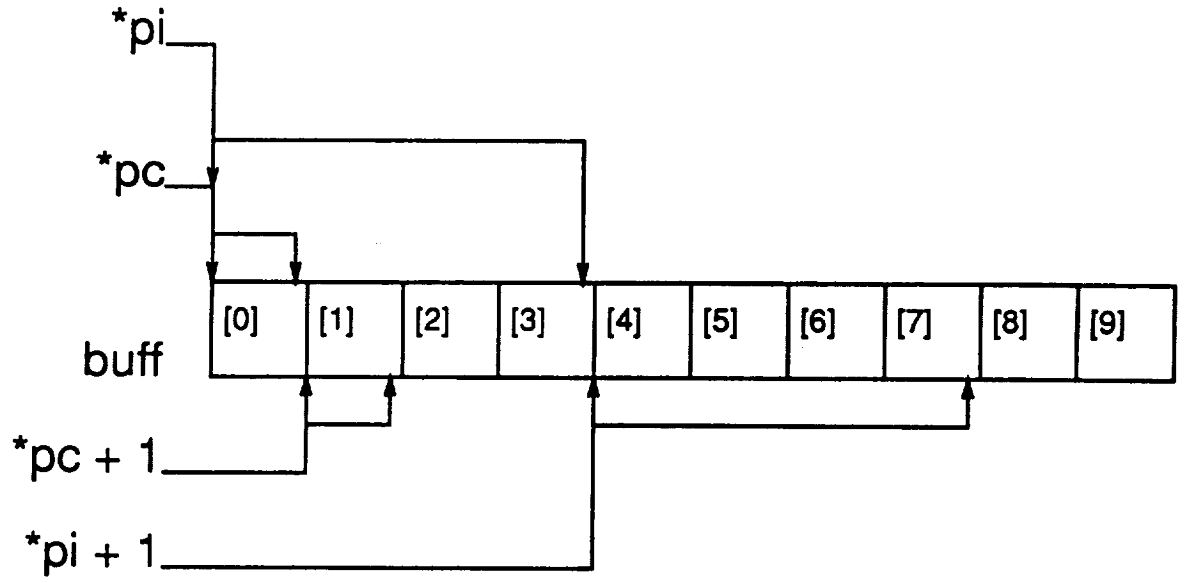
- Integers may be added and subtracted from pointers

- Conversion is done based pointer type

- Address exceptions can occur if alignment isn't heeded

```
char buff[10];
char *pc = &buff[0];
int  *pi = (int *)&buff[0];
```

C Pointers – Pointer arithmetic



The C Pre-Processor

- Processes a file *before* it is seen by the compiler

- Directives start with a # in the first column, keywords may be indented

- Used to define constants, macros and to textually include other C source or "header" files

The C Pre-Processor – Include Files

```
#ifndef  INCmyheaderh
#define  INCmyheaderh

/* Constants and key words */
#define  REDUCE     (1)
#define  EXPAND     (2)
#define  PI         (3)  /* For programming
                            in Georgia */
/* Macros */
#define  FOREVER        for(;;)
#define  MIN(_x,_y)     (_x > _y ? _y : _x )
#define  dataReduce( _what )  \
               dataManipulate(REDUCE,_what)
#define  dataExpand( _what )  \
               dataManipulate(EXPAND,_what)

/* Function prototypes */
int dataManipulate(int how; int what);
```

C Pre-Processor – Inline Functions

Consider the following code, when used in the macro MIN previously defined:

```
z = MIN(x++,y++);
```

Note that the arguments to the macro are "x++" and "y++", which will result in the increment being done twice for "x" and "y". Probably not the desired effect. One possible solution is *inline functions*: Included in header files as:

```
extern inline min(int x, int y)
{
    if (x > y) return(y);
    else       return(x);
}
```

C Pre-Processor – Inline Functions

- Not part of ANSI C but common and available with GCC

- Function replaces its call, but arguments and scoping of variables handled as with "normal" functions

Sources of Information

- *The C Programming Language, Second Edition*, Brian Kernighan and Dennis Ritchie (K&R)
- *Using and Porting GCC*, Richard Stallman
- The GCC manual page

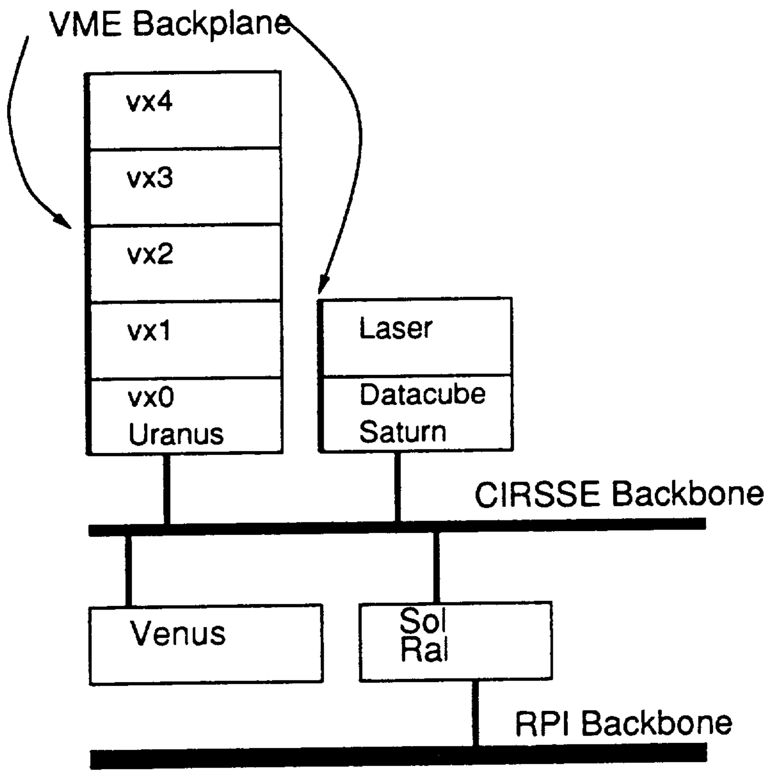
VxWorks

VxWorks is the real time operating system and development environment used at CIRSSE for motion control and Datacube based vision experimentation. Some features:

- runs on VME based single board computers
- Rich run time library
- Object code compatibility with UNIX
- Close network compatibility with UNIX
- An interactive shell for debugging and development

VxWorks – Networking

VxWorks, when installed on a VME cage, forms a *backplane* network. This is a TCP/IP (Internet) network which uses shared memory on the VME cage as a transport rather than Ethernet cable. All of the nodes become standard Internet nodes:



VxWorks – The Kernel

When a VxWorks system boots, it loads a VxWorks kernel over the network from its supporting host (Venus here at CIRSSE). This kernel contains the main entry point of the OS and all of the Wind River Supplied code that has not been expressly eliminated from the kernel. During the boot process, the kernel's entry routine may read and execute a user specified script of VxWorks shell commands, or it may load and call user specified code.

VxWorks – Utility Libraries

lstLib Doubly linked lists

rngLib Ring buffers

semLib Intra processor semaphores

spyLib CPU performance monitoring

stdioLib C Standard I/O library

sockLib UNIX 4.3BSD compatible network sockets

VxWorks – Kernel Selection

At CIRSSE there are numerous VxWorks kernels available. For the most part they contain the same set of VxWorks utility libraries. Some however are built for the Datacube, while others are built for the Control Cage. Further, some of the kernels support CTOS while others are built as raw VxWorks development environments. To select between VxWorks kernels, use the command `vxboot` on any of the CIRSSE UNIX systems.

VxWorks – Kernel Selection

When the `vxboot` command is used, it will present you with a list of the CIRSSE VxWorks processors for which you can select a kernel, and two pseudo processors:

@control	*vx0 vx1 vx2 vx3 vx4*
@vision	*laser datacube*
vx0	Control cage CPU 0 (MV135)
vx1	Control cage CPU 1 (MV135)
vx2	Control cage CPU 2 (MV135)
vx3	Control cage CPU 3 (MV135)
vx4	Control cage CPU 4 (MV135)
datacube	Datacube CPU 0 (MV147)
laser	Datacube CPU 1 (MV135)

VxWorks – Kernel Selection

Once you have selected the processors, you may select a kernel. The kernels with a $*$ in their names should be selected only for the Pseudo processors.

control.ctos.* *CTOS Kernels for Control Processors*

control.ctos.mv135 *Kernel with CTOS support for Control Cage*

control.default.* *Development Kernels for Control Processors*

control.default.mv147 *Kernel for Control Cage development (VxWorks V5)*

vision.ctos.* *CTOS Kernels for Vision Processors*

vision.default.* *Development Kernels for Vision Processors*

vision.default.mv135 *Kernel for laser control processor*

vision.default.mv147 *Kernel for datacube main processor*

VxWorks – The Shell

The VxWorks shell provides the user with a simple interactive interface to a system running VxWorks. It has the following commands/features

- `cd "/home/krf/vxworks"` will set the default directory to "/home/krf/vxworks"
- `ld < filename.o` will load the object code in "filename.o" into the running VxWorks system
- `< filename` will read a script of VxWorks shell commands from "filename"
- `i` will display a list of running processes
- function(5,6,7) will call any globally defined C function (which may be either VxWorks or user defined). In this case the function is passed the arguments "4", "5" and "6"

VxWorks – Dynamic Linking

VxWorks has the unique ability to dynamically link an object module with an already running system. This is accomplished by creating a standard UNIX object module and loading it with the shell's `ld` command. This dynamic linking has the following characteristics:

- All global symbols are added to the system symbol table
- When symbols are loaded which have the same name as already loaded symbols, the old symbols are effectively replaced
- Multiple UNIX object modules may be pre-linked with the UNIX `ld` command to form a single object module
- Unresolved references in an object module must be resolvable at load time

VxWorks – Dynamic Linking

Object files appropriate for the VxWorks environment here at CIRSSE may be created with the following command:

```
vxgcc filename.c
```

- Only creates object modules
- Causes C pre-processor to look in VxWorks directories
- Uses cross compiler on SPARC (Sun4) based systems

VxWorks – Further Information

- *VxWorks Programmer's Guide*
- *Using VxWorks at CIRSSE*, Tech Memo #3
- The VxWorks manual pages

  `vwman lstLib` For VxWorks utility functions

  `vwman mv135/sysBusTas` For board specific VxWorks functions

Realtime, Hardware and Distributed Programming

Realtime Programming Programming in which the correctness of an operation is dependent not only on its result but on the time at which the result is achieved

Realtime, . . .

Most of the development to date on CTOS and the MCS have been in the VxWorks based "realtime" hardware development environment. There are several characteristics of this environment that provide special challenges:

- The operating system is much less sophisticated and protective. Accessing memory that is more likely to crash the system than anything else
- Shared resources may be contended for among many processors as well as processes
- Communication must take place between processes and processors
- Hardware interfaces often must be built from scratch, utilizing the device registers, interrupts and other tools often hidden by multi-user Operating Systems such as UNIX

Realtime, ...– Hardware Programming

Often it is necessary to set bits in a control register on a particular hardware interface board. Consider the following Control Status Register on an I/O board. Bit 3 must be set to a 1 in order to enable the board:

```
#define IOCSR ((volatile char *) 0xffffdf0)
#define ENABLE  (0x04)

*IOCSR |= ENABLE; /* Enable the board */
*IOCSR &= ~ENABLE; /* Disable the board */
```

Realtime, ...– Hardware Programming

Often, in realtime programming, it is necessary to insure that a function is re-entrant (for ISR's, Event Handlers or functions that are called by same). This means that it must not be an error to call a function when some other version of that function is still running. To ensure re-entrancy keep the following in mind:

- Do not maintain static automatic variables
- Do not use global variables
- Do not arbitrarily use finite resources

Realtime, . . . – Resource Contention

In a distributed or multi-tasking environment it it often possible for multiple threads of execution to require the use of some limited resource. It is often necessary to arbitrate the use of this resource to prevent improper action. The semaphore can be used to construct protection for shared resources.

There are two basic semaphore operations:

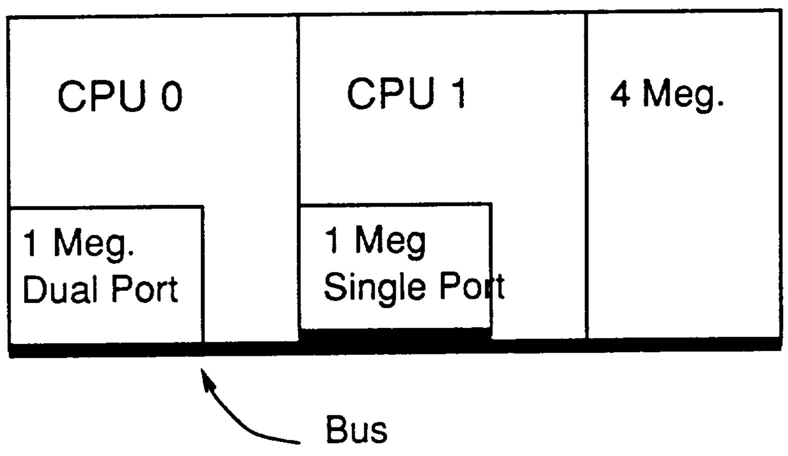
Realtime, . . . – Resource Contention

TAKE(s) The take operation determines if the semaphore "s" is available. If it is, it is removed (made unavailable to other processes) and the thread of execution may continue, using the protected resource. Note that the testing of the semaphore and the removal of it must be indivisible operations

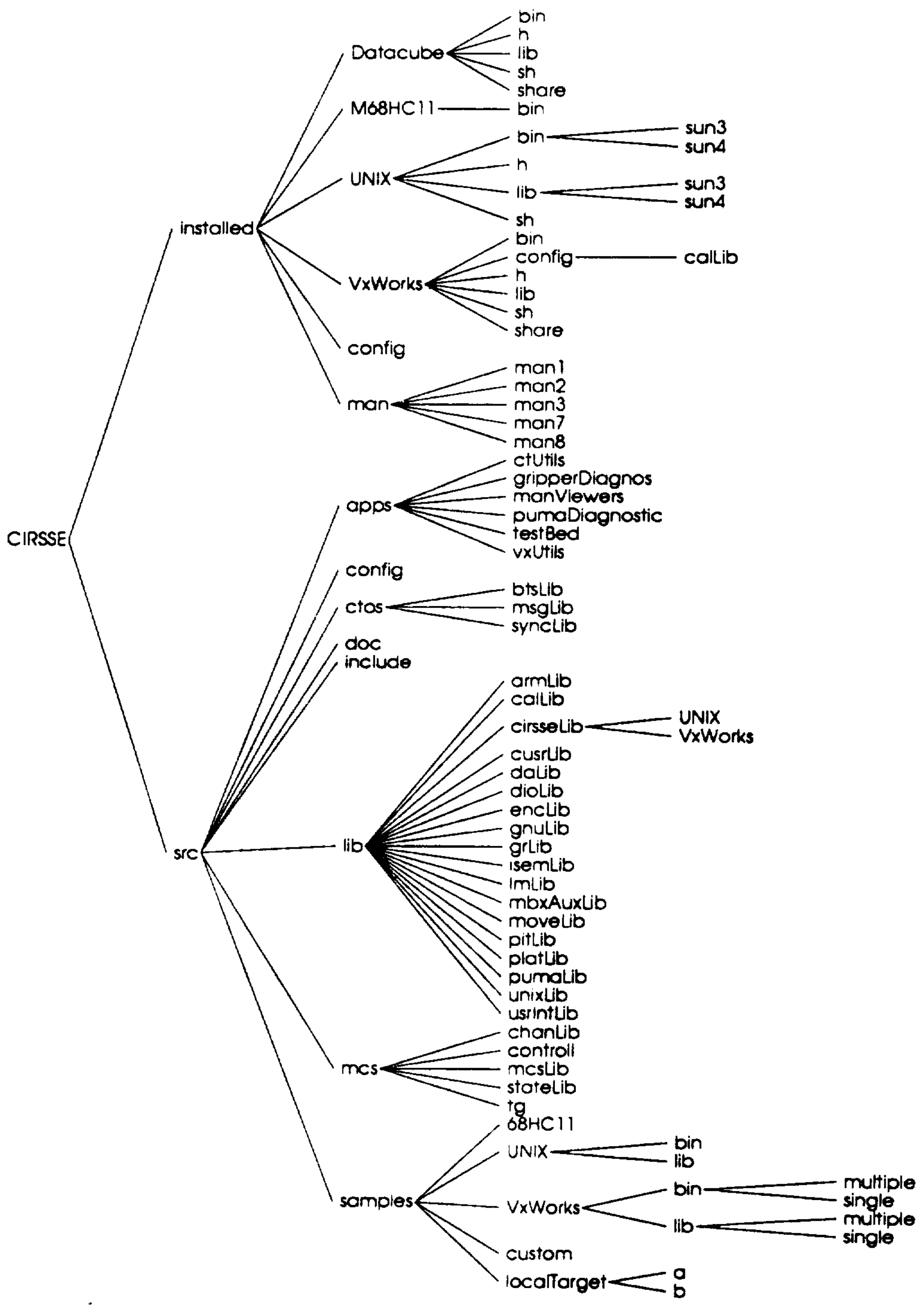
GIVE(s) The GIVE operation simply replaces an already removed semaphore

Realtime, ...– Memory

A peculiar aspect of many realtime programming environments (including the one at CIRSSE) is that memory is shared among all processes and often among processors. This provides a convenient method of inter process communication (when coupled with semaphores etc.).



Testbed Development



Testbed Development – Imake

In order to maintain some degree on manageability for software that has been developed for multiple platforms and multiple operating systems, the CIRSSE testbed development environment makes heavy use of the *Imake* system developed for the distribution of the X Window System.

Testbed Development – Imake

- A user of Imake creates creates an `Imakefile` in which he or she specifies the targets that should be built, and the files that make up that target
- When creating the Imakefile, the user makes use of pre-defined macros that are tailored to the specific system (in this case, the CIRSSE testbed) for which development is being done.
- Imake reads the user's `Imakefile` and the system macro definitions and creates a standard UNIX `Makefile` which can be called with the `make` utility
- To create a `Makefile`, type `cmkmf` in a directory in which an `Imakefile` exists. (Mnemonic: cmkmf == Cirsse MaKe MakeFile)
- If `cmkmf` is called with arguments, `make` is automatically called with those arguments once the `Imakefile` is converted

Testbed Development – Imake and cmkmf

```
AllTarget(ex1.o ex2.o)
VxWorksBinTarget(ex1.o,header.h, )
VxWorksBinTarget(ex2.o,header2.h, )
VxWorksBinTarget(ex3.o,header.h, )
```

Produces

```
all : ex1.o ex2.o ex3.o

ex1.o : ex1.c header.h

ex2.o : ex2.c header2.h

ex3.o : ex3.c header.h
```

Testbed Development – Naming Conventions

Project Prefix A 3 to 6 letter sequence that uniquely identifies a project or component. bts, msg, ipb

Functions Upper and Lower case, no underlines. Each word (but the first) is capitalized. Public functions start with the project prefix. Object verb arrangement. ipbClear, mcsSlotReserve

Variables Same as functions. mcsSMTid, ipbFlag

Constants All upper case. Each word separated by an underscore. Public constants start with the project prefix. MCS_MAX_SLOTS

Testbed Development – File Organization

```
/* %W% %G% */
/*
** File:
** Written By:
** Date:
** Purpose:
**
** Modification History:
*/

/* Include section */

/***************************************************
** Function:
** Purpose:
** Returns:
*/
```

Testbed Development – Other Conventions

- Separate system specific code as much as possible – code may very well be compiled for separate operating systems

- Use function prototypes to ensure type checking of parameters and return values

- Documentation for most components will include manual pages for public functions and Technical Memos for extensive libraries of functions

CTOS/MCS

Section II: CTOS

Outline of CTOS Topics

Processor/Task Configuration

- CTOS kernel & configuration files
- configuration file commands

Message Passing

- building messages
- message passing mechanisms
- managing message data

Event Handler Tasks

- designing an application
- format of event handler functions
- default processing of commands

CTOS Bootstrap Phases

- initialization phases
- application executive

Synchronous Processes

- creating & attaching sync processes
- communicating with sync processes

CIRSSE Testbed Operating System

CTOS supports development of distributed applications by providing means to:

- distribute processes among CPUs
- communicate between processes
- synchronize execution of processes

Configuration Files

Application Configuration File

- specifies chassis (pl.) used by application and names of chassis config files
- implicitly defines chassis interconnections
- currently (mid-July '91) being developed

Chassis CTOS Configuration Files

- one CTOS config file per chassis
- provides chassis-specific CTOS configuration information e.g. CPU interconnections & distribution of CTOS tasks

Chassis User Configuration Files

- one user config file per chassis in application
- defines where application software is loaded and what application tasks are created

CTOS Startup

Existing VME Chassis Startup

1. User defines application in <u>user config file</u>
2. User specifies user config file in 'ctconfig' command
3. VxWorks & CTOS kernels load & start when boot VME cage
4. CTOS reads <u>chassis CTOS config file</u> & starts remainder of CTOS
5. <u>User config file</u> is processed to load application software and create application tasks
6. CTOS broadcasts messages to synchronize initialization phases
7. "Application executive" takes over at start of AEXEC phase

CTOS Startup

Planned Sun/VME Multi-chassis Startup

1. User defines application in application config file and chassis user config files
2. CTOS kernels are preloaded and service daemons started to wait for application startup request
3. User starts application from command line of Sun or VME chassis

4. – 7. same as existing VME chassis startup

Config File Command Syntax

CPU_NUMBER   COMMAND   ARGUMENTS ...

- All CPUs on a chassis read the same config file, but only process lines that match their CPU number
- Except, lines with CPU_NUMBER of -1 are processed by all CPUs
- CPU_NUMBER must start in column 1
- COMMANDs are separated from CPU_NUMBER by one or more spaces, and may be upper or lower case
- ARGUMENTS are different for different commands, and are similarly separated by space(s)
- Comment lines begin with '#' or ' ' in column 1; hence blank lines are ignored

Config File Commands

- n LOAD /path/filename
  - load object module into local memory
  - order of loading files is important
    - usually load shared global variables first
    - must load C function before loading code that calls that function
    - all functions used by a task must be loaded before the task is created
  - uses /path/ if given, otherwise finds filename in current directory
- n SHARE /path/filename hex_address
  - load object module into specified memory address
  - primarily used to load global variables into shared memory
  - usually set hex_address to 0x0, which causes load into address immediately following previous SHAREd object module
  - must SHARE same files in same order on any CPU that receives SHAREd objects

Config File Commands, Con't

- n TASK sym_name func_call priority
  - create an event handler task
  - symbolic_name must be unique throughout application, and be < 24 characters
  - function_call specifies the name of the C function that executes the event handler code
  - application task priorities should be in the range of 100 - 255; CTOS and VxWorks use priorities < 100
- n INCLUDE /path/filename
  - suspends processing of current config file and begins processing commands from specified /path/filename
  - processing of original config file resumes after completion of included config file
  - include files may be nested to any depth
  - CAUTION: use of CHDIR within an include file will change current directory for original config file

Config File Commands, Con't

- n CHDIR /path/
  - changes the current directory to /path/ for subsequent LOAD, SHARE and INCLUDE commands that do not explicitly specify a path
- n ECHO ON | OFF | text
  - ECHO effects what is printed to the console display during config file processing
  - ECHO OFF will turn off information and warning messages, but error messages will be displayed
  - ECHO ON or ECHO followed by text will turn on all message printing, and will display 'text' to the console
- n LOGO /path/filename
  - specifies a file that will be displayed on the console when the application starts
  - the full /path/ to the logo file is REQUIRED

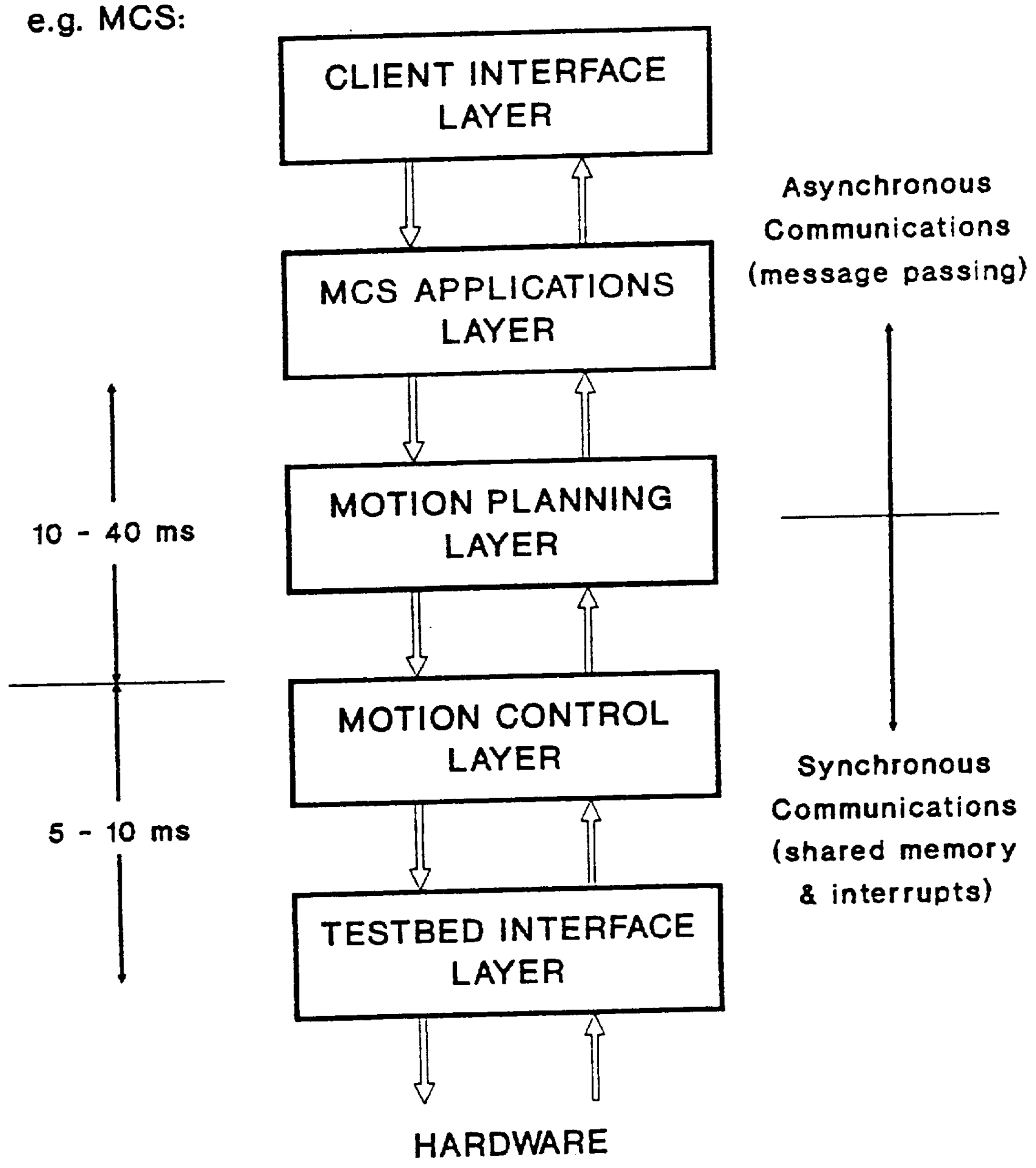
$\Longrightarrow$ refer to 'ctos_config' manual pages for the most current information on config file commands MCS/CTOS

Example User Configuration File

```
#  Configuration File for Example Application

#  'include' command reads another config file
-1 include  /home/mydir/some_standard_config_stuff

#  'chdir' command changes current directory
-1 chdir  /home/mydir/

#  'load' command loads obj module
-1 load   xyzLib.o
0  load   mcsControl.o
1  load   pidLoops.o
1  load   platIoChannel.o
2  load   armIoChannel.o
3  load   armIoChannel.o
2  load   trajGen.o
4  load   myApplication.o

#  'task' command creates event handler task
0  task   MCS_Control  mcsMain      100
1  task   PID_1        pidAlgo      150
1  task   PID_2        pidAlgo      150
1  task   PID_3        pidAlgo      150
1  task   platIO       platHandler  150
⋮

#  can mix load & task commands
3  load   debug.o
3  task   DataLogger   dbgLog        75
```

CTOS Supports Two Forms of Interprocess Communications

e.g. MCS:



Message Structure

```
struct MSG_TYPE
      {
      TID_TYPE    dest     ;
      TID_TYPE    source   ;
      CMD_TYPE    command  ;
      void       *data     ;
      int         datasize ;
      FLAG_TYPE   flags    ;
      }
```

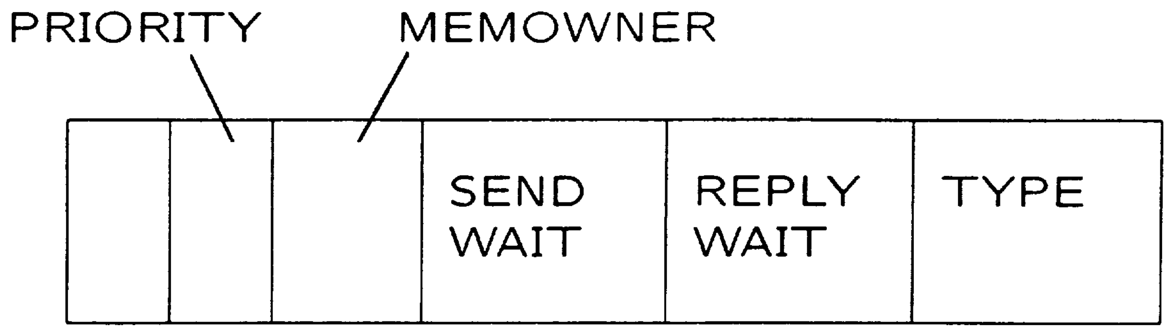
dest	TID of destination (receiving) task
source	TID of source (sending) task
command	indicates function of the message
data	points to additional message data
datasize	byte length of additional data
flags	specifies message handling options

Message Commands

- The .command member of MSG_TYPE structure is used to indicate the function of a message
  - CMD_TYPE is 2-byte unsigned int ⟶ over 65,000 unique commands
  - usually msg.command is equated to a predefined constant
- Message command conventions
  - names are upper case and begin with MSG_
  - values are assigned as offsets to blocks of commands
- Standard messages
  - MSG_PINIT: begin process initialization
  - MSG_AINIT: begin application initialization
  - MSG_AEXEC: begin application execution
- User-defined messages
  - define as offsets to MSG_USER, e.g.

```
#define  MSG_MY_MESSAGE    MSG_USER+1
#define  MSG_ANOTHER_MSG   MSG_USER+2
```

Message Flags



TYPE	normal, reply, etc. (used by system)
REPLY_WAIT	if set, sender will wait for reply
SEND_WAIT	waits if receiver queue is full
MEMOWNER	specifies who deallocates message data
PRIORITY	urgent msgs go to front of queue, normal to back

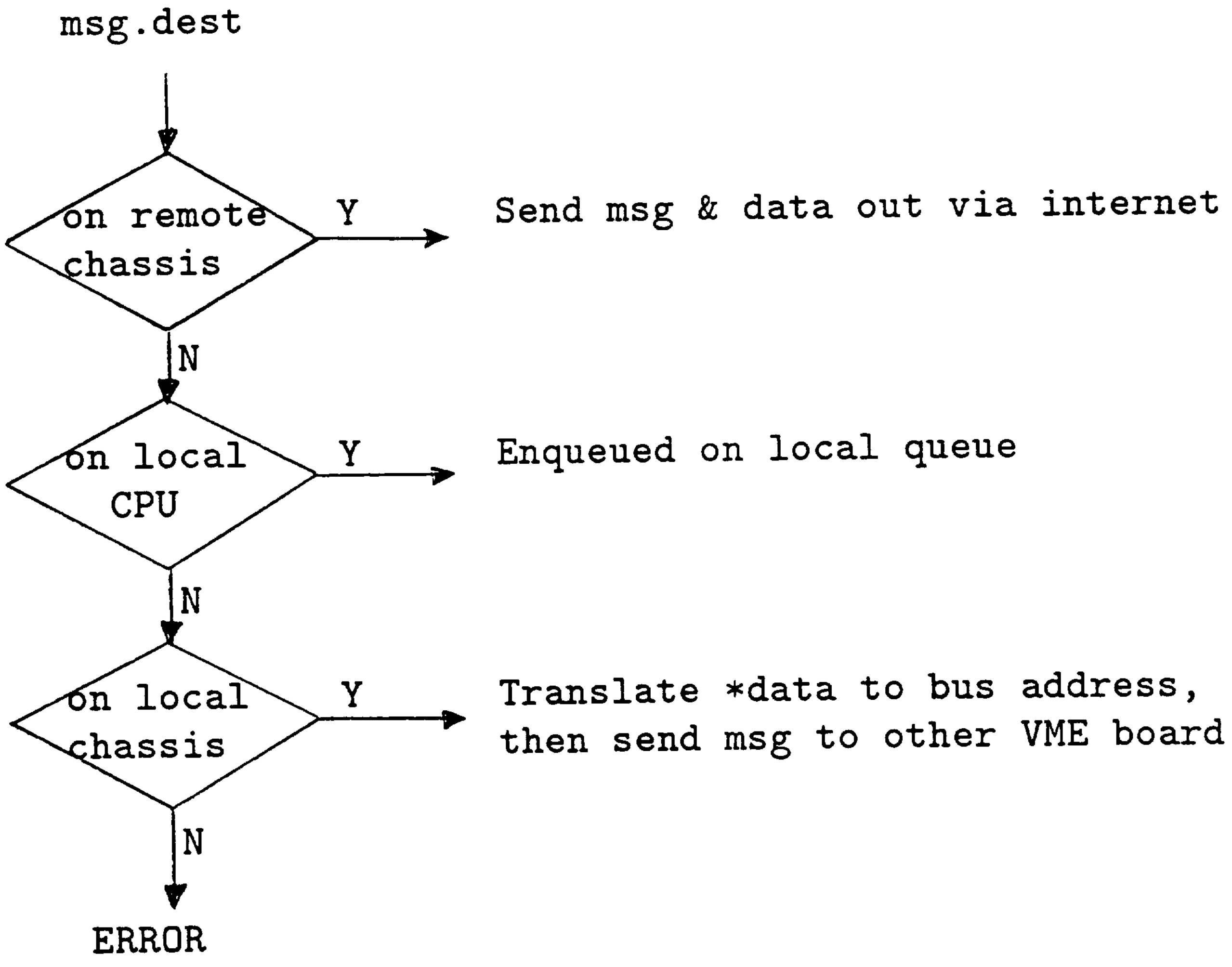
Using predefined message flags is recommended:

MF_STANDARD	normal priority, receiver owns memory, no waiting
MF_REPLYWAIT	normal priority, receiver owns memory, wait for reply

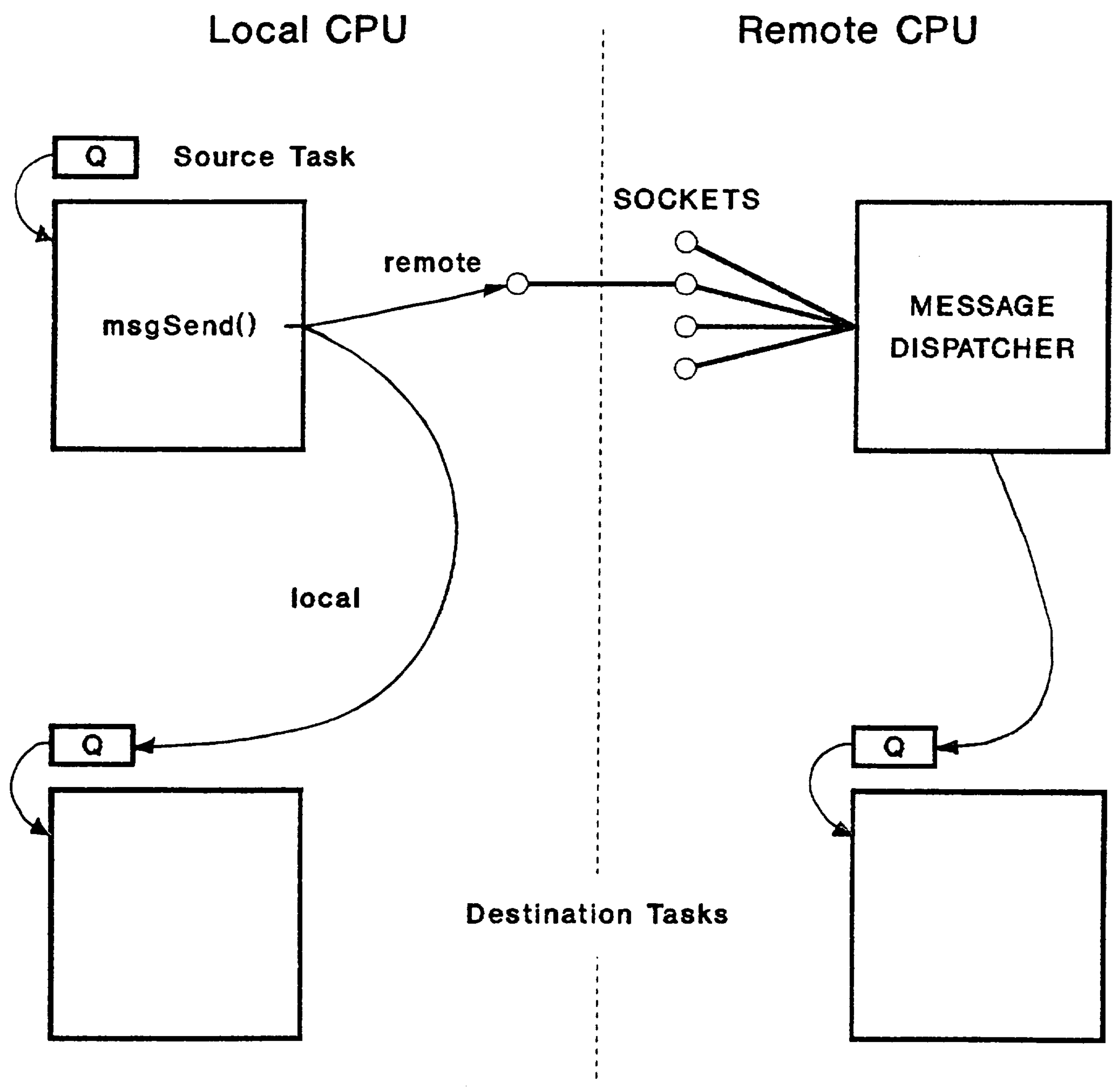
Task ID & Message Routing

TID = Chassis# + CPU# + LocalTask#

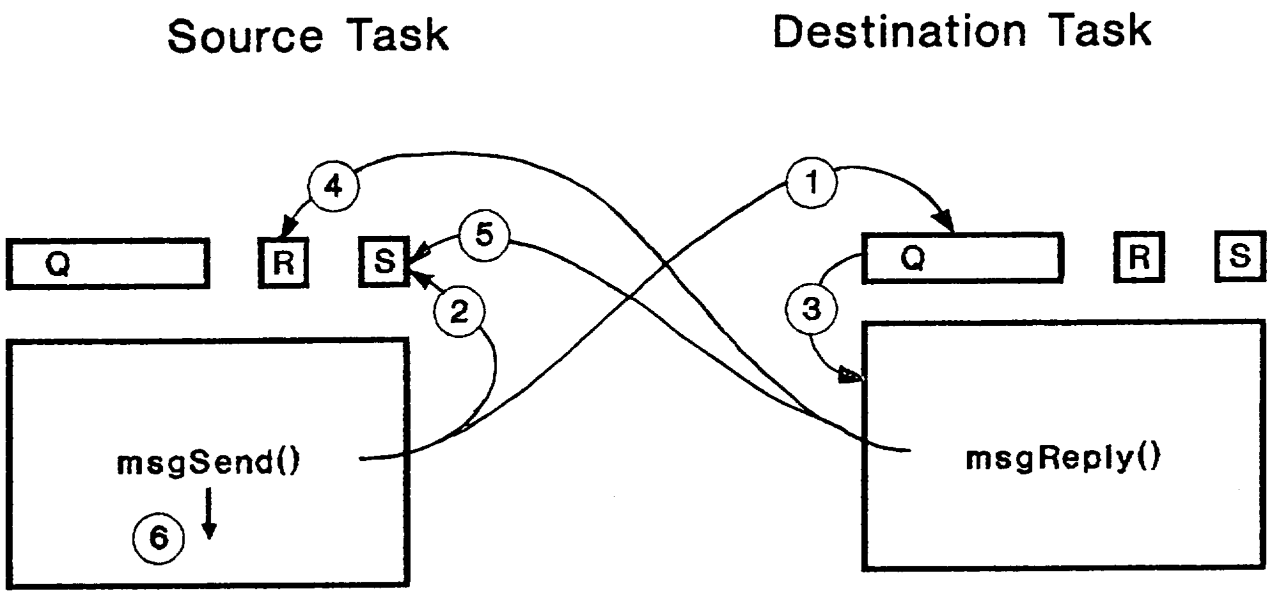
up to 16 Chassis	up to 16 CPUs	up to 256 Local Tasks


msg.dest
on remote chassis
Y
Send msg & data out via internet
N
on local CPU
Y
Enqueued on local queue
N
on local chassis
Y
Translate *data to bus address, then send msg to other VME board
N
ERROR

Normal Message Passing Mechanism



Message Reply Mechanism



1. Send message

2. Block sending task on semaphore

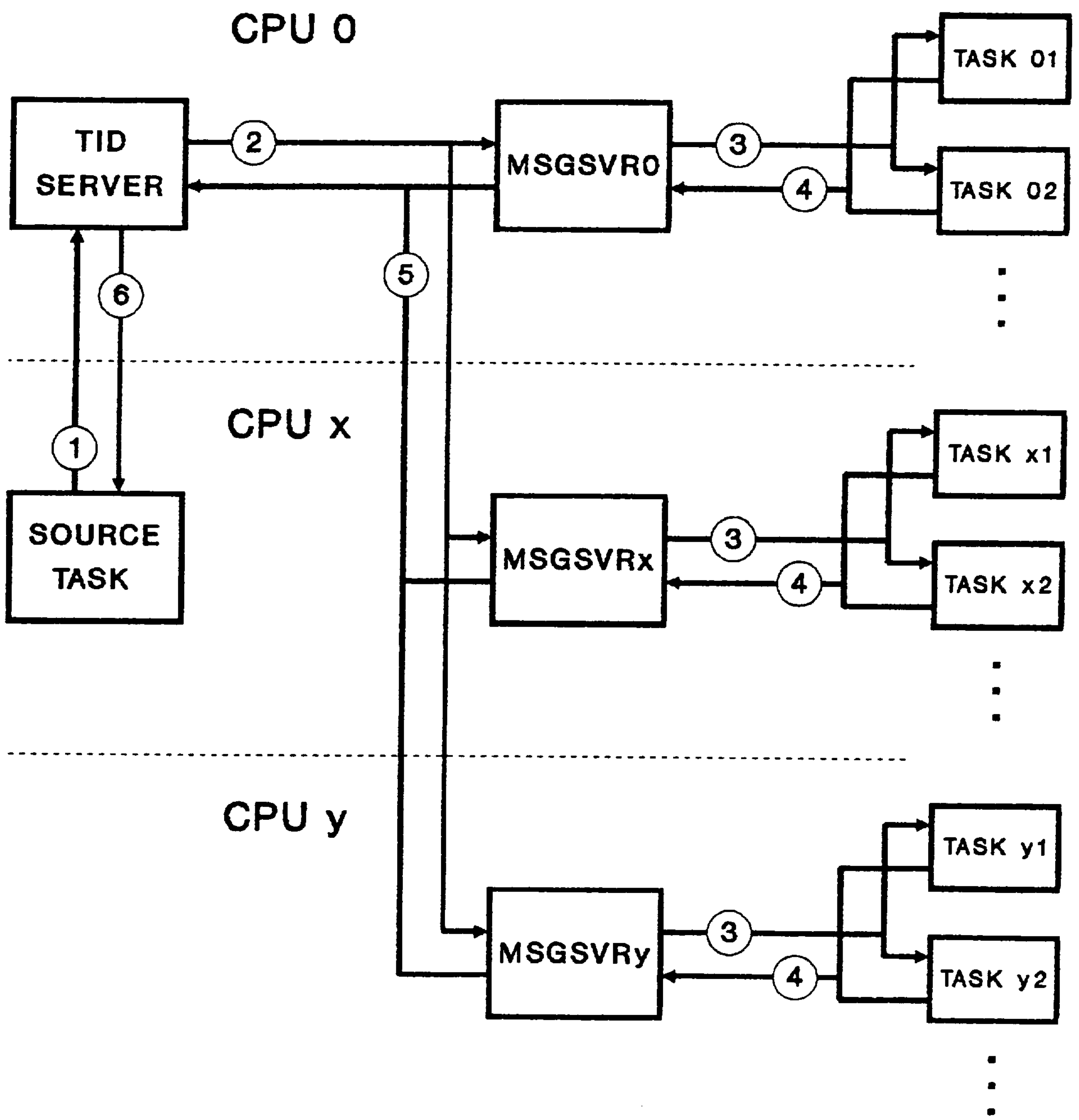
3. Receive message

4. Send reply

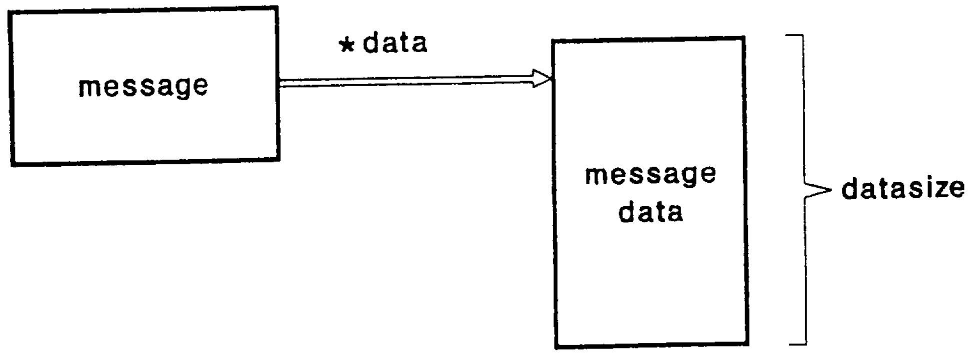
5. Unblock sending task

6. msgSend() returns reply data

Message Broadcast Mechanism



Managing Message Data



MEMOWNER = SENDER

- message sender "owns" memory allocated to message data
- message receiver should consider message data to be READ ONLY
- message sender is responsible for deallocating message data once it is no longer needed

MEMOWNER = RECEIVER

- message sender allocates memory for message data and "gives it away" to message receiver
- message data is automatically deallocated when receiving task exits event handler function
- use msgDataKeep or msgDataCopy to retain message data by receiver

msgLib Functions

- Sending Messages

msgSend	general form of send
msgPost	post msg returns immediately
msgBroadcast	send to all tasks
msgErrorLog	send string to Error Server
msgReply	reply to message
msgAcknowledge	acknowledge received msg
msgBuildSend	build then send msg

- Building Messages

msgBuild	set members of struct
msgTypeFlagSet, etc.	set fields of flags

- Working with Task Id's

msgTidQuery	find task id from name
msgTidGetCpu, etc.	get fields of tid
msgTidSetCpu, etc.	set fields of tid

- Queue Operations

msgDequeue	read message from local queue
msgQueueCount	count msgs in queue
msgRequeue	put message into local queue

msgLib Functions, Con't

- Memory Management

msgCopy	make copy of message
msgDataCopy	make copy of message data
msgDataKeep	keep message data
msgVarPtrSet	set pointer to variables
msgVarPtrGet	get pointer to variables

- Special Processing

msgAckAINIT	acknowledge AINIT
msgDefaultProc	default processing for msgs

⟹ See 'msgLib' manual pages for details of these functions

msgBuild Function

```
MSG_TYPE  *msgBuild ( MSG_TYPE  *msg      ,
                      TID_TYPE   dest     ,
                      TID_TYPE   source   ,
                      CMD_TYPE   command  ,
                      void      *data     ,
                      int        datasize ,
                      FLAG_TYPE  flags     )
```

```
MSG_TYPE  *msg      - pointer to message struct or NULL
TID_TYPE   dest     - address of destination task
TID_TYPE   source   - address of task sending message
CMD_TYPE   command  - message command
void      *data     - pointer to additional message data
int        datasize - number of bytes in message data
FLAG_TYPE  flags    - message flags
```

msgBuild provides a convenient way to define a message. The arguments to msgBuild are used to define the members of the message structure, whose address is passed in as the first argument. If *msg == NULL then msgBuild will allocate storage.

RETURNS: Pointer to message that was built

msgFlagSet Functions

```
msgMemownerFlagSet - set MEMOWNER field of flag
msgPriorityFlagSet - set PRIORITY field of flag
msgReplyFlagSet    - set REPLY_WAIT field of flag
msgSendFlagSet     - set SEND_WAIT field of flag
msgTypeFlagSet     - set TYPE field of flag
```

```
FLAG_TYPE  msgMemownerFlagSet (base_flag, field)
FLAG_TYPE  msgPriorityFlagSet (base_flag, field)
FLAG_TYPE  msgReplyFlagSet    (base_flag, field)
FLAG_TYPE  msgSendFlagSet     (base_flag, field)
FLAG_TYPE  msgTypeFlagSet     (base_flag, field)

FLAG_TYPE  base_flag   - base flag
FLAG_TYPE  field       - new value for flag field
```

These functions are used to manipulate the fields of a message .flags member. The actions of these functions are to replace the particular field of the base flag with a new value. For instance, the following function calls change the MEMOWNER field:

```
msg.flags = msgMemownerFlagSet (msg.flags, MF_MEMOWNER_SENDER);

msg.flags = msgMemownerFlagSet (MF_STANDARD, MF_MEMOWNER_SENDER);
```

RETURNS: flag resulting from changing 'field' of 'base flag'

msgSend Function

```
int  msgSend (MSG_TYPE *msg)

MSG_TYPE  *msg - pointer to message to be sent
```

msgSend is the most basic form of message passing, and the most frequently used. The message pointed to by the function argument contains all of the information needed by msgSend to route and handle the message.

RETURNS:

If reply flag set to REPLY_WAIT_NO:

OK message was successfully sent out.

ERROR error occurred during message passing; or if SEND_WAIT_NO is set, MsgDispatcher is busy or destination task's queue is full.

If reply flag other than REPLY_WAIT_NO:

msgSend returns *data from reply message (cast as an integer).

msgReply Function

```
STATUS  msgReply ( MSG_TYPE   *msg       ,
                   void       *data      ,
                   int         datasize  ,
                   FLAG_TYPE   flags      )
```

```
MSG_TYPE   *msg       - pointer to received message
void       *data      - pointer to reply data
int         datasize  - size of reply data
FLAG_TYPE   flags     - message flags
```

The msgReply function is used to reply to a received message. Its primary uses are to respond to requests, and to acknowledge synchronization messages.

The data pointed to by *data of msgReply is sent via the reply message and is received by the (now unblocked) originating task as the return value of msgSend. However, the *data pointer is ignored when replying to a broadcast message; so msgSend must be used (AFTER acknowledging the broadcast if required).

RETURNS: OK or ERROR indicating success of sending out reply

msgTidQuery Function

```
TID_TYPE  msgTidQuery (TID_TYPE tid, char *taskname)
```

```
TID_TYPE tid        - task id of task calling msgTidQuery

char    *taskname   - symbolic name of task whose TID is
                      sought
```

The msgTidQuery function sends a message to the Tid Server on CPU 0 requesting the TID of the task with symbolic name *taskname.

While msgTidQuery is waiting for a reply, the task that called msgTidQuery is blocked. As there is a potential delay, msgTidQuery should not be used within a fast synchronous process, except during initialization.

RETURNS

If query is successful, returns TID of *taskname.

If query does not succeed, returns 0.

msgTidSet & msgTidGet Functions

```
TID_TYPE  msgTidGetChassis (TID_TYPE tid)
TID_TYPE  msgTidGetCpu     (TID_TYPE tid)
TID_TYPE  msgTidGetLocal   (TID_TYPE tid)
TID_TYPE  msgTidSetChassis (TID_TYPE tid, int number)
TID_TYPE  msgTidSetCpu     (TID_TYPE tid, int number)
TID_TYPE  msgTidSetLocal   (TID_TYPE tid, int number)

TID_TYPE  tid    - task id to be manipulated
int       number - new value of TID field
```

These functions are used to access the fields of a TID. For instance, msgTidSetCpu will set the CPU field of a TID to a specified value, and msgTidGetCpu will return the value of the CPU field.

These functions are implemented as macros, and the msgTidSet functions will directly change the TID value. Hence, the following are legal statements and are equivalent:

```
msg->dest = msgTidSetCpu (msg->dest, 0) ;
msgTidSetCpu (msg->dest, 0) ;
```

RETURNS:

msgTidGet functions: value of the TID field

msgTidSet functions: whole TID after setting field

Structuring An Application

- Identify major operations & data flows
  - use standard software engineering techniques
- Group operations into tasks
  - logically group family of related operations into one task
  - concurrent operations should be separate tasks
  - consider single manager task for operations that must be serialized
  - assign unique symbolic name to each task
- Describe inter-task communications
  - define messages & data being passed
  - roughly, each message corresponds to a different operation
  - draw a diagram showing tasks & message exchanges
  - identify communication partners (who sends message and who receives it)
  - keep high volume communications on same CPU if possible, or at least same chassis

Structuring An Application, Con't

- Write event handler functions
  - design "application executive" to perform main execution sequence
  - build event handlers to interface to synchronous processes
- Build configuration files
  - assign tasks to CPUs
  - dependencies of function calls determine order to load object modules

Event Handler Tasks

- An event handler task consists of
  - event handler shell (with stack)
  - event handler function
  - message queue
  - storage for reply message
  - semaphore to wait for replies
  - pointer to saved variables
- Event handler shell manages the message queue and message data
- Event handler shell calls the event handler function when there is a message to process
- Event handler function is given TID of current instantiation of the function and pointer to the message
- Event handler function exits to shell after processing each message

Format of Event Handler Function

```
int  FunctionName (TID_TYPE myTid, MSG_TYPE *msg)
  {

  switch (msg->command)
    {
    case MSG_AINIT:
      /*  application initialization   */
      break ;

    case MSG_ONE;
      /*  process message one   */
      break ;

    case MSG_TWO;
      /*  process message two   */
      return (0) ;
    }

  /*  default processing of commands   */
  return (msgDefaultProc (myTid, msg));
  }
```

Tid Server Event Handler Function

```
int  btsTidSvr (TID_TYPE myTid, MSG_TYPE *msg)
  {
  TASKREC  *task ;
  TID_TYPE  result ;

  switch (msg->command)
    {
    case MSG_REGISTER_TID:
        /* add tid to symbol table */
        task = (TASKREC *) msg->data ;
        symAdd (tidTbl, task->name, &task->tid, 0)
        return (0) ;

    case MSG_QUERY_TID:
        /* find tid in symbol table */
        if (symFindByName (tidTbl, msg->data, &result,
                           NULL) == ERROR)
            result = 0 ;
        msgReply (msg, (void *)result,
                  MS_KEEP_ADRS, MF_STANDARD) ;
        return (0) ;
    }

  /* default processing of commands */
  return (msgDefaultProc (myTid, msg)) ;
  }
```

Request to Tid Server

```
TID_TYPE  msgTidQuery (TID_TYPE myTid, char *taskname)
  {
  MSG_TYPE  msg ;

  /*  send message to TID Server  */
  msgBuild (&msg,                 /* message  */
            TIDSVR,               /* dest     */
            myTid,                /* source   */
            MSG_QUERY_TID,        /* command  */
            taskname,             /* *data    */
            sizeof(taskname),     /* datasize */
            MF_REPLYWAIT          /* flags    */
            ) ;

  /*  return TID in reply message  */
  return ((TID_TYPE) msgSend (&msg)) ;
  }
```

msgDefaultProc()

- msgDefaultProc function provides default processing of system messages, such as PINIT and AINIT; plus acknowledges REPLY_WAIT messages

- Most event handler functions will have a similar format with switch/case statements used to decode the msg.command, and a call to msgDefaultProc at the end

- When the event handler function is ended by the recommended

  ```
  return( msgDefaultProc (tid, msg) )
  ```

  - ending case statement with <u>break</u> will cause a call to msgDefaultProc
  - ending case statement with <u>return(0)</u> will bypass default processing

- When an application fails to boot and run, a highly likely cause is an event handler task improperly responding to a system message due to bypassing msgDefaultProc

Reentrant Event Handler Functions

- Any number of event handler tasks can be created with the same event handler function provided:
  - task symbolic name is unique, and
  - event handler function is reentrant
- <u>Local variables are OK</u> because each task has its own stack
- Functions with <u>no static variables</u> are reentrant
- If need static variables
  1. define structure to hold all static variables
  2. during PINIT, allocate memory for static variable structure and initialize its members
  3. while still in PINIT, save pointer to this structure with msgVarPtrSet function
  4. use msgVarPtrGet function to retrieve pointer to static variable structure (may want before switch statement)

CTOS Bootstrap Synchronizes Startup by Stepping Through Phases

- Process Initialization Phase (PINIT)
  - can initialize an individual process
  - other processes may not yet exist
- Application Initialization Phase (AINIT)
  - all processes guaranteed to exist and to have completed PINIT phase
  - use msgTidQuery function to find TID of communication partners
  - can perform initialization between processes
- Application Execution Phase (AEXEC)
  - all processes guaranteed to have completed AINIT phase
  - begin execution of application when receive AEXEC message
  - likely will have only one task controlling the application (the application executive)

Default Processing of "Phase Messages"

- Bootstrap phase is begun via a broadcast message
  - CTOS_Boot is blocked while REPLY_WAITing
- Phase ends when all tasks have acknowledged the broadcast message
  - if one task fails to acknowledge it will block the whole application
  - for this reason it is important to call msgDefaultProc to ensure that all system messages are properly processed
- If you want to defer acknowledging completion of AINIT phase:
  1. end case AINIT with return(0) to bypass default acknowledgement
  2. complete application initialization processing
  3. explicitly acknowledge AINIT with msgAckAINIT function

Application Executive

- CTOS_Boot task controls initialization
  - loads application software & creates application tasks
  - broadcasts messages to start bootstrap phases
- Application executive task controls main execution sequence of the application
  - user writes new application executive for each new application
  - may be the only task that responds to AEXEC message
  - responsible for coordination of application tasks
  - provides synchronization if additional phases are needed
  - commonly will send messages to itself to provide opportunity for in-coming messages to get through
  - alternatively, can use queue management functions to access its message queue

Synchronous Service

presented by
Jim Watson

Other resources for follow-up information:

- Tech Memo #4
- On-line man pages (TBD)
- Kevin Holt and Dave Swift—designers of the robot channel drivers.

C-2

Outline

- Purpose
- Data- vs. Time-Synchronization
- Design And Implementation
  - desired functionality
  - architecture
  - PØ
  - LSPH
- Booting The Synchronous Service
- Initializing The Synchronous Service
- Use With Message Passing

Purpose

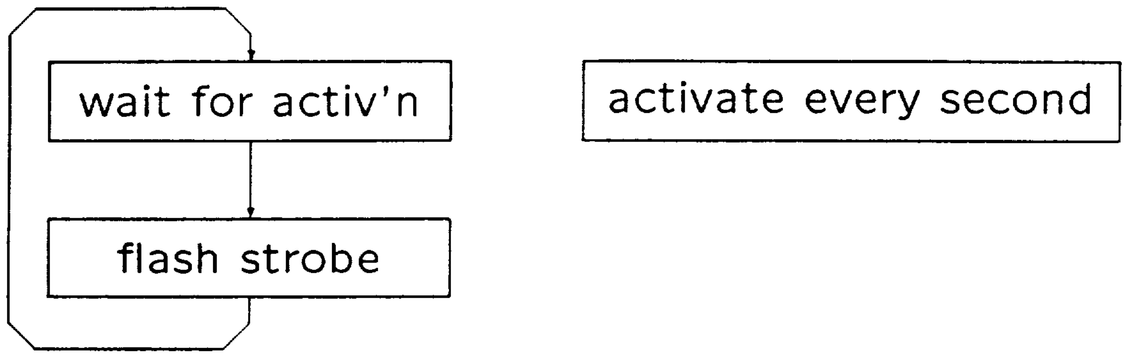
Primary purpose: provide a paradigm that supports high speed, low latency, *time*-synchronization of multiple processes distributed throughout the VME Cage. Typically, processes using this service will require synchronization every 5–40ms, although synchronization periods have no practical upperbound.

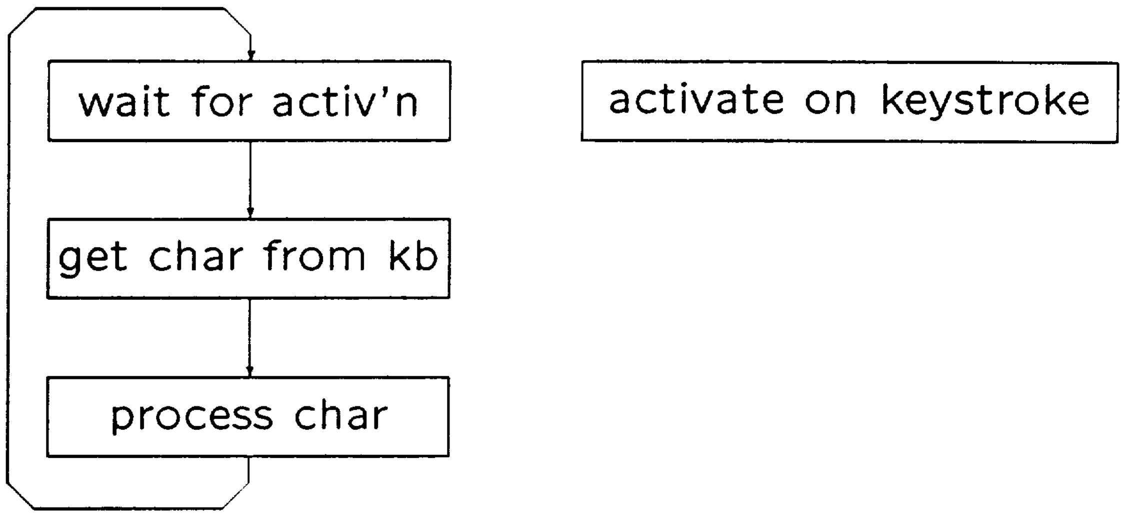
Secondary purpose: to maintain a system clock on the VME Cage.

Data- Vs. Time-Synchronization

The distinction is what makes the process *runable*.

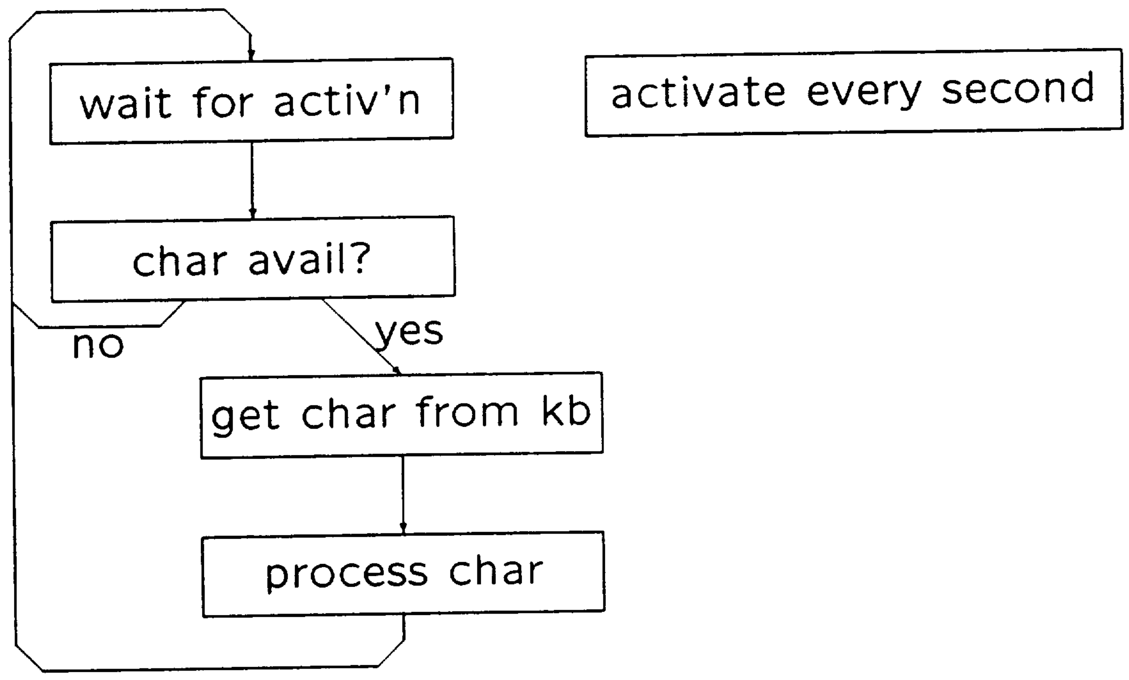
Example:





Data- Vs. Time-Synchronization

Data-synchronized processes can be forced to be time-synchronized. Risk losing data and/or wasting resources.



Data- Vs. Time-Synchronization

Robotic example:

- PUMA joints angles read & torques written every 5ms by the PUMA I/O driver
- 6 joint PID controller
- an independent safety process, running in the background, checks actual PUMA position every 500ms

The PUMA I/O driver and safety process are *time*-synchronous. The PID, although in lock-step with the I/O driver, is *data*-synchronous.

Design And Implementation

Desired functionality:

- System clock
- High speed, low latency synchronization of distributed processes
- Starting/stopping synchronization on-the-fly
- Detecting faults within synchronous service
- Detecting/processing overruns in user code
- Mechanism to control scheduler loading
- Compatiability with simulations and "real" experiments
- Aid debugging
- Minimal hardware resources (clocks, interrupts, etc.)

Design And Implementation

Architecture:

- Maintain local information on each CPU 1–4, with CPU 0 as master.
- Attach ISR on CPU 0 to the auxiliary clock chip. Choose an appropiate interrupt rate, and have CPU 0 maintain system clock, which is stored in global memory.
- System clock can be accessed by user on all CPUs via function call that extracts clock value from global memory.
- CPU 0 generates bus interrupt (using LM interrupt). This serves as the synchronization heartbeat.
- ISRs on CPUs 1–4 respond to LM interrupt and manage local synchronous process activations. The ISR is the guts of the local synchronous process handler (LSPH).
- Functions on CPUs 1–4 provide user with interface to the LSPH.

Design And Implementation

PØ, (the ISR on CPU 0):

- Responds to clock chip interrupts throughout entire experiment.
- Maintains state flags for system clock on/off & LM interrupt enabled/disabled.
- Time Units: MCS-TU = 0.1ms. Time stored in system clock is an integer number of MCS-TUs.
- Clock Update Rate: MCS-CUR = 0.9ms. Period between clock chip interrupts.
- Time Scale: MCS-TS, integer ≥ 1, set by user (typically 1). Number of clock chip interrupts between system clock updates and LM interrupts. Thus, factor between real-time and system-time.
- Time Phase: MCS-CP, integer ≥ 1, set by user (typically 1). Number of clock chip interrupts before the first ISR action.
- Small errors in system clock can occur due to hardware limitations (see Tech Memo).

PØ: *State Flags*

clock on

LM interrupts enabled

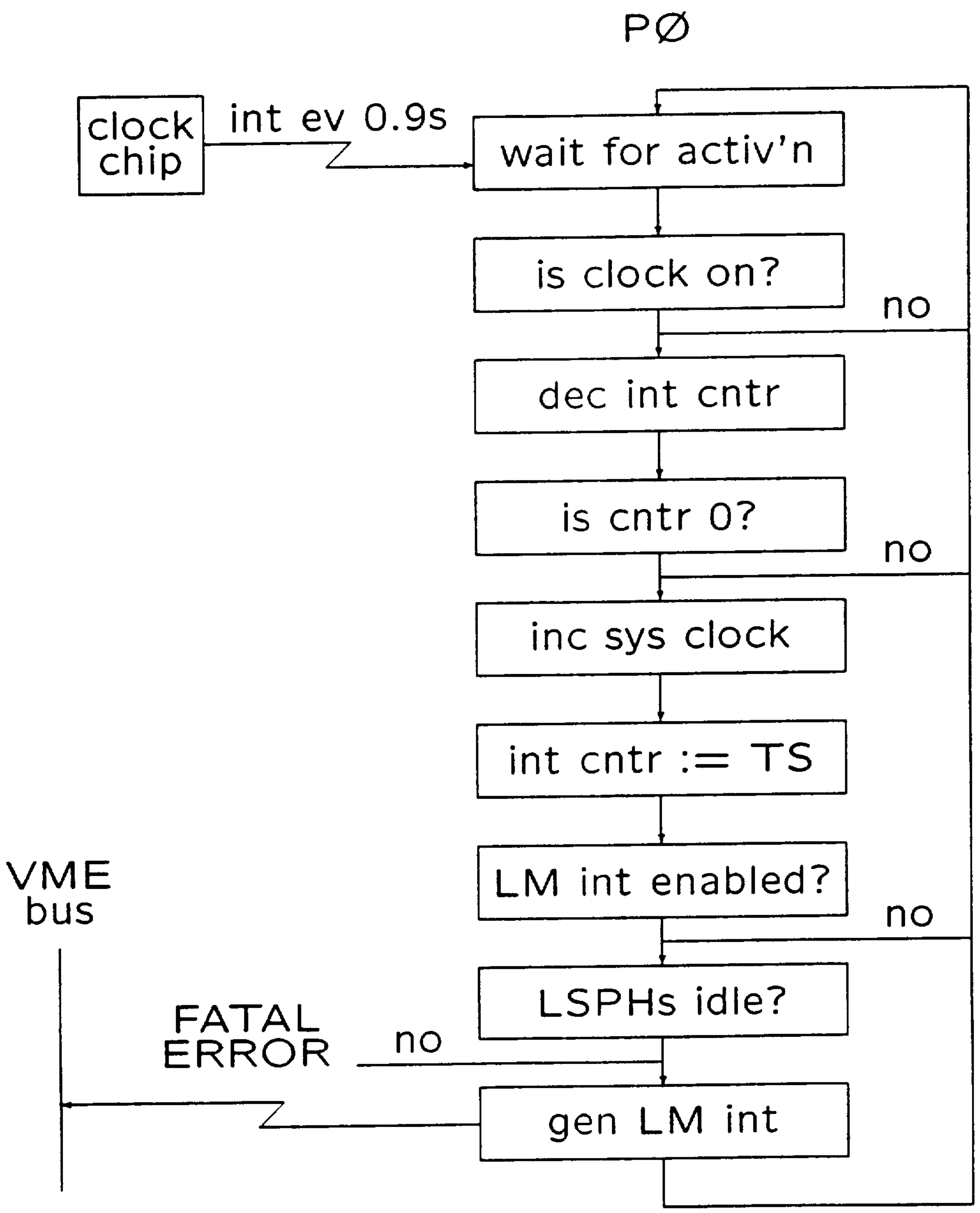
clock on

LM interrupts disabled

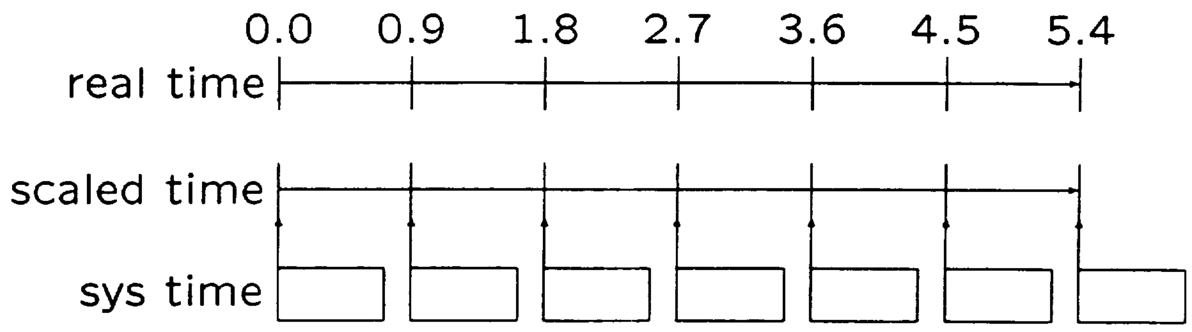
clock off

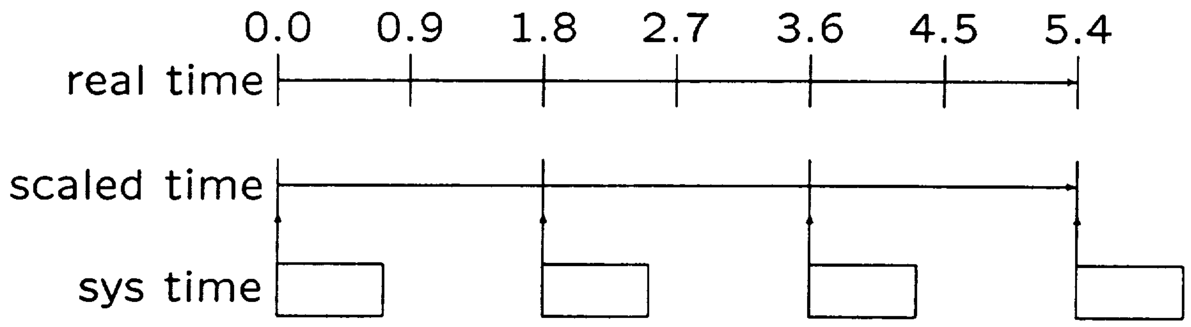
LM interrupts disabled

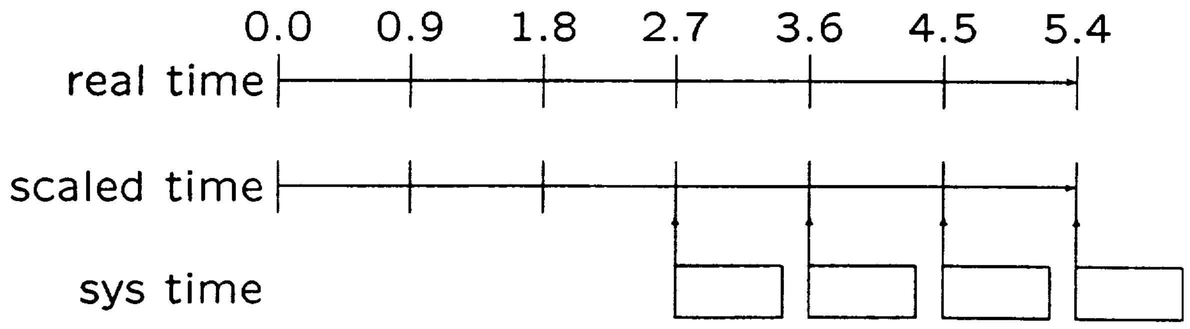
PØ: *Interrupts*



PØ: *System Clock*







Design And Implementation

LSPH (Local Synchronous Process Handler):

- Responds to LM interrupts and manages synchronous processes on local CPU.
- A synchronous process (referenced by handle) includes:
  - a synchronous task and synchronous semaphore
  - an overrun task and overrun semaphore
  - a synchronization period and phase
  - a running flag
  - status registers and data (synchronization enabled/disabled, disable pending and disable time, overrun pending and overrun time)
- Contains a functional interface for user to add/delete/control synchronous processes.

LSPH

Two ways of attaching a synchronous process:

- More flexible, low-level function:
  - synchronous and overrun tasks are spawned by user with taskSpawn with arbitrary parameters (e.g., stack size, priority)
  - semaphores are created by user with semCreate
  - user provides LSPH with these IDs, running flag, phase, and period
  - user gets synchronous handle

```
SYNC_HANDLE syncProcAttach(
        SEM_ID sync_sem,  int sync_task_id,
        SEM_ID or_sem,    int or_task_id,
        BOOL *flag,  int phase,  int period )
```

LSPH

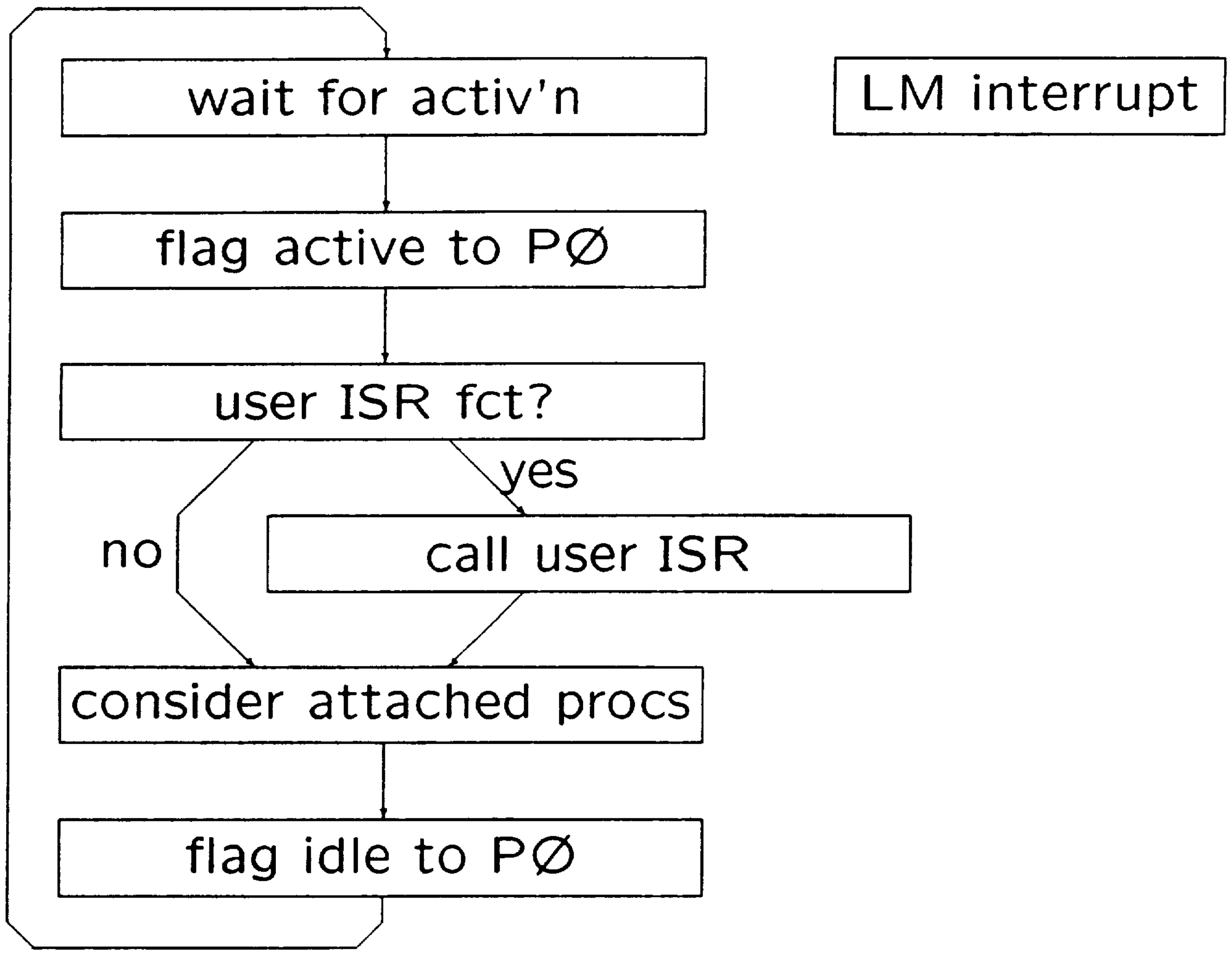
- Less flexible, high-level function:
  - tasks are spawned using default parameters and semaphores are created by LSPH
  - user minimally provides function to be spawned as synchronous task, symbolic name, running flag, phase, and period (LSPH uses default arguments for task spawn and attaches a default overrun task)
  - user may provide overrun function to be used
  - user may provide one argument to be passed to the synchronous and overrun tasks

```
SYNC_HANDLE syncProcSpawn(
        SEM_ID *pSync_sem,  VOIDFUNCPTR pSync_func,
        char *pSync_name,   int sync_arg,
        SEM_ID *pOr_sem,    VOIDFUNCPTR pOr_func,
        char *pOr_name,     int or_arg,
        BOOL *flag,  int phase,  int period )
```

Synchronous (and overrun) task called with arguments:

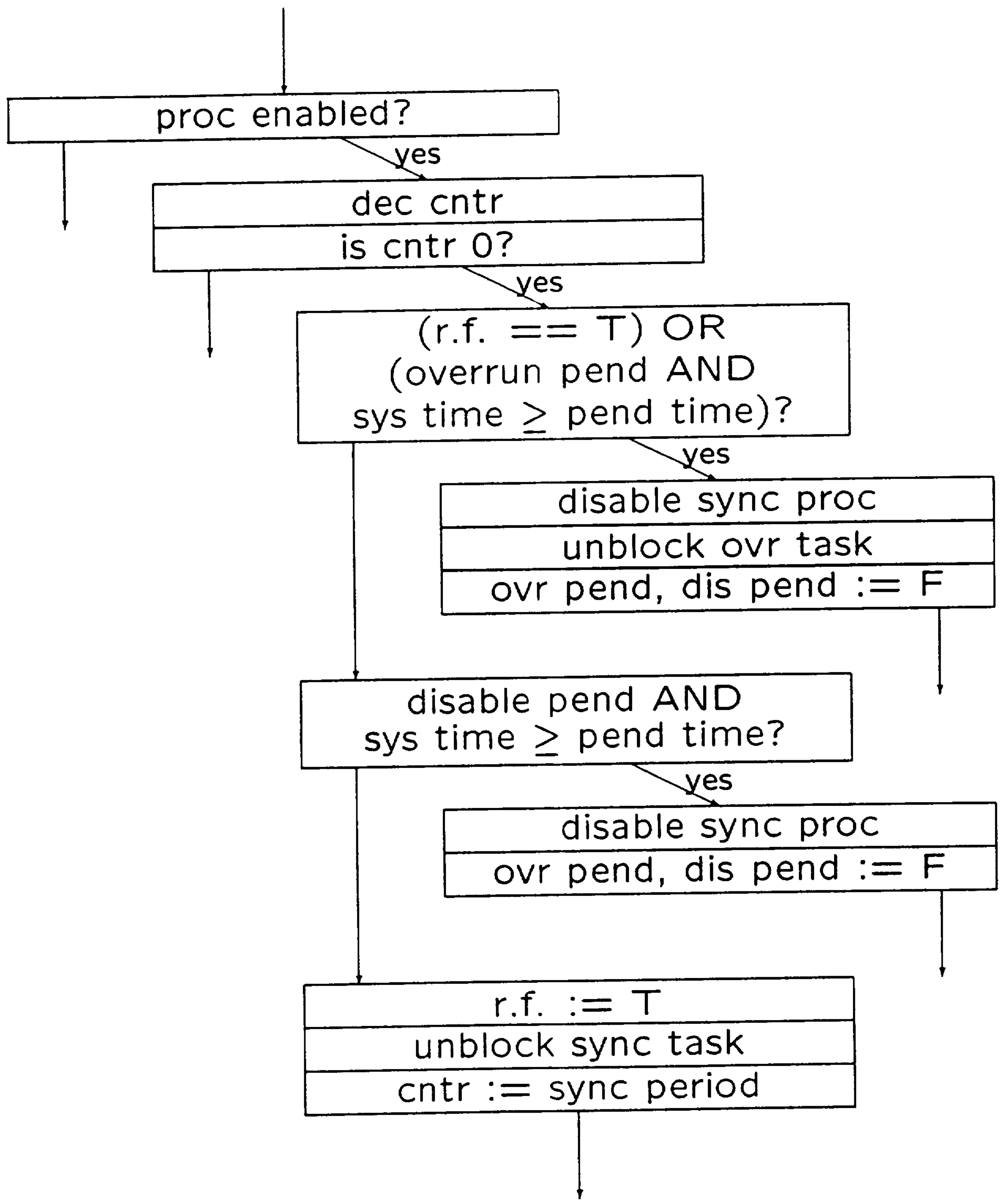
```
     syncFuncName(  SEM_ID syncSem,  BOOL *rf,
                    int sysProcNum,  int syncOptArg  )
```

LSPH

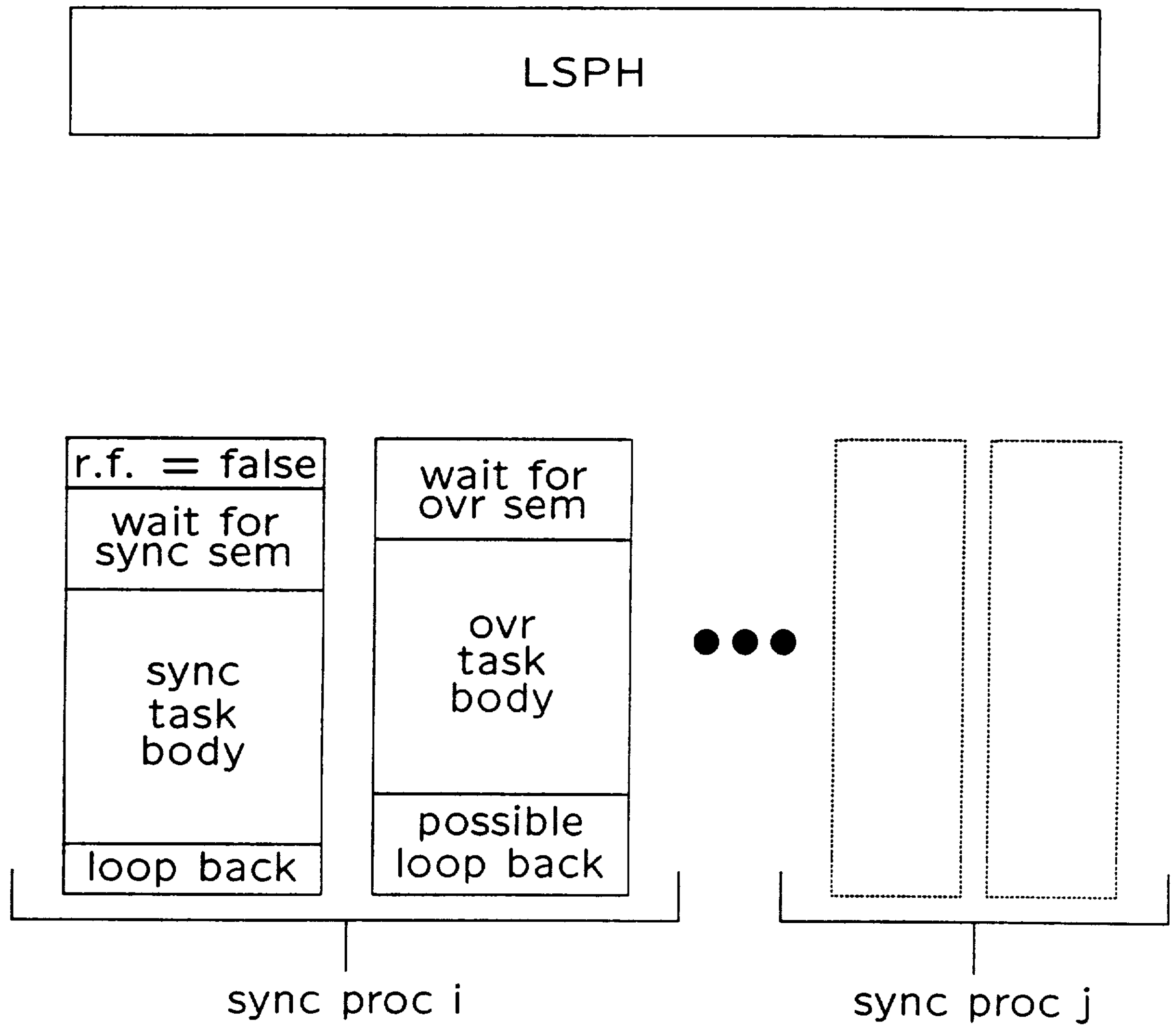


Using ISR Voids Warranty

LSPH: *attached procs*



LSPH


LSPH
r.f. = false
wait for sync sem
sync task body
loop back
wait for ovr sem
ovr task body
possible loop back
sync proc i
sync proc j

LSPH

- Once a synchronous process is enabled, a detected overrun disables it and unblocks the overrun task.
- Disabling and re-enabling can be done by the user.
- Overruns can be “forced” by the user.
- Disables and forced overruns are time-stamped.
- High level functions are provided to do task spawns and semaphore creations for the user.
- Low level functions allow user much more flexibility but with less hand-holding.

Booting The Synchronous Service

These loads are performed by the CTOS System Configuration File:

- Want the clock and shut-down functions available on all CPUs

```
-1 load   syncSupport.o
```

- Want PØ on CPU 0

```
0  load   syncMaster.o
```

- Want the LSPH on CPUs 1–4

```
1  load   syncLib.o
2  load   syncLib.o
3  load   syncLib.o
4  load   syncLib.o
```

Booting The Synchronous Service

These spawns are performed by the CTOS System Configuration File:

- Want PØ Message Handler activated on CPU 0

```
0  task  p0          syncP0MsgHandler      50
```

- Want the LSPH Message Handlers activated on CPUs 1–4

```
1  task  Lsph_Svr1  syncLsphMsgHandler   50
2  task  Lsph_Svr2  syncLsphMsgHandler   50
3  task  Lsph_Svr3  syncLsphMsgHandler   50
4  task  Lsph_Svr4  syncLsphMsgHandler   50
```

Initializing The Synchronous Service

- PØ and LSPHs initialize data structures in response to `MSG_CINIT`. Additionally, the LSPHs notify PØ, using `MSG_SYNC_CPU_CHECK_IN`, that they will be responding to LM interrupts.

- PØ requires that the phase and time-scale be set for the system clock prior to turning it on. Messages are used for this:

  - `MSG_SYNC_CLK_RESET`
  - `MSG_SYNC_CLK_PHASE_SET` (integer data)
  - `MSG_SYNC_CLK_SCALE_SET` (integer data)

- Messages to PØ are used for the *empowering* state changes of the system clock:

  - `MSG_SYNC_CLK_ON`
  - `MSG_SYNC_CLK_PROC_ON`
  - `MSG_SYNC_CLK_PROC_ENB`

Use With Message Passing

- Messages are used by CTOS to initialize the synchronous service.
- The synchronous service uses messages to establish communication between PØ and the LSPHs.
- Messages can be used between event handler tasks to establish and control synchronous processes.
- Synchronous processes can be used to periodically generate messages.

Inter-processor Blocks (IPB)

- VxWorks Semaphores do not work between processors on a VME chassis

- While there are primitives (*e.g.* `sysBusTas`) that can be used to construct semaphores, they have disadvantages

  - They must be polled in order to block the "taking" process, this could either flood the bus, or if delays are used, introduce unacceptable latencies

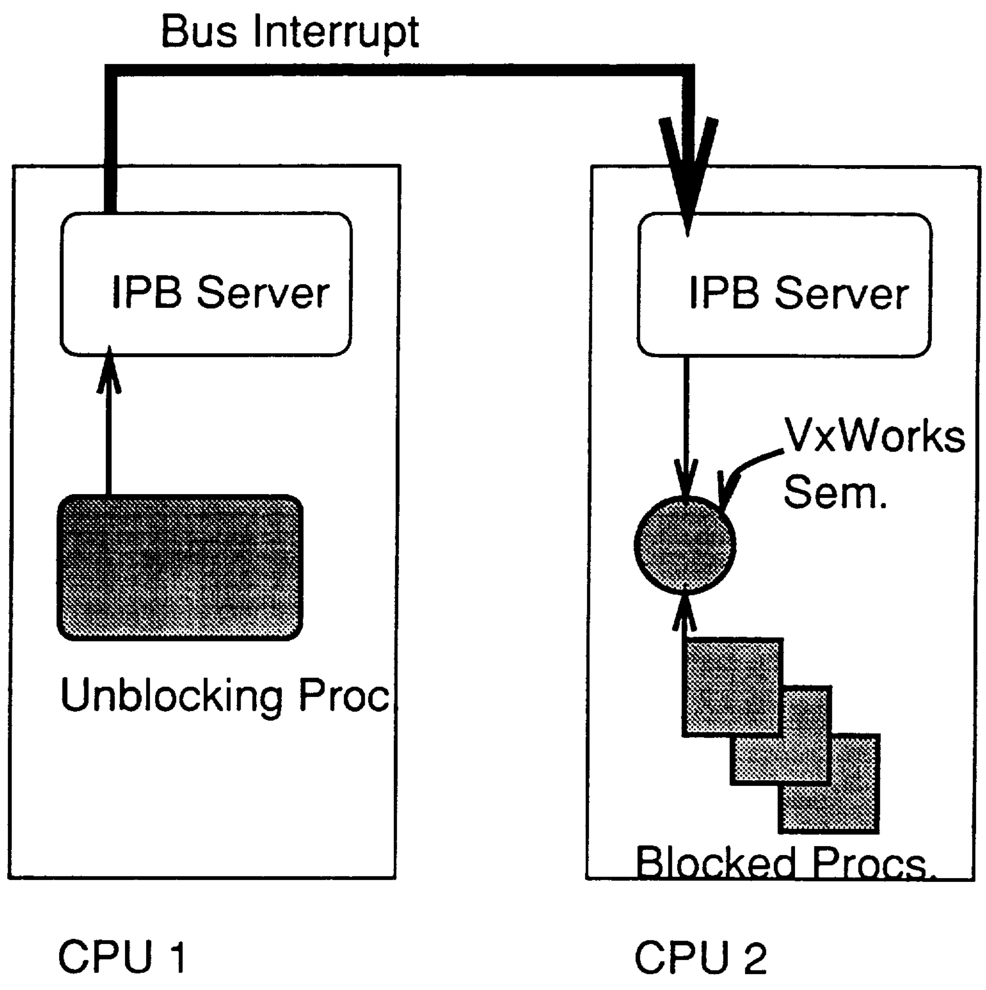
  - The polling process remains "ready" rather than blocked

IPB Functions

IPBs attempt to eliminate these problems by utilizing the VxWorks semaphore library and bus interrupts:

```
IPB_FLAG ipbCreate(IPB_STATE init)
IPB_STATE     init       - the initial state of the IPB
                           IPB_CLEARED or IPB_BLOCKED

IPB_FLAG ipbTake(IPB_FLAG flag)
IPB_FLAG      flag       - the IPB flag to take

void ipbUnblock(IPB_FLAG flag, IPB_STATE state)
IPB_FLAG      flag       - the flag to Unblock
IPB_STATE     state      - the state to leave the flag in
                           after unblocking. (IPB_CLEARED
                           or IPB_BLOCKED
```

IPB Implementation



CTOS/MCS

Section III: MCS

Motion Control System – Introduction

- Designed to be the interface to the manipulators of the CIRSSE testbed
- Effort kicked off in November, 1990 and started in earnest in January 1991 by the MCS design team.
- Basic functionality (with the exception of a complete TG) in place by early July 1991
- Continued effort to enhance and complete MCS and complete its integration with the CIRSSE Intelligent Control System

Motion Control System – Features

- Designed as a control server and and as a testbed for control research

- Individual component interface designed to allow easy replacement for research.

- Developed on top of (and in conjunction with) CTOS, thus providing for seamless integration with the rest of the CIRSSE Integlligent Control System

- Provides a convenient, well understood framework for testbed software development

Motion Control System – Components

A functioning Motion Control System is configured by including several MCS components and an application manager.

The application manager may function as the driver for a particular experment. Or, it may act as a "client interface" to systems outside of the MCS, such as the Coordinator.

Motion Control System – Components

MCS State Manager Monitors and maintains the state of the Motion Control System. Provides the implementation of the interface between the application and the other MCS components

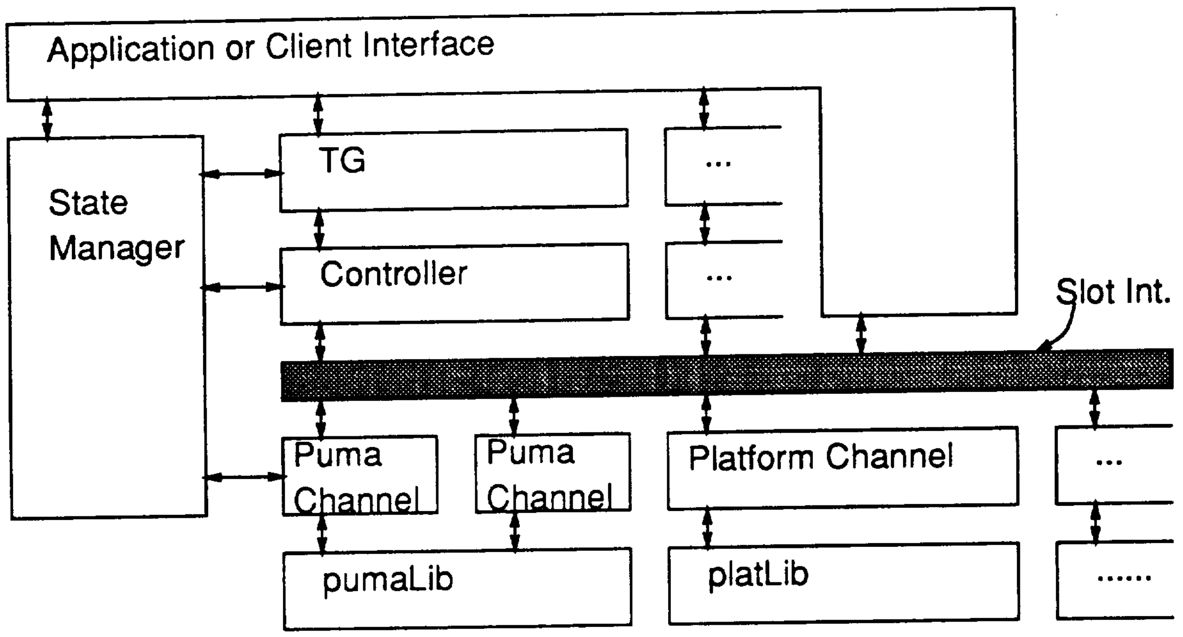
Channel Drivers Low level interface between the hardware that the MCS controls and higher levels of the MCS Hierarchy. Maps MCS "slots" to I/O areas on the hardware

Controllers Provides control for those MCS slots which require it

Trajectory Generator Provides trajectory generation for those slots which require it

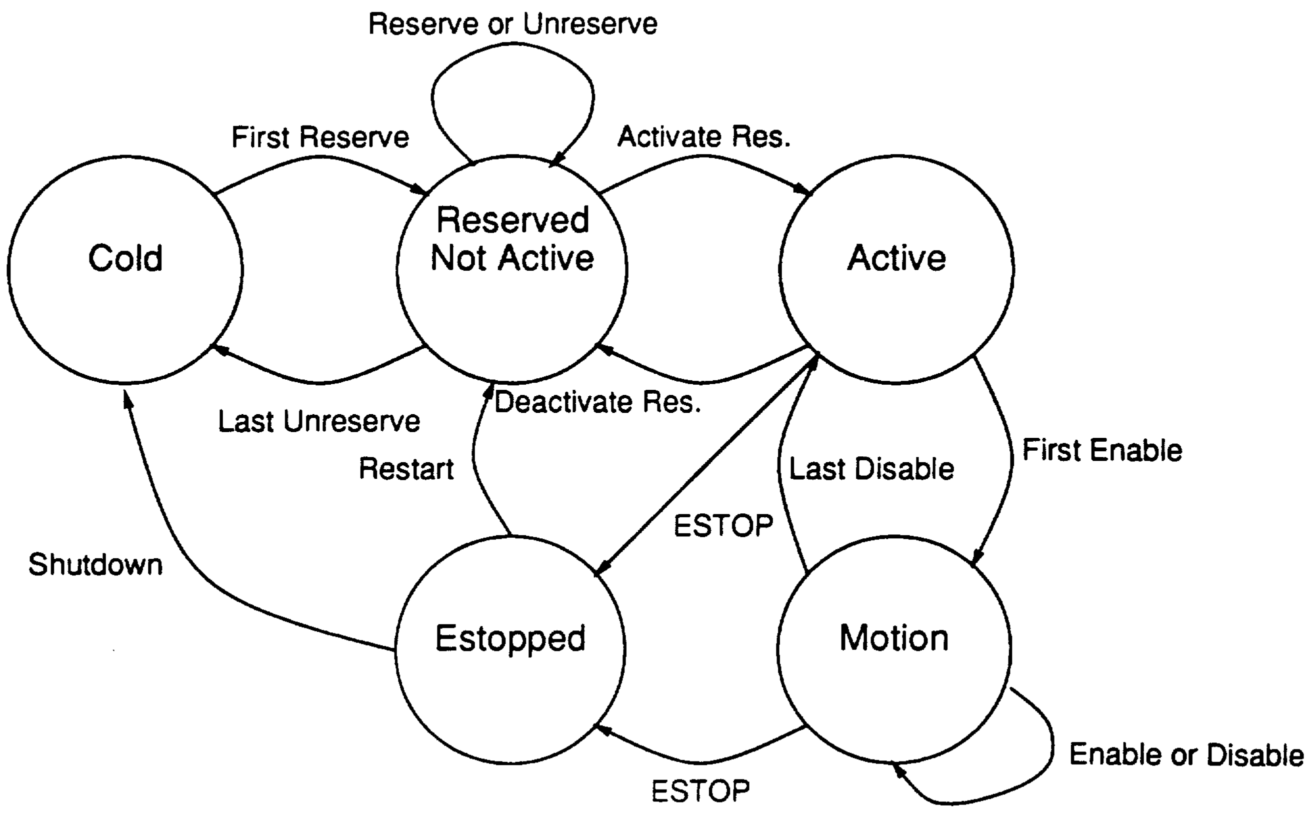
Note that all of the components may be allocated and distributed as the user wishes using the CTOS Configuration mechanism.

Motion Control System – Components



Motion Control System – State Diagram

To interact with the Motion Control System, an application makes transitions along the MCS State Diagram:



Motion Control System – State Manager Messages

MSG_PINIT

- Initializes data structures
- Reponds to registration by Channels, Controllers, and TGs. Channels describe slots

MSG_AINIT

- Creates IPBs for each channel, then distributes correct IPBs to the appropriate controllers
- Sends initial timing information to channesl an TG
- Notifies MCS Components that they may establish their default configuration

MSG_AEXEC

- Responds to other State Manager/mcsLib messages

Motion Control System – State Manager Messages

MSG_MCS_component_GET Returns a MCS_SLOT_LIST filled with the TIDs of the requested component (TG, CONTROLLER, CHANNEL)

MSG_MCS_RATE_GET Returns a MCS_SLOT_LIST filled with the rates at which the slots are being servoed

MSG_MCS_RESERVE Notes the reservation of the slot

MSG_MCS_ACTIVATE

- Calibrates any reserved slots that should be.
- Ensures that power has been enabled for all reserved slots
- Allows positioning of slots that are capable of it (and are reserved)

Motion Control System – State Manager Messages

MSG_MCS_ENABLE

- If this is the first ENABLE, notify slot's TG, CHANNEL and CONTROLLER that MCS is moving into the Motion state
- Notify slot's CHANNEL that slot has been enabled
- Notify TG that slot has been enabled

MSG_MCS_DISABLE

- If this is the last DIABLE, notify slot's TG, CHANNEL and CONTROLLER that MCS is moving out of the Motion state
- Notify TG that slot has been disabled
- Notify slot's CHANNEL that slot has been disabled

MSG_MCS_DEACTIVATE

Notifies active channels to disable power

MSG_MCS_UNRESERVE Notes the unreservation of the slot

Motion Control System – mcsLib

In general, an application does not explicitly send messages to the MCS State Manager. Rather, an application can use *mcsLib* functions, which encapsulate the sending of the appropriate messages to the State Manager.

Motion Control System – mcsLib

```
MCS_STATUS mcsSlotReserve(TID_TYPE callTid, int slot);

MCS_STATUS mcsSlotUnreserve(TID_TYPE callTid, int slot);

MCS_STATUS mcsReservationsActivate(TID_TYPE callTid);

MCS_STATUS mcsReservationsDeactivate(TID_TYPE callTid);

MCS_STATUS mcsSlotEnable(TID_TYPE callTid, int slot);

MCS_STATUS mcsSlotDisable(TID_TYPE callTid, int slot);

TID_TYPE  callTid      - TID of the calling task
int       slot         - slot of interest
```

Motion Control System – mcsLib

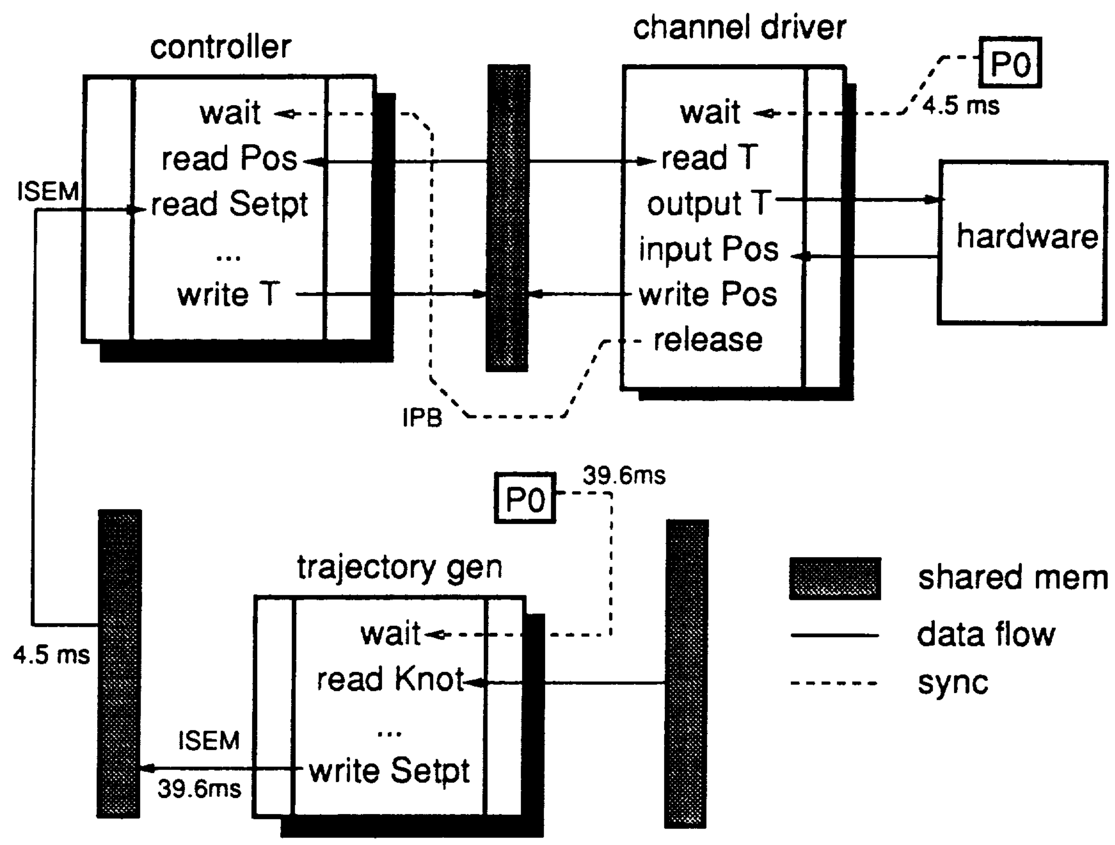
The mcsLib functions that return information obtained from the State Manager use a specially defined data type called an MCS_SLOT_LIST:

```
typedef union  {
    TID_TYPE tid[MCS_MAX_SLOTS];
    INT      period[MCS_MAX_SLOTS];
    INT      phase[MCS_MAX_SLOTS];
    REAL     position[MCS_MAX_SLOTS];
    BOOL     bool[MCS_MAX_SLOTS];
    MCS_SLOT_WORD slotWord[MCS_MAX_SLOTS];
    MCS_STATUS status[MCS_MAX_SLOTS];
    } MCS_SLOT_LIST;
```

Motion Control System – mcsLib

```
MCS_STATUS mcsChannelGet(TID_TYPE callTid,
                         MCS_SLOT_LIST *slotList);

MCS_STATUS mcsControllerGet(TID_TYPE callTid,
                            MCS_SLOT_LIST *slotList);

MCS_STATUS mcsTGGet(TID_TYPE callTid,
                    MCS_SLOTS_LIST *slotList);


TID_TYPE      callTid   - TID of the calling task
MCS_SLOT_LIST *slotList - Pointer to storage for a list
                          or NULL
```

MCS Synchronous Interface



Channel Driver Overview

- Purpose of the Channel Drivers
- Channel Driver Message Handler
- Channel Driver Synchronous Task and Overrun Task
- Channel Driver Interfaces
- Current Implementation of chanPuma and chanPlat
- Future Developments and Additions

Purpose of the Channel Drivers

- Interface between hardware and controllers
- Handles the synchronization for discrete control
- Handles error conditions
- Creates a device independent layer (for controller interface)

Channel Driver Message Handler

- Purpose/Features
  - Handles asynchronous messages from state manager
  - Transfers message information to sync task
  - Handles failure of state manager
  - Initializes data and hardware

Channel Driver Messages

- PINIT
  - Initialize data
  - Register joints
- TIME_SET (during AINIT)
  - Set channel driver period and phase
- IPB_SET (during AINIT)
  - Set channel driver interprocessor block flag

Channel Driver Messages (cont.)

- DEFAULT_CONFIG (during AINIT)
  - Check hardware
  - Spawn sync task and overrun task
  - Install channel driver
- CONFIG_GET/CONFIG_SET (?)
  - Not Defined

Channel Driver Messages (cont.)

- CALIBRATE (one time only)
  - Turn on high power
    (for joint channel drivers)
  - Calibrate hardware
- PREPARE_MOTION
  (transition into activate state)
  - Prepare robot for motion state
  - Turn on high power if not already on
    (for joint channel drivers)
  - Update shared memory

Channel Driver Messages (cont.)

- POSITION (in activate state)
  - Position the robot using hardware
  - Not supported by all hardware
  - Update shared memory

Channel Driver Messages (cont.)

- MOTION (transition into motion state)
  - Enable clocking of sync process
  - Joints do not move until an ENABLE is received
- ENABLE (enable selected joint for motion)
  - Enable joint for motion
  - Brakes off for selected joint
  - One at a time

Channel Driver Messages (cont.)

- DISABLE (disable selected joint)
  - Disable joint motion
  - Brakes on for selected joint
  - One at a time
- NO_MOTION (transition out of motion state)
  - Disable clocking of sync process

Channel Driver Messages (cont.)

- DEACTIVATE
  (transition out of activate state)
  - Turn off high power
    (for joint channel drivers)

Channel Driver Messages (cont.)

- ESTOP (any time after AEXEC)
  - Software ESTOP
  - Stop all joints
  - Turn off high power
- KILL (any time after AEXEC)
  - Remove channel driver

Channel Driver Synchronous Task

- Purpose/Features
  - Gathers data from hardware
  - Outputs data to hardware in a synchronous fashion
  - Handles hardware to software conversions (encoder ticks to radians, etc.)
  - Releases data driven tasks when data is available (data sync mode)
  - Handles controller data write delays (ex. torque not fresh)
  - Alerts state manager of "forced" state transitions

Channel Driver Synchronous Task

- Purpose/Features (cont.)
  - Stops joints before they hit hardware limits
  - Checks for ESTOP
  - Stops robot when an overrun occurs or hardware fails

Synchronous Task Code

(1) One time initialization.

(2) Wait for P0 to release.

(3) Read torque data from shared memory.

(4) Check torque data freshness.
   - If not fresh then decrement count.
   - If fresh then reset count.
   - If count has expired set disable pending.

(5) Clip torque data (optional).

(6) Convert torque data from Nm to hardware specific values.

(7) Output torque vector to robot.

(8) Read joint encoder positions.

(9) Convert joint encoder counts to radians/mm.

(10) Write position data into shared memory.

(11) Release ipb for controllers.

Synchronous Task Code (cont.)

(12) Check for joint limits.

- If joint at limit, set disable pending.

(13) Check for enable/disable transitions.

- Requested by message handler, or
- Requested by above code.

(14) Check for ESTOP.

(15) Notify state manager of forced transitions.

(16) Loop back to wait (2).

Overrun Task Code

(1) Issue a taskLock.

(2) Stop all robot joints.

(3) Turn off high power.

(4) Suspend synchronous task.

(5) Check if overrun was forced by sync task.

(6) Send message to state manager.

(7) Issue taskUnlock.

(8) Halt.

Channel Driver Interfaces

- Channel to Hardware
  - Calls to pumaLib and platLib
- Channel to Controller
  - Reads and writes to shared memory directly
  - Controller calls chanLib to access data

Current Implementation

- chanPuma
  - One message handler for both drivers (reentrant)
  - One sync task (spawned twice)
  - All joints on one driver must be at same time period
- chanPlat
  - One channel driver for both platforms
  - All joints on both platforms must be at same period

Current Implementation (cont.)

- Options
  - Synchronous task priority
  - Overrun task priority
  - Synchronous Period and Phase
  - ipb Flag to release
  - Number of torque overruns before disable
  - Torque clipping

Future Developments and Additions

- Split chanPlat driver into right and left channel drivers
- Ability to enable more than one joint at a time
- Force torque sensor channel driver
- Gripper channel driver
- Driver for the GCA arm

MCS Controllers

Overview:

- The Message Handler
- The Synchronous Task
- The Controller / Channel Interface
- The Controller / Trajectory Generator Interface
- Putting Together the Pieces
- Future Developments

The Message Handler

The Controller Message Handler Must Respond to the Following Messages:

- MSG_PINIT
- MSG_MCS_IPB_SET
- MSG_MCS_TIME_SET
- MSG_MCS_DEFAULT_CONFIG
- MSG_MCS_MOTION
- MSG_MCS_NO_MOTION

The Message Handler (cont.)

- MSG_PINIT
  - Register with the State Manager
  - Example:

```
/* Define List of Joints to Control */
int jointList = {1,2,3,10,11,12};
int numJoints = 6;

/* Register with State Manager */
mcsControllerRegister( myTid,
                       jointList,
                       numJoints );
```

The Message Handler (cont.)

- MSG_MCS_IPB_SET
  - Receive an IPB flag via message data
  - Example:

```
/* Get IPB Flag */
myIpbFlag = (IPB_FLAG)(msg->data);
```

The Message Handler (cont.)

- MSG_MCS_TIME_SET
  - Receive period via message data
  - Example:

```
/* Cast Message Data to Structure */
myTimeInfo =
(MCS_SLOT_TIME_TYPE *)(msg->data);

/* Get Period */
j = jointList[0];
period =
(myTimeInfo->slotPeriodInfo).period[j];

/* Convert Period to Seconds */
periodInSecs = (period * MCS_CUP) / 1000;
```

The Message Handler (cont.)

- MSG_MCS_DEFAULT_CONFIG
  - Read files
  - Initialize data structures
  - Spawn the synchronous task
- MSG_MCS_MOTION
  - Get the current robot position
  - Example:

```
/* Initialize interpLib */
interpLibInit( numJoints,
               jointList,
               initPos );
```

The Message Handler (cont.)

Future Messages Will Include:

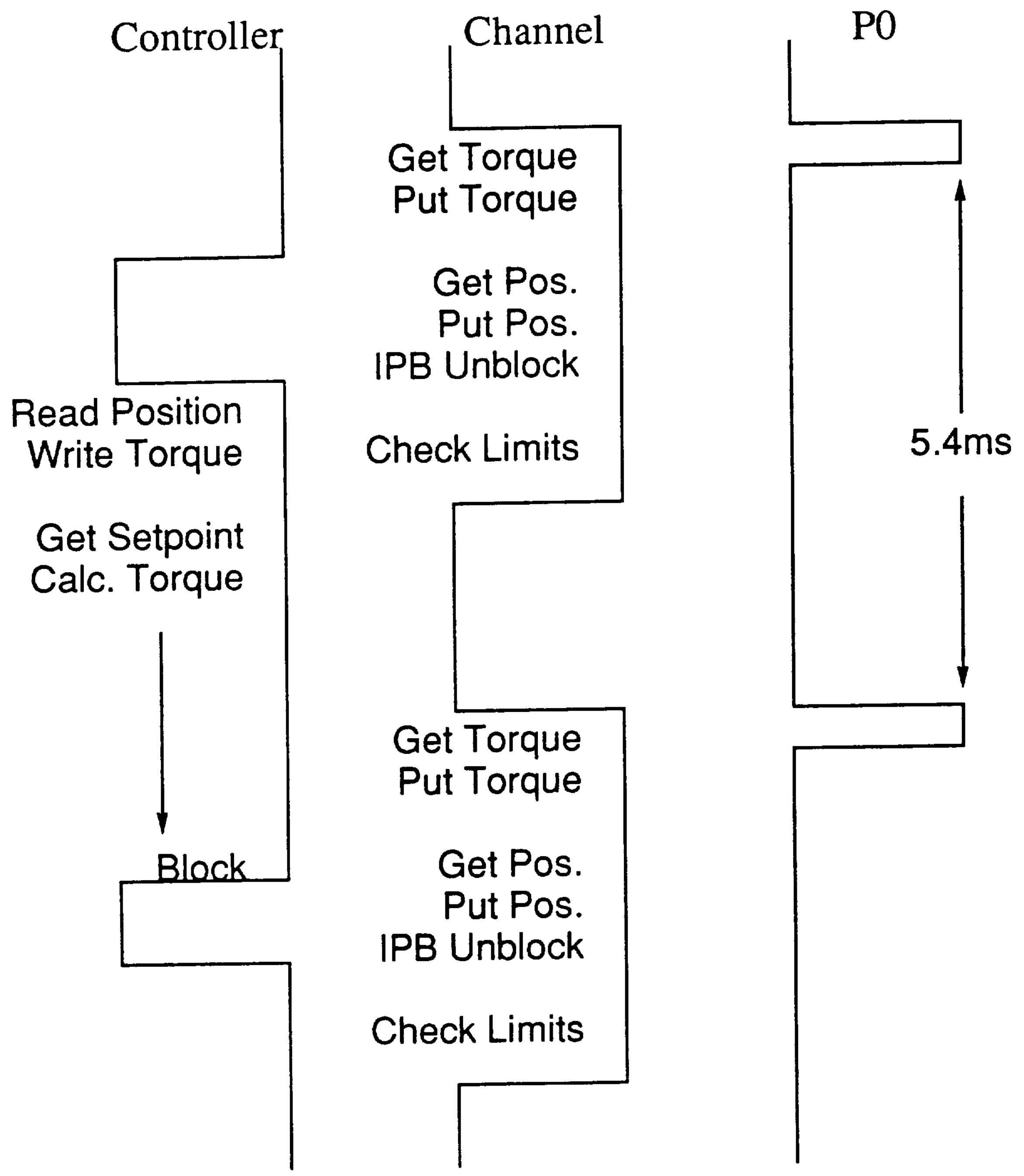
- MSG_MCS_CONFIG_GET
- MSG_MCS_CONFIG_SET
- MSG_MCS_ENABLE / DISABLE
- MSG_MCS_KILL

The Synchronous Task

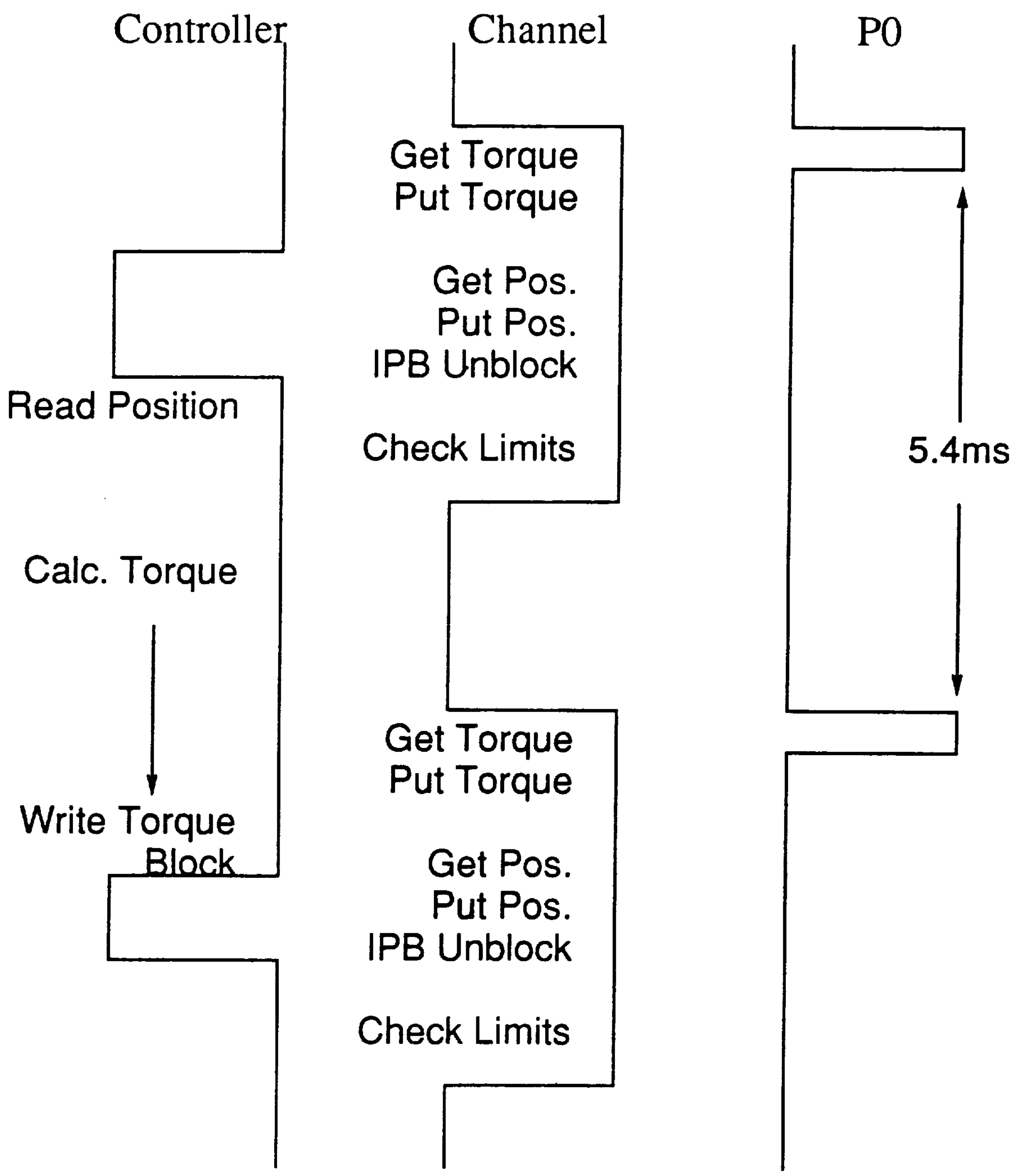
Important Issues:

- Blocking on an IPB flag
- Data Overruns
- The Control Structure

The Synchronous Task (cont.)



The Synchronous Task – Data Overrun



The Synchronous Task (cont.)

- Blocking on an IPB flag
  - Example:

```
while (TRUE) {

   /* Wait for Channel */
   ipbTake( myIpbFlag );

             :
             :

}   /* end of while */
```

The Synchronous Task (cont.)

- Data Overruns
  - A data overrun occurs when the positions or torques are not FRESH
  - If a torque overrun occurs, the channel uses the old torque value
  - The channel will allow N data overruns before the joint is disabled

The Synchronous Task (cont.)

- The Control Structure
  - The control loop must:
    - * block on an IPB flag
    - * read positions and write torques
    - * get setpoints
    - * compute torque
  - The order of these operations is a trade-off between computational speed and lag

The Controller / Channel Interface

Controllers read positions and write torques using chanLib.

- Reading positions

```
chanScalarRead( int joint,
                float *pos,
                short mode );

chanVectorRead( int numJoints,
                int jointList[],
                float posVector[],
                short mode );
```

- Position units are rad (revolute) and mm (prismatic)

The Controller / Channel Interface (cont.)

- Modes for Reading Positions
  - CHAN_CONTROLLER
  - CHAN_OBSERVER
- Writing Torques

```
chanScalarWrite( int joint,
                 float trq );

chanVectorWrite( int numJoints,
                 int jointList[],
                 float trqVector[] );
```

- Torque units are Nm

The Controller / Channel Interface (cont.)

- Checking for Enable / Disable Transitions

```
chanJointState( int joint );
```

- chanLib Return Codes
  - CHAN_OKAY
  - CHAN_ERROR
  - CHAN_DISABLED
  - CHAN_NOTFRESH
  - CHAN_OVERRUN
  - CHAN_ENABLED

The Controller / Trajectory Generator Interface

Controllers get setpoints using interpLib.

```
interpScalarRead( int joint,
                  float *pos,
                  float *vel,
                  float *acc,
                  short dataSelect );

interpVectorRead( int numJoints,
                  int jointList[],
                  float posVector[],
                  float velVector[],
                  float accVector[],
                  short dataSelect );
```

Putting Together the Pieces

Or, How To Write An MCS Controller

Step 1: Write a Message Handler

Step 2: Write a Sync Task That:

- a) Blocks on an IPB Flag
- b) Writes Torques
- c) Reads Positions
- d) Gets Setpoints
- e) Computes Torques (Control Algorithm)

Putting Together the Pieces (cont.)

Example: Synchronous Task

```
static void
ctrlPid( TID_TYPE myTid )
   {
   float trq[NUM_JOINTS];     /* torques */
   float pos_k[NUM_JOINTS];   /* current position */
   float pos_d[NUM_JOINTS];   /* desired position */
   float vel_d[NUM_JOINTS];   /* desired velocity */

   while (TRUE)
      {
a)    /* wait for channel to unblock */
      ipbTake(myIpbFlag);

b)    /* write torques */
      chanVectorWrite(NUM_JOINTS,
                      jointList,
                      trq);

c)    /* read positions */
      chanVectorRead(NUM_JOINTS,
                     jointList,
                     pos_k,
                     CHAN_CONTROLLER);
```

Putting Together the Pieces (cont.)

```
d)    /* get position and velocity setpoints */
      interpVectorRead(NUM_JOINTS,
                       jointList,
                       pos_d,
                       vel_d,
                       NULL,
                       INTERP_POS_VEL);


e)    /***** insert control algorithm here *****/


      }   /* end of while */
   }   /* end of ctrlPid() */
```

Future Developments

- Trans-Channel Controllers
  - Requires ANDing IPB flags
  - Servo rate limited by slowest channel
- Swapping Controllers
- Better Algorithms
  Currently available:
  - Gravity compensation
  - PID with integral windup compensation

MCS Client Interface

- MCS Client Interface will provide access to MCS functions for
  - higher levels of "intelligent machine"
  - experiments coordinating vision & motion
- Client Interface will be implemented as library of C functions
  - these C functions will exchange messages with MCS
  - library will be available on VME and Suns
- Library will include
  - motion commands
  - gripper commands
  - access to internal sensors
  - transform operations
  - trajectory generation functions
- First application will be a teach pendant

CTOS/MCS

Section IV: Case Study

Case Study: Master/Slave Control

vx0	CTOS Support Tasks
vx1	chanRPmaDrv
vx2	chanLPmaDrv
vx3	ctrlRGrav ctrlPid5
vx4	tgen Application Manager

Case Study: Master/Slave Control

- Configuration File
- Application Code
- "Trajectory Generator" Code
- Controller Code
  - Gravity Compensation
  - PID

CTOS/MCS

Section V: Excercises

CTOS/MCS Course Exercises

1. VxWorks "Print String" Function

   - Lessons
     - using Imake to compile
     - working with bare VxWorks
   - Procedure

     (a) write function that prints "From task xx: 'string' "

     (b) function prototype: void xyzPrtStr (TID_TYPE id, const char *s)

     (c) copy header file /home/lefebvre/vxworks/bootstrap/course/ex.h and change function prototypes to match your function names

     (d) create Imakefile - be sure to include the following directories

     `-I/home/lefebvre/vxworks/bootstrap`
     `-I/home/watson/cirsse/mcs/sync`

     (e) run **cmkmf**, then compile your function

     (f) run under VxWorks:

       i. cd "/home/yourdir/"

       ii. ld < xyzPrtStr.o

       iii. xyzPrtStr (123, "Hello World")

2. Simple Event Handler Task

   - Lessons
     - format of event handler function
     - CTOS bootstrap phases
     - building a config file
   - Procedure

     (a) you will write an event handler function for a task with symbolic name 'Team_n' - where 'n' is your team number, e.g. Team_2

     (b) the task is to report when it receives the bootstrap phase messages MSG_PINIT, MSG_AINIT, & MSG_AEXEC, e.g. "From task xx: Team 1 received PINIT"

     (c) build a User Config File for entire class

     (d) use **ctconfig** to point to your config file

     (e) use **vxboot** to change to CTOS VxWorks kernel

     (f) run the application

3. Send Messages to Other Tasks

- Lessons
  - finding TIDs of other tasks
  - saving data between calls to EH function
  - building & sending messages
- Procedure

  (a) add to event handler function of exercise 2 to send messages to the other teams

  (b) during AINIT: find TIDs of other teams' event handler functions via their symbolic names (i.e. use **msgTidQuery**), print out the names & TIDs

  (c) during AEXEC: send different MSG_STRING message to each team

  (d) be prepared to print out received messages

  (e) run it

4. Set up Synchronous Task

- Lessons
  - creating a synchronous process
  - connecting to synchronous services
- Procedure

  (a) write synchronous task that posts a MSG_STRING message to your event handler function

    i. use prototype: void xyzSyncTask(), and put in separate file

    ii. create global variables for: EH task's TID, sync process semaphore & running flag

    iii. sync task loops forever

    iv. remember to set running flag = FALSE and to take semaphore at begining of loop

  (b) add to event handler function of exercise 3 to set up the synchronous process

    i. use **syncProcSpawn** in PINIT to create sync process

    ii. use 2000 ticks for clock rate (1.8 seconds)

    iii. use **syncProcEnb** in AEXEC to start it

  (c) update config file to load sync task

  (d) add following lines to config file to create Application Executive task that starts clock

```
0  load  /home/lefebvre/vxworks/bootstrap/course/app_exec.o
0  task  App_Exec   app_exec   50
```

  (e) compile everything & run it

5. Communicate with Synchronous Task

   - Lessons
     - communication between synchronous & non-synchronous processes
   - Procedure

     (a) Application Executive will periodically send MSG_START_SYNCTASK & MSG_STOP_SYNCTASK messages

     (b) your event handler task must communicate with your synchronous process to start/stop posting messages

       i. use **syncProcDis** to stop it

       ii. use **syncProcEnb** to restart it

       iii. print message to report start/stop

     (c) compile everything & run it

```
/*

  Header file for Exercises  - ex.h

*/

/*------------------------------ include files  -------------------------*/
#include "stdioLib.h"
#include "string.h"
#include "logLib.h"
#include "msgLib.h"
#include "syncLib.h"

/*------------------------------ function prototypes  -------------------*/
void  printString (TID_TYPE t, const char *s) ;
int   userfcn     (TID_TYPE myTid, MSG_TYPE *msg) ;
void  syncTask    () ;
```

```
/*

  TEAM 1 Exercise 1  - strprt.c

*/

#include "ex.h"


/*******************************************************************************
  printString
*******************************************************************************/
void  printString (TID_TYPE t, const char *s)
    {
    printf ("From task %x: '%s'\n", t, s) ;
    }
```

```
/*

  TEAM 1 Event Handler Function  - ex2.c

*/

#include "ex.h"


/*****************************************************************************
  userfcn - Event Handler Function for Exercise 2
*****************************************************************************/
int userfcn (TID_TYPE myTid, MSG_TYPE *msg)
    {
    switch (msg->command)
        {
        case MSG_PINIT:
          printString (myTid, "Team 1 received PINIT") ;
          break ;

        case MSG_AINIT:
          printString (myTid, "Team 1 received AINIT") ;
          break ;

        case MSG_AEXEC:
          printString (myTid, "Team 1 received AEXEC") ;
          break ;
        }
    return (msgDefaultProc (myTid, msg)) ;
    }
```

```
/*

  TEAM 1 Event Handler Function  - ex3.c

*/

#include "ex.h"


/*******************************************************************************
  userfcn - Event Handler Function for Exercise 3
*******************************************************************************/
int userfcn (TID_TYPE myTid, MSG_TYPE *msg)
    {
    static TID_TYPE  t1, t2, t3, t4 ;
    static char      msg1[] = {"Hello team 1 from myself"} ;
    static char      msg2 [20] ;
           char     *msg3 ;

    switch (msg->command)
        {
        case MSG_PINIT:
          /*  report receiving bootstrap message  */
          printString (myTid, "Team 1 received PINIT") ;

          /*  break to get default processing  */
          break ;


        case MSG_AINIT:
          /*  report receiving bootstrap message  */
          printString (myTid, "Team 1 received AINIT") ;

          /*  find TIDs of other tasks  */
          printf ("Team 1's TID = %x\n", t1 = msgTidQuery(myTid, "Team_1")) ;
          printf ("Team 2's TID = %x\n", t2 = msgTidQuery(myTid, "Team_2")) ;
          printf ("Team 3's TID = %x\n", t3 = msgTidQuery(myTid, "Team_3")) ;
          printf ("Team 4's TID = %x\n", t4 = msgTidQuery(myTid, "Team_4")) ;

          /*  break to get default processing  */
          break ;


        case MSG_AEXEC:
          /*  report receiving bootstrap message  */
          printString (myTid, "Team 1 received AEXEC") ;

          /*  send msg to other teams  */
          msgBuildSend (t1, myTid, MSG_STRING,
                        msg1, strlen(msg1),
                        MF_STANDARD) ;

          strcpy (msg2, "Hello team 2 from team 1") ;
          msgBuildSend (t2, myTid, MSG_STRING,
                        msg2, strlen(msg2),
                        MF_STANDARD) ;

          msg3 = (char *) malloc (25) ;
          strcpy (msg3, "Hello team 3 from team 1") ;
          msgBuildSend (t3, myTid, MSG_STRING,
                        msg3, strlen(msg3),
                        MF_STANDARD) ;

          msgBuildSend (t4, myTid, MSG_STRING,
                        "Hello team 4 from team 1", 25,
                        MF_STANDARD) ;

          /*  break to get default processing  */
          break ;


        case MSG_STRING:
          /*  report received string  */
          printf ("Task %x received string from Task %x: '%s'\n",
                  myTid, msg->source, (char *)msg->data) ;

          /*  break to get default processing  */
          break ;


        case MSG_INTEGER:
          /*  report received string  */
          printf ("Task %x received integer from Task %x: %i\n",
                  myTid, msg->source, (int)msg->data) ;

          /*  break to get default processing  */
          break ;
        }
    return (msgDefaultProc (myTid, msg)) ;
    }
```

```
/*

  TEAM 1 Event Handler Function  - ex4.c

*/

#include "ex.h"

extern TID_TYPE  parent ;            /*  global TID of parent EH function    */
extern SEM_ID    semSync ;           /*  global vars needed by sync process  */
extern BOOL      runSync ;


/*************************************************************************
  userfcn - Event Handler Function for Exercise 4
*************************************************************************/
int userfcn (TID_TYPE myTid, MSG_TYPE *msg)
    {
    static TID_TYPE     t1, t2, t3, t4 ;
    static SYNC_HANDLE  hSync ;
    static char         msg1[] = {"Hello team 1 from myself"} ;
    static char         msg2 [40] ;
           char        *msg3 ;

    switch (msg->command)
        {
        case MSG_PINIT:
          /*  report receiving bootstrap message  */
          printString (myTid, "Team 1 received PINIT") ;

          /*  set up synchronous task  */
          parent = myTid ;
          hSync = syncProcSpawn (&semSync, syncTask, "Sync_Task", 0,
                                 NULL, NULL, "", SYNC_OVR_MILD,
                                 &runSync, 1, 2000) ;
          if (hSync == ERROR)
              {
              printf ("ERROR: Could not create Sync Task\n") ;
              break ;
              }


          /*  break to get default processing  */
          break ;


        case MSG_AINIT:
          /*  report receiving bootstrap message  */
          printString (myTid, "Team 1 received AINIT") ;

          /*  find TIDs of other tasks  */
          printf ("Team 1's TID = %x\n", t1 = msgTidQuery(myTid, "Team_1")) ;
          printf ("Team 2's TID = %x\n", t2 = msgTidQuery(myTid, "Team_2")) ;
          printf ("Team 3's TID = %x\n", t3 = msgTidQuery(myTid, "Team_3")) ;
          printf ("Team 4's TID = %x\n", t4 = msgTidQuery(myTid, "Team_4")) ;

          /*  break to get default processing  */
          break ;


        case MSG_AEXEC:
          /*  report receiving bootstrap message  */
          printString (myTid, "Team 1 received AEXEC") ;
        /*  send msg to other teams  */
        msgBuildSend (t1, myTid, MSG_STRING,
                      msg1, strlen(msg1),
                      MF_STANDARD) ;

        strcpy (msg2, "Hello team 2 from team 1") ;
        msgBuildSend (t2, myTid, MSG_STRING,
                      msg2, strlen(msg2),
                      MF_STANDARD) ;

        msg3 = (char *) malloc (25) ;
        strcpy (msg3, "Hello team 3 from team 1") ;
        msgBuildSend (t3, myTid, MSG_STRING,
                      msg3, strlen(msg3),
                      MF_STANDARD) ;

        msgBuildSend (t4, myTid, MSG_STRING,
                      "Hello team 4 from team 1", 25,
                      MF_STANDARD) ;

        /*  enable Sync Task  */
        if (syncProcEnb (hSync) == ERROR)
            printf("ERROR: could not enable Sync Task, hSync=%x\n", hSync) ;
        else
            printf ("Sync Task was Enabled\n") ;

        /*  break to get default processing  */
        break ;


      case MSG_STRING:
        /*  report received string  */
        printf ("Task %x received string from Task %x: '%s'\n",
                myTid, msg->source, (char *)msg->data) ;

        /*  break to get default processing  */
        break ;


      case MSG_INTEGER:
        /*  report received string  */
        printf ("Task %x received integer from Task %x: %i\n",
                myTid, msg->source, (int)msg->data) ;

        /*  break to get default processing  */
        break ;
      }
   return (msgDefaultProc (myTid, msg)) ;
   }
```

```
/*

  TEAM 1 Synchronous Task  - sync.c

*/

#include "ex.h"

TID_TYPE  parent ;                    /*  global TID of parent EH function    */
SEM_ID    semSync ;                   /*  global vars needed by sync process  */
BOOL      runSync = FALSE ;


/*******************************************************************************
  syncTask
*******************************************************************************/
void  syncTask ()
    {
    MSG_TYPE  msg ;
    char      s[80] ;
    int       i = 1 ;

    while (TRUE)
        {
        /*  block on semaphore  */
        runSync = FALSE ;
        if (semTake (semSync, WAIT_FOREVER) == ERROR)
            logMsg("***  ERROR: Invalid semaphore  ***\n") ;

        /*  create string  */
        sprintf (s, "Hi daddy, msg #%i", i++) ;

        /*  create message  */
        msgBuild (&msg, parent, parent, MSG_STRING, s, strlen(s), MF_STANDARD);

        /*  post message  */
        msgPost (&msg) ;
        }
    }
```

```
/*

  TEAM 1 Event Handler Function  - ex5.c

*/

#include "ex.h"

extern TID_TYPE  parent ;            /*  global TID of parent EH function    */
extern SEM_ID    semSync ;           /*  global vars needed by sync process  */
extern BOOL      runSync ;


/*****************************************************************************
 userfcn - Event Handler Function for Exercise 5
*****************************************************************************/
int userfcn (TID_TYPE myTid, MSG_TYPE *msg)
    {
    static TID_TYPE     t1, t2, t3, t4 ;
    static SYNC_HANDLE  hSync ;
    static char         msg1[] = {"Hello team 1 from myself"} ;
    static char         msg2 [40] ;
           char        *msg3 ;

    switch (msg->command)
        {
        case MSG_PINIT:
          /*  report receiving bootstrap message  */
          printString (myTid, "Team 1 received PINIT") ;

          /*  set up synchronous task  */
          parent = myTid ;
          hSync = syncProcSpawn (&semSync, syncTask, "Sync_Task", 0,
                                 NULL, NULL, "", SYNC_OVR_MILD,
                                 &runSync, 1, 2000) ;
          if (hSync == ERROR)
              {
              printf ("ERROR: Could not create Sync Task\n") ;
              break ;
              }


          /*  break to get default processing  */
          break ;


        case MSG_AINIT:
          /*  report receiving bootstrap message  */
          printString (myTid, "Team 1 received AINIT") ;

          /*  find TIDs of other tasks  */
          printf ("Team 1's TID = %x\n", t1 = msgTidQuery(myTid, "Team_1")) ;
          printf ("Team 2's TID = %x\n", t2 = msgTidQuery(myTid, "Team_2")) ;
          printf ("Team 3's TID = %x\n", t3 = msgTidQuery(myTid, "Team_3")) ;
          printf ("Team 4's TID = %x\n", t4 = msgTidQuery(myTid, "Team_4")) ;

          /*  break to get default processing  */
          break ;


        case MSG_AEXEC:
          /*  report receiving bootstrap message  */
          printString (myTid, "Team 1 received AEXEC") ;
```

```
            /*  send msg to other teams  */
            msgBuildSend (t1, myTid, MSG_STRING,
                          msg1, strlen(msg1),
                          MF_STANDARD) ;

            strcpy (msg2, "Hello team 2 from team 1") ;
            msgBuildSend (t2, myTid, MSG_STRING,
                          msg2, strlen(msg2),
                          MF_STANDARD) ;

            msg3 = (char *) malloc (25) ;
            strcpy (msg3, "Hello team 3 from team 1") ;
            msgBuildSend (t3, myTid, MSG_STRING,
                          msg3, strlen(msg3),
                          MF_STANDARD) ;

            msgBuildSend (t4, myTid, MSG_STRING,
                          "Hello team 4 from team 1", 25,
                          MF_STANDARD) ;

            /*  enable Sync Task  */
            if (syncProcEnb (hSync) == ERROR)
                printf("ERROR: could not enable Sync Task, hSync=%x\n", hSync) ;
            else
                printf ("Sync Task was Enabled\n") ;

            /*  break to get default processing  */
            break ;


          case MSG_STRING:
            /*  report received string  */
            printf ("Task %x received string from Task %x: '%s'\n",
                    myTid, msg->source, (char *)msg->data) ;

            /*  break to get default processing  */
            break ;


          case MSG_INTEGER:
            /*  report received string  */
            printf ("Task %x received integer from Task %x: %i\n",
                    myTid, msg->source, (int)msg->data) ;

            /*  break to get default processing  */
            break ;


          case MSG_START_SYNCTASK:
            /*  start Sync Task  */
            if (syncProcEnb (hSync) == ERROR)
                printf("ERROR: could not enable Sync Task\n") ;
            else
                printf ("Sync Task was Enabled\n") ;

            /*  break to get default processing  */
            break ;


          case MSG_STOP_SYNCTASK:
            /*  stop Sync Task  */
            if (syncProcDis (hSync) == ERROR)
                printf("ERROR: could not disable Sync Task\n") ;
            else
            printf ("Sync Task was Disabled\n") ;

        /*  break to get default processing  */
        break ;
      }
  return (msgDefaultProc (myTid, msg)) ;
  }
```

```
/*  Imakefile for CTOS/MCS Course Exercises  */

CPPFLAGS += -I/home/lefebvre/vxworks/bootstrap  -I/home/watson/cirsse/mcs/sync

AllTarget(strprt.o ex5.o sync.o app_exec.o)

VxWorksBinTarget(strprt.o, ex.h, )
VxWorksBinTarget(ex5.o   , ex.h, )
VxWorksBinTarget(sync.o  , ex.h, )

VxWorksBinTarget(app_exec.o, , )
```

```
#  User Configuration File for CTOS/MCS Course

#  load & start Application Executive
0  load  /home/lefebvre/vxworks/bootstrap/course/app_exec.o
0  task  App_Exec   app_exec   50


-1 load  /home/lefebvre/vxworks/bootstrap/course/strprt.o
-1 load  /home/lefebvre/vxworks/bootstrap/course/sync.o

1  load  /home/lefebvre/vxworks/bootstrap/course/ex5.o
2  load  /home/lefebvre/vxworks/bootstrap/course/ex5.o
3  load  /home/lefebvre/vxworks/bootstrap/course/ex5.o
4  load  /home/lefebvre/vxworks/bootstrap/course/ex5.o

#  load & start each team's event handler task
#1  load  /home/lefebvre/vxworks/bootstrap/course/team_1.o
1  task  Team_1   userfcn   150

#2  load  /home/lefebvre/vxworks/bootstrap/course/team_2.o
2  task  Team_2   userfcn   150

#3  load  /home/lefebvre/vxworks/bootstrap/course/team_3.o
3  task  Team_3   userfcn   150

#4  load  /home/lefebvre/vxworks/bootstrap/course/team_4.o
4  task  Team_4   userfcn   150
```

```
/*

  Application Executive for CTOS/MCS Course  - app_exec.c

*/

#include "stdioLib.h"
#include "logLib.h"
#include "msgLib.h"
#include "syncLib.h"


/*****************************************************************************
  app_exec - Application Executive
*****************************************************************************/
int app_exec (TID_TYPE myTid, MSG_TYPE *msg)
    {
    TID_TYPE  p0Tid ;
    MSG_TYPE  bmsg ;

    switch (msg->command)
        {
        case MSG_CINIT:
          /*  start synchronous clock  */
          if ((p0Tid = msgTidQuery(myTid,"p0")) == 0)
              {
              msgErrorLog(myTid,"App_Exec ERROR: Couldn't find P0\n") ;
              break ;
              }
          if (msgBuildSend (p0Tid, myTid, MSG_SYNC_CLK_RESET,
                            NULL, MS_NONE, MF_STANDARD) != OK)
              {
              msgErrorLog(myTid, "Appl_Exec ERROR: could not sync clock") ;
              break ;
              }
          if (msgBuildSend (p0Tid, myTid, MSG_SYNC_CLK_PROC_ON,
                            NULL, MS_NONE, MF_STANDARD) != OK)
              {
              msgErrorLog(myTid, "App_Exec ERROR: no clock on") ;
              break ;
              }

          /*  break to get default processing  */
          break ;


        case MSG_AEXEC:
          /*  periodically send START/STOP_SYNCTASK messages  */
          msgBuild (&bmsg, 0, myTid, MSG_START_SYNCTASK,
                    NULL, MS_NONE, MF_STANDARD) ;
          FOREVER
              {
              taskDelay (60*15) ;
              bmsg.command = MSG_STOP_SYNCTASK ;
              msgBroadcast (&bmsg, MB_CHASSIS) ;
              printf("Broadcasting STOP Sync Task\n") ;

              taskDelay (60*15) ;
              bmsg.command = MSG_START_SYNCTASK ;
              msgBroadcast (&bmsg, MB_CHASSIS) ;
              printf("Broadcasting START Sync Task\n") ;
              }
          break ;
        }
    return (msgDefaultProc (myTid, msg)) ;
    }
```

CTOS/MCS

Section VI: Supplement

CIRSSE Technical Memorandum

To:	CIRSSE	**Date:**	April, 1991
From:	Keith R. Fieldhouse	**Number:**	3 vers. 1
Group:	All		
Title:	VxWorks at CIRSSE		

1 Introduction

VxWorks is the real time operating system and development environment used at CIRSSE for motion control and Datacube based vision experimentation. VxWorks runs on VME based Single Board Computers (SBCs). The system was developed by Wind River Systems of Alameda, California.

This document is intended as an introduction to VxWorks for those members of CIRSSE who will be doing development on our real time systems.

An important characteristic of VxWorks is that when several SBCs are installed on one backplane, they are configured as an Internet subnet with their own addresses and node names, much as the CIRSSE Sun systems are also collected into a subnet. Tables 1 and 2 show the two VME cage subnetworks here at CIRSSE. Note that CPU 0 on each cage has two names, indicating its role as the gateway between the cage's backplane network and the CIRSSE thinwire (coaxial cable) network.

The existence of each SBC as a node on CIRSSE's network allows one to `rlogin` to any of the nodes. Given the nature of VxWorks, only one active session (either via `rlogin` or attached to the console) can be allowed per SBC at a given time. Care should be taken when using `rlogin` to attach to a board, as other users may (inadvertently or not) reboot the cage from beneath you. In general it is best to be in the lab when using a one of the VME cages. You should always be in the lab when you are actually causing a manipulator to move.

Motion Control Cage				
CPU 0	CPU 1	CPU 2	CPU 3	CPU 4
uranus 128.113.30.17				
vx0 128.113.47.254	vx1 128.113.47.10	vx2 128.113.47.11	vx3 128.113.47.12	vx4 128.113.47.13

Table 1: Motion Control CPU List

Datacube Cage	
CPU 0	CPU 1
saturn 128.113.30.16	
datacube 128.113.52.254	laser 128.113.52.10

Table 2: Datacube CPU List

2 Getting Started

To get started using VxWorks here at CIRSSE you must add some directories to your **path** so that your shell can find the commands that Wind River and CIRSSE provide to assist in the use of the system.

In order to use the Wind River provided tools you must add the directory:

```
/usr2/testbed/vxworks/vxworks5.0/bin/`arch`
```

to your path. Note that the **`arch`** part of the command (exactly as printed above, "arch" surrounded by backticks) is important to ensure that the appropriate directory is used for the Sun architecture (sun3 or sun4) you are running on.

You should also add

```
/usr2/testbed/CIRSSE/installed/UNIX/bin/`arch`
```

to your path. Again, you must use the "arch" in backticks in order to get the appropriate directory. This directory will make the CIRSSE developed VxWorks tools available to you.

2.1 Hello World

What follows is a simple example of writing some code that will run on a VxWorks node. This is by far the simplest of applications that can be written for VxWorks and thus does not go into great detail on the use of some of the more esoteric features of the operating system. For more details see the *VxWorks Programmer's Guide* which is available in the documentation cabinet or from Keith Fieldhouse.

This example is a program that will simply print "Hello World" on the standard output channel. Figure 1 shows the source code that we will be using for this example.

There are several things to note about this code. The first is the inclusion of the file "vx-Works.h", this file contains data structures, type definitions and macros that are used to produce VxWorks compatible code. We also include the "stdioLib.h" file, which gives us the function definitions necessary to uses parts of Wind River's "stdioLib", in this case, "printf".

Another aspect of note is that this code does not have a "main" function. This is because when VxWorks loads an object module it is actually linking that module with itself, thus, any externally available (non static) functions in an object file are available to the VxWorks system as a whole and the VxWorks shell in particular. This means that you may call any function directly from the shell without the need of a "main" entry point.

```
#include "vxWorks.h"
#include "stdioLib.h"

void hello()

{
printf("Hello World\n");
}
```

Figure 1: Hello World Example Code

To compile the code we'll use the vxgcc tool available here at CIRSSE. **Vxgcc** automatically uses the appropriate compiler options for compiling code for later downloading to a VxWorks node. In particular, vxgcc uses the -c option to prevent the compiler from linking the code (thus producing a linkable object module). **Vxgcc** also includes the correct VxWorks directories when it searches for "#include" files. To compile the code (which we'll presume is in a file called. hello.c) we would do the following:

```
    sol.ral.rpi.edu% vxgcc hello.c
```

This will produce the file hello.o in our working directory. Let us presume that our working directory is /home/krf/vxworks. If that is the case then the following sequence of events can be followed to download the file to the VxWorks node vx3 and run it:

```
sol.ral.rpi.edu% rlogin vx1

-> iam "vxworks"
-> cd "/home/krf/vxworks"
value = 0 = 0x0
-> ld < hello.o
value = 0 = 0x0
-> hello
Hello World
value = 12 = 0xc
-> logout

connection closed.
sol.ral.rpi.edu%
```

What follows is a step by step description of this procedure. The **iam "vxworks"** command is necessary because of a bug in the VxWorks rlogin daemon that causes VxWorks to "forget" which username it is supposed to use for network access.

Once you have connected and set up the VxWorks session, you may then set the working directory to the directory on the Sun systems in which you have placed your code. You can then use the VxWorks ld[1] command to load your object module into VxWorks (Note that, instead of

[1]It is worth pointing our here that while the cd commands expects its argument to be in double quote marks, the ld command does not

using the `cd` command you could also have specified the entire "path" of the object file that you wished to load: `ld < /home/krf/vxworks/hello.o`).

Finally, since all non static (global) function are available to the shell, you can simply type "hello" (the name of your function) which will execute the function. Since there is no explicit return value in the `hello` function, the "value" returned by VxWorks (in this case, 12) is meaningless.

3 Tools

There are several tools available to users of VxWorks. Some of these are provided by Wind River Systems while others were developed here at CIRSSE. The following sections describe some of the more useful of these tools.

3.1 Wind River VxWorks Tools

3.1.1 vwman

Perhaps the most important Wind River tool is their manual page viewer **vwman**. The **vwman** command allows you to look at any of the Wind River function manual pages. For example:

```
vwman semTake
```

will give you the manual page for the **semTake** function call. There are also manual pages for entire libraries (e.g. `semLib`). To get to the board specific information available in the VxWorks manual you must prefix the topic with the board type and a slash. For example

```
vwman mv135/sysBusTas
```

will give you information on the **sysBusTas** command as it applies to the Motorola MV-135 board.

The VxWorks manual pages are divided into sections. The available sections are: (1) Libraries, (2) Subroutines, (3) Drivers, (4) Tools, (t) Targets.

To get a Table of Contents for any of the sections, use a command of the form `vwman 3 Toc`. Note that in section "t", you must specify the target you are interested in: `vwman t mv135/Toc`. A list of the available functions is available from Keith Fieldhouse or in the Documenation cabinet.

For further details on **vwman**, try `vwman vwman`.

3.1.2 vxgdb

Wind River system has provided a specially modified version of the Free Software Foundation's GNU Debugger (`gdb`) called **vxgdb**. Details on the use of **vxgdb** can be found in the manual from Wind River (available from Keith Fieldhouse or the documentation cabitnet). To successfully use the debugger, however, there are some details you must take care of first:

You must modify your path to include `/usr2/testbed/vxworks`. This is so that **vxgdb** can find our VxWorks kernels.

You should also create a file in your home directory called `.vxgdbinit` which should contain (at least) the following two lines:

```
dir /usr2/testbed/CIRSSE/installed/VxWorks/bin
dir /usr2/testbed/CIRSSE/installed/VxWorks/lib
```

This will allow **vxgdb** to find any modules that are loaded at boot time by the kernel (most notably, `testBed.o`). You may, of course, have to add other search directories to accommodate the files you are working on as detailed in the manual.

3.2 CIRSSE Tools

3.2.1 cmkmf

The `cmkmf` command is a simple interface to `Imake` that will construct a `makefile` based on an associated Imakefile and then call `make` with the arguments that were specified on the call to `cmkmf`. Details on the use of `cmkmf` can be found in the CIRSSE Technical Memo on testbed development and in the manual pages (TBD).

3.2.2 manc & xmanc

These two commands are similar to the VxWorks `vwman` command in that they allow the viewing of manual pages, in this case, the CIRSSE specific testbed manual pages. The commands work by calling the UNIX `man` or `xman` commands with the appropriate value of MANPATH. Thus, these commands may be used to find out details about any of the commands described in the CIRSSE Tools section of this manual.

3.2.3 vxgcc

The GNU C compiler is used for most of the development for the CIRSSE testbed. In general, the compiler produces more efficient code than does the native Sun C compiler. Further, the GNU C compiler is an implementation of the new ANSI C standard which the native C compiler is not.

The `vxgcc` command is designed to make it somewhat easier to compiler code for a VxWorks target. By default it informs the compiler that it should search the Wind River "include" directories when compiling. It will also invoke the appropriate compiler for production of 680X0 code regardless of the platform on which the compilation is being done.

For example, the command `vxgcc test.c` will produce a 680X0 object module `test.o` in your current directory. If you included any VxWorks header files (e.g. "vxWorks.h") they will be searched for by the compiler in the correct places.

3.2.4 vxstart

The `vxstart` command allows the user to specify that a VxWorks node or nodes should execute a specified file at boot time. The nodes are specified by their network names (e.g. vx1) or by a collection name (@control or @datacube). Without an argument, `vxstart` will list the available nodes. The command also provides the `-c` option which cancels a particular start up file and which should be used when work with a node or collection of nodes has completed.

For example `vxstart @control mystart` will cause the commands in the file `mystart` in the current directory to be executed at boot time by all of the control cage processor boards. The command `vxstart -c vx3` will cancel any personal start files associated with node `vx3`. When you have finished using the VME cage, as a matter of courtesy you should always `vxstart -c` any startup files that you have established.

3.2.5 vxboot

Over time, several different VxWorks kernels have been developed here at CIRSSE. The `vxboot` command is a menu driven interface that allows the selection of a particular kernel for a VxWorks node or collection of nodes. Table 3 shows the currently available VxWorks kernels and their descriptions.

datacube.v4	Kernel for datacube main processor (VxWorks V4) (MV-147)
datacube.v5	Kernel for datacube main processor (VxWorks V5) (MV-147)
laser.v4	Kernel for laser control processor (VxWorks V4) (MV-135)
laser.v5	Kernel for laser control processor (VxWorks V5) (MV-135)
control.v4.mv135	Kernel for Control Cage development (VxWorks V4.0.3)
control.v5.mv135	Kernel for Control Cage development (VxWorks V5.0)
kali-demo-nasa-90	Kernel used for Kali demo for NASA 11/29/90
kali-experimental	Experimental Kali Kernel. Use at own risk!

Table 3: Available VxWorks Kernels

Currently the default kernel for the Motion Control cage is `control.v5.mv135`, while for the Datacube it is `datacube.v4` and `laser.v4`.[2]

The `vxboot` command takes no paramters. When it is run (from a Unix prompt) it will first present a list of processors, after which it will present a list of kernels. Selecting a processor/kernel pair will cause the specified processor to use the specified kernel the next time it is booted.

[2]The defaults for the Datacube processors will change to Version 5 of VxWorks when the Datacube ImageFlow software supports VxWorks V5.0

CIRSSE Technical Memorandum

To:	CIRSSE	**Date:**	July, 1991
From:	Keith R. Fieldhouse	**Number:**	5 vers. 1
Group:	All		
Title:	Testbed Software Development at CIRSSE		

1 Introduction

The CIRSSE testbed is a large, hardware/software project designed to be an arena in which various issues relating to Intelligent Robotic Control can be researched. In large measure, the CIRSSE Testbed is centered around two PUMA 6 degree of freedom manipulators and a 6 degree of freedom dual teletransporter on which the PUMA arms are mounted. The PUMAs, the transporter and their associated sensors are controlled from a VME based control cage running the VxWorks operating system. An auxiliary of the CIRSSE testbed is a VME based Datacube vision system which is used for vision and other forms of sensing. The Datacube also runs the VxWorks operating system.

This document is intended as a set of guidelines and instructions for those members of CIRSSE who wish to develop software for the testbed. Readers of this document will probably also wish to read CIRSSE Technical Memo #3 "VxWorks at CIRSSE" for details on the use of the the VxWorks operating system as it relates to the CIRSSE Testbed.

1.1 The CIRSSE Testbed Software Directory

All of the CIRSSE Testbed software is maintained, once completed, in the directory structure illustrated in Figure 1. As you can see from the figure, there are two main nodes of the CIRSSE tree: `src` and `installed`. The `installed` tree is where "header" files, libraries, applications and viewable manual pages are kept. The `src` subdirectory contains the archived sources for these elements of the testbed software.

In general, this means that users who wish to include CIRSSE Testbed ".h" files should look in (or direct their compiler to look in) `testbed/CIRSSE/installed/VxWorks/h`. Users wishing to use UNIX tools associated with the testbed would set their paths to

`~testbed/CIRSSE/installed/UNIX/bin/‘arch‘`. [1]

2 Conventions

This portion of the this memo will describe the conventions and standards to be applied to software that is being developed for the CIRSSE testbed. These standards are applied to such software to provide for consistency and ease of maintenance since as it is expected that the software will have a long and productive life, and, quite possibly will be made available to other research institutions.

[1]Currently the CIRSSE testbed directory is rooted at /usr2 on all of the CIRSSE systems.

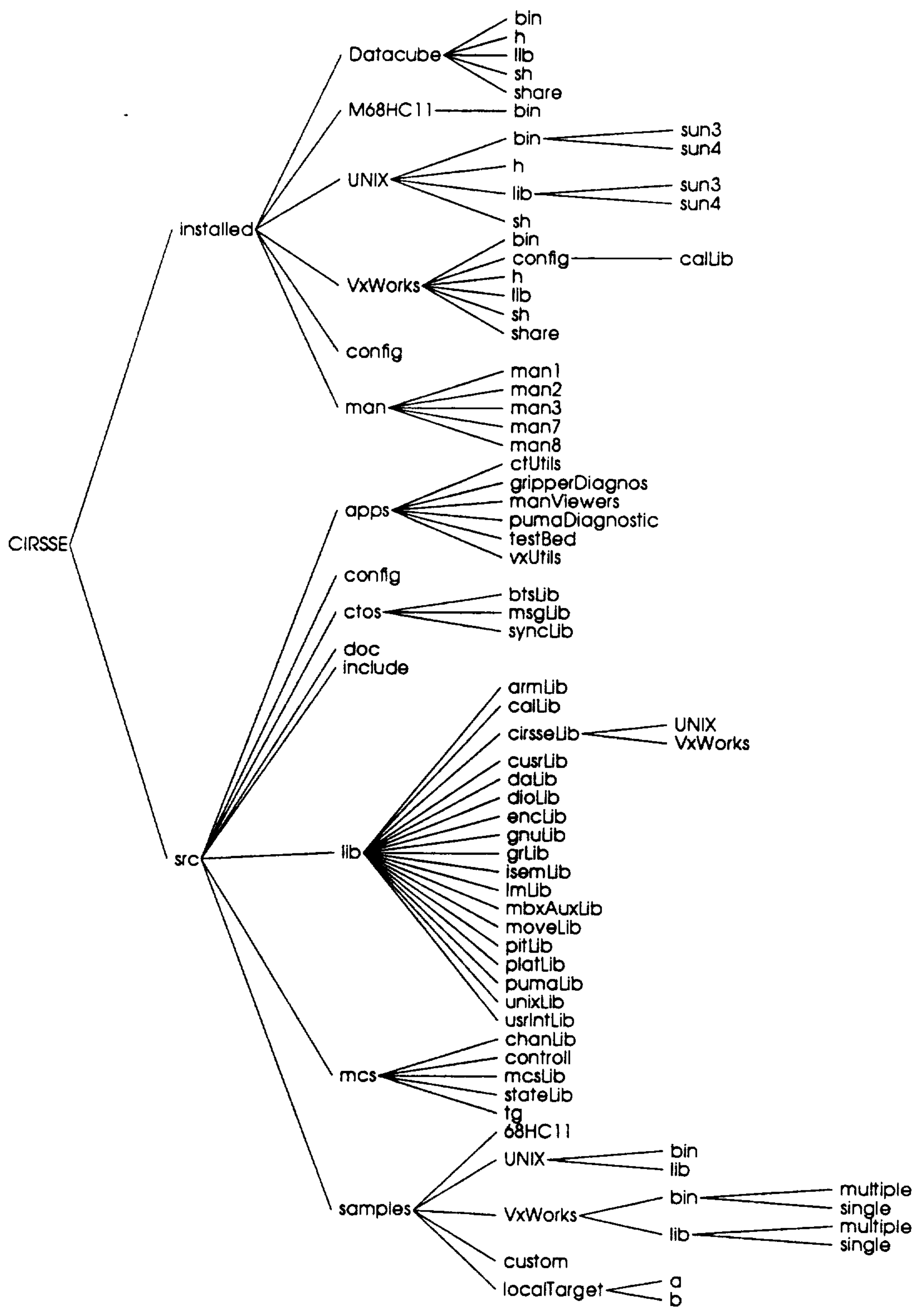


Figure 1: CIRSSE Testbed Directory Structure

2.1 Project Organization

When starting a project for the CIRSSE Testbed you must determine whether the project will be UNIX, VxWorks, or Datacube based and whether it will be an application or a library. This will help you understand where the code will ultimately reside in the Testbed hierarchy and will also allow you to apply the standards described below properly.

One further note. Not all software that is developed on the CIRSSE Testbed has necessarily been developed *for* the testbed. As the testbed matures and it is used more and more as a research tool, there will be significant amounts of experimental code developed that runs on the testbed but which is not a part of the testbed system itself. Such code will generally not be installed in **~testbed** but will instead be saved in other CIRSSE archive areas. This code, should nevertheless follow the standards and conventions outlined in this file as much as possible.

2.2 Naming Conventions

On any given software project there are numerous items which must be named. The following guidelines will help properly identify such items in context and will help prevent conflict between like named items.

Before any naming of objects can begin, a project prefix must be chosen. This prefix is used to identify modules that belong to a project, and the routines and data items that are available outside of the project. The prefix should in some way identify the function of the project (for example, "bts" for a set of boot strap routines, "isem" for a set of inter-processor semaphores etc.). Check with the CIRSSE software engineer to be sure that your project prefix is unique.

Once the project prefix is chosen it is possible to use it to derive the names of other objects in the project:

- **Modules:** Often, the words "module" and "file" are used interchangeably. For our purposes, however, a "module" will refer to a file that can be loaded by VxWorks. Often a single C source file will be compiled into a single object "module" loadable by VxWorks. Sometimes, though, it is desirable to link many object modules together into a single VxWorks module.

  There are three types of modules that can be created for VxWorks:

  - *Application Modules:* Application modules are modules which contain a single VxWorks runnable application or tool. Such modules should be named in such a way that their purpose is readily identifiable. Further, the main entry point of the tool should be the same as the name of its containing module. For example, the module `pumaDiagnostics.o` can be started by typing `pumaDiagnostics` at the VxWorks prompt.
  - *Library Modules:* Library modules are modules which contain one or more routines which are designed to be called by code that resides outside of the project. All library modules should have names that consist of their 2 to 6 letter prefix followed by the letters "Lib" (*e.g.* syncLib.o).
  - *Shareable Modules:* Shareable modules are those modules which contain data structures that are to be loaded at some location in memory which is available to all processors on a given VME cage chassis. These modules should have names which contain their project prefix followed by the letters "Share". When building a complete system for the testbed, many such source files will be included together into one large shareable module.

- **UNIX Libraries:** In order to satisfy the expectations of the link editors that run under UNIX, UNIX library files should begin with the letters "lib" followed by the project prefix (though in this case it isn't a prefix) and have a ".a" as their extension (*e.g.* libsync.a).
- **UNIX Commands** The names of testbed commands should be meaningful and reasonably mnemonic.
- **CTOS Tasks** CTOS event handler tasks have two names: a symbolic name and the name of its C function. Several tasks may call the same C function provided each task has a unique symbolic name. These symbolic names are defined and associated with a C function in your application's CTOS configuration file (see manual pages for ctos_config). A task's symbolic name is used by other tasks to find its address for routing messages.

  Because most tasks are application-specific, the user has considerable freedom to create task names. The only restriction at present is that the name be less than 32 characters. However, due to the central role played by symbolic names in connecting together tasks, it is recommended that the user adopt a consistent naming convention early in code development. Note that CTOS system tasks are typically accessed via library functions which encapsulate system task names. Please consult the CTOS administrator if you are naming tasks that are to be part of MCS or VSS.

  The C function associated with a task is named in the same manner as any other function (see below). The UNIX task creation process requires that the name of a task's linked object module must match that task's C function name. For this reason, the source code for an event handler function should be maintained in an individual file with the same name as the function (plus a ".c" extension).
- **Files** The naming of individual files is somewhat less formal. This is because a group of files can generally be identified as being a part of a project simply by their aggregation in a particular directory. Nevertheless, there are some conventions to be followed:
  - Files should have meaningful names. A file that contains string parsing routines is better named `parser.c` rather than `stuff.c`.
  - For projects that produce a single module from a single source file, the filename should be the same as the module name (and thus will follow the module naming conventions) with the exception of the extension. In the case of multi-file projects that produce library modules, if feasible, keep the externally available routines in a file with the same name as the module.
  - For projects which provide external routines, a header file with the same name as the module (with a `.h` extension) should be created. It should contain constant declarations and function prototypes for the use of the users of the module's routines. See section 2.3 for details on the construction of these header files.
- **Functions:** Function names are written with upper and lowercase letters and no underlines. Each "word", with the exception of the first one is capitalized. For example: `sysProcNumGet()`. The general form of the name of the function should be *object/verb* as opposed to *verb/object* (*e.g.* `sysProcNumGet()` rather than `sysGetProcNum()`. Functions which are externally available should begin with the project prefix.
- **Variables:** Variables should also be named with upper and lower case letters, each word but the first capitalized. Externally available variables should begin with their project prefix

(For example msgMessageNum. There is slightly different handling for variables that are to be loaded into shared memory for access by multiple processors. These variable name should be prefixed by the entire module name with which they are associated and an underscore. Following the underscore they may be named as usual: ipbLib_commLink. This serves to identify the variable both as associated with the project and as a shared memory variable.

- **Constants:** Constants are named with all upper case letters. Each "word" in the name is separated with underscores. Again, externally available constant names should begin with the project prefix. For example MBXAUX_MAXCPUS.
- **Defined Types:** Defined types follow the same naming convention as constants.
- **Macros:** Macros follow the the same naming convention as defined types and constants. In the case of macros that are created in lieu of functions, use the GCC "extern inline" keywords to create an inline function (see the GCC documentation for details).

2.3 File Organization

The internal organization of a file is broken into several sections separated by blank lines. These sections are slightly different for source and include files. All files, however, share some common features:

Each should have as its first line a comment of the following form (though the comment delimiters may differ from language to language):

```
/* %W% %G% */
```

This will allow the SCCS source code archiving system to insert revision information in the file once the code has been transferred to the testbed/CIRSSE area.

Following the SCCS comment, the CIRSSE copyright notice should be applied to the file. This copyright notice can be found in /usr/local/lib/cirsse-copyright.

After the copyright should come a block comment with the following form

```
/*
** File:
** Written By:
** Date:
** Purpose:
*/
```

Where each of the items is filled in appropriately.

After this identifying block should come the modification history of the file. This modification history should be updated each time a file is updated and is re-installed in the testbed/CIRSSE area.

Each modification history line should look similar to the following:

```
/*
** Modification History:
**
** 10 May 1991 Archie Goodwin Added doEverything Function call
** 15 Jul 1991  Purley Stebbins Deleted doEverything Function (didn't work)
**                              Add doMostEverthing call
*/
```

Subsequent to this initial set of commentary are any include directives needed in order to compile the module correctly. Generally any OS include files (for the VxWorks or UNIX operating systems) should come first. These would be followed by any CIRSSE testbed include and finally by the include files that belong to the project that is being worked on. Note that this is strictly a rule of thumb and can be overridden when necessary. **Never** use absolute path names (path names which begin with the / or . character) to include a file. To search for include files in other directories use the -I compiler option.

After the include directives come constant and type definitions followed by any function prototypes needed by the module.

Note that header or ".h" files will have some slightly different organizational requirements from standard source or ".c" files. These differences will be discussed now.

2.3.1 Header Files

In general header files should be organized as above (Block Commentary, include directives, constants and types etc.). It important that a public header file be created for function libraries that contains all of the information needed by a user of that library to properly compile their code. In particular, ANSI C style function prototypes for all externally available functions must be available in the library's public header file. Generally, this header file should have the same name as the library but with a ".h" extension (thus, the public header file file for `isemLib.o` would be `isemLib.h`). Private header files may have whatever (meaningful) name the author chooses.

Header files should also contain some logic that protects against multiple of the header file. Such mutliple inclusing can lead to compiler errors (due4 to multiply defined variables etc.) and causes needless useage of processor time. The pre-processor logic that helps prevent this is as follows.

```
... banner information ....
#ifndef INCfilenameh
#define INCfilenameh
#pragma once
... body of header file ...
#endif
```

The above code fragment has the follwing meaning. The C pre-processor checks for the existance of a variable constructed by concatenating the letters "INC" with name of the include file (less the "." since that character is not allowed in variables in the C pre-processor). If the variable is not defined (#ifndef) the code following the statement (and before its associated #endif) is included in the compiler stream. The first order of bussiness is to define the variable (#define) so that should the header file be included again the body of the file will not be processed. The "#pragma once" command is a special command that is understood by some C compilers (including the GCC compiler that we use) and causes them to refrain (when possible) from even reading an include file after the first time. The use of the #ifndef logic and the #pragma provide the most reliable way to prevent the inclusion of the body of a header file more than once.

2.3.2 Source Files

The fundamental characteristic of source files is that they contain function implementations. Functions should be organized in the following manner.

Each function must be preceded by a function comment. A function comment consists of the following:

1. **Banner:** A comment consisting of a line of asterisks across the page. This serves to identify the start of new functions.
2. **Title:** A line containing the name of the function and a one line description of its purpose.
3. **Description:** A complete description of a function' s purpose and usage. Only necessary if the title description is insufficient.
4. **Returns:** A description of the possible return values of the function.
5. **Parameters:** Parameters may be described either in the block comment proceeding the function or in the function declaration itself.

Function definitions should be arranged (where possible) to obviate the need for `forward` declarations. Grouping functions logically so that functions with similar or exactly opposite effects are near each other is also desirable. (e.g. `semaphoreGive` should be near `semaphoreTake`.

2.4 Style

Coding style refers to the actual layout of code in a module. It is traditionally a rather contentious issue that rarely lends significant value to a project. The use of C styling programs such as `indent(1)` can often subvert and make irrational previously well formed code. Thus, the following set of guidelines is presented to offer a minimal standard without cramping individual style:

- Be consistent. Use the same coding style throughout a project.
- Use indentation to make control structures more visible. Using 4 characters per indentation level is suggested.
- Use vertical whitespace to visually break up logical portions of code.
- Separate binary operators (+,-, *, etc.) with a space.
- When modifying someone else's code, use *their* coding style throughout (even if you, personally, find it hideous).
- Comment code through out, even beyond the block comments described earlier in this document. When commenting a block of code, indent the level of comment to the same level as the code.
- Except in rare cases, place one statement per line.

2.5 Code Documentation

There are two distinct types of documentation for CIRSSE Testbed software. The first is the CIRSSE Technical Memo and the second is the online manual page. These will be considered individualy.

2.5.1 CIRSSE Technical Memos

The CIRSSE Technical memo is the medium through which the overall design philosophy and functionality for a particular library or application can be described. The content of the memo is very much up to the author of the software but should spend time placing the software in the context of the CIRSSE testbed, and should provide a high level overview of the software. In particular, such technical memos should spend time explaining to a reader and potential user of the software, why the software is useful and the philosophical underpinnings of its existence.

To produce a technical memo, get the file

```
/usr/local/lib/techmemo-template.tex
```

and edit it to suit your work. You may then use LaTeX to format the document. Documentation on the use of the LaTeX formatting package can be obtained from the system administrator.

Once you have produced your Technical Memo it is a good idea to send it our for review to interested parties (for example, a Technical Memo that deals with software for the Motion Control System might best be sent to the MCS design team). Once you are completely satisfied with your memo, you can contact the Technical Memo administrator and have your memo published.

2.5.2 Online Manual Pages

As their name implies, online manual pages are available for viewing electronically. Their intent to provide a quick reference to users of your software as to its purpose, calling conventions, return values etc.

All of the testbed manual pages are produced (as are the UNIX manual pages) using the `man` macros for the text formatting program `nroff`.

The following is a skeleton of manual page in it's pre-formatted state:

```
.TH name section "date"
.SH NAME
.SH SYNOPSIS
.SH DESCRIPTION
.SH OPTIONS or .SH RETURNS
.SH FILEs
.SH SEE ALSO
.SH DIAGNOSTICS
.SH BUGS
.SH AUTHOR
```

Each of these sections has the following meaning:

.TH name section "date" There are three parameters to the Title/Heading (.TH) command. The first is the name of the manual page itself. This should be the name of the command or function that the manual page is associated with. Following this is the section of the manual system that the manual page belongs in. The sections are divided as follows:

1 Commands

2 System Services

3 User Level Library Functions

4 Device Drivers, Protocols & Network Interfaces

5 File Formats

6 Games and Demos

7 Miscellaneous Useful information

8 System Maintenance and Administration

Following the section number should be the date the manual page was last updated in the format `"DD MMM YY"`. Note that the date must be contained in quotation marks.

.SH NAME The line following this directive should contain the name of the command or function (which should be the same as the name used in the .TH directive) followed by a short (one sentence) description of the command.

.SH SYNOPSIS The lines followed by this directive should contain a short synopsis of the command and functions followed by its arguments.

.SH DESCRIPTION A narrative describing the function of the command or function should follow this directive.

.SH OPTIONS For commands, the options that modify the command's behavior should be enumerated in this section.

.SH RETURNS For functions, the possible return values of the function should be described in this section.

.SH FILES This section should describe (and name) and files that the command uses or creates.

.SH SEE ALSO This section should list other relevant manual pages

.SH DIAGNOSTICS Any error messages that the command can generate should be documented in this section.

.SH BUGS Known deficiencies should be described here.

.SH AUTHOR The author or authors of the command or function should have their names listed here.

Note that in some cases a particular section of the manual page may be omitted as not relevant. Further, in some cases it is worthwhile to add a section not described here. The goal of the manual page is to provide useful information in a consistent manner. For further information on the creation of online manual pages see the "Formatting Documents" section of the *SunOS Documentation Tools* volume of the SunOS documentation set. By far, the best way to create a manual page is to obtain a sample manual page for a similar command and modify it to suit. CIRSSE testbed manual pages can be found in the testbed area under

```
~testbed/CIRSSE/installed/man/man?/
```

where ? is replaced with a section number.

3 Development Environment

The basic tools for CIRSSE's Testbed development environment are the Free Software Foundation's C compiler packages and MIT's Imake. In the following description of the development environment, the assumption will be made the the reader is familiar with the UNIX `make` command and the use of "makefiles" in general.

3.1 Imake

`Imake` is a program developed at MIT for the Athena Project and is currently available with the MIT X Window System distribution. The purpose of `imake` is to allow a developer to concentrate on the development of his or her code without concern for configuration details of a project. The configuration details of a project consist of the commands and options used to build software programs and libraries, the directories into which finished code should be installed, the locations of libraries, header files and commands and the names and locations of the tools needed to successfully build software for the project.

To use `Imake` here at CIRSSE, the user must first build a file named `Imakefile`. This file specifies the names and interdependencies of the files that make up the software package. This `Imakefile` can then be converted to a `Makefile` through the use of the `cmkmf` command. Once the `Makefile` is produced, the standard `make` make command can be used to build the software.

While much of the information contained in an `Imakefile` is the same as that contained in a standard `Makefile` it is generally specified in a much different way. Available to the writer of an `Imakefile` is a set of macros that automatically specify the appropriate build rules for a particular piece of software. Some of the more useful macros in the CIRSSE Imake system follow:

AllTarget(targets) This macros should simply contain a blank separated list of targets that should be constructed when the `make all` command is used.

UNIXBinTarget(target,inclist,objlist) This macro specifies the name and dependencies of a UNIX command. The "target" is the name of the actual command. "Inclist" should be a blank separated list of the "header" files that the command depends on while "objlist" should be a blank separated list of ".o" files that the command depends on. Note that "inclist" can be empty. It might be somewhat clearer to those familar with traditional `make` to translate this macro into the `make` text that will be produced.

Consider the following macro definition in an `Imakefile`:

```
UNIXBinTarget(ctosboot,ctos.h ctosunix.h,ctos.o parser.o process.o)
```

This will produce a `Makefile` target similar to the following:

```
ctosboot : ctos.o parser.o process.o ctos.h ctosunix.h
```

UNIXLibTarget(target,inclist,objlist) In this macro, "target" is the name of a UNIX object library (".a") file. "Inclist" and "objlist" have the same form and meaning as they do in **UNIXBinTarget**.

VxWorksBinTarget(target,inclist,objlist) This macro is used for the construction of VxWorks shell commands (e.g. `pumaDiagnostics`). "Inclist" and "objlist" serve the same purpose as the do for the UNIX macros. In the case of VxWorks, however, it is very common for "target" to have a ".o" extension. If this is the case do *not* place the same ".o" file name in "objlist" as this will cause circular dependencies. In this case leave "objlist" empty or omit the target ".o" file from that list.

VxWorksLibTarget(target,inclist,objlist) Due to the nature of VxWorks load files, this macro and the VxWorksBinTarget are functionally identical. They do, nevertheless, serve to distinguish the functionality of various targets.

- **VxWorksShareTarget(target,inclist,objlist)** This macro is similar to the VxWorksLibTarget macro save that it uses the UNIX linker loader to alter the module to be appropriate for loading into shared memory on multiple processors (see the ctos_config manual page for details on the SHARE configuartion command).

DatacubeBinTarget(target,inclist,objlist) This macro is similar to the VxWorksBinTarget save that it "knows" to search the Datacube include and library directories when building a target.

DatacubeLibTarget(target,inclist,objlist) This macro is similar to the VxWorksLibTarget save that it "knows" to search the Datacube include and library directories when building a target.

- **DatacubeShareTarget(target,inclist,objlist)** This macro is similar to the DatacubeLibTarget macro save that, like the VxWorksShareTarget macro, it uses the UNIX linker loader to alter the module to be appropriate for loading into shared memory on multiple processors.

There are also macro that are used to insure that particular targets are installed in their correct location when the software is "published" in the ~ `testbed` directory. When these are used a `make install` command will install the targets in their appropriate public directory:

manInstall(manlist) In this macro, "manlist" is a blank separated list of manual pages that should be installed in the manual page directory tree. The **manInstall** macro determines the correct manual page directory by reading the `.TH` directive in the actual manual pages.

UNIXBinInstall(binary) The indicated binary should be a UNIX command that will be installed in the "bin" directory of the appropriate UNIX architecture. There are similar macros **VxWorksBinInstall** and **DatacubeBinInstall**

UNIXLibInstall(binary) The indicated binary should be a UNIX command that will be installed in the "bin" directory of the appropriate UNIX architecture.There are similar macros **VxWorksLibInstall** and **DatacubeLibInstall**

For details on other macros available with the CIRSSE Imake configuration see the `CONFIG` manual page in the testbed manual pages. Examples of the various kinds of `Imakefiles` can be found in the directories underneath

`~testbed/CIRSSE/src/samples/`

Further, all of the directories underneath the testbed `src` sub-branch contain `Imakefiles` that are used to build their associated projects. As with manual pages, finding an `Imakefile` for a similar project and modifying it to suit your own needs is the most effective way to produce a correct `Imakefile`.

Once an Imakefile has been created, the following `cmkmf` commands are of use:

cmkmf all Build all targets contained in the macro `AllTargets`

cmkmf clean Remove temporary and re-buildable files (e.g. ".o" files)

cmkmf install Install targets in their appropriate public directories. Generally, this command is only of use to the testbed administrator.

The following items are also of note to users of the system:

- The `cmkmf` command works by simply converting an `Imakefile` to a `Makefile` and passing its arguments on to `make`. Thus any `make` argument is a possible `cmkmf` argument.
- If no changes to the `Imakefile` are made, the `make` command can be used directly.
- The **UNIX***, **VxWorks*** and **Datacube*** macros cannot be used in the same `Imakefile`. UNIX, VxWorks and Datacube projects should be kept in separate directories for clarity.
- The CPPFLAGS `make` macro can be redefined to add new search directories for include files. The proper syntax would be to put a line similar to the following near the beginning of the `Imakefile`

  ```
  CPPFLAGS += /home/lefebvre/vxworks/bootstrap
  ```
- There are two types of comments that may be place in an `Imakefile`. With the first type, each line is preceded with a `/**/#`, these comments are copied in to the `Makefile` as standard comments. You may also place standard C language style comments in the `Imakefile`. This type of comment, however, is not copied to the created `Makefile`.
- Any text in an `Imakefile` that does not comprise a comment or an `Imakefile` macro is simply copied into the resulting `Makefile`. Thus, custom targets etc. can be kept in an `Imakefile` and will find thier way into the `Makefile`.

For more details on CIRSSE Testbed configuration management see the CIRSSE testbed manual pages for: CONFIG(7), Imakefile(7) and `cmkmf(7)`.

NAME

ctos_boot_phases

SYNOPSIS

The CIRSSE Testbed Operating System (CTOS) supports the startup of distributed applications by stepping through several startup phases. Certain characteristics of the state of the system are guaranteed for each phase.

DESCRIPTION

CTOS boot phases are initiated by the broadcast messages: MSG_PINIT, MSG_AINIT, and MSG_AEXEC representing the Process INITialization, Application INITialization, and Application EXECution phases. Each of these boot phases are described below.

PINIT PHASE

The Process Initialization phase is begun after all CPUs on the chassis have processed their configuration files. This is an opportunity to initialize individual processes (tasks).

All tasks have been created and all CTOS functions are guaranteed to be available.

Because all tasks perform process initialization concurrently, they will complete initialization in an unpredictable order; for this reason no initialization BETWEEN processes should be done during PINIT phase.

AINIT PHASE

The Application Initialization phase begins after all tasks have completed PINIT processing. This is when you should perform initialization between processes. Now is a good time to use the msgTidQuery function to find the TID of a task's communication partners, and to store the TID for future use.

All tasks are guaranteed to have completed Process Initialization.

Three somewhat different event handler structures are possible to accomodate different AINIT processing requirements:

1) No application initialization required

there should be no 'case MSG_AINIT:' in event handler 'switch' - msgDefaultProc responds to message.

2) No messages required during application initialization

   put all AINIT processing in 'case MSG_AINIT:' - the case should end with 'break' so that msgDefaultProc responds to message.

3) Must receive messages during application initialization

   put initial AINIT processing in 'case MSG_AINIT:' and end case with 'return(0)' to bypass msgDefaultProc, thereby postponing acknowledgement of the AINIT message.

   when AINIT processing is complete, call msgAckAINIT to explicitly acknowledge the AINIT message.

The distinguishing feature between cases 2) and 3) is whether messages need to be received; messages may be sent out and replies may be received in both cases.

AEXEC PHASE

The Application Execution phase begins after all tasks have completed AINIT processing. As the name suggests this is when the application is executed.

All tasks are guaranteed to have completed Application Initialization.

Most event handler tasks will not process MSG_AEXEC but instead will respond to application-specific messages [see message_commands manual pages] that request particular actions. Likely there will be one task responding to MSG_AEXEC that then takes control of the application and issues the application-specific messages.

IMPORTANT NOTE

The broadcast messages initiating PINIT and AINIT must be acknowledged to confirm that every task has completed the phase. Therefore, at the end of EVERY event handler function there MUST be a call to msgDefaultProc in order to obtain correct handling of system messages. If the CTOS system fails to complete the boot process after configuration files are read, the most likely cause is an event handler function improperly responding to a system message such as PINIT or AINIT.

SEE ALSO

message_commands(2) msgDefaultProc(2) ctos_config(2)
msgAckAINIT(2)

AUTHOR

Don Lefebvre

NAME

ctos_config

SYNOPSIS

The CIRSSE Testbed Operating System (CTOS) is configured by specifying the distribution of software and processes via two configuration files. These two files, known as the system config file and the application config file, are read whenever CTOS is started. First, the system config file provides information to configure CTOS for a particular chassis; it is a read-only file maintained by the System Administrator. After the system config file is processed, CTOS reads your application config file to load software and start the processes of your application. [See the ctconfig manual pages for how to install your application config file]

DESCRIPTION

CTOS configuration files support the following commands:

```
load    - load object module into local memory
share   - load object module into shared memory
task    - create an event handler task
include - include another config file
chdir   - change present working directory
echo    - echo text to screen or turn off printing
logo    - specify file containing logo
connect - specify connections between cpus
```

All command lines in the config file have the same syntax:

```
CPU_NUMBER  COMMAND  COMMAND_ARGUMENTS.......
```

All CPUs on a chassis read the same configuration file, but do not process every command. The lines with CPU_NUMBER = 0 are processed by CPU 0, and the lines with CPU_NUMBER = 1 are processed by CPU 1, and so on. If CPU_NUMBER is set to -1 then the command is processed by all CPUs.

COMMANDs are separated from CPU_NUMBER by one or more spaces, and may be in upper or lower case. COMMAND_ARGUMENTS are similarly separated by space(s), and are different for different commands as described below.

Comment lines begin with '#' or ' ' in column one, and blank lines are ignored. Command lines MUST begin in column one.

CHDIR COMMAND

```
n -CHDIR  /path/
```

The CHDIR command changes the present working directory. Subsequent file reads, e.g. for a load command, will be performed from this directory.

CONNECT COMMAND

```
n  CONNECT  hostname  cpu_num
```

The CONNECT command specifies the socket interconnections that are to be built between cpus on a chassis. This command is only used in the system configuration file. YOU SHOULD NOT USE THIS COMMAND IN YOUR APPLICATION CONFIGURATION FILE.

ECHO COMMAND

```
n  ECHO  ON | OFF | text
```

The ECHO command effects what is printed to the console display during config file processing. An ECHO OFF command will turn off information and warning messages, but error messages will be displayed. ECHO ON or ECHO followed by text will cause all messages to be printed. The text following ECHO is printed to the console, providing a convenient method of displaying comments in the config file. Config file processing begins with echo turned on.

INCLUDE COMMAND

```
n  INCLUDE  /path/filename
```

The INCLUDE command suspends processing of the current config file and begins processing of the config file specified in the command argument. Processing of the original config file resumes after completion of the included config file. Config file includes can be nested, i.e. you can put an INCLUDE command inside an included file.

The full /path/ to a file is used if given, otherwise the file on the current working directory is read [see CHDIR command].

LOAD COMMAND

```
n  LOAD  /path/filename
```

The LOAD command loads an object file into local processor memory. The order in which object modules are loaded is important. Basically, the object code for a C function MUST be loaded before loading object code which calls that C function. Additionally, all C functions used by a task MUST be loaded before the task is created [see TASK command below]. Loads by both LOAD and SHARE [see below] should be accounted for when determining the order of loading object files.

The full /path/ to a file is used if given, otherwise the file on the current working directory is read [see CHDIR command].

LOGO COMMAND

```
n  LOGO  /path/filename
```

The LOGO command provides a means to display a logo when the application starts (when AEXEC begins). The logo should be defined in a readable file 79 columns wide by 15-20 lines long.

The full /path/ to the logo file is REQUIRED because the current working directory when the command was read is not remembered.

SHARE COMMAND

```
n  SHARE  /path/filename  memory_hex_address | 0
```

The SHARE command loads an object file at a specified memory location; its primary use is to initialize shared memory. The order of loading object files follows the same rules as for the LOAD command. Loads by both LOAD [see above] and SHARE should be accounted for when determining the order of loading object files.

Be careful to avoid loading object modules at overlapping addresses. One way to prevent this potential problem is to link together all objects going into shared memory and load them as a single module. However, the preferred way to avoid memory address conflicts is to specify a hex address of 0 (zero). When the address is zero the object file will be loaded at an address immediately following the previous SHARE file. Since the CTOS system loads some shared memory items and hence will initialize the SHARE address, you can specify a hex address of 0 in your application configuration file for virtually all cases.

IMPORTANT: when the "zero hex address" option is used, the SAME shared object files must be loaded in the SAME order on all cpus in order to get the correct addresses.

The memory_hex_address argument (if non-zero) must be specified in hexadecimal format; specifically, the address must begin with 0x, i.e. 0x1600000.

The full /path/ to a file is used if given, otherwise the file on the current working directory is read [see CHDIR command].

TASK COMMAND

```
n TASK symbolic_name function_call priority
```

The TASK command creates an event handler process with a message queue and a unique task id (TID) that serves as its address for message routing.

The TID of any task can be found from its symbolic name, hence the symbolic_name argument must be unique throughout an application. Length of symbolic names should be limited to 24 characters.

The function_call argument specifies the name of the C function that executes the event handler code. [See the manual pages for msgDefaultProc for a description of event handler format.] Any number of tasks may be created that call the same C function provided that each task has a unique symbolic name and that the C function is reentraint. Length of function_call C function names should be limited to 32 characters.

As noted above for the LOAD command, all C functions used by a task MUST be loaded before the task is created. As long as this requirement is met, LOAD and TASK commands can be intermixed in the config file.

The priority argument specifies the priority at which the task will execute. Tasks may have priorities ranging from 0 (highest priority) to 255 (lowest). Priorities below 100 are reserved for CTOS and VxWorks processes, so normal application tasks will have priorities in the range of 100 to 255.

IMPORTANT NOTE

A portion of CTOS itself is loaded via a system configuration file. In an earlier version of CTOS it was necessary to INCLUDE this system configuration file; however, this is no longer required and your configuration file should not include 'ctos_system_config'.

SEE ALSO

ctconfig(1) ctos_boot_phases(2)

AUTHOR

Don Lefebvre

NAME

message_commands

SYNOPSIS

The .command member of the MSG_TYPE structure is used to indicate the function of a message.

DESCRIPTION

The contents of a message are defined by the MSG_TYPE struct shown below. The .command member of the message is used to indicate its function. For instance, when the msgTidQuery function is called, it sends a message command of MSG_QUERY_TID to the Tid Server on CPU 0. This command informs the Tid Server that it should look up a TID and reply to the message sender. The other members of the MSG_TYPE struct are defined in the manual pages for msgLib and message_flags.

```
struct MSG_TYPE
    {
    TID_TYPE    dest     ;
    TID_TYPE    source   ;
    CMD_TYPE    command  ;
    void       *data     ;
    int         datasize ;
    FLAG_TYPE   flags    ;
    }
```

Commands are declared as type CMD_TYPE which is a 2-byte unsigned integer - allowing the definition of over 65,000 unique commands. Customarily, the message .command is equated to a predefined constant when the message is built. For example, the msgLib.h header contains the definition:

```
     #define  MSG_TID_QUERY    MSG_STANDARD+4
```

And, a message using this command would use this definition as the command value in an assignment, msgBuild or msgCommandSet function call:

```
     msg.command = MSG_TID_QUERY ;
or
     msgBuild (&msg,  ,  , MSG_TID_QUERY, ... ) ;
or
     msgCommandSet (&msg, MSG_TID_QUERY) ;
```

The above message command definition also illustrates two conventions that have been adopted for the CIRSSE testbed operating system:

1) Command symbolic names are uppercase and begin with MSG_

2) Commands values are assigned as offsets to blocks of commands, e.g. the MSG_TID_QUERY command is the 4th in the MSG_STANDARD block.

By assigning command values by blocks we can manage commands to ensure that they are unique across the CTOS system and all applications.

USER-DEFINED COMMANDS

Users may create their own messages by defining commands in the MSG_USER block:

```
#define  MSG_PID_LOOPS       MSG_USER
#define  MSG_MY_MESSAGE      MSG_PID_LOOPS+1
#define  MSG_ANOTHER_MSG     MSG_PID_LOOPS+2
```

If your application is later adopted as a standard application, the CTOS system administrator will assign MSG_PID_LOOPS to its own block so that other users can access the application without a conflict of command values.

STANDARD MESSAGES

There are several standard messages that all event handler tasks should respond to - these are listed below. For convenience and to ensure uniform response, a msgDefaultProc function is provided to perform default processing of standard messages. Your event handler function should pass standard messages to msgDefaultProc after completing your application-specific processing of the message. [See msgDefaultProc manual pages or DECODING MESSAGES topic below for an example]

STANDARD COMMANDS

MSG_PINIT	- Broadcast message begining process initialization phase (must be acknowledged).
MSG_AINIT	- Broadcast message begining application initialization phase (must be acknowledged).
MSG_AEXEC	- Broadcast message begining application execution phase.

DECODING MESSAGES

Most event handler functions will have a similar structure as suggested by the example shown below. Specifically, the function must decode the .command member of the message, perform the appropriate processing, and pass standard messages plus "unrecognized" messages to the default processing. Message commands are "decoded" via a SWITCH statement in which each recognized command is a CASE. If the event handler function performs all required processing, then the CASE ends with RETURN(0); otherwise, the CASE should end with BREAK so that the message can be passed to msgDefaultProc.

```
int  UserEventHandler (TID_TYPE myTid, MSG_TYPE *msg)
    {
    switch (msg->command)
        {
        case MSG_AINIT:
            /*  application-specific AINIT processing  */
            break ;

        case MSG_MY_MESSAGE:
            /*  process this message command  */
            return (0) ;

        case MSG_ANOTHER_MSG:
            /*  process this message command  */
            return (0) ;
        }
    return (msgDefaultProc (myTid, msg)) ;
    }
```

SEE ALSO

msgLib(2) msgDefaultProc(2) ctos_boot_phases(2)

AUTHOR

Don Lefebvre

NAME
message_flags

SYNOPSIS
The .flags member of the MSG_TYPE structure is used to specify handling of a message.

DESCRIPTION
The contents of a message are defined by the MSG_TYPE struct shown below. The .flags member of the message is used to specify how the message and message data are handled. The other members of the MSG_TYPE struct are defined in the manual pages for msgLib and message_commands.

```
struct MSG_TYPE
     {
     TID_TYPE    dest     ;
     TID_TYPE    source   ;
     CMD_TYPE    command  ;
     void       *data     ;
     int         datasize ;
     FLAG_TYPE   flags    ;
     }
```

Message flags have five fields:

TYPE	- indicates normal, reply, or broadcast
PRIORITY	- urgent msgs go to front of dest queue
REPLY_WAIT	- if set, task blocks and waits for reply
SEND_WAIT	- optionally waits if dest queue is full
MEMOWNER	- specifies who deallocates message data

Message flags are created by OR'ing together defined constants for each field - either in an assignment statement or by using the flag building routines in msgLib. [See FLAGS MANIPULATION FUNCTIONS topic below] Message flag masks are also defined in the msgLib.h header to assist flag manipulation, but their use is discouraged.

TYPE FIELD
The type field of .flags selects the messaging mechanism that will be used to deliver the message. Message type should not be confused with message priority. With only extremely rare exceptions, users will specify MF_TYPE_NORMAL. A normal message is routed to the message queue of the destination task directly through msgSend and/or the MsgDispatcher.

Reply messages, on-the-other-hand, are not put in a queue;

rather, they are sent to a separate storage location for replies, and the original message sender (now the destination of the reply) is unblocked. [See REPLY_WAIT FIELD topic below] The MF_TYPE_REPLY field is set by the msgReply function, and used by the system to effect this alternate routing.

Similarly, the type field of broadcast messages are set by the system and cause messages to be differently routed. [See msgBroadcast manual pages for a description]. These message types should not be used by application developers.

DEFINED CONSTANTS:

MF_TYPE_NORMAL	- normal message, ALL USER MESSAGES SHOULD BE OF THIS TYPE.
MF_TYPE_REPLY	- reply message, USED BY SYSTEM.
MF_TYPE_BC_NORMAL	- broadcast message, USED BY SYSTEM.
MF_TYPE_BC_REPLY	- reply to broadcast message, USED BY SYSTEM.

PRIORITY FIELD

The priority field of .flags specifies how the message is added to the message queue of the destination task. Messages are normally added to the back of the queue, where they must wait until the destination task has processed all proceeding messages. When a message is given urgent priority it is placed in the front of the queue and will be the next message processed after the destination task completes its current operation.

DEFINED CONSTANTS:

MF_PRI_NORMAL	- message sent to back of dest queue.
MF_PRI_URGENT	- message sent to front of dest queue.

REPLY_WAIT FIELD

The reply_wait field of .flags indicates whether the sending task wishes to wait for a reply. If reply_wait is not MF_REPLY_WAIT_NO, then the sending task will be blocked until a reply message is received or the wait times out. [See msgSend and msgReply manual pages for return values when reply_wait is set]

DEFINED CONSTANTS:

MF_REPLY_WAIT_NO	- do not wait for reply.
MF_REPLY_WAIT_FOR	- wait forever for reply.
MF_REPLY_WAIT_1SEC	- wait for reply, but timeout after 1 second.

SEND_WAIT FIELD

The send_wait field of .flags determines what the system does when the messaging system is busy or the destination task queue is full. With send_wait set to MF_SEND_WAIT_NO, the system will discard messages that it cannot deliver immediately. Most of the time this option is adequate. If the message absolutely must get through, then use MF_SEND_WAIT_FOR; but be warned that the sending task my be blocked waiting to send the message. [See msgPost manual pages for an alternative that ensures the sending task will not block]

DEFINED CONSTANTS:

MF_SEND_WAIT_NO	- do not wait to send message.
MF_SEND_WAIT_FOR	- wait forever to send message.

MEMOWNER FIELD

One of the primary uses of a message is to transmit a pointer to additional data. The CIRSSE messaging system provides a means to manage this "message data". The MEMOWNER field of msg.flags specifies whether the sending task or the receiving task has the responsibility for deal-locating message data storage. If MF_MEMOWNER_RECEIVER is chosen, then the storage allocated to message data is AUTOMATICALLY deallocated by the event handler shell follow-ing processing of the message. [Note that the msgDataKeep function can be used by the receiving task to prevent automatic deallocation] With MF_MEMOWNER_SENDER the sending task must deallocate message data storage if desired, no automatic deallocation occurs.

DEFINED CONSTANTS:

MF_MEMOWNER_RECEIVER	- event handler shell automatically deallocates message data memory.
MF_MEMOWNER_SENDER	- message sender is responsible for deallocating message data memory.

PREDEFINED MESSAGE FLAGS

For convenience, flags for the most common cases have been defined. Most messages can use the predefined standard flag:

```
MF_STANDARD   =  MF_TYPE_NORMAL     |  MF_PRI_NORMAL      |
                 MF_REPLY_WAIT_NO   |  MF_SEND_WAIT_NO    |
                 MF_MEMOWNER_RECEIVER
```

Other predefined message flags are:

```
MF_REPLYWAIT  =  MF_TYPE_NORMAL     |  MF_PRI_NORMAL      |
                 MF_REPLY_WAIT_FOR  |  MF_SEND_WAIT_NO    |
                 MF_MEMOWNER_RECEIVER

MF_SYSTEM     =  MF_TYPE_NORMAL     |  MF_PRI_URGENT      |
                 MF_REPLY_WAIT_NO   |  MF_SEND_WAIT_FOR   |
                 MF_MEMOWNER_RECEIVER
```

FLAGS MANIPULATION FUNCTIONS

Message .flags are been designed with expansion in mind, and may be redefined in future. For this reason it is important to avoid direct manipulation of .flags fields. Users are encouraged to only use flag manipulation functions [listed in the "Building Messages" section of msgLib manual pages] or predefined message flags.

For example, to specify that the task is to wait for a reply, we would use the following:

```
msg.flags = msgReplyFlagSet (msg.flags, MF_REPLY_WAIT_FOR);
```

SEE ALSO

msgLib(2) msgSend(2) msgFlag_macros(2)

AUTHOR

Don Lefebvre

NAME

msgAckAINIT - explicitly acknowledge AINIT message

SYNOPSIS

```
STATUS  msgAckAINIT (TID_TYPE tid)


TID_TYPE  tid   - task id of task calling msgAckAINIT
```

DESCRIPTION

For some event handler tasks it is desirable to postpone acknowledging the AINIT message so that the task can receive other messages during the Application Initialization phase. Once the task has completed Application Initialization it MUST acknowledge the AINIT message so that CTOS can continue with its next boot phase. The msgAckAINIT function should be used to generate the AINIT acknowledgement message.

Please read the AINIT PHASE topic of the ctos_boot_phases manual pages for tips on receiving messages during the Application Initialization phase.

RETURNS

OK or ERROR indicating success of sending acknowledgement.

SEE ALSO

ctos_boot_phases(2) msgLib(2) msgDefaultProc(2)

AUTHOR

Don Lefebvre

NAME

msgAcknowledge - acknowledge a received message

SYNOPSIS

```
STATUS  msgAcknowledge ( MSG_TYPE  *msg )


MSG_TYPE  *msg      - pointer to received message
```

DESCRIPTION

msgAcknowledge is used to reply to a message without returning data. It is implemented as a macro:

```
     msgReply (msg, NULL, MS_NONE, MF_STANDARD)
```

RETURNS

OK or ERROR indicating success of sending out reply.

SEE ALSO

msgLib(2) msgSend(2) msgReply(2)

AUTHOR

Don Lefebvre

NAME

msgBroadcast - broadcast message to all tasks

SYNOPSIS

```
int  msgBroadcast (MSG_TYPE *msg, FLAG_TYPE scope)

MSG_TYPE  *msg   - pointer to message to be sent
FLAG_TYPE  scope - scope of broadcast (defined constant)
```

DESCRIPTION

msgBroadcast sends a message to many tasks at once. The broadcast can be limited to the local CPU or chassis, or may be broadcast application-wide.

The scope of the broadcast is specified as one of the following defined constants:

```
MB_LOCAL    - broadcast to local cpu only
MB_CHASSIS  - broadcast to local chassis only
MB_SYSTEM   - broadcast to entire application
```

When msgBroadcast is called, a message is sent to the local Msg Server (local scope) or to the TID Server (chassis & system scope) requesting that the message be broadcast. These requests are distributed to all Msg Servers on the chassis; and, for system-wide broadcasts, to all TID Servers throughout the system. Acknowledgements of REPLY_WAIT broadcast messages are gathered via the reverse routing. IMPORTANT: the message is NOT sent to the task which called msgBroadcast, so that deadlock is prevented when REPLY_WAIT is set.

All message flags are supported; for instance, a task can broadcast a message and wait for all tasks to reply. However, msgBroadcast cannot return data to a waiting task. REPLY_WAIT should be used with caution as replies MUST be received from ALL tasks. The default processing provided by msgDefaultProc does acknowledge broadcast messages - making use of REPLY_WAIT practical.

Typically a task acknowledges a broadcast through a call to msgAcknowledge (often from within msgDefaultProc) which replies with a null msg.data pointer. To facilitate counting a subset of responding tasks, the msgBroadcast function will return a positive integer which is the count of tasks that replied with non-null msg.data pointers. Thus, to have a task counted it should reply with something like msgReply(msg, (void *) 1, MS_KEEP_ADRS, MF_STANDARD).

RETURNS

Positive integer indicating number of non-zero acknowledgements received; or

-1 when error occurred during broadcast.

SEE ALSO

msgLib(2) msgSend(2) msgBuild(2) message_flags(2)

AUTHOR

Don Lefebvre

NAME

msgBuild - build message by filling message structure

SYNOPSIS

```
MSG_TYPE  *msgBuild ( MSG_TYPE  *msg      ,
                      TID_TYPE   dest     ,
                      TID_TYPE   source   ,
                      CMD_TYPE   command  ,
                      void      *data     ,
                      int        datasize ,
                      FLAG_TYPE  flags     )


MSG_TYPE  *msg      - pointer to message struct or NULL
TID_TYPE   dest     - address of destination task
TID_TYPE   source   - address of task sending message
CMD_TYPE   command  - message command
void      *data     - pointer to additional message data
int        datasize - number of bytes in message data
FLAG_TYPE  flags    - message flags
```

DESCRIPTION

msgBuild provides a convenient way to define a message. The arguments to msgBuild are used to define the members of the message structure, whose address is passed in as the first argument. If *msg == NULL then msgBuild will allocate storage.

For a description of message structure members see the manual pages for msgSend, message_commands, and message_flags.

RETURNS

Pointer to message that was built.

SEE ALSO

msgLib(2) msgSend(2) message_commands(2) message_flags(2)

AUTHOR

Don Lefebvre

NAME

msgBuildSend - build then send a message

SYNOPSIS

```
int  *msgBuildSend ( TID_TYPE   dest     ,
                     TID_TYPE   source   ,
                     CMD_TYPE   command  ,
                     void      *data     ,
                     int        datasize ,
                     FLAG_TYPE  flags      )
```

TID_TYPE	dest	- address of destination task
TID_TYPE	source	- address of task sending message
CMD_TYPE	command	- message command
void	*data	- pointer to additional message data
int	datasize	- number of bytes in message data
FLAG_TYPE	flags	- message flags

DESCRIPTION

As the function name suggests, msgBuildSend is a combination of msgBuild and msgSend. The arguments to msgBuildSend are used to define the members of a message structure, and then the message is sent.

Internally, msgBuildSend allocates storage for the message, and later frees it after the message is sent.

For a description of message structure members see the manual pages for msgSend, message_commands, and message_flags.

RETURNS

Same as msgSend.

SEE ALSO

msgLib(2) msgBuild(2) msgSend(2) message_commands(2) message_flags(2)

AUTHOR

Don Lefebvre

NAME

msgCopy - make local copy of message

SYNOPSIS

```
MSG_TYPE  *msgCopy ( MSG_TYPE *msg )

MSG_TYPE  *msg - pointer to received message to be copied
```

DESCRIPTION

The msgCopy function allocates memory and copies the message pointed to by *msg. This is the only mechanism for retaining a message between successive calls of an event handler function because a message is normally lost when the event handler function exits. [Remember to use a static variable for the new message pointer]

msgCopy DOES NOT COPY MESSAGE DATA. Use msgDataCopy or msgDataKeep to copy message data.

RETURNS

pointer to copy of message

SEE ALSO

msgLib(2) msgSend(2) msgDataCopy msgDataKeep

AUTHOR

Don Lefebvre

NAME
    msgDataCopy - make local copy of message data

SYNOPSIS
```
void  *msgDataCopy ( MSG_TYPE *msg )

MSG_TYPE  *msg - pointer to message whose data is to be copied
```

DESCRIPTION

The msgDataCopy function allocates memory and copies the message data pointed to by msg->data.

When a message is received with its MEMOWNER flag set to SENDER the message data should be considered to be READ ONLY. In this case you might want to use msgDataCopy to get a copy of the data that you can change.

When a message is received with its MEMOWNER flag set to RECEIVER then the receiving task "owns" the data and can change it as desired. However, when the current call to the event handler function exits the message data will be automatically deallocated (unless msgDataKeep was called). In this case you may want to copy the message data to retain it between calls to the event handler function.

msgDataCopy DOES NOT COPY THE MESSAGE ITSELF. Use msgCopy to copy the message.

Note that the receiving task now has the responsibility to deallocate the memory used by the copied message data after it is no longer needed.

RETURNS

pointer to copy of message data

SEE ALSO

msgLib(2) msgSend(2) msgDataKeep msgCopy

AUTHOR

Don Lefebvre

NAME

msgDataKeep - keep message data by preventing deallocation

SYNOPSIS

```
void  *msgDataKeep ( MSG_TYPE *msg )

MSG_TYPE  *msg - pointer to message whose data is to be kept
```

DESCRIPTION

The msgDataKeep function prevents the automatic deallocation of message data that occurs when an event handler function exits and the message's MEMOWNER flag is RECEIVER.

msgDataKeep performs somewhat differently in VME and UNIX versions:

For VME, msgDataKeep simply adjusts the MEMOWNER flag to prevent the deallocation. The msg->data pointer is not effected, but may point to memory on another CPU. If a local copy of the message data is desired use the msgDataCopy function.

For UNIX, msgDataKeep performs identically to msgDataCopy. TO MAINTAIN COMPATIBILITY in event handler functions that may run in either VME or UNIX environments, it is recommended that code be developed to account for the possibility that msgDataKeep may change the message data pointer to point to newly allocated memory.

Note that the receiving task may now have the responsibility to deallocate the memory used by the kept message data after it is no longer needed.

RETURNS

pointer to message data

SEE ALSO

msgLib(2) msgSend(2) msgDataCopy msgCopy

AUTHOR

Don Lefebvre

NAME

msgDefaultProc - default processing for system messages

SYNOPSIS

```
int  msgDefaultProc (TID_TYPE tid, MSG_TYPE *msg)


TID_TYPE   tid - address of current task
MSG_TYPE  *msg - pointer to received message
```

DESCRIPTION

The msgDefaultProc function provides default processing of system messages such as PINIT and AINIT in event handler functions. Even in cases where your event handler function responds to a system message (such as performing application initialization in response to AINIT), a call should be made to msgDefaultProc following your processing.

Most event handler functions will have a similar structure as suggested by the example shown below. Specifically, the function must decode the .command member of the message, perform the appropriate processing, and pass standard messages plus "unrecognized" messages to msgDefaultProc. Message commands are "decoded" via a SWITCH statement in which each recognized command is a CASE. If the event handler function performs all required processing, then the CASE ends with RETURN(0); otherwise, the CASE should end with BREAK so that the message can be passed to msgDefaultProc.

```
int  UserEventHandler (TID_TYPE myTid, MSG_TYPE *msg)
    {
    switch (msg->command)
        {
        case MSG_AINIT:
            /*  application-specific AINIT processing   */
            break ;

        case MSG_MY_MESSAGE:
            /*  process this message command   */
            return (0) ;

        case MSG_ANOTHER_MSG:
            /*  process this message command   */
            return (0) ;
        }
    return (msgDefaultProc (myTid, msg)) ;
    }
```

DEFAULT PROCESSING

msgDefaultProc provides the following default processing:

MSG_PINIT - acknowledges broadcast message.

MSG_AINIT - acknowledges broadcast message.

MSG_AEXEC - displays logo.

IMPORTANT NOTE

At the end of EVERY event handler function there MUST be a call to msgDefaultProc in order to obtain correct handling of system messages. If the CTOS system fails to complete the boot process after configuration files are read, the most likely cause is an event handler function improperly responding to a system message such as PINIT or AINIT.

RETURNS

[System return code to event handler shell]

SEE ALSO

msgLib(2) message_commands(2) ctos_boot_phases(2)

AUTHOR

Don Lefebvre

NAME

msgDequeue - read message directly from local queue

SYNOPSIS

```
STATUS  msgDequeue (TID_TYPE tid, MSG_TYPE *msg)


TID_TYPE   tid - task id of queue owner
MSG_TYPE  *msg - ptr to storage for dequeued message
```

DESCRIPTION

msgDequeue removes a message from the local message queue identified by tid. While it is possible to read a message from any task on the local CPU, it is recommended that a task only manipulate its own message queue.

Note that there is no checking that the input tid is local. If the tid is not local, the result is unpredictable.

RETURNS

```
OK:  if queue not empty, msg contains the dequeued message.
     if queue was empty, msg->command is set to MSG_QUEUE_EMPTY.

ERROR: dequeue operation failed, e.g. tid was invalid.
```

SEE ALSO

msgLib(2) msgRequeue(2) msgQueueCount(2)

AUTHOR

Don Lefebvre

NAME

msgErrorLog - send a string to the Error Server

SYNOPSIS

```
STATUS  msgErrorLog (TID_TYPE tid, char *string)


TID_TYPE   tid     - address of task calling msgErrorLog
char      *string - error message string
```

DESCRIPTION

msgErrorLog copies the error message string, and sends a message pointing to this string to the Error Server. At present, the Error Server simply prints the error message to the console.

Since msgErrorLog makes a copy of the input error string, the task calling this function does not need to maintain storage for *string.

RETURNS

OK or ERROR indicating result of msgSend to Error Server.

SEE ALSO

msgLib(2)

AUTHOR

Don Lefebvre

NAME

```
msgMemownerFlagSet - set MEMOWNER field of flag
msgPriorityFlagSet - set PRIORITY field of flag
msgReplyFlagSet    - set REPLY_WAIT field of flag
msgSendFlagSet     - set SEND_WAIT field of flag
msgTypeFlagSet     - set TYPE field of flag
```

SYNOPSIS

```
FLAG_TYPE  msgMemownerFlagSet (FLAG_TYPE flag, FLAG_TYPE field)

FLAG_TYPE  msgPriorityFlagSet (FLAG_TYPE flag, FLAG_TYPE field)

FLAG_TYPE  msgReplyFlagSet    (FLAG_TYPE flag, FLAG_TYPE field)

FLAG_TYPE  msgSendFlagSet     (FLAG_TYPE flag, FLAG_TYPE field)

FLAG_TYPE  msgTypeFlagSet     (FLAG_TYPE flag, FLAG_TYPE field)


FLAG_TYPE  flag  - base flag
FLAG_TYPE  field - new value for flag field (defined constant)
```

DESCRIPTION

These functions are used to manipulate the fields of a message .flags member. Message flags have five fields:

```
MEMOWNER    - specifies who deallocates message data
PRIORITY    - urgent msgs go to front of dest queue
REPLY_WAIT  - if set, task blocks and waits for reply
SEND_WAIT   - optionally waits if dest queue is full
TYPE        - indicates normal, reply, or broadcast
```

Please read the message_flags manual pages for descriptions of the actions defined by these flag fields, and the defined constants that are acceptable values for flag fields.

The actions of these functions are to replace the particular field of the base flag with a new value. For instance, the following function calls change the MEMOWNER field:

```
msg.flags = msgMemownerFlagSet (msg.flags, MF_MEMOWNER_SENDER);

msg.flags = msgMemownerFlagSet (MF_STANDARD, MF_MEMOWNER_SENDER);
```

RETURNS

flag resulting from changing 'field' of 'base flag'.

SEE ALSO
     msgLib(2) msgSend(2) message_flags(2)

AUTHOR
     Don Lefebvre

NAME

msgLib.[ch] - Messaging Routines

SYNOPSIS

```
----------------- SENDING MESSAGES -----------------------------
msgSend              - send message to event handler task
msgPost              - post message (returns immediately)
msgBroadcast         - broadcast message to all tasks
msgErrorLog          - send a string to the Error Server
msgReply             - reply to received message
msgAcknowledge       - acknowledge a received message
msgBuildSend         - build then send a message

----------------- BUILDING MESSAGES ----------------------------
msgBuild             - set all members of message structure
msgMemownerFlagSet   - set MEMOWNER field of msg.flag
msgPriorityFlagSet   - set PRIORITY field of msg.flag
msgReplyFlagSet      - set REPLY_WAIT field of msg.flag
msgSendFlagSet       - set SEND_WAIT field of msg.flag
msgTypeFlagSet       - set TYPE field of msg.flag

----------------- WORKING WITH TIDS ----------------------------
msgTidQuery          - find task id from symbolic task name
msgTidGetChassis     - get CHASSIS field of task id
msgTidGetCpu         - get CPU field of task id
msgTidGetLocal       - get LOCAL field of task id
msgTidSetChassis     - set CHASSIS field of task id
msgTidSetCpu         - set CPU field of task id
msgTidSetLocal       - set LOCAL field of task id

----------------- QUEUE OPERATIONS -----------------------------
msgDequeue           - read message directly from local queue
msgQueueCount        - count messages in local queue
msgRequeue           - put message directly into local queue

----------------- MEMORY MANAGEMENT ----------------------------
msgCopy              - make local copy of message
msgDataCopy          - make local copy of message data
msgDataKeep          - keep message data (prevents dealloc)
msgVarPtrSet         - set pointer to saved variables
msgVarPtrGet         - get pointer to saved variables

----------------- SPECIAL PROCESSING ---------------------------
msgAckAINIT          - explicit acknowledgement of AINIT message
msgDefaultProc       - default processing for system messages
```

DESCRIPTION

These msgLib functions are the interface to the CIRSSE testbed message passing system. They are the primary means for communicating between "event handler" tasks distributed over the CPUs of a VME chassis; and, eventually, between VME chassis and SUN workstations.

The steps to sending a message are 1) determine the task id (TID) of the message's destination, 2) build a message, and 3) send it. When an event handler task is created, it is given a unique TID which serves as its address. While each task "knows" its own TID, it needs to find the TID of its communication partners - the msgTidQuery function is used to do this. Rather than calling msgTidQuery every time a message is sent, a program will usually determine destination TIDs during initialization (AINIT phase) and save them for later use.

Building the message is the next step. The structure definition shown below lists the components of a message. The first two components are the TIDs of the destination of the message and its source. Next is the message command which describes the function of the message. For instance, when the msgTidQuery function is called it sends a message command of MSG_QUERY_TID to the Tid Server on CPU 0. This command informs the Tid Server that it should look up a TID and reply to the message sender. [For more discussion see the manual pages for message_commands]

The *data and datasize components of a message point to additional message data. Continuing the msgTidQuery example: the request to the Tid Server includes the symbolic name of the task whose TID is desired. This information is transmitted by setting the *data pointer equal to the character string containing the symbolic name, and datasize equal to the length of the string.

Lastly, the message flags specify options for handling the message such as whether to wait for a reply, who owns the message data, and priority of the message. [For more discussion see the manual pages for message_flags] Functions are available to assist in building a message and in defining values for individual members of the message struct or individual fields of task ids and message flags.

```
struct MSG_TYPE
    {
    TID_TYPE    dest     ;
    TID_TYPE    source   ;
    CMD_TYPE    command  ;
    void       *data     ;
    int         datasize ;
    FLAG_TYPE   flags    ;
    }
```

The final step in using the messaging system is to send out the message. The most basic form is msgSend which simply takes a pointer to the message as an argument. In most instances you will use msgSend or msgBuildSend (which, predictably, is a combination of msgBuild and msgSend). Use msgBroadcast to send a message to many tasks at once; the broadcast can be limited to the local CPU or chassis, or may be broadcast system-wide. To simplify error logging, the msgErrorLog function sends a string to the Error Server. The functions msgReply and msgAcknowledge are provided to return information or to acknowledge a received message. A rarely used function is msgPost; it is intended for low-level routines that need to transmit a message with ensurance that the sending task will not be delayed.

Message queue operations are provided to directly manipulate local message queues. Normally, a task should only access its own message queue.

Memory management functions provide means to manipulate storage of messages and message data, and to build reentrant event handler tasks.

To use the msgLib functions described here simply include the header file msgLib.h at the begining of your source code.

PROTOTYPES
```
     STATUS      msgAckAINIT        (TID_TYPE tid)

     STATUS      msgAcknowledge     (MSG_TYPE *msg)

     int         msgBroadcast       (MSG_TYPE *msg, FLAG_TYPE dest)

     MSG_TYPE   *msgBuild           (MSG_TYPE *message,
                                     TID_TYPE dest, TID_TYPE source,
                                     CMD_TYPE command, void *data,
                                     int datasize, FLAG_TYPE flag)

     int         msgBuildSend       (TID_TYPE dest, TID_TYPE source,
                                     CMD_TYPE command, void *data,
                                     int datasize, FLAG_TYPE flag)

     MSG_TYPE   *msgCopy            (MSG_TYPE *msg)

     void       *msgDataCopy        (MSG_TYPE *msg)

     void       *msgDataKeep        (MSG_TYPE *msg)

     int         msgDefaultProc     (TID_TYPE tid, MSG_TYPE *msg)

     STATUS      msgDequeue         (TID_TYPE tid, MSG_TYPE *msg)
     STATUS  -   msgErrorLog          (TID_TYPE tid, char *string)

     FLAG_TYPE   msgMemownerFlagSet   (MSG_TYPE *msg, FLAG_TYPE flag)

     STATUS      msgPost              (MSG_TYPE *msg)

     FLAG_TYPE   msgPriorityFlagSet   (MSG_TYPE *msg, FLAG_TYPE flag)

     int         msgQueueCount        (TID_TYPE tid)

     STATUS      msgReply             (MSG_TYPE *msg, void *data,
                                       int datasize, FLAG_TYPE flags)

     FLAG_TYPE   msgReplyFlagSet      (MSG_TYPE *msg, FLAG_TYPE flag)

     STATUS      msgRequeue           (TID_TYPE tid, MSG_TYPE *msg,
                                       FLAG_TYPE prty)

     int         msgSend              (MSG_TYPE *msg)

     FLAG_TYPE   msgSendFlagSet       (MSG_TYPE *msg, FLAG_TYPE flag)

     TID_TYPE    msgTidGetChassis     (TID_TYPE tid)

     TID_TYPE    msgTidGetCpu         (TID_TYPE tid)

     TID_TYPE    msgTidGetLocal       (TID_TYPE tid)

     TID_TYPE    msgTidQuery          (TID_TYPE tid, char *taskname)

     TID_TYPE    msgTidSetChassis     (TID_TYPE tid, int number)

     TID_TYPE    msgTidSetCpu         (TID_TYPE tid, int number)

     TID_TYPE    msgTidSetLocal       (TID_TYPE tid, int number)

     FLAG_TYPE   msgTypeFlagSet       (MSG_TYPE *msg, FLAG_TYPE flag)

     void       *msgVarPtrGet         (TID_TYPE t)

     STATUS      msgVarPtrSet         (TID_TYPE t, void *p)
```

SEE ALSO
     message_commands(2) message_flags(2)

AUTHOR
     Don Lefebvre

NAME
    msgPost - send message and return immediately

SYNOPSIS
    STATUS msgPost (MSG_TYPE *msg)

    MSG_TYPE *msg - pointer to message to be sent

DESCRIPTION
    msgPost copies the message and enqueues it for retransmission by the local Msg Server. By doing this msgPost can ensure that the task sending the message will not be delayed. However, because the message may sit in the Msg Server queue for a while, the message itself may be delayed. The msgPost function is primarily intended for low-level routines which need to avoid unpredictable delays.

    Sending a message while waiting for a reply is inconsistent with the spirit of msgPost; hence, the REPLY_WAIT field of msg.flags is ignored by msgPost.

RETURNS
    OK or ERROR indicating success of enqueuing the message at the Msg Server.

SEE ALSO
    msgLib(2) msgSend(2) msgBuild(2) message_commands(2) message_flags(2)

AUTHOR
    Don Lefebvre

NAME

msgQueueCount - count messages in local queue

SYNOPSIS

```
int  msgQueueCount (TID_TYPE tid)


TID_TYPE   tid  - task id of queue owner
```

DESCRIPTION

msgQueueCount counts the number of messages that are waiting on the local message queue identified by tid.

Note that there is no checking that the input tid is local. If the tid is not local, the result is unpredictable.

RETURNS

The number of messages in the message queue.

SEE ALSO

msgLib(2) msgDequeue(2) msgRequeue(2)

AUTHOR

Don Lefebvre

NAME

msgReply - reply to received message

SYNOPSIS

```
STATUS  msgReply ( MSG_TYPE  *msg      ,
                   void      *data     ,
                   int        datasize ,
                   FLAG_TYPE  flags      )

MSG_TYPE  *msg      - pointer to received message
void      *data     - pointer to reply data
int        datasize - size of reply data
FLAG_TYPE  flags    - message flags
```

DESCRIPTION

The msgReply function is used to reply to a received message. Its primary uses are to respond to requests, and to acknowledge synchronization messages.

When a task originates a message with the REPLY_WAIT flag set [See manual pages for msgSend] the task is blocked pending receipt of a reply. To unblock the originating task, the receiving task MUST call msgReply or msgAcknowledge.

The data pointed to by *data of msgReply is sent via the reply message and is received by the (now unblocked) originating task as the return value of msgSend. However, the *data pointer is ignored when replying to a broadcast message; so msgSend must be used (AFTER acknowledging the broadcast if required).

The flags argument can be used to specify message PRIORITY or SEND_WAIT fields; other options are ignored, in particular REPLY_WAIT. Unfortunately it is not possible to support MEMOWNER_RECEIVER for reply messages; thus reply data must be managed explicitly by the application.

RETURNS

OK or ERROR indicating success of sending out reply.

SEE ALSO

msgLib(2) msgSend(2) message_flags(2) msgAcknowledge(2)

AUTHOR

Don Lefebvre

NAME

msgRequeue - put message directly into local queue

SYNOPSIS

```
STATUS  msgRequeue (TID_TYPE tid, MSG_TYPE *msg, FLAG_TYPE prty)


TID_TYPE   tid  - task id of queue owner
MSG_TYPE  *msg  - ptr to message to be requeued
FLAG_TYPE  prty - message priority
```

DESCRIPTION

msgRequeue enqueues a message on the local message queue identified by tid. While it is possible to enqueue a message to any task on the local CPU, it is recommended that a task only manipulate its own message queue. One reason for this recommendation is that msgRequeue bypasses the additional processing provided by msgSend such as blocking on REPLY_WAIT.

Note that there is no checking that the input tid is local. If the tid is not local, the result is unpredictable.

The message priority may have values of MF_PRI_NORMAL or MF_PRI_URGENT, indicating whether the message will be placed at the back or front of the message queue, respectively.

RETURNS

OK or ERROR indicating success of enqueuing operation.

SEE ALSO

msgLib(2) msgDequeue(2) msgQueueCount(2)

AUTHOR

Don Lefebvre

NAME

msgSend - send message to event handler task

SYNOPSIS

```
int  msgSend (MSG_TYPE *msg)

MSG_TYPE  *msg - pointer to message to be sent
```

DESCRIPTION

msgSend is the most basic form of message passing, and the most frequently used. The message pointed to by the function argument contains all of the information needed by msgSend to route and handle the message.

```
struct MSG_TYPE
    {
    TID_TYPE    dest     ;
    TID_TYPE    source   ;
    CMD_TYPE    command  ;
    void       *data     ;
    int         datasize ;
    FLAG_TYPE   flags    ;
    }
```

The contents of a message are defined by the MSG_TYPE struct shown above. The first two struct members are the TIDs of the destination of the message and its source. Next is the message command which describes the function of the message. For instance, when the msgTidQuery function is called it sends a message command of MSG_QUERY_TID to the Tid Server on CPU 0. This command informs the Tid Server that it should look up a TID and reply to the message sender. Users can define their own commands, and are urged to read the message_commands manual pages for a description of how to do so.

The *data and datasize struct members point to additional message data. For instance, continuing the msgTidQuery example, the request to the Tid Server includes the symbolic name of the task whose TID is desired. This information is transmitted by setting the *data pointer equal to the character string containing the symbolic name, and datasize equal to the length of the string. The *data pointer is the size of an integer, therefore integer data may be passed directly in the message by casting *data to integer.

When a message destination is off the local CPU, msgSend performs address translation of the *data pointer. There are circumstances when address translation is not desired, such as when *data is the data itself rather than a pointer.

To accomodate these cases the following constants, defined in msgLib.h, should be used as the datasize argument:

MS_KEEP_ADRS	- prevents address translation
MS_NONE	- *data is ignored
MS_CONVERT_ADRS	- forces address translation

Lastly, the message flags specify options for handling the message. Message flags have five fields:

TYPE	- indicates normal, reply, or broadcast
PRIORITY	- urgent msgs go to front of dest queue
REPLY_WAIT	- if set, task blocks and waits for reply
SEND_WAIT	- optionally waits if dest queue is full
MEMOWNER	- specifies who deallocates message data

Message flags are created by OR'ing together defined constants for each field - either in an assignment statement or by using the flag building routines in msgLib. [See the message_flags manual pages for a list of these defined constants] For convenience, flags for the most common cases have been defined. Most messages can use the predefined standard flag:

```
MF_STANDARD = MF_TYPE_NORMAL     |  MF_PRI_NORMAL     |
              MF_REPLY_WAIT_NO   |  MF_SEND_WAIT_NO   |
              MF_MEMOWNER_RECEIVER
```

As the above description suggests, one of the primary uses of a message is to transmit a pointer to additional data. The CIRSSE messaging system provides a means to manage this "message data". The MEMOWNER field of msg.flags specifies whether the sending task or the receiving task has the responsibility for deallocating message data storage. If MF_MEMOWNER_RECEIVER is chosen, then the storage allocated to message data is AUTOMATICALLY deallocated by the event handler shell following processing of the message. [Note that the msgDataKeep function can be used by the receiving task to prevent automatic deallocation] With MF_MEMOWNER_SENDER the sending task must deallocate message data storage if desired, no automatic deallocation occurs.

For most messages, e.g. those using MF_STANDARD, the sending (source) task allocates LOCAL storage for the message data, sets msg.data to point to this local storage, and then sends the message. The receiving (dest) task processes the message, accessing the message data on the sending task's CPU as desired, and then exits. When the receiving task exits, its event handler shell will deallocate the storage space of the message data (which may entail sending a message to the sender's CPU requesting the deallocation). The deallocation is performed automatically, and so the application developer need not explicitly write code to manage this storage.

RETURNS

If reply flag set to REPLY_WAIT_NO, msgSend returns:

OK - message was successfully sent out.

ERROR - error occurred during message passing; or if SEND_WAIT_NO is set, MsgDispatcher is busy or destination task's queue is full.

If reply flag set to value other than REPLY_WAIT_NO:

msgSend returns *data from reply message (cast as an integer).

SEE ALSO

msgLib(2) msgPost(2) msgBuild(2) message_commands(2) message_flags(2)

AUTHOR

Don Lefebvre

NAME

msgTidQuery - find task id from symbolic task name

SYNOPSIS

```
TID_TYPE  msgTidQuery (TID_TYPE tid, char *taskname)


TID_TYPE tid        - task id of task calling msgTidQuery

char     *taskname  - symbolic name of task whose TID is
                      sought
```

DESCRIPTION

When an event handler task is created it is given a unique task id, called a TID. This TID also identifies the chassis and cpu on which the task is executing, and serves as an address for routing messages to the task. Additionally when the task is created, its symbolic name (specified in the .vxconfig file) and associated TID are saved by the TID Server so that any task on the system can later find the task's TID.

The msgTidQuery function sends a message to the Tid Server on CPU 0 requesting the TID of the task with symbolic name *taskname. While msgTidQuery is waiting for a reply, the task that called msgTidQuery is blocked. As there is a potential delay, msgTidQuery should not be used within a fast synchronous process, except during initialization.

RETURNS

If query is successful, msgTidQuery returns the TID of *taskname.

If query does not succeed, msgTidQuery returns 0.

SEE ALSO

msgLib(2) msgSend(2)

AUTHOR

Don Lefebvre

NAME

```
msgTidGetChassis - get CHASSIS field of task id
msgTidGetCpu     - get CPU field of task id
msgTidGetLocal   - get LOCAL field of task id

msgTidSetChassis - set CHASSIS field of task id
msgTidSetCpu     - set CPU field of task id
msgTidSetLocal   - set LOCAL field of task id
```

SYNOPSIS

```
TID_TYPE  msgTidGetChassis (TID_TYPE tid)

TID_TYPE  msgTidGetCpu     (TID_TYPE tid)

TID_TYPE  msgTidGetLocal   (TID_TYPE tid)

TID_TYPE  msgTidSetChassis (TID_TYPE tid, int number)

TID_TYPE  msgTidSetCpu     (TID_TYPE tid, int number)

TID_TYPE  msgTidSetLocal   (TID_TYPE tid, int number)


TID_TYPE  tid    - task id to be manipulated
int       number - new value of TID field
```

DESCRIPTION

Every event handler task is given a unique task id (TID) when it is created. The TID has three fields:

```
CHASSIS - id of VME or Sun chassis   (4 bits)
CPU     - id of cpu on local chassis (4 bits)
LOCAL   - id of task on local cpu    (8 bits)
```

These functions are used to access the fields of a TID. For instance, msgTidSetCpu will set the CPU field of a TID to a specified value, and msgTidGetCpu will return the value of a field.

Note that these functions are implemented as macros, and that the TID argument is the actual variable not its address. The following are legal statements and are equivalent:

```
msg->dest = msgTidSetCpu (msg->dest, 0) ;

msgTidSetCpu (msg->dest, 0) ;
```

RETURNS

msgTidGet functions return the value of the TID field.

msgTidSet functions return the whole TID after setting the field.

SEE ALSO

msgLib(2) msgSend(2) msgTidQuery(2)

AUTHOR

Don Lefebvre

NAME

msgVarPtrGet - get pointer to saved variables
msgVarPtrSet - set pointer to saved variables

SYNOPSIS

```
void    *msgVarPtrGet ( TID_TYPE t )

STATUS  msgVarPtrSet ( TID_TYPE t, void *p )


TID_TYPE  t - task id of current event handler task
void     *p - pointer to saved variables
```

DESCRIPTION

The msgVarPtr functions assist in building reentrant event handler functions by associating a pointer with the current instantiation of the function.

To define static variables for a reentrant event handler function, follow these steps:

1. Define a structure to hold all static variables

2. During PINIT phase processing, allocate memory for the static variable structure and initialize its members

3. While still in PINIT, save a pointer to this structure with the msgVarPtrSet function

4. Use msgVarPtrGet at the beginning of the event handler function to retrieve the pointer to the saved variables structure, and reference all static variables through this pointer.

RETURNS

msgVarPtrGet returns a pointer to the saved variable structure, or NULL if task id is invalid.

msgVarPtrSet returns OK or ERROR indicating validity of task id.

SEE ALSO

msgLib(2)

AUTHOR

Don Lefebvre

CIRSSE Technical Memorandum

To:	Users/Developers of the MCS and VSS	**Date:**	9 May 1991
From:	Jim Watson	**Number:**	4 vers. 2
Group:	Motion Control Group		
Title:	Using The CIRSSE Testbed Synchronous Service		

1 Purpose:

The CIRSSE Testbed Operating System (CTOS) contains those functions that are common between the Motion Control System (MCS) and the Vision Services System (VSS). CTOS will be composed of several fundamental building blocks to aid in the development of higher-level functions of CTOS and the development of Testbed experiments and applications. The purpose of this memo is to explain the synchronous service component, abbreviated CTOS-SS, that will be available on both the MCS and the VSS. Any process that requires *time*-synchronization will use the CTOS-SS. Thus, the synchronization service will be utilized by other developers of the MCS and the VSS infrastructures and developers of Testbed experiments and applications. This memo provides design and functional interface summaries for the synchronization service. This design has been primarily motivated by application to the MCS, though it is intended to be general enough to be useful in the VSS. Whenever possible, the similarities and differences of the synchronization service between the two systems are noted.

2 Data- vs. Time-Synchronization:

The CTOS-SS is intended to manage the *time*-synchronization of various processes. Process synchronization, however, is not limited to only time-synchronization, but also includes *data*-synchronization. This section clarifies the difference between these two synchronization paradigms.

Consider the following scenario. The PUMA joint angles are to be read and torques written every 5 milliseconds. The reading and writing is accomplished by the PUMA channel driver. The controller is a 6 joint PID algorithm that requires only its current state, current joint data, and the current setpoints, and produces the output torques that are sent to the channel driver. The setpoints are produced by an off-line trajectory generator, and the PID controller maintains its own state via private non-volatile variables. An automatic safety monitoring process runs in parallel, checking the PUMA's actual position every 0.5 seconds.

In the above scenario, the channel driver and safety process would be termed time-synchronous, whereas the PID controller would be data-synchronous (for sake of a simple discussion, the trajectory generator is not considered). The distinction is made based on what makes the process "runable." The channel driver needs to read angles and write torques at known time instances. Likewise, the safety monitor is also "runable" periodically. However, the periodicity of the PID controller is only implied by its coupling with the channel driver—the PID controller is really "runable" when the joint data and setpoint data have become available. One could massage the PID controller into a time-synchronous process, but as soon as the time-synchronous PID controller became "runable," it would have to wait until the joint and setpoint values were guaranteed to be fresh.

There is no doubt that the PID controller needs to be efficient and finish its computations in a timely manner. However, since processes can be momentarily swapped out by the VxWorks scheduler, it is better to consider the PID controller as part of a data-synchronous paradigm rather than a time-synchronous paradigm. In either case, the PID controller and channel driver are in lock-step, but with the PID controller as data-synchronized and the channel driver as time-synchronized, the detection and handling of tardy torque computations can be made in the channel driver.

3 Synchronous Service Design:

This section briefly describes the design of the CTOS-SS, without providing too much detail that would only confuse the issue. Understanding of the design basics is necessary to take full advantage of the service and to use the provided functions effectively. The discussion is primarily centered around the MCS. It is unlikely that the backplanes of the MCS VME Cage and the VSS VME Cage will be directly connected. Therefore, the CTOS-SS, while identical on the two systems, will probably function independently. The issue of how the MCS-SS and VSS-SS will communicate is largely unresolved and spans many aspects of the CTOS, including data exchange, common addressing, shared resources, etc.

The MCS VME Cage consists of five CPU boards (CPUBs), numbered CPUB 0 to CPUB 4. The five CPUBs are functionally identical, with the exception that CPUB 0 serves as the network gateway for the VME Cage. The current MCS architecture assumes that the user will have no processes running on CPUB 0. Among other things, CPUB 0 will serve a special function for the CTOS-SS. This functionality is described below. No other assumptions about the MCS architecture are made in the following discussion.

The MCS-SS consists of two major functional areas: management of the MCS system clock, and management of time-synchronous processes on CPUBs 1 through 4. The system clock is conceptually simpler and is discussed first.

3.1 System Clock

The MCS system clock measures relative time since the beginning of an experiment, i.e., since the time at which the clock was turned on. The clock has two states: on and off. The current time can be accessed via the function `syncClkTimeGet()`, whereby time is measured in integer multiples of 0.1 milliseconds. This atomic unit is referred to as an MCS Time Unit, in short, MCS-TU, and cannot be changed by the user. Choosing this value for the MCS-TU was a trade-off between representing sufficiently long experiments with a single 32-bit clock register and having a fine enough grain for time-synchronization of high frequency processes. The maximum length of an experiment is therefore $0.1\text{ms} * 2^{32}$, approximately 111 hours.

Distinct from the MCS-TU is the actual rate at which the clock register is updated, i.e., the MCS clock update rate (MCS-CUR). Hardware limitations permit update rates between 32 and 5000 Hz, i.e., periods of 0.2 to 31.25 milliseconds. However, additional considerations also influence the choice of the MCS-CUR. For purposes of PUMA control, it is known that the encoders are updated every 0.9 milliseconds. Thus, preferable choices for the clock update period, (MCS-CUP = 1 / MCS-CUR), would be 0.3, 0.9, 1.8, 2.7, etc., milliseconds. Regardless of the actual choice for the MCS-CUR, **syncClkTimeGet()** returns the number of MCS-TUs that have elapsed. The current MCS design does not allow the user to modify the MCS-CUR.

Since connection of the two VME Cage buses is unlikely, the VSS VME Cage will probably maintain its own system clock and associated clock update rate. Thus, the VSS-TU and VSS-CUR would be chosen to be compatible with vision synchronization frequencies (vision applications would probably prefer the clock update frequency to be a multiple of 30 Hz). Issues of coordinating the clock starts on both systems, and how turning off the system clock on one Cage should affect the other system clock are still open. These types of operations will probably best reside at a higher level in the intelligent machine hierarchy.

Most experiments will be run in real-time, that is. the elapsed time reflected in the clock register will agree with the elapsed time on someone's (working) wrist watch. However, given the possibility that the MCS will be driving a Testbed simulator rather than actual hardware, it may be useful to run an experiment slower than real-time. So that the application code would not have to change with the corresponding change from real-time to slow-time, the clock register update rate can be modified by an integer "time scale," MCS-TS. MCS-TS is the number of ticks to be ignored by the clock register plus 1. Thus, if MCS-TS is 1, the clock is updated in real-time—if MCS-TS is 2, then **syncClkTimeGet()** would indicate that τ MCS-TUs have elapsed, whereas your trusty wrist watch would indicate that 2τ MCS-TUs have elapsed. In other words, MCS-TS times the return value of **syncClkTimeGet()** is always the number of MCS-TUs that have elapsed in real-time. While Testbed simulation capability is somewhat in the future, time scaling, which is also useful for debugging, is available today. A function to set MCS-TS is available to the user, (see section 4).

The last issue regarding the system clock is its phase. Simply put, a delay between turning on the clock and registering the first tick may be desirable for the initial synchronization—this delay is referred to as clock phase (MCS-CP). Explicitly, the MCS-CP is the initial value of the tick counting register is the system clock routine. Thus, MCS-CP should be set to one plus the number of ticks to be initially ignored by the clock register, (e.g., MCS-CP equal to 1 results in an immediate update after turning on the clock; and MCS-CP equal to 10 would cause the first clock register update to occur after 10 MCS-CUPs have elapsed). A function to set MCS-CP is available to the user, (see section 4).

To summarize, the MCS system clock is controlled by an on/off flag, its update rate, the clock phase and time scale. The clock phase influences start-up behavior, and the time scale influences steady-state behavior. The clock update rate and clock update period are related to the grain of the clock register. Independent of all of these parameters, system time is always measured in MCS-TUs, which correspond to 0.1 milliseconds.

3.2 Time-Synchronous Processes

Each of the four user CPUBs, (i.e., CPUBs 1 through 4), may have up to five time-synchronous processes that are supported by the MCS-SS. Note, in the following, the word synchronous will be used to mean exclusively time-synchronous.

CPUB 0, in addition to maintaining the MCS system clock, aids the other CPUBs to maintain their synchronous processes. CPUB 0 maintains flags to determine whether the system clock is currently on or off, and whether process synchronization is currently enabled or disabled. Figure 1 depicts the states of these flags and the functions to change their states. Note that process synchronization can be enabled or disabled with the clock on, however, the clock **must** be on to enable process synchronization. If process synchronization is enabled, each clock register update

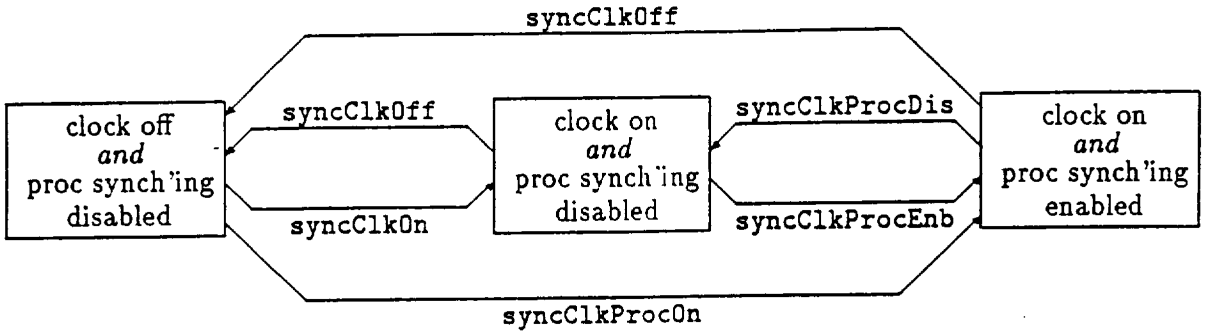

syncClkOff
clock off *and* proc synch'ing disabled
syncClkOff
syncClkOn
clock on *and* proc synch'ing disabled
syncClkProcDis
syncClkProcEnb
clock on *and* proc synch'ing enabled
syncClkProcOn

Figure 1: Clock/Process Synchronization State Diagram

is accompanied by an interrupt sent to CPUBs 1–4. Thus, the synchronous processes are affected by MCS-TS is a way similar to the system clock, again minimizing code changes associated with time-scaling.

Having received this interrupt, each CPUB manages its local synchronous processes independently. Since the functionality of the CTOS-SS as seen from each of the CPUBs is identical, the remaining discussion is given based on a single CPUB.

Assume that a process is currently being synchronized by the local synchronous process handler, LSPH. How this process would go about "attaching" itself to the LSPH is discussed below. Each process handled by the LSPH has two tasks and two semaphores associated with it. The two tasks are the synchronous task and the overrun task, and each is unblocked by one of the two semaphores. (NOTE: pay careful attention to the distinction between synchronous process, synchronous task, and overrun task. The former embodies the latter two.) The LSPH maintains a countdown timer for each process. When this timer expires, the LSPH would normally make the synchronous task "runable" by "giving" the synchronous task semaphore, and reset the timer. The synchronous task could then carry out its function. Under some hopefully rare circumstances, the timer could expire and the LSPH would instead make the overrun task "runable" by "giving" its semaphore. These circumstances include the case that the LSPH detects that the synchronous task has not completed its function from the previous timer expiration and the case that the user wants to force the overrun task to execute. Thus, in normal operation the synchronous task would become "runable" with user specified periodicity, and the overrun task would **never** become "runable."

While the above gave a functional overview of how process synchronization occurs, issues of attaching processes, determining periodicity, detecting overruns, and the design of synchronous and overrun tasks have to be discussed. One important note to make is that the synchronous and overrun functions execute as normal VxWorks tasks, not as interrupt-level functions, and thus can take full advantage of VxWorks and CTOS. Attaching a process is the mechanism by which the LSPH can maintain its data structures and determine what actions to take when various timers expire. At current design, up to five synchronous processes can be attached on each CPUB.

The LSPH needs to know the functions that will serve as the synchronous and overrun tasks, and the semaphores that can unblock these tasks. Additionally, there needs to be a guarantee that the tasks have been successfully spawned, have completed their initialization and are waiting at their blocks, before they are made "runable" the first time. Lastly, the LSPH needs to know the frequency of synchronization and how to detect overruns. All of these issues, with the exception of detecting overruns, are handled straightforwardly by providing the correct information in a function call to the LSPH. In the case that the user does not want to provide an overrun task, the LSPH will associate a system default overrun task to the synchronous process. Two default overrun tasks are available: one task presumes that overruns are **serious**, and thus attempts to shutdown the MCS; another task presumes that overruns are **mild**, and thus generates an error message that can be logged and/or detected by the user. The use of the default overrun tasks are indicate by the parameters SYNC_OVR_SERIOUS and SYNC_OVR_MILD, respectively.

Overrun detection is accomplished by the following paradigm. Immediately after the synchronous task becomes "runable," a flag is set to TRUE, i.e., true that the function is executing. Having completed the function, this flag is set to FALSE immediately prior to the task block. The address of this flag is made available to the LSPH. Thus, before the LSPH unblocks the synchronous task, this flag is checked and if it is TRUE, the overrun task is unblocked. Otherwise, the flag is set to TRUE by the LSPH and the synchronous task is unblocked.

The following pseudo-code fragments illustrate the steps involved in attaching a process and designing tasks that conform to the paradigm described above. The last section of this memo summarizes the actual functions that can be used to perform much of this code.

```
/*pseudo-code to perform synchronous task*/

void mySyncTask( SEM_ID block,  BOOL *pointer_to_running_flag )
{
  <do initialization of synchronous function...that is, code that>
  <will only execute once immediately after this task is spawned>

  loop forever,
    *pointer_to_running_flag = FALSE;
    semTake(block, WAIT_FOREVER);          /*VxWorks indefinite block*/

    <do function that needs to be synchronized>

  end loop;
}



/*pseudo-code to perform overrun task*/

void myOverrunTask( SEM_ID block )
{
  <do initialization of overrun function...that is, code that>
  <will only execute once immediately after this task is spawned>

  semTake(block, WAIT_FOREVER);

  <do function to handle overruns in synchronous function>

  <possible loop back to semTake, if overrun is not major error>
}
```

```
/*pseudo-code to attach synchronous process*/

SEM_ID sem1, sem2;                  /*declarations*/
BOOL running_flag;                  /*declarations*/
int task1, task2;                   /*declarations*/
SYNC_HANDLE sync_proc_handle;       /*declarations*/

< ... >

sem1 = semBCreate(SEM_Q_PRIORITY, SEM_EMPTY); /*VxWorks semaphore creation*/
sem2 = semBCreate(SEM_Q_PRIORITY, SEM_EMPTY); /*VxWorks semaphore creation*/

task1 = <spawn synchronous task with VxWorks taskSpawn including
        arguments of sem1 and running_flag>
task2 = <spawn overrun task with VxWorks taskSpawn including
        arguments of sem2>

syncTaskBlockGuarantee( task1 )     /*guarantee task1 blocking*/
syncTaskBlockGuarantee( task2 )     /*guarantee task2 blocking*/

running_flag = FALSE;               /*set overrun detection flag to false*/


/*Attach synchronous process to the LSPH using task1 as the synchronous
**task, task2 as the overrun task, a phase of 1, and a period of 1000*/

sync_proc_handle = syncProcAttach( sem1, task1, sem2, task2,
          &running_flag, 1, 1000 )

< ... >

syncProcEnb( sync_proc_handle );       /*enable the synchronous process*/

< ... >
```

Finally, the issues of enabling/disabling an individual synchronous process, and determining its period and phase, are discussed. When a synchronous process is attached, it is initially disabled. Thus, its countdown timer is not affected by clock update interrupts, and neither its synchronous or overrun tasks are unblocked. Enabling a synchronous process makes its countdown timer responsive to the clock interrupts. Recall, though, that these interrupts are not generated unless process synchronization has been enabled on CPUB 0. Thus, there is a two-layer hierarchy for enabling and disabling process synchronization. CPUB 0 serves as a master switch, with the ability to disable synchronization system-wide. Only when process synchronization on CPUB 0 has been enabled **and** a synchronous process has been individually enabled with the LSPH, will unblocking of the synchronous task occur.

Two parameters used when attaching a synchronous process are its phase and period. The phase is the start-up delay, i.e., the initial value of the countdown timer. If immediate action is needed after enabling the synchronous process, then the phase should be 1. The period is the reset value of the countdown timer. A subtle use of the phase is to minimize thrashing in the VxWorks scheduler among synchronous tasks that have the same period. For example, say two synchronous processes each have periods of 9 milliseconds. If both processes had equal phases, then every 9 milliseconds, the VxWorks scheduler would have to deal with two tasks unblocking at the same instant, and thus give each task CPU time slices. Switching between tasks consumes CPU time. However, if one process had a phase of 1, and the other had a phase of, say 3, then the first task would have 2 milliseconds of uncontended CPU time. If the first task could finish in this period, then when the second task was unblocked, it would also have uncontended CPU time.

Local disabling of a synchronous process can be achieved in a variety of ways. Disabling is required before the process can be detached from the LSPH. The user can call **syncProcDis()** to change the state of the synchronous process from enabled to disabled, thereby causing the countdown timer to ignore interrupts, and the synchronous and overrun tasks to remain blocked. Re-enabling a synchronous process is allowed. Prior to re-enabling, functions are available to the user to inspect and modify the period and current value of the countdown timer for the synchronous process. Upon re-enabling, the countdown timer is not reset, and thus acts as a re-enabling phase. Since some of the synchronous tasks may have high frequencies, it may be difficult to generate the disable function call exactly when needed. The user is allowed to make a disable pending on a system clock time, whereby the synchronous process will be disabled at the first attempted unblock that occurs at system time greater than or equal to the pending time.

Whenever the overrun task is unblocked, the synchronous process is disabled immediately. Overruns can be generated by the user either immediately or with a pending time. Of course, the overrun task is automatically unblocked if an overrun condition is detected.

To summarize, the local structure of synchronous processes involves a synchronous task, a overrun task, two semaphores for unblocking, a flag to detect overruns, and states of attached/detached, enabled/disabled/disable_pending, and overrun occurred/not_occured/pending. Period and phase parameters control the response of the synchronous task to the clock interrupts.

4 Synchronous Service Functional Interface:

This section describes the CTOS-SS function calls currently available. All of these functions are not directly callable from application codes—*protected functions*, and can only be accessed via messages to the system clock message handler, (also called the P0 message handler for historical reasons). It is important to note when local versus global data structures are being manipulated. As a general rule, the system clock functions work with global data structures, whereas the process synchronization functions work locally.

The first subsection below presents the higher-level functions, which will typically be used by the MCS applications programer. The second subsection presents the lower-level functions, which will be used by experienced MCS applications programmer and the MCS developers. It is important to understand the functionality of the protected functions, although they can only be accesseb by the MCS applications programmer via message passing. These functions appear in the third subsection.

4.1 High-level Functions

`int syncClkStatus( void )`

returns the status of the system clocking using the following bit coding:

- the least significant bit (LSB) indicates whether the system clock is on or off—on is a 1. The bit mask `SYNC_MASK_CLOCK_ON` can be used.
- the next most significant bit indicates whether process synchronization is enabled or disabled—enabled is a 1. The bit mask `SYNC_MASK_PS_ENABLED` can be used.

Note that the return value 10 binary indicates an illegal state.

`int syncClkScaleGet( void )`

returns the value of MCS-TS, the time-scaling factor. MCS-TS is not changeable once the clock has been set. The user is responsible for setting the value of MCS-TS before this function is called. The return value of `-1` indicates that the MCS-TS has not been set yet.

`int syncClkTimeGet( void )`

returns the number of MCS-TUs that have elapsed since the clock was turned on.

`void syncClkOff( void )`

disables process synchronization and turns off the system clock, regardless of their states prior to the call.

`void syncClkProcDis( void )`

disables process synchronization without affecting the current state of the system clock.

`VOID syncTaskBlockGuarantee( int task_id )`

delays until the task with VxWorks task id `task_id` is known to be spawned and blocking on a semaphore.

`STATUS syncFlagSemTake( SEM_ID sem,  BOOL *running_flag )` provides the application code with a compact call to set the running flag to FALSE and then do a VxWorks semTake(sem, WAIT_FOREVER). The return code is the return from the semTake function.

```
SYNC_HANDLE syncProcSpawn(
               SEM_ID *pSync_sem,   VOIDFUNCPTR pSync_func,
               char *pSync_name,    int sync_arg,
               SEM_ID *pOr_sem,     VOIDFUNCPTR pOr_func,
               char *pOr_name,      int or_arg,
               BOOL *flag,  int phase,  int period )
```

provides a higher-level of support than syncProcAttach for attaching synchronous processes. This function will do all of the necessary semaphore creation and task spawning and blocking. The default overrun process is used if pOr_func is NULL, in which case pOr_name is arbitrary and or_arg is used to indicate the severity of overruns for this process (i.e., either SYNC_OVR_SERIOUS or SYNC_OVR_MILD). The synchronous handle or ERROR is returned. The semaphore addresses, i.e., SEM_IDs are written to pSync_sem and pOr_sem (if the default overrun task is not used). The tasks are spawned with MCS default arguments. The user can specify an additional integer argument for each task.

```
STATUS syncProcDetach( SYNC_HANDLE n )
```

deletes the synchronous process in the LSPH data structure. The synchronous handle n is used to refer to the process, and it must be disabled. The associated semaphores and tasks are deleted. The return code indicates the success of the detach.

```
STATUS syncProcRemove( SYNC_HANDLE n )
```

removes the synchronous process in the LSPH data structure, without deleting the associated tasks or semaphores. The synchronous handle n is used to refer to the process, and it must be disabled. The return code indicates the validity of success of the removal.

```
STATUS syncProcEnb( SYNC_HANDLE n )
```

enables synchronous process with handle n. Successful enable requires that the process phase and period have been set to values greater than or equal to 1, and the synchronous and overrun tasks are blocking. The overrun pending and disable pending flags are set to FALSE. The return code indicates the success of the enable.

```
STATUS syncProcOnce( SYNC_HANDLE n )
```

is identical to `syncProcEnb` except the disable pending flag is set to TRUE, thus resulting in only one unblock of the synchronous task.

`STATUS syncOverrunForce( SYNC_HANDLE n )`

unblocks the overrun task associated with process `n`. This function returns immediately, with the return code indicating the validity of `n`. If the process is currently enabled, the overrun pending flag is set to TRUE. Thus, on the next period, the overrun task will be unblocked. If the process is currently disabled, the overrun task is unblocked immediately.

`STATUS syncOverrunNow( SYNC_HANDLE n )`

is identical to `syncOverrunForce` except regardless of the state of the synchronous process, the overrun task is unblocked immediately.

`STATUS syncOverrunPost( SYNC_HANDLE n,  int time )`

requires that synchronous process `n` be enabled, and will unblock the overrun task at the first time the countdown timer expires with the system time greater than or equal to `time`. If the system time is already greater than or equal to `time`, or if the synchronous process is disabled, or if the handle is invalid, then ERROR is returned. Successful posting of the overrun results in the overrun pending flag being set to TRUE. Special case: `time` of 0 causes the overrun pending flag to be set to FALSE and any previously pending overrun to be disregarded—the return code is OK, unless the handle is not valid or the process is not enabled.

`STATUS syncProcDis( SYNC_HANDLE n )`

disables the synchronous task associated with process `n`. That is, no more unblocks of the synchronous task occur, however, if the synchronous task is still executing due to the previous unblock, its execution continues. The return code is OK unless `n` is an invalid process handle.

`STATUS syncProcDisPost( SYNC_HANDLE n,  int time )`

sets the disable pending flag of process n to TRUE. This will allow the synchronous task to finish any current execution and be unblocked until the system time is greater than or equal to `time`, at which time it is disabled. The return code is OK unless n is an invalid process handle, or the process is not enabled, or the system time is already greater than or equal to `time`. Successful posting of the disable results in the disable pending flag being set to TRUE. Special case: `time` of 0 causes the disable pending flag to be set to FALSE and any previously pending disable to be disregarded—the return code is OK, unless the handle is not valid or the process is not enabled.

`int syncProcStatus( SYNC_HANDLE n )`

returns the bit-coded status of process n. The following bits are used:

- the least significant bit (LSB) is 1 if the handle is valid. If this bit is 0, the other bits are meaningless. The bit mask `SYNC_MASK_HNDL_VALID` can be used.
- the next most significant bit is 1 if the synchronous process is enabled. `SYNC_MASK_PROC_ENABLED` can be used.
- the next most significant bit is 1 if the disable pending flag is TRUE. `SYNC_MASK_PROC_DIS_PEND` can be used.
- the next most significant bit is 1 if the overrun pending flag is TRUE. `SYNC_MASK_PROC_OVR_PEND` can be used.

`BOOL syncTableVacancy( void )`

returns TRUE if there is a vacancy in the LSPH entry table.

`SYNC_HANDLE syncSyncTaskIdToHandle( int task_id )`

`SYNC_HANDLE syncOvrTaskIdToHandle( int task_id )`

`SYNC_HANDLE syncSyncSemIdToHandle( SEM_ID sem_id )`

`SYNC_HANDLE syncOvrSemIdToHandle( SEM_ID sem_id )`

return the synchronous process handle given various data that are maintained by the LSPH. A return code of ERROR indicates that the handle could not be identified in the set of currently attached processes (processes with enabled and disabled synchronous tasks are both considered).

4.2 Low-level Functions

```
BOOL syncTaskBlocking( int task_id )
```

returns immediately with TRUE indicating that VxWorks task with id **task_id** is currently in the VxWorks task table and is not ready, i.e., blocking on a semaphore.

```
void default_overrun_task1( SEM_ID blockID,  int localProcNum,
                            int syncVXWTaskId )
```

is one of the function choices for the default overrun task. This function is never called directly by the application code. It is attached by the LSPH as the overrun task in cases when the user does not provide an overrun task for a particular synchronous process and indicates that overruns are *serious*. If syncProcSpawn is used with an user supplied overrun task, the argument list for the overrun task should be the same as above with the addition of an integer argument (this argument is specified by the user in the call to syncProcSpawn).

```
void default_overrun_task2( SEM_ID blockID,  int localProcNum,
                            int syncVXWTaskId )
```

is one of the function choices for the default overrun task. This function is never called directly by the application code. It is attached by the LSPH as the overrun task in cases when the user does not provide an overrun task for a particular synchronous process and indicates that overruns are *mild*. If syncProcSpawn is used with an user supplied overrun task, the argument list for the overrun task should be the same as above with the addition of an integer argument (this argument is specified by the user in the call to syncProcSpawn).

```
SYNC_HANDLE syncProcAttach( SEM_ID sync_sem,  int sync_task_id,
                            SEM_ID or_sem,    int or_task_id,
                            BOOL *flag,  int phase,  int period )
```

is the lowest-level support function for attaching a synchronous process to the LSPH. The return code is the synchronous handle, with ERROR indicating an unsuccessful attach. The synchronous semaphore must have been created and the synchronous task spawned and blocking prior to calling this function. The task spawn arguments were completely up to the discretion of the application code. If the default overrun task is desired for this process, then **or_sem** should be NULL, and **or_task_id** should indicate the severity of overruns for this process (i.e., either SYNC_OVR_SERIOUS or SYNC_OVR_MILD). Otherwise, the overrun task should have been spawned and blocking prior to this call. The synchronous process phase and period are also set by this function.

`int syncProcPeriodGet( SYNC_HANDLE n )`

returns the period for the synchronous process with handle **n**. The return of ERROR (-1) results if **n** is not valid.

`STATUS syncProcPeriodSet( SYNC_HANDLE n,  int per )`

attempts to set the period for synchronous process n to **per**. The return of ERROR results if **n** is not valid, or the synchronous process is enabled, or **per** is not greater than 0.

`int syncProcCounterGet( SYNC_HANDLE n )`

returns the value in the countdown timer for the synchronous process **n**. The return of ERROR (-1) results if **n** is not valid or if the synchronous process is enabled.

`STATUS syncProcCounterSet( SYNC_HANDLE n,  int cnt )`

attempts to set the countdown timer for synchronous process **n** to **cnt**. The return of ERROR results if **n** is not valid, or the synchronous process is enabled, or **cnt** is not greater than 0. Setting the countdown timer prior to re-enabling acts like a re-enabling phase.

`VOIDFUNCPTR syncUsrISRAttach( VOIDFUNCPTR  f )`

attaches the specified function to the LSPH's ISR. The function **f** will be called at interrupt-level with no arguments with the receipt of every location monitor interrupt from CPUB 0. Thus calls to this function are affected by time-scaling. The return value is the pointer to the current function attached. Any function chaining is the responsibility of the caller. The NULL pointer indicates no function.

`VOIDFUNCPTR syncUsrISRClear( void )`

behaves like `syncUsrISRAttach( NULL )`.

4.3 Protected Functions

`STATUS syncPOInit( void )`

puts the system clock and process synchronization data structures in a known state. This function is not available to the application code, and is used internally by the MCS bootstrap. After this call, the values of clock phase and scale are set to illegal values, thus requiring the user to perform explicit initialization. The clock is off and process synchronization is disabled. This function also attaches the clock interrupt service routine (ISR) to the Motorola-135 auxiliary clock chip and enables the location monitor interrupts. The return code indicates the success of attaching to the clock and enabling the interrupts. This is a *protected function.*

`STATUS syncClkPhaseSet( int  n )`

sets the MCS-CP, which is the initial delay (in integer multiples of CTOS-TUs) between the call to **syncClkOn** or **syncClkProcOn** and the first update of the system clock register. A phase of 1 indicates no delay. Setting the clock phase can only be done once per experiment, and must be done before calls to turn on the clock will be successful. Setting the clock phase to 1 can be achieved via **syncClkReset**. This is a *protected function.*

`STATUS syncClkScaleSet( int  n )`

sets the time scale of the system clock, i.e., MCS-TS. Setting the time scale can only be done once per experiment, and must be done before calls to turn on the clock will be successful. Setting the time scale to 1 can be achieved via **syncClkReset**. This is a *protected function.*

`STATUS syncClkReset( void )`

sets MCS-CP to 1, MCS-TS to 1, and the clock register to 0. This can only be done if the clock is off, and the return code indicates if the function was successful. This is a *protected function.*

`STATUS syncClkOn( void )`

turns on the system clock and leaves the process synchronization disabled. The clock must currently be in the off state, and the phase and scale must have values greater than or equal to 1. The return code indicates if the function was successful. This is a *protected function.*

```
STATUS syncClkProcOn( void )
```

enables process synchronization and turns on the system clock. The clock must currently be in the off state, and the phase and scale must have values greater than or equal to 1. The return code indicates if the function was successful. This is a *protected function.*

```
STATUS syncClkProcEnb( void )
```

enables process synchronization. The system clock must currently be on and process synchronization currently disabled. The return code indicates if the function was successful. This is a *protected function.*

```
STATUS syncProcInit( void )
```

puts the LSPH data structures in a known state. This function is not available to the application code and is only called internally by the MCS bootstrap. This function also enables location monitor interrupts for the CPUB, and the return code indicates the success of this operation. This is a *protected function.*

5 Analysis Of System Clock Errors:

This section describes the circumstances under which an error between the system clock and real-time may occur. Assume for the moment that the time scale, i.e., MCS-TS, is set to 1. Then the value in the system clock register *should* reflect real-time as measured in the MCS-TU of 0.1 milliseconds.

The auxiliary clock chip used to maintain the system clock has the following limitations:

- auxiliary clock interrupts can be requested to have integer frequency $f_r \in [32, 5000]$ interrupts per second.
- the crystal in the auxiliary clock chip oscillates at 2,048,000 Hz.
- interrupts can only be generated after an integer number of oscillations, $c \in \mathcal{N}$ have been counted in the chip.

Thus, the auxiliary clock chip really counts oscillations of its internal crystal and after a specified number of these oscillations, an interrupt is generated. The actual period between interrupts is given by

$$T_a = \frac{c}{2048000} \text{ seconds,}$$

where $c = \text{int}(2048000/f_r)$.

Consider the case that we want to generate interrupts every 0.9 milliseconds. Since $1/0.0009 = 1111.11$, we can chose f_r to be 1111 or 1112. Using $f_r = 1111$, results in $c = 1843$, and $T_a = 0.000899902$ seconds. Thus, after the system clock register is updated 1111 times, its value will be

9999 MCS-TUs (equivalent to 0.9999 seconds), but only 0.99979 seconds will have actually elapsed. The relative error is 0.01%, a very small amount.

Now consider that we want to generate interrupts every 4.5 milliseconds. Then after 222 updates of the system clock register, its value would be 9990 MCS-TUs (equivalent to 0.9990 seconds), but really 0.9999756 seconds will have elapsed. This results in a relative error of 0.1%, still a small amount, but an order of magnitude larger than above. Furthermore, the relative error associated with generating interrupts every 0.3 milliseconds is 0.07%. Note that no error is incurred if 2048000 is divided by f_r with no remainder. The amount of relative error is obviously cyclic.

It should be clear that time scaling will not improve or worsen the relative timing errors. For an example, consider that MCS-TS is 2, and we are generating interrupts every 4.5 milliseconds. After 444 interrupts, the system clock register would have been updated 222 times, thereby indicating that 9990 MCS-TUs (equivalent to 0.9990 seconds). In reality, 1.99995 seconds would have elapsed. The relative error is computed based on the 1.99995 seconds that really elapsed and the $2*0.9990 = 1.998$ seconds that should have elapsed. This relative error is again 0.1%.

6 CODING EXAMPLES:

The purpose of this section is to present explicit coding examples, whereby actual function calls are illustrated without the use of pseudo-code. Meaningful examples that are applicable to robot control require the functional interface specifications for the Message Passing Service, Robot State Manager, and the Channel Drivers. These interfaces are not currently available, therefore, the completion of this section is postponed for a future version of this memorandum.

NAME

ipbUnblock - Unblock processes blocked on an IPB

SYNOPSIS

```
void ipbUnblock(IPB_FLAG flag, IPB_STATE mode);

IPB_FLAG flag;       /* IPB to unblock */
IPB_STATE mode;          /* State in which to leave IPB */
```

DESCRIPTION

The ipbUnblock(2) function will release all processes that are currently blocked on the specified IPB. If the mode argument is IPB_CLEARED, subsequent "takes" on that IPB will not block until and ipbSet(2) on that IPB is called. A mode of IPB_RELEASED will cause only those processes currently waiting on the IPB to be realesed, immediately following "takers" will be blocked.

The ipbUnblock(2) function will release all processes that are currently blocked on the specified IPB. If the mode argument is IPB_CLEARED, subsequent "takes" on that IPB will not block until and ipbSet(2) on that IPB is called. A mode of IPB_RELEASED will cause only those processes currently waiting on the IPB to be realesed, immediately following "takers" will be blocked.

INCLUDE FILE

ipbLib.h

SEE ALSO

ipbLib(2), ipbCreate(2), ipbTake(2), ipbSet(2), msgLib(2), mbxAuxLib(2)

AUTHOR

Keith Fieldhouse

NAME
    ipbTake.c - Take an Inter Processor Block

SYNOPSIS
```
void     ipbTake(IPB_FLAG flag);

IPB_FLAG flag;      /* The IPB Flag to take */
```

DESCRIPTION
    The ipbTake function simply attempts to take an IPB. If that IPB is in the BLOCKED state, the calling process is blocked on a local VxWorks binary semaphore. The process will not be made runnable again until some process, somewhere on the VME cage calls ipbUnblock on that IPB.

    If the ipb is note in the BLOCKED state, the process simply continues.

NOTE
    The state of and IPB is never altered by an ipbTake call.

INCLUDE FILE
    ipbLib.h

SEE ALSO
    ipbLib(2), ipbCreate(2), ipbSet(2), ipbUnblock(2), msgLib(2), mbxAuxLib(2)

AUTHOR
    Keith Fieldhouse

NAME

ipbSet - Set an Inter Processor Block

SYNOPSIS

```
void     ipbSet(IPB_FLAG flag);

IPB_FLAG flag;      /* The IPB Flag to set */
```

DESCRIPTION

The ipbSet function simply places the specified IPB in the BLOCKED state. Any subsequent ipbTake's on that particular IPB flag will cause the "taking" process to block until the IPB is unblocked.

INCLUDE FILE

ipbLib.h

SEE ALSO

ipbLib(2), ipbCreate(2), ipbTake(2), ipbUnblock(2), msgLib(2), mbxAuxLib(2)

AUTHOR

Keith Fieldhouse

INCLUDE FILE
    ipbLib.h

SEE ALSO
    ipbCreate(2), ipbTake(2), ipbSet(2), ipbUnblock(2), msgLib(2), mbxAuxLib(2)

AUTHOR
    Keith Fieldhouse

NAME

ipbLib.c - Interprocessor Block Library

SYNOPSIS

```
ipbCreate     - Create and interprocessor block
ipbTake       - Take an interprocessor block
ipbSet          - Set an interprocessor block
ipbUnblock    - Unblock all processes which have taken an IPB

IPB_FLAG ipbCreate(IPB_STATE init);
void     ipbTake(IPB_FLAG flag);
void     ipbSet(IPB_FLAG flag);
void     ipbUnblock(IPB_FLAG flag, IPB_STATE mode);
```

DESCRIPTION

The VxWorks, semaphore library, while rich, is limited to intra-processor situations. Often, it is desirable to allow one process to block on a semaphore (ala the VxWorks library) and to be freed by a process on another processor on the same VME cage. The TAS primitives are not sufficient for this task since the require that polling be done on an unavailable semaphore leading to busy waits or unacceptably high latencies. The InterProcessor Block library is designed to address these issues.

The ipbLib provides routines to create Inter Processor Blocks. When a processor "takes" one of these blocks, if it is in the blocking state, the process is blocked on a VxWorks binary semaphore (see semLib(2)) thus giving the scheduler the opportunity to movethe process to the BLOCKED state. When a processor unblocks an IPB, all of the processes affilated with that IPB are unblocked on all of the processes on the VME cage. The user is given the option of leaving the IPB in the blocked state, eliminating a window of non blocking on that IPB or the actually clear the semaphore until it is reset.

NOTE

In order to achieve it's function, ibpLib connects an ISR to the mbxAux interrupt on each processor on which it is initialized. When an IPB is unblocked, the mbxAux interrupt is generated for each CPU. The ISR checks a status table that it maintains in shared memory (ipbLib_comm) and unblocks the appropriate VxWorks semaphore on its processor.

IPBs are constant through out the cage and can be transmitted by whatever means are convenient. The message passign abilities of msgLib are particularly well suited to this.