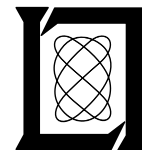
Project Report
ATC-267

A 9PAC System and Application Programmer's Guide

O. J. Newell
G. R. Elkin
W. S. Heath

16 February 1999

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS



Prepared for the Federal Aviation Administration,
Washington, D.C. 20591

This document is available to the public through
the National Technical Information Service,
Springfield, VA 22161

This document is disseminated under the sponsorship of the Department of Transportation in the interest of information exchange. The United States Government assumes no liability for its contents or use thereof.

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
ATC-267		

4. Title and Subtitle	5. Report Date
A 9PAC System and Application Programmer's Guide	16 February 1999
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
O.J. Newell, G.R. Elkin, W.S. Heath	ATC-267
9. Performing Organization Name and Address	10. Work Unit No. (TRAIS)
MIT Lincoln Laboratory 244 Wood Street Lexington, MA 02420-9108	11. Contract or Grant No. DTFA01-93-Z-02012
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
Department of Transportation Federal Aviation Administration Systems Research and Development Service Washington, DC 20591	ATC / February 1999
	14. Sponsoring Agency Code AND-410

15. Supplementary Notes

This report is based on studies performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology, under Air Force Contract F19628-95-C-0002.

16. Abstract

The ASR-9 Processor Augmentation card (9PAC) is a custom processing card that provides the ASR-9 system with increased beacon and radar processing performance. This paper describes the system and application software that executes on the prototype board, with an emphasis on the interaction between software modules.

The application software on the 9PAC determines the position of radar and beacon target reports, replacing software that previously ran on the ASR-9 Array Signal Processor (ASP). The software is organized as a set of cooperating tasks executing under the control of a real-time operating system, PAC/OS, which provides all the services typical of an embedded kernel such as interrupt handling, pre-emptive multitasking, queues, signals, semaphores, mailboxes, and memory management. The deployment of 9PAC will occur in two phases. The Phase I application replaces only the beacon target detector (BTD) and radar/beacon target merge (MRG) functions of the ASP. The Phase I application consists of two executable programs since Phase I uses only two of the C44 processors on the 9PAC. One program, the housekeeping processor, is responsible for all I/O functions and performs the radar/beacon merge operation. The second program, the beacon processor, is dedicated to processing the raw beacon replies and generating beacon targets which are then returned to the first processor for the merge operation. The Phase II application consists of three executable programs, one for each of the C44 processors on the 9PAC and performs much of the Phase I functionality and adds primary radar processing.

The intent of this paper is to provide the 9PAC software support personnel with sufficient information to implement future enhancements without unintentionally compromising some aspect of the overall system.

17. Key Words	18. Distribution Statement
	This document is available to the public through the National Technical Information Service, Springfield, VA 22161.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	88	

FORM DOT F 1700.7 (8-72) Reproduction of completed page authorized

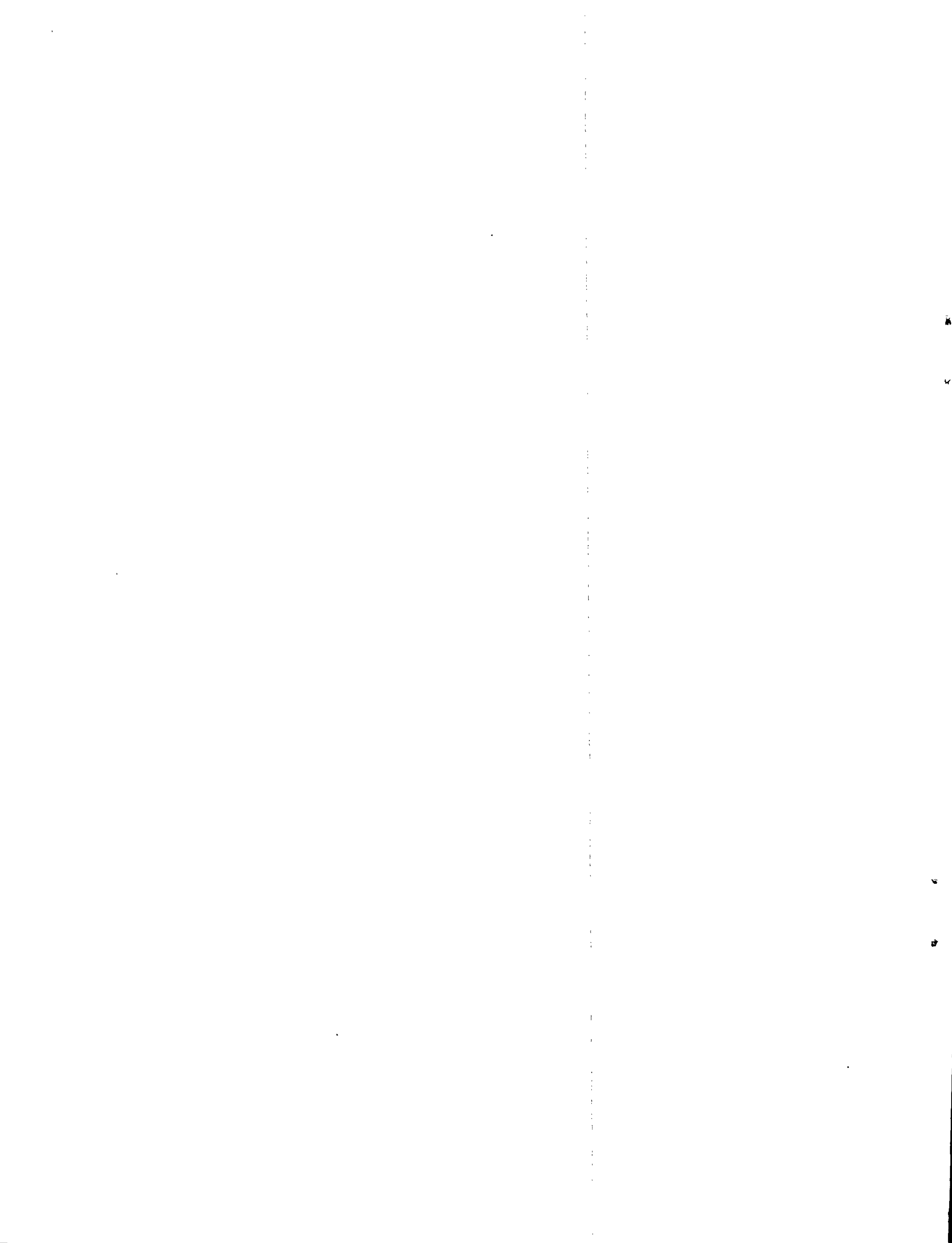


TABLE OF CONTENTS

Section		Page
1.	Introduction	1
	1.1 Hardware Description	1
	1.2 System Software	2
	1.3 Application Software Architecture	2
	1.4 Document Organization	3
2.	PAC/OS Kernel Routines	5
	2.1 Multi-tasking	5
	2.2 Signals	6
	2.3 Semaphores	7
	2.4 Queues	7
	2.5 Memory Management	9
	2.6 Mailboxes	9
	2.7 Interrupt Handling	10
	2.8 C44 Register Access Routines	10
	2.9 Trace Buffer Support	11
	2.10 Linked Lists	11
	2.11 Stack Routines	12
3.	Intertask Communications (ITC) Facilities	13
	3.1 Networking Layer	13
	3.2 Multiplexing	14
	3.3 Network Frames	14
	3.4 Server-Client Protocol Layer	15
	3.5 Server-Client Stream Configuration	15
	3.6 Server-Client Messaging Transport Model	15
	3.7 Server-Client Data Transport Model	17
4.	Serial I/O	19
	4.1 High-Level Architecture	19
5.	Flash File System	21
	5.1 High-Level Architecture	21
	5.2 Low-Level Flash Filesystem Architecture	22
	5.2.1 Special Features/Limitations	23
	5.2.2 Derivative File Systems	23
	5.2.3 File System Design Strategy	24
	5.2.4 File System Component Relationships	27
	5.2.5 File System Size Specifications	31
	5.2.6 File System Function Descriptions	31
6.	Built-In Test	33
7.	Software Application Architecture	35
	7.1 The Phase I Application	35
	7.1.1 The Housekeeping Processor	36
	7.1.2 The Beacon Processor	38

TABLE OF CONTENTS
(Continued)

Section	Page
7.2 The Phase II Application	40
7.2.1 The Housekeeping Processor	40
7.2.2 The Radar Processor	43
7.2.3 The Beacon Processor	44
APPENDIX A: PAC/OS FUNCTION REFERENCE	45
A.1 Flash Filesystem Functions	45
A.2 Mailbox Functions	51
A.3 Memory Functions	53
A.4 Queue Functions	54
A.5 List Functions	56
A.6 Stack Functions	63
A.7 Register Access Functions	66
A.8 Semaphore Functions	67
A.9 Signal Functions	68
A.10 Task Management Functions	69
APPENDIX B: 9PAC Memory Map	73
GLOSSARY	75
REFERENCES	77

LIST OF ILLUSTRATIONS

Figure		Page
1.	PAC Hardware Block Diagram.	1
2.	PAC/OS Components.	2
3.	Phase I Application-Level Software Module Layout.	3
4.	9PAC Network Physical Link Layout.	13
5.	Server-Client Message Transport Model.	16
6.	Server-Client Data Transport Model.	17
7.	Serial I/O Facilities.	19
8.	Distributed Flash Filesystem Architecture.	21
9.	Flash Card Files Structure.	27
10.	Flash Memory Structure.	28
11.	RAM Disk Files Structure.	29
12.	BIT Tasks and Communication Paths.	33
13.	Phase I Application Software Block Diagram.	36
14.	Phase I Data Extraction Block Diagram.	39
15.	Phase II Application Software IBI Mode Block Diagram.	41
16.	Phase II Application Software Monopulse Mode Block Diagram.	42
17.	Phase II Data Extraction Block Diagram.	44

LIST OF TABLES

Table		Page
1.	Housekeeping Processor Tasks	37
2.	Beacon Processor Tasks	40

1. INTRODUCTION

The ASR-9 Processor Augmentation Card (9PAC) is a custom processing card that, when inserted in an ASR-9 system, provides for increased beacon and radar processing performance. This document describes the system and application software that executes on the prototype board, with an emphasis on how the software modules interact. The intent is to provide the 9PAC software support personnel with sufficient information to implement future enhancements without unintentionally compromising some aspect of the overall system.

1.1 Hardware Description

A block diagram of the 9PAC hardware is shown in Figure 1. The board is built around three TMS 320C44 processors, each configured with 1 MB zero wait-state static RAM and either 16 MB or 32 MB single wait-state dynamic RAM. The static RAM (SRAM) is connected to the local bus on each processor, while the dynamic RAM (DRAM) resides on the global bus. The local bus on one of the C44 processors is also connected to a set of peripherals, including a boot PROM, an ASR-9/9PAC multi-port memory, two serial controller chips, and a 20 MB flash card. Each of the three processors is connected to the other two processors via the C44 hi-speed communications ports. Note that although the C44 communications port hardware provides support

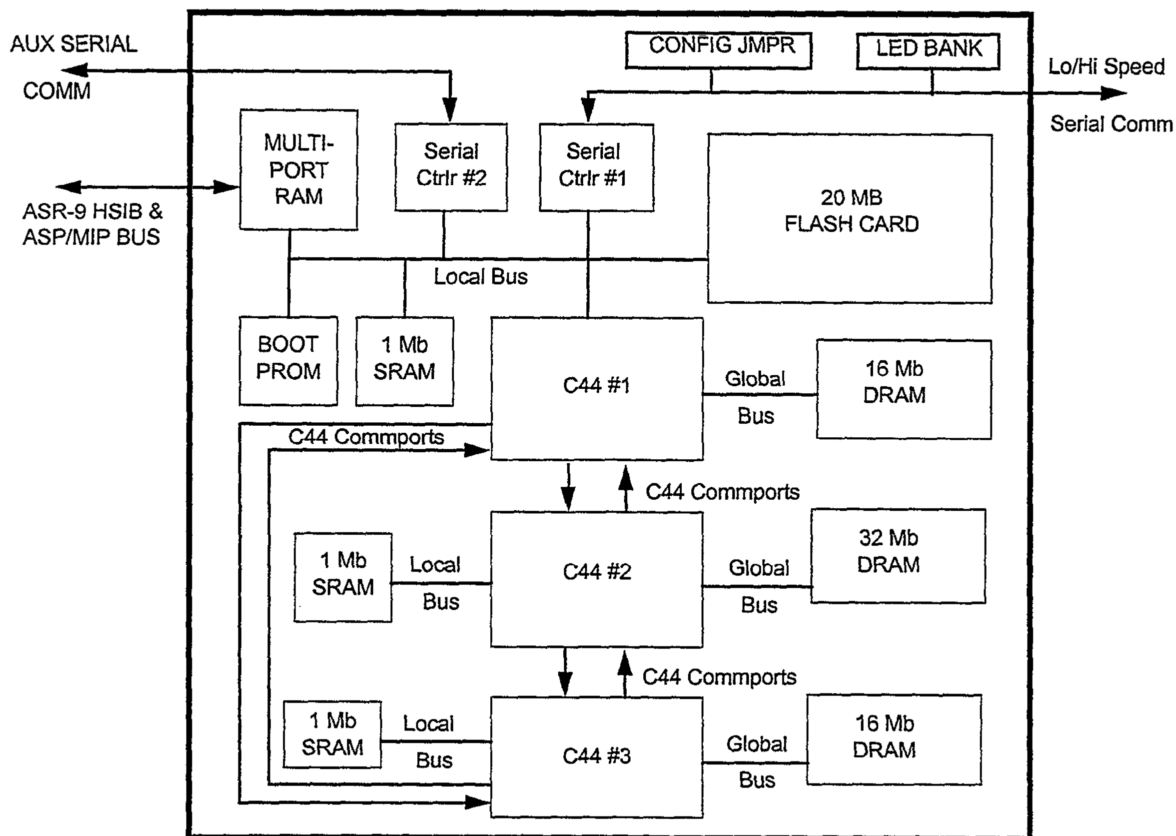


Figure 1. PAC Hardware Block Diagram.

for bi-directional links, the 9PAC uses two unidirectional (one read/one write) ports to avoid problems encountered with early C40 silicon. Note that the component location in the diagram does not correspond to the physical location on the actual 9PAC board.

1.2 System Software

The software running on the ASR-9 Processor Augmentation Card (9PAC) is organized as a set of cooperating tasks, executing under the control of a real-time operating system. The operating system, PAC/OS, provides all the services typical of an embedded kernel such as interrupt handling, pre-emptive multitasking, queues, signals, semaphores, mailboxes, and memory management. In addition, higher-level services exist to support C44 interprocessor communication, high-speed serial I/O, and flash card file access.

The PAC/OS components are shown in Figure 2. At the lowest level are the multitasking kernel and the device drivers for the various 9PAC I/O devices. At an intermediate level, a set of I/O manager tasks exist to provide shared access to the commport, serial, and flash card devices. At the top level exist the 9PAC system tasks such as CPU monitoring and memory testing and the application tasks such as radar and beacon processing.

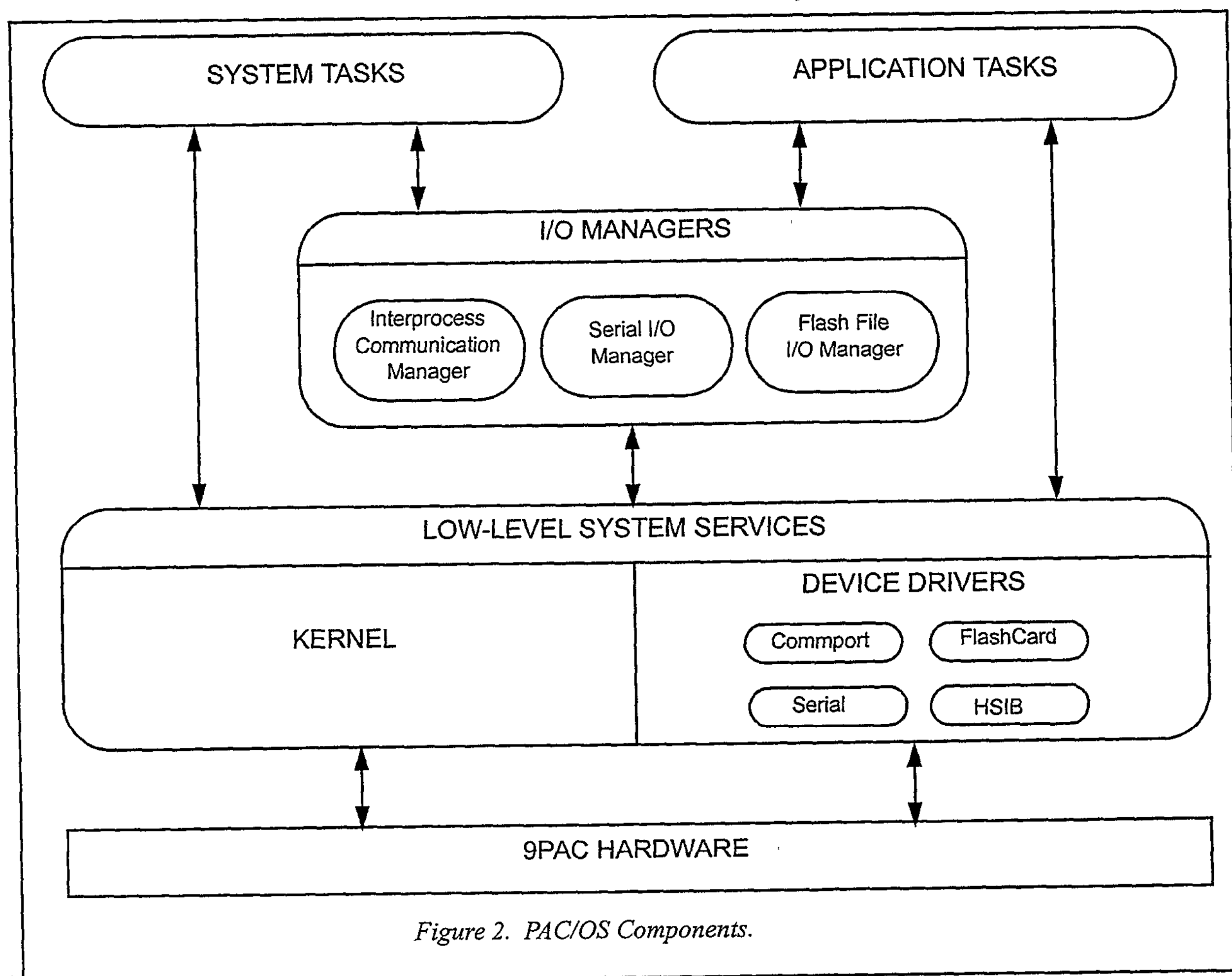


Figure 2. PAC/OS Components.

1.3 Application Software Architecture

A high-level diagram showing the major functional blocks for the Phase I 9PAC application software is shown in Figure 3. As can be seen by the diagram, the software makes heavy use of the lower-level system services discussed above. In the Phase I case, only two of the processors are used. One processor is responsible for all I/O functions, and in addition performs the radar/

beacon merge operation. The second processor is dedicated to processing the raw beacon replies and generating beacon targets which are then returned to the first processor for the merge operation. Note that the beacon processing is split into two tasks, a primary, high priority task that actually generates the beacon targets, and a second, lower priority task responsible for periodically updating the dynamic reflector database and writing it out to the 9PAC flash memory card. Additional details concerning the Phase I and Phase II applications are provided in Section 7.

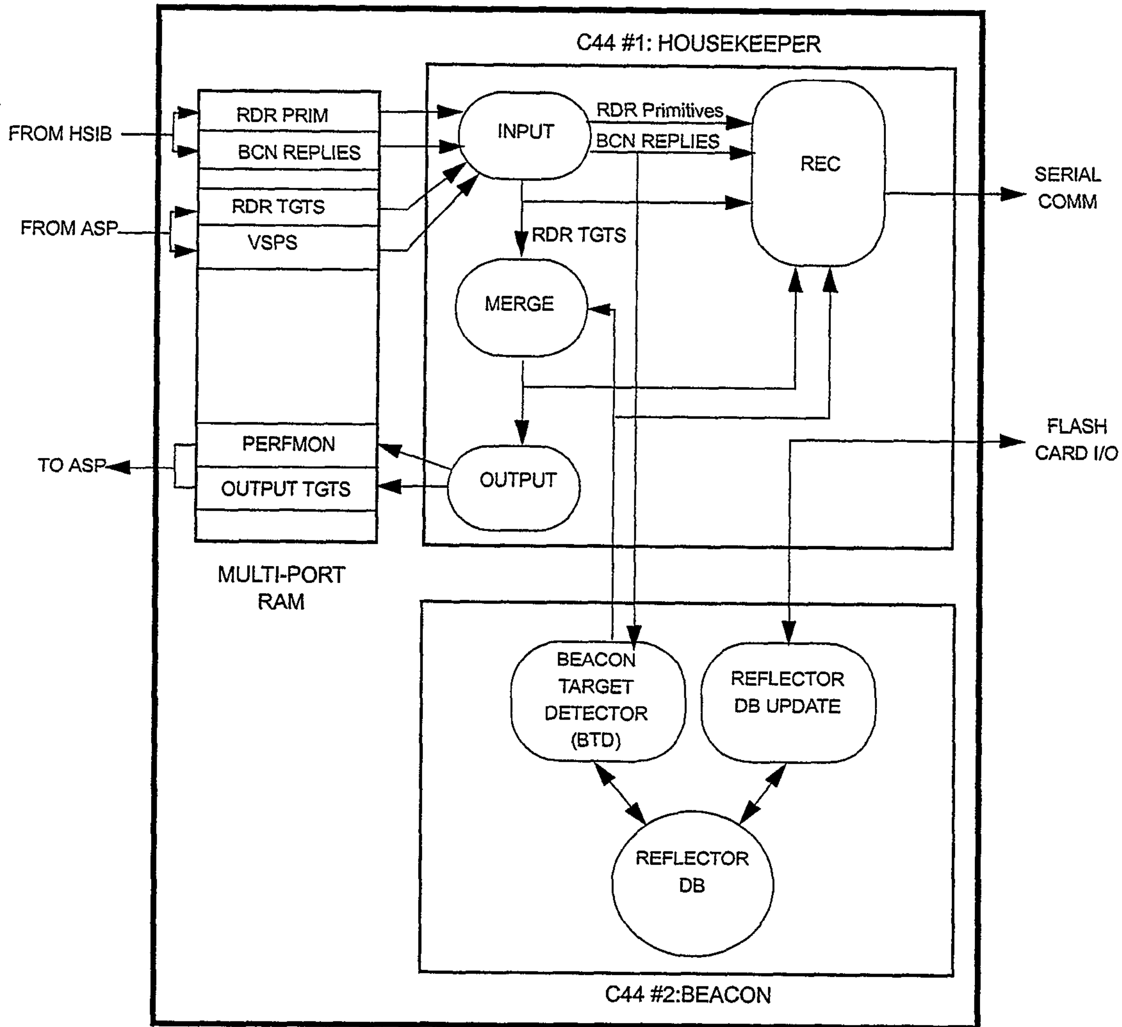
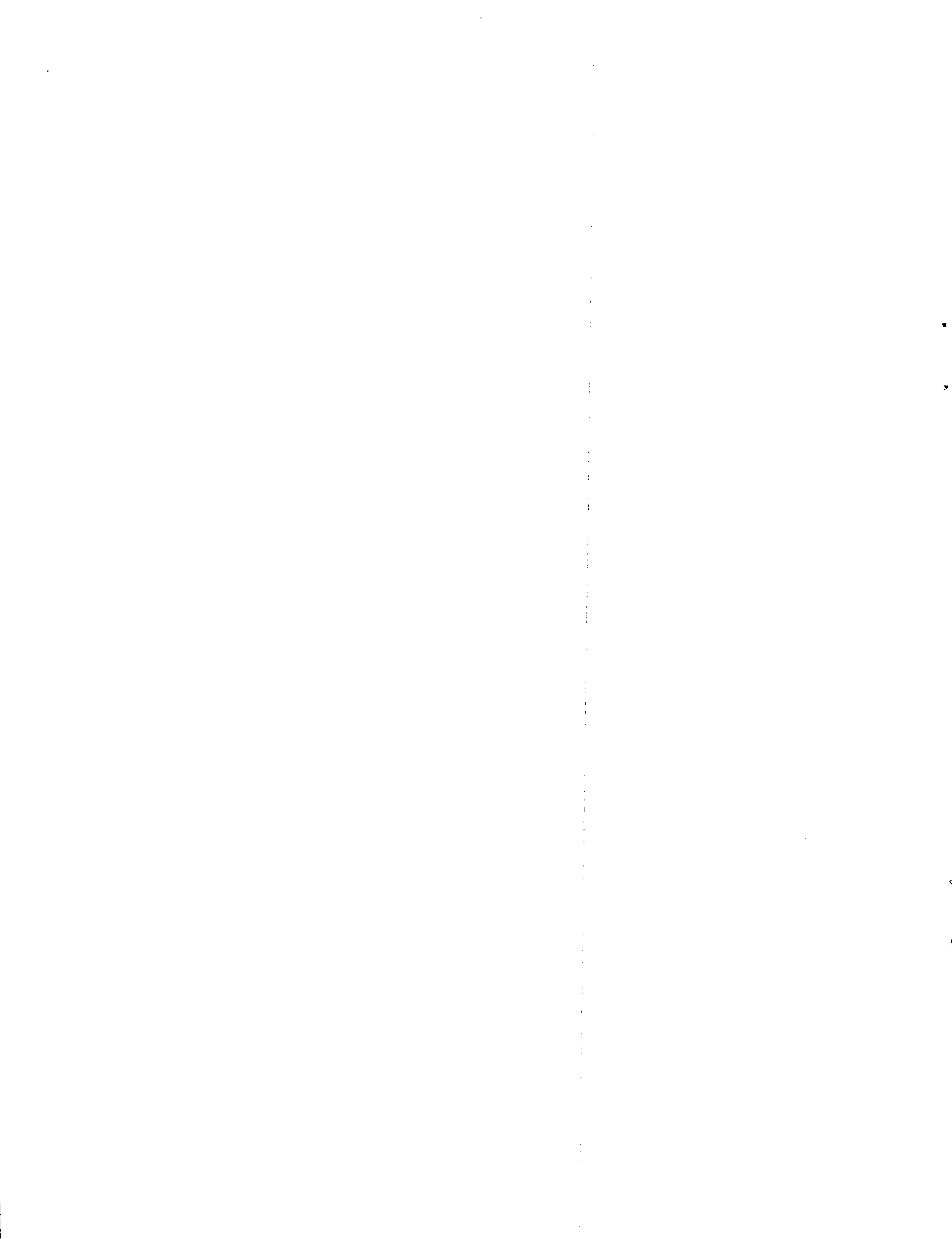


Figure 3. Phase I Application-Level Software Module Layout.

1.4 Document Organization

The remainder of this document is split into six sections and two appendices. Sections 2 and 3 describe the PAC/OS kernel and interprocess communications facilities, respectively, with short programming examples provided. Section 4 describes the serial input/output capability, Section 5 describes the flash file system, and Section 6 describes the built-in test function. Section 7 provides an overview of the 9PAC application software architecture. Appendix A provides a PAC/OS function reference and appendix B details the 9PAC memory map.



2. PAC/OS KERNEL ROUTINES

2.1 Multitasking

PAC/OS Tasks are always in one of five possible states:

- **TASK_RUNNING**
- **TASK_READY**
- **TASK_SLEEPING**
- **TASK_BLOCKED**, or
- **TASK_BLOCKED_WITH_TIMEOUT**

Tasks are scheduled to run based on their priority level, and the running task is always the highest priority task that has reached the TASK_READY state. Each task maintains its own private stack area. Once a task is created and started, it can never be terminated. This is an intentional characteristic of PAC/OS. It disallows the dynamic creation and deletion of tasks while the 9PAC is running, a risky proposition in terms of memory management for a system that must be capable of running indefinitely.

With the exception of the currently running task, all tasks are members of a linked list. Tasks in the **TASK_READY** state are kept in the ready list. Tasks in state **TASK_SLEEPING** or **TASK_BLOCKED_WITH_TIMEOUT** are kept in the sleeping list. Tasks that are simply blocked awaiting a signal with no timeout restriction are kept on a separate list, the blocking list. Maintaining separate lists for sleeping and strictly blocking tasks reduces the number of tasks that need to be examined each time a (1000Hz) timer interrupt occurs, preventing the system performance from degrading unnecessarily as the number of tasks increases.

Semaphore operations, unlike signals, can affect the state of multiple tasks. To deal with this conveniently, tasks that are blocked on a semaphore (state **TASK_BLOCKED** or **TASK_BLOCKED_WITH_TIMEOUT**) are not on any of the previously mentioned lists but instead are maintained on a separate list that is private to each semaphore. A list of active semaphores is also maintained so that a timer interrupt can access and wake up any tasks that are blocked on a semaphore with a timeout restriction. Semaphores and signals are explained in more detail later in this section.

The following example spawns a process that prints "hello" once per second.

```
#include <os.h>
void hello( int );

smain()
{
    Task        *helloTask ;
    TaskAttrs   attrs ;
    int         arg1 = 99 ;

    attrs = TASK_ATTRS ;   /* Copy default attributes to structure. */

    attrs.priority = 5 ;
    helloTask = TaskCreate( hello, &attrs, (void *)arg1, NULL, NULL, NULL );

    /* Lower this tasks priority below helloTask to allow it to run. */
    TaskSetPri( TaskSelf(), 2 );
```

```
    while( FOREVER ) ;  /* Loop forever  (not allowed to return from smain() in PACOS)
}
void hello( int arg )
{
    while( FOREVER )
    {
        SysPrintf("Hello, arg = %d", arg );
        TaskSleep(1000);
    }
}
```

2.2 Signals

Signals are sent from one task to another to indicate that some type of event, such as the arrival of data at a commport, has occurred. Signals are managed internally using a 32-bit mask, resulting in each task having 32 separate signals that it can be "listening" for. Currently, none of the signal bit positions is reserved, and all 32 signals are available for general use. To prevent signal numbers from being inadvertently reused, a dedicated routine, TaskGetAvailSignal(), is provided to allocate signal numbers to the user.

The following example utilizes two tasks, one that waits for a signal and one that sends the signal. The task that is waiting on the signal runs at the higher priority of the two, guaranteeing that it will become the running task each time it receives a signal. Every time the signal is received, a message is printed. Note that the order of the function calls in this example ensures that recvTaskSignum is initialized prior to being used by either the sender or receiver.

```
#include <os.h>

void recvSignal();
Uns recvTaskSignum ;   /* Global signal number. */

smain()
{
    Task *recvTask ;
    TaskAttrs attrs ;

    attrs = TASK_ATTRS ;   /* Copy default attributes to structure. */
    attrs.priority = 5 ;
    recvTask = TaskCreate(  recvSignal, &attrs, NULL, NULL, NULL, NULL );

    TaskSetPri( TaskSelf(), 4 );  /* Lower priority below receiver task. (Causes task switch) */

    /*
     * When control returns to this task, the receiver has already initialized the global
     * recvTaskSignum and is blocked awaiting a signal.
     */
    while( FOREVER )
    {
        SignalSend( recvTask, recvTaskSignum );
        TaskSleep( 1000 );
    }
}
void recvSignal()
{
    recvTaskSignal = SignalCreate( recvTask, NULL );

    while( FOREVER )
    {
        SignalWait( recvTaskSignum, 0 );  /* No timeout specified. */
        SysPrintf("Got signal!\n");
    }
}
```

2.3 Semaphores

PAC/OS semaphores are counting semaphores that within the 9PAC software are typically used for mutual exclusion in one of two instances: when multiple processes are competing for access to a single resource or for task synchronization between two tasks (a data producer and a data consumer) accessing a first in, first out (FIFO) queue. The next example uses a semaphore to provide mutually exclusive access to a single resource–a flash memory card. The following section on FIFO queues illustrates how semaphores can be used to help coordinate access to a FIFO queue.

```
/*
 *  File: sem_mutex.c
 *
 *  Example program to run three tasks  that all access Flash memory.  Since only one task can
 *  access  Flash memory at a given time, the tasks use a semaphore to provide mutual exclusion.
 */

#include <os.h>

void flashWriter( int );
Sem *flashSem ;

smain()
{
    Task *task2, *task3 ;
    TaskAttrs attrs ;
    int sleepPeriod ;

    flashSem = SemCreate( 1, NULL );   /* Create a semaphore with an initial count of 1 */

     attrs = TASK_ATTRS ;   /* Copy default attributes to structure. */       attrs.priority = 5 ;
    sleepPeriod = 1000 ;
    task2 = TaskCreate( flashWriter, &attrs, (void *)sleepPeriod, NULL, NULL, NULL );

    sleepPeriod = 1500 ;
     task3 = TaskCreate( flashWriter, &attrs, (void *)sleepPeriod, NULL, NULL, NULL );

    sleepPeriod = 1300 ;

    while( FOREVER )
    {
        SemPend( flashSem, 0 );    /* Get exclusive access. */
        FlashIO( );                  /* Hypothetical flash access routine. */
        SemPost( flashSem );       /* Give up exclusive access. */
        TaskSleep( sleepPeriod );
    }
}

void flashWriter( int sleepPeriod )
{
    while( FOREVER )
    {
        SemPend( flashSem, 0 );  /* Get exclusive access. */
        FlashIO( );
        SemPost( flashSem );     /* Give up exclusive access. */
        TaskSleep( sleepPeriod );
    }
}
```

2.4 Queues

Queues are most often used either to pass data between tasks or between an interrupt handler routine and a task. Because multiple processes access the queue structure, some form of mutual exclusion is necessary. For performance reasons (queues are heavily used in device drivers), it is not a good idea to use semaphores for this purpose. Instead, each of the queue routines briefly dis-

ables interrupts while it is altering the queue structures. Some form of task synchronization is still necessary, however, to put tasks to sleep when they are waiting on an empty queue and to wake a task when an item is placed on a queue. Both signals and semaphores can be used for this purpose, but signals should really be used only when a queue reader process needs the capability of blocking reads on more than a single queue (the 9PAC server-client package uses this feature). The following example uses a semaphore to coordinate queue accesses between a reader and a writer task. It also is the first example to make use of the memory manager via the MemAlloc() call.

```
#include <os.h>

#define NUM_PACKETS 4

typedef struct {
     QueueElem *elem ;
     char           data[128] ;
} DataBuf ;

void reader();

smain()
{
     int i ;
     Queue *freeQueue, *dataQueue ;
     Sem *freeSem, *dataSem ;
     Task *readerTask ;

     freeQueue = QueueCreate( NULL );
     dataQueue = QueueCreate( NULL );
     for( i = 0 ; i  < NUM_PACKETS ; i++ )
     {
          buf = MemAlloc( PAC_DRAM, sizeof(DataBuf), 0 );
          QueuePut( freeQueue, (QueueElem *)buf );
}
     freeSem = SemCreate( NUM_PACKETS, NULL );
     dataSem = SemCreate( 0, NULL );

     readerTask = TaskCreate( reader, &attrs, NULL, NULL, NULL, NULL );

     /* Send data to reader process. */
     while( FOREVER )
     {
          SemPend( freeSem, 0 );  /* Wait for a free queue packet. */
          buf = (DataBuf *)QueueGet( freeQueue ) ;
          QueuePut( dataQueue, (QueueElem *)buf );
          SemPost( dataSem );
          SysPrintf("Sent Queue packet\n");
     }
}

void reader()
{
     DataBuf *buf ;

     /*
      * Wait for a data packet to appear on the data queue, then just put it back on the free
      * queue.
      */
     while( FOREVER )
     {
          SemPend( dataSem, 0 );
          buf = QueueGet( dataQueue );
          SysPrintf("Got Queue packet\n");
          QueuePut( freeQueue, (QueueElem *)buf );
          SemPost( freeSem );
     }
}
```

2.5 Memory Management

The PAC/OS memory manager is extremely simple. At startup, three memory segments are initialized: one for static RAM (PAC_SRAM), one for dynamic RAM (PAC_DRAM), and one for C44 on-chip RAM (PAC_CRAM). Memory is allocated using the MemAlloc() or MemAllocVerboseErr() function calls. Once allocated, memory can never be freed, so all required memory must be allocated only at startup. This is an intentional characteristic of PAC/OS. It ensures that memory fragmentation will not occur and cause the 9PAC to crash after an indeterminate period. When a dynamic 'pool' of fixed-size memory buffers is required (such as in the case of interprocessor communication frames), the buffers are typically allocated at startup and placed on a free buffer 'stack' from which they can be popped (allocated) or pushed (freed back to the stack). Neither function call returns NULL on failure, but instead, both halt the processor after putting an error message in the trace buffer. MemAllocVerboseErr() allows an added error message to be passed as an argument for added debugging capability. Memory statistics can be obtained at any time via the MemStat() function. The following code fragment illustrates the use of the MemAlloc() functions, along with the trace buffer output for each if an error occurred.

```
void allocBuf()
{
    char *buf1, *buf2 ;
    buf = MemAlloc( PAC_SRAM, 1024, 0 );      /* No special alignment. */
    buf = MemAllocVerboseErr( PAC_DRAM, 1024, 4, "allocBuf" );  /* Align on a 4 word boundry.*/
}
```

If the memory request failed for MemAlloc(), the error message in the trace buffer would be:

```
MemAlloc:  Out of Memory, seg = 0
```

or if the call to MemAllocVerboseErr() failed:

```
allocBuf:  MemAllocVerboseErr: Out of Memory, seg = 1
```

2.6 Mailboxes

Mailboxes are a useful mechanism for providing buffered access to a single resource. In the 9PAC system software, for example, a mailbox is used to allow multiple processes to send messages/data to the 9PAC serial port on C44 processor #1. The buffering capability is important to avoid holding up processes unnecessarily in a real-time environment. The following example starts up one task that reads a mailbox and two tasks that write to it. Note that an additional argument, 'senderArg,' is provided by the mailbox routines to allow each message to be conveniently tagged with information (usually a structure pointer, but the example just uses an integer) specific to each sender. This can be useful in directing mailbox message feedback to the original sender process.

```
#include <os.h>

#define MBX_MSG_SIZE   1024
#define MBX_NUM_MSGS  10

void writer( int );

Mailbox *mbx ;

smain()
{
    TaskAttrs attrs ;
    int senderArg ;

    mbx = MailboxCreate( MBX_MSG_SIZE, MBX_NUM_MSGS, NULL );
```

```
        /* Create 2 tasks that write to the mailbox, then sit and read it. */
        attrs = TASK_ATTRS ;
        attrs.priority = 4 ;
        TaskCreate( writer, &attrs, (void *)32, NULL, NULL, NULL );

        attrs.priority = 5 ;
        TaskCreate( writer, &attrs, (void *)32, NULL, NULL, NULL );

        while( FOREVER )
        {
            MailboxPend( mbx, buf, &length, &senderArg, 0 );   /* No timeout specified. */
            SysPrintf("smain: Got mail msg, len = %d, arg = %d\n", length, senderArg ) ;
        }
    }

    void writer( int msgLen )
    {
        char *msg = MemAlloc( PAC_DRAM, msgLen, 0 );
        int senderArg = 9 ;

        while( FOREVER )
        {
            SysPrintf("writer: Posting msg, len = %d\n", msgLen );
            MailboxPost( mbx, msg, msgLen, (void *)senderArg, 0 );
            TaskSleep( 1000 );
        }
    }
```

2.7 Interrupt Handling

Whenever an interrupt occurs on one of the 9PAC processors, a C interrupt service routine is called. At startup, all interrupts are by default vectored to the routine ISR_unrecognized(), which simply puts a message describing the occurrence in the trace buffer (see Section 2.9) and halts the processor. Before enabling a specific interrupt, the routine ISRAttach() should be called to cause the processor to vector to the specified interrupt handler routine. The following example attaches an interrupt handler routine to the non-maskable interrupt (NMI). If an NMI occurs, a message will be printed to the trace buffer. Note that an optional argument can be passed to the handler routine. This is typically a pointer to a structure, usually containing some state information about the interrupt source device (such as a C44 commport), but here it is just set to a constant value to illustrate its use.

```
#include <os.h>

static int devInfo = 99 ;

void intrHandler( int devInfo )
{
    SysPrintf("NMI interrupt occurred, devInfo = %d\n", devInfo );
}

smain()
{
    ISRAttach( 0x1, intrHandler, devInfo );
    while( FOREVER ) ;
}
```

2.8 C44 Register Access Routines

There are certain C44 registers that are not directly accessible from C code. PAC/OS provides a variety of assembly language routines (SetST, SetIIE, OrST, OrIIE, etc.) to allow the registers to be read and/or modified. See Appendix A for more details.

2.9 Trace Buffer Support

While debugging low-level routines it is often useful to access a trace buffer to get an idea of what is happening. PAC/OS provides a trace buffer facility that saves messages in static memory for later analysis. Normally, a portion of the C44 on-chip static RAM is used for this purpose. Many of the following examples leave evidence of their execution in the trace buffer using the SysPrintf() function call. Calls to SysPrintf() fill the trace buffer with ordinary ASCII text, accessible via a memory dump using a C44 JTAG-based debugger.

2.10 Linked Lists

General-purpose, doubly-linked list routines are provided by PAC/OS. As with queues, any item that is to be added to a list must contain a linked list node header structure (consisting of the next and previous pointers) as its first member. Unlike queues, the linked list code is not intended for use as an interprocess communication mechanism but instead as a lighter-weight mechanism for organizing a list of items (such as aircraft targets) within a single task. The following code fragment declares an array of buffer data structures that contain the node header, creates a list, adds a few of the structures to the list, an then iterates through the list.

```
#include <linklist.h>

typedef struct {
     ListNodeHdr  nodeHdr ;
     int size ;
     char data[256] ;
} Buf ;

Buf bufArray[10];
Buf *bufPtr ;
LinkList *buffList ;

     /*
      * Create the list, and place 10 buffers on it.
      */
     bufList = ListCreate( NULL );

     for( i = 0 ; i < 10 ; i++ )
          ListAppendItem( bufList, &bufArray[i] );

     /*
      * Starting from the beginning of the list, iterate through the entire list.
      */
     bufPtr = ListGetFirstNode( bufList );
     while( bufPtr != NULL )
          bufPtr = ListGetNextNode( bufPtr );
...
```

See Appendix A for a more complete description of all of the list functions.

2.11 Stack Routines

The PAC/OS stack routines should not be confused with the C44 processor stack - they are totally unrelated. Stacks are a generally useful data structure in a number of situations, such as the buffer 'pool' situation described in Appendix A, Section A.3, Memory Functions. The following code illustrates how a buffer pool can be set up and managed using the stack routines:

```
# include <stack.h>
#define DATA_SIZE 10

STACK_T *freeBufStack ;
bufStruct *bufPtr ;
bufStruct inData[DATA_SIZE} ;
.
.
freeBufStack = MemStackCreate( PAC_DRAM,  NUM_POOL_BUFS_, sizeof(BufStruct) ) ;

/*
 * Hypothetical polling loop that reads data from somewhere...
 */
while( poll == TRUE )
{
     /*
      * Grab a buffer, fill it with data, process it, then put it back on the free stack.
      */
     if( incomingData )
     {
          bufPtr = StackPop( freeBufStack );
          fillBuffer( bufPtr, inData, DATA_SIZE);
     }

     processActiveBuffers( ) ;

     /* Put  all buffers that are no longer active back on the free stack. */
     for(  all buffers )
          if( !bufPtr->active )
               StackPush( freeBufStack, bufPtr );

}
.
.
```

This is certainly not the only way to handle a memory pool (queues could easily be used as well), but the stack implementation is extremely simple and is suitable for compiler inlining. Since the position of the bottom of the stack is known, it is also a trivial operation to see how many items are being held on the stack without having to keep track of it with a separate variable.

3. INTERTASK COMMUNICATIONS (ITC) FACILITIES

In any multiprocessing environment such as the 9PAC, it is desirable to make as transparent as possible the fact that the code is spread among multiple processors. In applications that parallelize well, such as large numerical models, this is usually done by running the same program on every processor, with each processor handling a subset of the data to be processed. The 9PAC software, however, does not conform to this model. It more closely approximates a computer network where each node performs an independent set of tasks and communicates with other nodes over network connections which, in the case of 9PAC, are the C44 high-speed communication ports. The PAC/OS intertask communications facilities provide server-client based transport mechanisms that allow node-independent communications to take place; that is, the interface looks the same whether or not the two tasks doing the communicating reside on the same C44 node or on different C44 nodes.

The communications software is comprised of two main layers: the networking layer and the server-client layer. The networking layer utilizes pairs of C44 commports as bidirectional links between C44 nodes and provides for multiple multiplexed channels over a single link. The server-client layer lies above the networking layer and provides network frame fragmentation/defragmentation and connection-based data/message protocols.

3.1 Networking Layer

The 9PAC board contains three TI320C44 DSP chips which are interconnected by the C44 high-speed communication ports. Each C44 processor has four available commports (numbered 0-5), all of which are used to link each processor with its two neighbors. Each commport is capable of bidirectional transfers, but problems with early C40 chips led to a design where each link is used in a unidirectional manner and pairs of commports are used to implement bidirectional transfers. A diagram of the connections is shown in Figure 4.

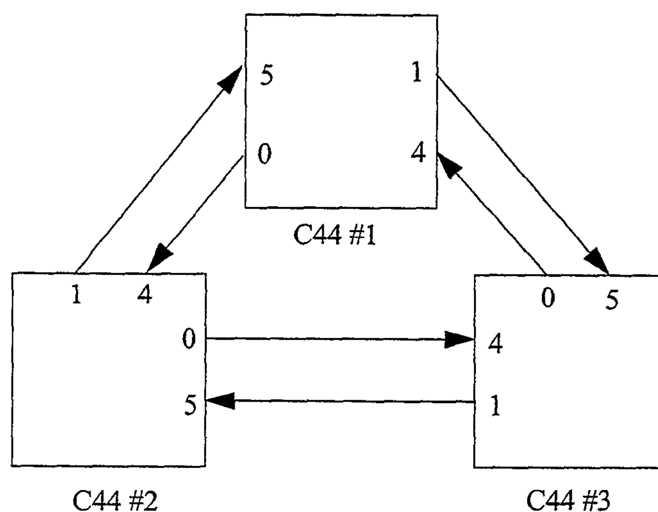


Figure 4. 9PAC Network Physical Link Layout.

Each C44 has an on-chip DMA coprocessor capable of managing commport data transfers independently of the CPU. When a packet is ready to be sent, the CPU need only set up a parameter block for the DMA coprocessor and start it running. When the transfer completes, the DMA coprocessor raises an interrupt and the CPU can then free up any resources (buffers) that were used during the transfer and restart the DMA if another transfer is ready and waiting.

The networking layer is designed to provide intertask communication support between tasks on the same processor as well as on different processors, so in addition to the actual physical interfaces shown above, there is a 'loopback' interface implemented in software. Above the level of the networking driver, the loopback interface appears identical to the physical commport interface, resulting in transparent communication between local or remote tasks.

3.2 Multiplexing

There is only one bidirectional link between each pair of C44 nodes, and it must be shared among multiple tasks needing to use it. Each link is therefore multiplexed into a number of 'channels' (The maximum number of channels per link is currently 20). Each channel operates independently, maintaining its own separate frame queues and status. When a channel is being utilized in a connection-based manner (the line blurs a bit here between the server-client layer and the networking layer), information regarding the status of the channel on the other side of the connection is maintained so that data will not be sent if no more free frames are available on the other side. This is accomplished by requiring each channel that is receiving data to send a 'frames freed' status message to the sender whenever <N> frames have been processed and freed, where <N> is configurable, but it usually set to 1/2 the total number of frames in the receive queue. This status message is not actually seen by the sender process but is intercepted and processed in the network layer. This design allows some channels to be blocked, waiting for the receiver process on the other side to catch up, while at the same time other processes are free to transmit if frames are available on the other side.

In PAC/OS, no channel has strict priority over another channel, but the design is such that channels belonging to tasks with higher priority will generally be serviced first. Additional prioritization at the networking level could be added rather easily at a later time if necessary.

3.3 Network Frames

Data are transmitted across channels using 'frames.' A 10-word header at the beginning of each frame contains addressing and protocol information as well as the number of data words carried by the frame. The data portion of each frame is one of two fixed sizes (currently 256 words/frame for 'small' frames or 4096 words/frame for 'large' frames). The 9PAC software currently uses small frames exclusively. The larger frame is included for future serial interface networking support, where the larger frame will be helpful in keeping interrupt overhead to a minimum. When transmitting a frame between C44 nodes (across a commport link), the entire size of the frame is always transmitted regardless of how 'full' the frame is. This may seem wasteful, but receiving variable size frames using DMA requires two DMA operations: one for the header describing how much data will follow and one for the data transfer itself. It turns out that the interrupt processing overhead for the additional DMA operation is actually *greater* than the overhead of sending partially empty frames, so the latter choice was taken. Note that when frames are being transmitted using the loopback interface, simple pointer manipulation is used and no copy operation actually takes place, and the above situation does not apply.

In general, the software writer will not use the network layer routines but will instead use the server-client routines described below.

3.4 Server-Client Protocol Layer

The server-client protocol layer implements a connection-based protocol using the lower-level networking layer. Basically, when a process initializes a server stream, a stream ID and serving process ID are registered with a connection manager process that runs on each C44 node. When a process wishes to register as a client of the server it opens a channel, and using that channel, sends a connection request (specifying a stream ID) to the connection manager on the appropriate C44 node. Assuming that the specified stream is being served, a channel is opened on the server side, a connection is established, and communication between server and client may then proceed.

3.5 Server-Client Stream Configuration

Each application has its own stream configuration, including stream names, server nodes, number of send and receive frames, and whether or not statistics should be gathered for each stream. This information is centralized in a single include file, typically included in the same file as smain(), that is compiled and linked into the application. An example of the stream configuration file is shown below:

```
/*
 * File: teststreams.h
 *
 *  typedef struct {
 * char name[24] ;
 * int streamId ;
 * int srcNode;
 * int serverSendFrames;
 * int serverRecvFrames;
 * int clientSendFrames ;
 * int clientRecvFrames ;
 * int frameSize ;
 * int debugLevel ;
 * int reportStats ;
 * } STREAM_INFO_T ;
 *
 */
STREAM_INFO_T *SC_streams[] = {
     { "node1_test", 10, 1, 4, 2, 2, 4, IP_SMALL_FRAMES, 0, FALSE },
     { "node2_test", 11, 2, 4, 2, 2, 4, IP_SMALL_FRAMES, 0, FALSE }
};
int SC_nstreams = sizeof(SC_streams)/sizeof(STREAM_INFO_T) ;
```

This example sets up the configuration for two streams: one that will be served by a task on node1 and one that will be served by a task on node2.

3.6 Server-Client Messaging Transport Model

PAC/OS provides support for two server-client models. The first model, the messaging model is useful when a task wishes to listen for messages from other tasks, perform some action, and send replies back to the originating tasks. In the 9PAC software, this is most often used to implement server tasks that provide multiplexed access to a single resource (such as the flash memory card).

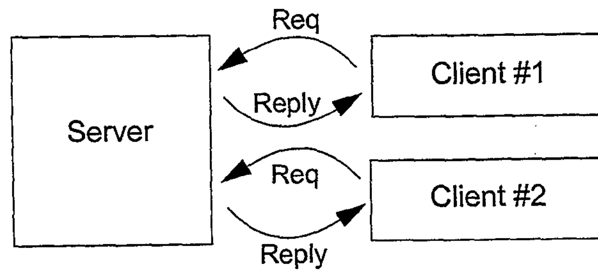


Figure 5. Server-Client Message Transport Model.

For reasons of simplicity, client tasks currently always block while awaiting a reply from a server (messaging model only!). This means there is never more than one outstanding server request per client at a given time. Given sufficient reason, this behavior may be modified in the future.

```
#include <os.h>
#include <ip.h>
#include <sc.h>
#include <teststreams.h>
#include <pac_net_config.h>

void client( void );

smain()
{
    SERVER_T *sp ;
    SC_SERVER_ATTRS_T *attrs ;
    TaskAttrs *attrs ;

    /*
     * Initialize Server-Client layer. This starts up a connection manager process on this node.
     */
    SC_init() ;
    /*
     * Hardcode the node number here. In some situations, it will be automatically set during
     * the boot process, but typically not when the debugger is being used.
     */
    IP_node( 1 );

    /*
     * node_vport_config is a configuration table defined in pac_net_config.h that describes how
     * the processors are interconnected (i.e. which commports on one processor hook to those
     * on another processor.
     */
    IP_init(node_vport_config);

    /*
     * Initialize the server task. This registers with the connection manager so client
     * connection requests can be honored.
     */
    attrs.mode = SC_MSG_MODE ;
    sp = SC_server_init("teststream", &attrs );
    /*
     * Start up client task.
     */
    attrs = TASK_ATTRS ;
    attrs.pri = 6 ;
    TaskCreate( client, &attrs, NULL, NULL, NULL, NULL );
    /*
     * Sit in a loop waiting for requests, and replying to the client that sent the request. The
     * reply in this case is simply an echo of the received message.
```

```
    */
    while( 1 )
    {
        words = SC_server_get_msg( sp, msg, MAX_LEN, &clientId );
        SysPrintf("Server: Got message of %d words\n", words );
        SC_server_reply( sp, msg, 10, clientId );
    }
}

/*
 * Client task to send message and receive reply every 2 seconds.
 */
void client(void)
{
    int i ;
    CLIENT_T *cp ;
    SC_CLIENT_ATTRS_T *attrs ;
    int msg[10] ;

    for( i = 0 ; i < 10 ; i++ )
        msg[i] = i ;

    attrs.mode = SC_MSG_MODE ;
    cp = SC_client_init("teststream", &attrs) ;

    while( 1 )
    {
        SC_client_send_msg( cp, msg, 10 );
        TaskSleep( 2000 );
    }
}
```

3.7 Server-Client Data Transport Model

The second model, the data model, allows a single task to send data to multiple clients using a single data stream. As shown in Figure 6, the communication in this case is unidirectional, with each connected client receiving an exact copy of the data being transmitted by the server. If no clients are connected to a server, the data are simply discarded until a client does connect. The amount of buffering between server and client is user-configurable, as is the blocking mode used. This model is used extensively in the 9PAC software to pass Beacon and Radar information between tasks.

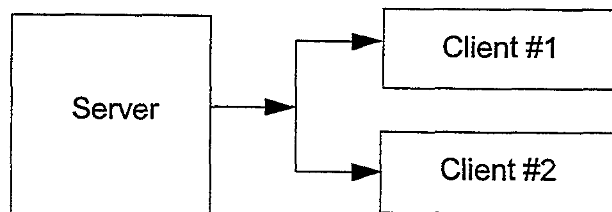


Figure 6. Server-Client Data Transport Model.

The following example starts up a server and a client in data transport mode and sends data from the server to the client.

```
#include <os.h>
#include <ip.h>
#include <sc.h>
#include <teststreams.h>
```

```
#include <pac_net_config.h>

void client( void );

smain()
{
	SERVER_T *sp ;
	TaskAttrs *attrs ;

	SC_init() ;
	IP_node( 1 );
	IP_init(node_vport_config);

	/*
	 * Initialize the server task. Default mode is data mode, so no attributes need be
	 * specified.
	 */
	sp = SC_server_init("teststream", NULL );

	/*
	 *  Start up client task.
	 */
	attrs = TASK_ATTRS ;
	attrs.pri = 6 ;
	TaskCreate( client, &attrs, NULL, NULL, NULL, NULL );

	/*
	 * Send a 1000 words of  data every second.
	 */
	while( 1 )
	{
		words_sent = SC_server_send_data( sp, data, 1024 );
		SysPrintf("Server: Sent message of %d words\n", words_sent );
		TaskSleep(1000);
	}
}
/*
 * Client task to send message and receive reply every 2 seconds.
 */
void client(void)
{
	CLIENT_T *cp ;
	int data[1000] ;
	int words ;
	cp = SC_client_init("teststream", NULL) ;
	while( 1 )
	{
		words = SC_client_get_data( cp, data, 1000 );
		SysPrintf("client: Received %d words of data\n", words );
	}Serial IO
}
```

4. SERIAL I/O

4.1 High-Level Architecture

PAC/OS provides two levels of support for the 9PAC serial ports. At the lowest level, a device driver layer provides an interface between a serial port device and a task executing on C44 #1. At a higher level, a set of cooperating system-level tasks provides true distributed write access to *one* of the serial ports, allowing any task on one of the three processors to send messages to a host computer (normally a Sun workstation outfitted with a high-speed serial card). For convenience, support for the printf() family of functions is also provided as part of the higher layer. The basic architecture of the serial I/O facilities is shown in Figure 7.

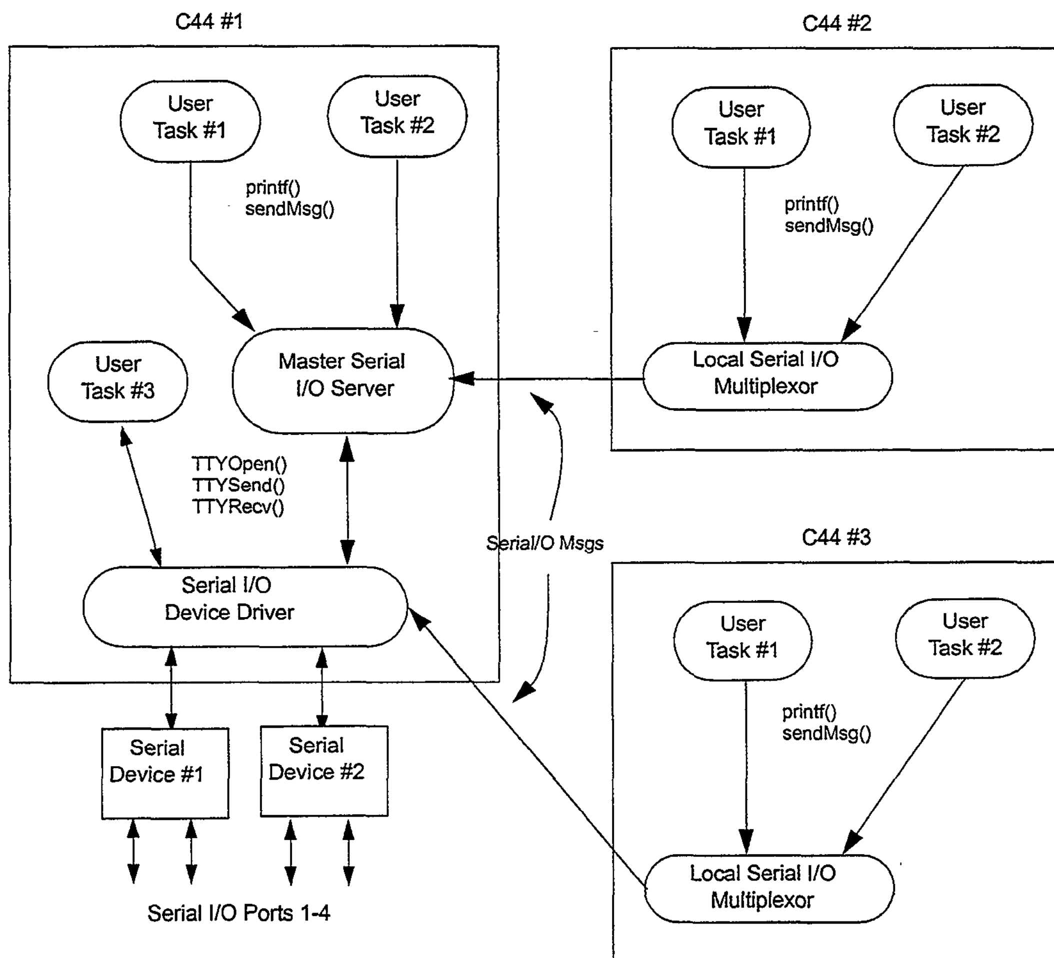


Figure 7. Serial I/O Facilities.

At the application level, tasks make calls to printf(). In the Phase I 9PAC software, the serial I/O facilities are primarily used to output diagnostic messages and radar/beacon target data to a connected Sun workstation using a single serial line. This utilizes the distributed serial I/O mechanism whereby calls to printf() and sendMsg() are made by the various 9PAC application tasks.

On processor #1, these calls are set up to communicate directly with a serial I/O server task which provides shared access to a single serial port. On processors #2 and #3, the serial requests are first transformed into messages and then forwarded to the serial I/O server using the PAC/OS ITC facilities described in Section 3.

Note that in the current implementation, serial communication using these distributed facilities is strictly unidirectional (output). In addition, the entire chain of serial I/O communications is set up to be non-blocking to prevent unanticipated delays in application software due to serial I/O. If the aggregate serial I/O data rate exceeds the capacity of the serial link at any point, the excess data are simply discarded. (Generous buffering provided by the serial I/O multiplexor and server tasks is normally sufficient to prevent this, however).

In future 9PAC applications, it may become necessary to begin using additional serial channels (to support direct output of CD-2 data, for example). In such a case, a second task communicating directly with the serial I/O driver would likely be used. In general, though it is possible to control more than one serial port from a single task, it is advisable for reasons of flexibility to allocate a separate task to each serial connection. If the connection is to be used for a single purpose, it is likely that all the desired functionality and the serial I/O driver calls can be implemented in the same task.

5. FLASH FILE SYSTEM

5.1 High-Level Architecture

PAC/OS provides for true distributed access to the 20 MB flash card. Any task on one of the three processors can access files stored on the card, and accesses from multiple tasks are permitted to occur simultaneously. This is in keeping with the philosophy of the ITC layer–applications tasks which can in general be moved between physical C44 processors with no loss of access to basic system services.

The file access capability is implemented in an analogous manner to a UNIX distributed file system, using the layered architecture shown in Figure 8. At the highest level, application-level tasks make calls to a set of standard functions, including FileOpen(), FileRead(), FileWrite(), and FileClose(). These calls generate file I/O messages which are forwarded to a multiplexor task local to each C44 node. The purpose of the multiplexor process is to limit the number of inter-processor connections related to file system access to two–one for each 'remote' processor. This helps to minimize the total number of network frames that must be allocated for interprocessor communication. After reaching the multiplexor, file I/O messages are transmitted to the master file server on processor #1.

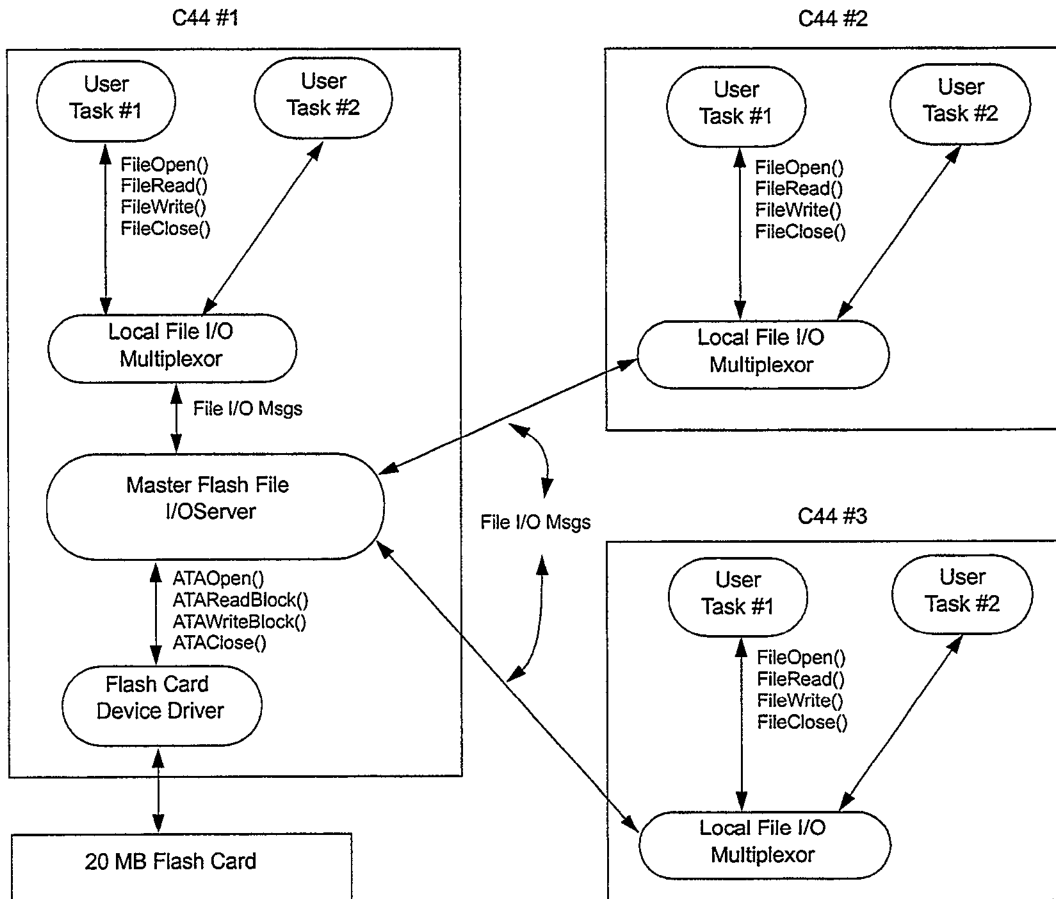


Figure 8. Distributed Flash Filesystem Architecture.

This server is responsible for providing multiplexed access to the single flash card. It services requests one at a time on a first-in, first-out basis (FIFO). Each transaction generally consists of an arriving request, a flash card read or write access, and a return message containing data and any error status information.

The master file server contains the majority of the flash file system internals such as maintaining lists of allocated vs. free blocks on the flash card in a MS-DOS-compatible file allocation table (FAT). It also translates the high-level read/write requests to the low-level block-oriented accesses provided by the flash card device driver (ATAOpen(), ATAReadBlock(), ATAWriteBlock(), etc.). Note that although the file server module provides hooks to support multiple device types (the original 9PAC prototype contained 4MB on-board flash in addition to the 20 MB flash card), only the ATAxxx() driver routines are currently in use.

The modules comprising the distributed flash filesystem are all configured as low-priority tasks to avoid stealing CPU cycles from the real-time radar/beacon processing tasks. That fact, in conjunction with the relatively slow write rate of flash memory, results in file I/O taking a relatively long time (up to 0.1 seconds per 256-byte transfer). Obviously, the real-time radar/beacon processing tasks cannot tolerate such delays. For that reason, the real-time processing functions that contain an element of file I/O (such as the beacon processing code and its associated reflector database file) are typically implemented as two tasks: the high-priority processing task and a lower priority file I/O task. Both tasks access the same in-memory copy of the information saved on the flash card, but only the low priority task does actual reads/writes from/to the flash card.

5.2 Low-Level Flash Filesystem Architecture

The Flash File System (FFS) driver library is compatible with the following technologies:

- DOS file system
- PCMCIA ATA flash card standard
- TMS320C4x DSP processor
- ANSI C language

The library supports two interfaces: 1) a user command interface and 2) a programming interface. Functions that implement the user command interface support the following operations:

format	-- DOS-format the flash card
getFFS	-- copy the FFS from the flash card to on-board RAM
dir	-- display file names and sizes
delete <filename>	-- delete a file
mv <SRCfile> <DESTfile>	-- rename a file
cpTo <FFSfile> <HOSTfile>	-- copy <HOSTfile> to <FFSfile>
cpFrom <FFSfile> <HOSTfile>	-- copy <FFSfile> to <HOSTfile>
csFFS <FFSfile>	-- checksum file <FFSfile>
putFFS	-- copy the on-board FFS to the flash card

For the 9PAC system, these operations may be performed through a serial port interface to a host computer.

The programming interface supports the following operations:

format	-- DOS-format the flash card
getFFS	-- get the FFS from the flash card
open	-- open a file (RDONLY, WRONLY, APPEND)
close	-- close a file
read	-- read file data
write	-- write data to a file
seek	-- move file access point
tell	-- get current file access point
size	-- get current file size
flush	-- write file's sector buffer to the flash card
putFFS	-- copy the FFS to the flash card

These functions are available to the real-time kernel and application code.

The driver implements a standard DOS file system (FAT16). All files contain 32-bit integers with DOS and TMS320C4x compatible byte ordering (little endian).

5.2.1 Special Features/Limitations

The 9PAC project required a fail-safe file system. As a result, additional redundancy (beyond that provided by DOS) and integrity are built in. Redundant copies of the FAT table (part of DOS) and root directory are implemented. In addition, all alterable file system data structures are checksum checked. Provision also was made to ensure that one consistent file system is always present on the card. This assures protection against power failure or card removal while the file system is being updated.

To accommodate redundant root directories, the standard DOS 512 entry root directory was replaced with two 256 entry copies. The resulting file system is still compatible with DOS FAT16 but limited to 256 file entries in the root directory. Since support for this many files was considered more than adequate for the 9PAC application, subdirectory support was not implemented.

5.2.2 Derivative File Systems

Although the requirement was later dropped, the 9PAC project initially had a requirement to support a file system for flash memory mounted directly on the processor board. This memory presented an additional design challenge: the memory must be written in 64 Kilobyte blocks. A copy of the flash card file system was adapted to support this requirement by implementing a 64 Kilobyte software cache between the file system and the memory. As an intermediate step in the conversion process, the file system was first converted to support a RAM disk—that is, a file system for a region of SRAM or DRAM memory. The net result of these efforts is that three separate file systems were created:

- PCMCIA ATA flash card
- On-board flash memory

- On-board RAM disk

The user command and programming interfaces are the same for all file systems.

A RAM disk is useful for situations in which a real-time system must access files at high speed. Files may be transferred between it and a slower flash file system during less time-critical periods.

5.2.3 File System Design Strategy

The file system design strategy was primarily constrained by the desire to implement a file system that was compatible with DOS FAT16 and by the characteristics of the specific storage media supported. For example, the DOS file system implements a sector size of 512 bytes and uses a FAT to specify access to clusters of file sectors. It was also necessary to minimize write operations to flash memory since write operations are relatively slow and flash memory has limited write cycle life expectancy (see Section 5.2.3.3, Wear Leveling).

As a result of these and other constraints, the file system was designed to support a mode of operation in which the file system data structures in a formatted flash memory file system are first copied into RAM before file operations are performed. The in-memory copy of the file system is then modified as file operations take place. The data structures in the actual flash memory file system may then be synchronized with the current in-memory version by periodically copying the in-memory image at convenient times. Note, however, that application file sectors are written to flash memory as they are filled by the application program. Only the file system data structures (e.g., the FAT and root directory structures) are maintained in RAM and written periodically. This approach has advantages and shortcomings.

Clearly, by maintaining and accessing an image of the file system data structures in RAM, file operations are faster. Indeed for a real-time system, this is probably the only practical approach that will satisfy access timing requirements since flash memory writes are relatively slow. An additional advantage is that flash memory sectors that hold file system structures sustain less wear. On the other hand, only the in-memory file system image is current all the time. If the system should loose power, some of the recently written data could be lost. However, the existing flash memory file system would still be consistent and files would still be accessible.

5.2.3.1 Data Structures

The design associates a file descriptor data structure with each file created. Among other data items, this structure contains a 512 byte sector buffer. Sector size was chosen to comply with DOS FAT16 and the PCMCIA ATA specification. The sector buffer is managed like a cache. That is, on a file read operation, the file sector containing the data to be read is copied into the buffer. As long as program reads require data from that sector, data are read from the buffer. When reads are required from a different sector, the new sector is copied into the buffer.

Similar operations take place in a file write operation. When the buffer has been filled, its contents are copied to the appropriate file sector in flash memory. The file descriptor structure maintains a "buffer dirty" flag which is set whenever data are written to the buffer. The flag is used to signal the need to copy the buffer back to the file system before it is overwritten with data from a different sector. This mechanism supports random read/write operations.

The DOS file system maintains a FAT data structure that implements a link list mechanism for locating clusters of file sectors. A cluster is a contiguous block of sectors and is the smallest

unit of storage managed by the file system. The various implementations of DOS use different numbers of sectors per cluster. The number used may be read from the boot sector data structure stored in the file system.

Since sectors are allocated to files in units of clusters, data storage efficiency is impacted by the sectors-per-cluster value chosen. The value also affects the size of the FAT table and, as a result, the time it takes to update the flash memory file system from the in-memory copy. Large values result in smaller FATs and hence faster updates. Once a file system is formatted, the number of sectors per cluster is read from the boot sector data structure when the file system is loaded into the in-memory image.

The **format** function for the flash card file system programming interface has an argument for specifying the number of sectors per cluster. The **format** command in the user command interface for this file system defaults to eight sectors per cluster. But this value may be overridden by specifying a command option. The design is different for the on-board flash memory and RAM disk file systems, however.

As indicated earlier, on-board flash memory must be written in 64 Kilobyte blocks. To support this, an additional level of buffering was implemented between the file descriptor sector buffers and memory. The new buffer also contains 64 Kilobytes and is accessed as a secondary cache called the current cluster cache. That is, the file system accesses file sector buffers in the current cluster cache rather than directly from flash memory. The current cluster cache is read and written to/from flash memory as needed to support file requests. It holds an entire cluster of sectors.

Since the current cluster cache contains 64 Kilobytes and a sector is 512 bytes, the number of sectors per cluster for this file system is fixed at 128. This (rather large) value is a direct result of the requirement that writes to flash memory must be in 64 Kilobyte blocks. The operational impact of this is that files are allocated storage in 64 Kilobyte blocks.

In the case of the RAM disk, the number of sectors per cluster is specified by a symbolic constant, which is currently set to eight. This value has been found to provide a reasonable compromise between memory access speed and usage efficiency. A different value may be chosen by changing the value of the symbolic constant and rebuilding the library.

5.2.3.2 Redundancy and Integrity

As indicated earlier, the file systems were designed to be fail safe. Failure mechanisms include power failure or card removal while the file system is being updated and actual memory chip failure at any time. Two mechanisms were implemented to detect and recover from these failures: 1) checksum calculations and 2) file system redundancy.

To detect errors, a checksum is computed for each file system data structure as it is written to the flash memory. The checksum values are stored in a separate structure within the file system. When the file system is later read back into its RAM memory image, its checksums are again computed and compared to the previously stored values. If the previous attempt to update the file system experienced a failure, the checksums will not agree and the failure will be detected.

It is possible to recover from a file system failure by maintaining redundant copies of file system data structures in the flash memory. If all of the data structures for each copy are updated consecutively, then at least one copy will be consistent at all times. The code is written to attempt to read one copy, and if a checksum error is detected, to read the second copy.

An additional level of redundancy is built into the on-board flash memory file system. The redundant file system is first built in the 64 Kilobyte current cluster cache buffer. It is then stored in the first two 64 Kilobyte blocks of flash memory. This is done to guard against the possibility of a single write operation failure. If errors are detected in the redundant copies read from the first block, the second block is read and accessed. For this file system, a total of four complete sets of file system data structures are maintained in flash memory.

An additional mechanism is implemented to support file system redundancy. The standard DOS file system maintains two copies of the FAT structure but only one copy of the root directory. As indicated earlier, the FFS file system maintains two copies of the root directory in the same space that would ordinarily be occupied by the DOS root directory. The number of files that may be created in the root directory is therefore reduced from 512 to 256 (actually 255—one entry is used for a file system volume label). A checksum is maintained for each copy.

5.2.3.3 Wear Leveling

Flash memory wears out (fails) after a large number of write operations (several hundred thousand, depending on the technology used). The PCMCIA ATA flash card standard provides a means to extend the life of a flash card by changing the mapping of logical sectors to physical sectors. This operation is called "wear leveling." Two commands are provided. The first returns status indicating whether wear leveling is needed; the second performs the remapping operation.

Since a wear level check can take up to 12 seconds to complete and the actual leveling operation can take up to 30 seconds, these operations should usually be performed by a low level, preemptable task and only occasionally. The FFS driver provides functions to perform the two operations and includes calls to them. The check function is called each time the in-memory copy of the file system is written to the card (by the putFFS operation). If this call indicates that wear leveling is needed, the second function is called to perform the actual operation.

It should be noted that the maximum execution times stated for these operations are rarely seen in practice since they assume a fully loaded flash card. Also, since the card actually performs the operations (it contains its own processor), the driver functions spend most of their time polling for a completion status. This time may be more productively used by higher priority application tasks that preempt the polling task. Finally, since the write operation cycle limit is very large, the wear leveling operation will occur only very rarely—perhaps never in some designs.

5.2.3.4 Compatibility

When the flash card file system software was completed, it was possible to demonstrate compatibility with the drivers supplied with laptop/notebook computers manufactured by NEC, HP and Epson. That is, flash cards formatted by these machines could be accessed by the FFS and flash cards formatted by the FFS could be accessed by these computers. The only qualification to this was that the flash card first had to be formatted by the laptop/notebook since their DOS drivers put code on the card that is run when a card is inserted. The FFS does not alter this code when it formats a card.

Although the file systems were compatible at one point in time, it was not practical to keep them compatible as new versions of manufacturer drivers were released. Therefore, no assurance may be given that the FFS is compatible with a given manufacturer's equipment. However, this does not negate the usefulness of the FFS in real-time systems. The file system may still be

accessed through the host command interface via a serial port and through the programming interface. If required, the file system could be brought back into compliance with current PC drivers.

The laptop/notebook and FFS drivers referred to all implement standard DOS FAT16 file systems. The primary difference is the sectors-per-cluster value used. The NEC driver used two sectors per cluster while the Hewlett-Packard (HP) and Epson drivers implemented four. By contrast, the FFS formats use eight by default, but may format or access a file system with any reasonable value. These differences reflect different anticipated user requirements. For laptops and notebooks, support for a large number of small files is desirable and very fast access time is not too important. However, in a real-time system fewer files of larger size are common and access time is usually critical. As indicated earlier, a larger sector-per-cluster value results in a larger file block granularity and shorter file access time.

5.2.4 File System Component Relationships

Figures 9, 10, and 11 show the relationships between library files, application programs and memory interface hardware for the three types of file systems supported. At the highest ***Application Interface*** level, the user programming view is the same for all file system types. Application programs typically include the real-time program and an EPROM monitor program that accesses a serial port in support of a host command interface.

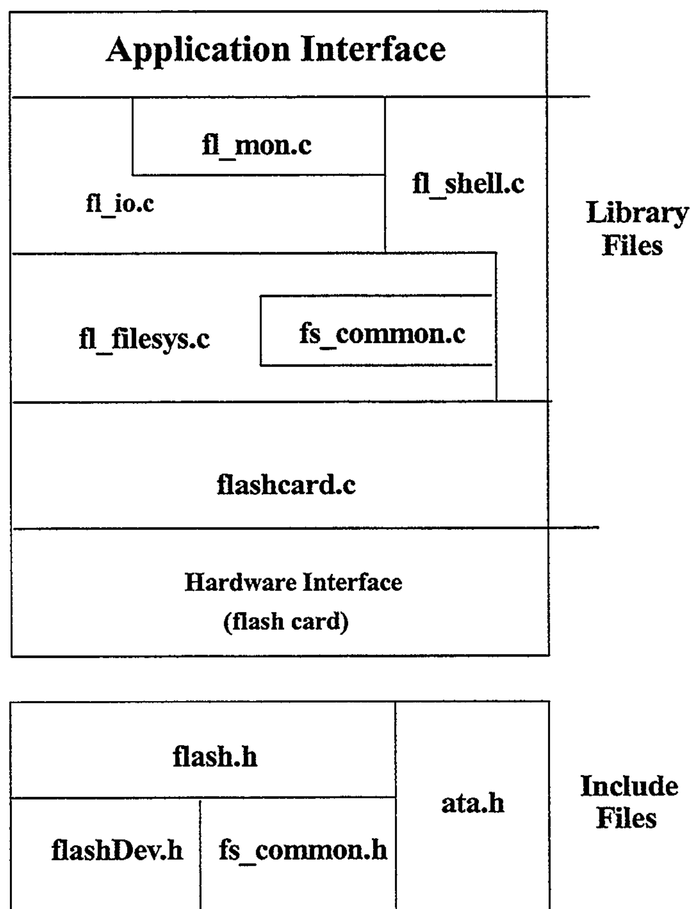


Figure 9. Flash Card Files Structure.

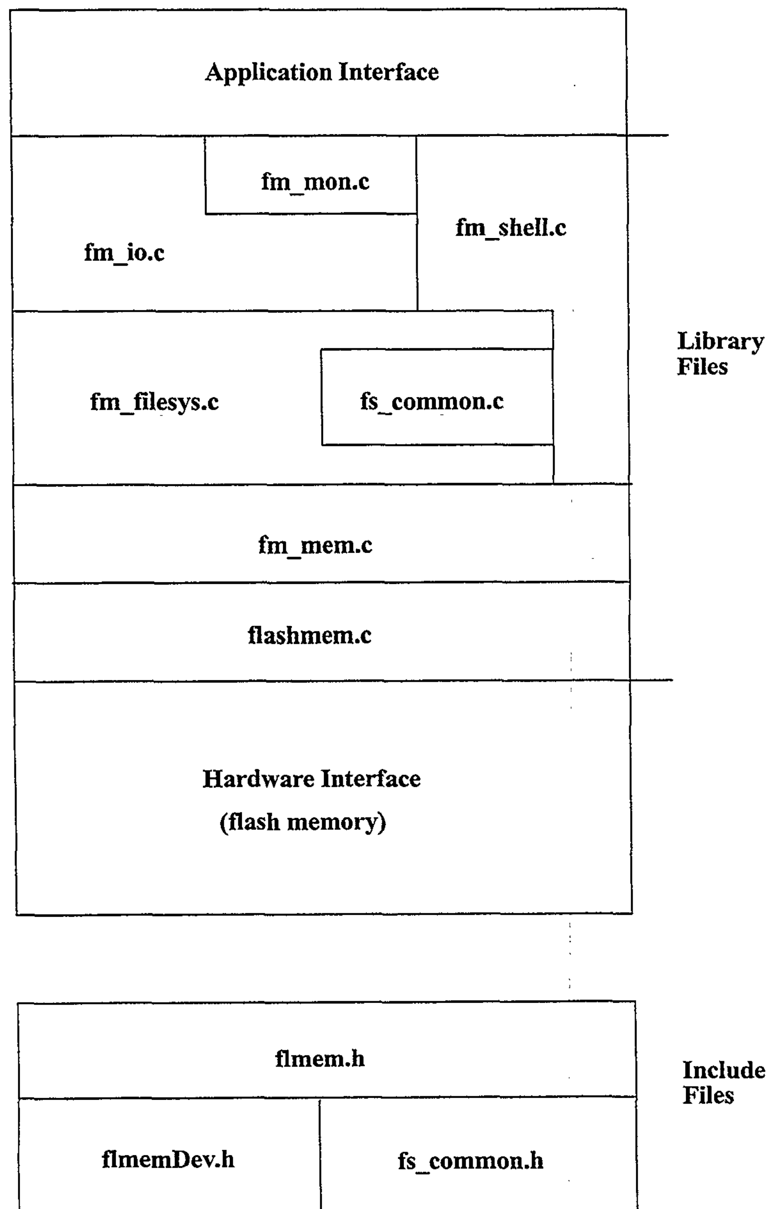


Figure 10. Flash Memory Structure.

At the lowest ***Hardware Interface*** level, the files are custom tailored to specific hardware requirements. For the flash card and flash memory file systems, the hardware interfaces are sufficiently complex to warrant separate interfaces contained in files *flashcard.c and flashmem.c*, respectively. In addition, the flash memory file system requires a software cache mechanism (the current cluster cache) that is implemented in file *fm_mem.c*. By contrast, the RAM disk file system memory interface is straightforward and is contained in file *rd_mem.c*.

Certain operations are common to all three file systems and are included in common file *fs_common.c*. These consist of functions to pack and unpack DOS file system data structures and a checksum function. Since these functions are shared, they must be reentrant.

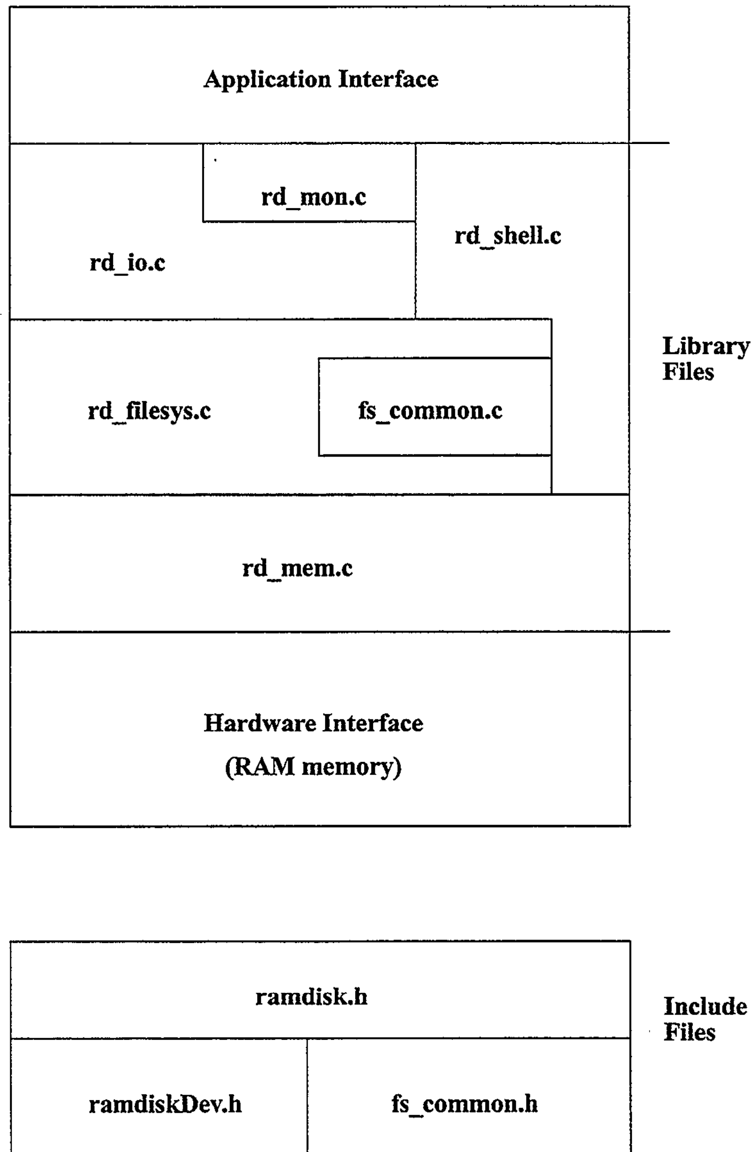


Figure 11. RAM Disk Files Structure.

5.2.4.1 File Access Operations

Files *xx_io.c* (where *xx* is *fl, fm or rd*, depending on the file system) contain functions that implement the file access operations required by a real-time program. Operations *format, open, close, read, write, flush, seek, tell* and *size* are implemented here. Files may be opened with access option RDONLY, WRONLY or WRONLY | APPEND. In all cases except RDONLY, if the file does not exist, it is created.

The *seek* operation may be performed on any successfully opened file. Since the smallest unit of storage in a file is a 32-bit integer, the *seek* index specifies the integer offset from the beginning of the file.

5.2.4.2 Maintenance Operations

Functions that maintain the file systems are present in files xx_shell.c. These functions would normally be called by a real-time program's operator console task or as a result of commands received from a host computer via a serial port. Functions are provided to format a file system, read it into its in-memory image and write it back out, delete or rename a file and collect data needed to display a file system directory.

5.2.4.3 Command Interface Operations

Functions specifically designed to ease the implementation of an operator console or EPROM monitor interface are present in files *xx_mon.c*. Interface functions are provided for all of the functions in files *xx_shell.c*. In addition, functions to copy files to and from the host and to checksum a file are provided. The functions implement an ASCII text dialog with an operator console task or host computer to support operator requests and indicate operator entry errors.

5.2.4.4 DOS Access Operations

The functions that read and write file system data structures and file sectors are contained in files *xx_filesys.c*. In the DOS FAT16 file system, parameters are packed into data structures on (8-bit) byte boundaries. By contrast, the TMS320C4x stores all parameters in 32-bit integers. To accommodate these differences, a set of functions was needed to unpack DOS FAT16 data structures as they are read in and to pack them as they are written out. These functions are present in file *fs_common.c* and are called by the functions in files *xx_filesys.c* as a file system is read or written. The functions in these files implement much of the interface to the DOS FAT16 file system.

5.2.4.5 Cluster Cache Operations

The on-board flash memory file system buffers an entire cluster of sectors in an internal buffer called the current cluster cache. Functions to read and write this buffer are present in file *fm_mem.c*. The buffer is actually used for two purposes. When the file system is being read or written, it is used to store a file system memory block image. As indicated earlier, the file system is stored in the first two 64 Kilobyte blocks of flash memory. However, during normal file access operations the current cluster cache buffer is used to store file clusters. Two sets of buffer access functions are included in file *fm_mem.c* to support these two sets of buffering operations.

5.2.4.6 Include Files

File system data structure definitions and symbolic constants are declared in include files *flash.h*, *flmem.h*, *ramdisk.h* and *fs_common.h*. Symbolic constants that define the "dimensions" and default parameter values for the file systems are defined in these files. For example, symbolic constant MAX_OPEN_FILES specifies the maximum number of files that may be open at one time for a file system (currently set to 20). The default number of sectors per cluster symbolic constant SECTORS_PER_CLUSTER is defined to be eight. Constants such as these are only needed when the library is being built.

Symbolic constants and function prototypes needed by an application program to access the file system functions present in the library have been broken out into separate include files: *flash-Dev.h*, *flmemDev.h* and *ramdiskDev.h*. These are the only files that need to be included in an application program.

5.2.5 File System Size Specifications

The target flash card used during the file system design was an Epson ATA212SD00. This is a 20 Mbyte card that conforms to the PCMCIA ATA command set specification. The file system should work with any other card conforming to this size and specification. DOS file systems that access disks larger than 32 Mbytes use extended partitions. The current FFS design does not implement this concept and is therefore limited to card memory capacity of 32 Mbytes.

The size and location of on-board flash memory accessed by the flash memory file system are specified by parameters set in include files. As a result, the file system library must be rebuilt to change these values. In the current design, parameter FLASH_MEM_SIZE (in file *flmem.h*) is set to 4 Mbytes and the memory starts at address 0x12000000.

The RAM disk file system uses memory allocated from the program heap. Its size is specified by parameter RAM_DISK_DATA_SIZE, which is set to 128 Kilobytes in the current design. Again, the library must be rebuilt to accommodate a different value.

5.2.6 File System Function Descriptions

Appendix A contains summary programming interface descriptions for all of the flash card file system functions callable from a real-time program or host command interface. As stated earlier, the interfaces for the flash memory and RAM disk file systems are similar—only function names are modified.

6. BUILT-IN TEST

The 9PAC software performs a set of built-in test (BIT) procedures at run-time to ensure the hardware and software are operating correctly. Tests include basic memory checks, detection of non-responsive tasks, and detection of non-responsive processors. For efficiency's sake, some of these conditions are inferred from the condition of the 'algorithmic' task data paths, while others are allocated to a dedicated BIT task executing on each processor node.

A block diagram of the tasks involved in BIT for 9PAC Phase I is shown in Figure 12. A BIT task executing on each processor is responsible for performing run-time memory diagnostics on the majority of 9PAC SRAM and DRAM memories. The SRAM attached to each processor is used almost exclusively for program code and intertask communication (ITC) buffers. In the case of program code, a simple checksum test is performed, verifying that the checksum matches the sum computed when the processor started running. The SRAM allocated to ITC buffers is *not* checked, as it is being accessed simultaneously by the C44 processor and the on-chip DMA coprocessor and is never in a known, testable state with respect to the C44. In this case, error checking logic in the ITC software is responsible for detecting any corrupted data packets and halting the affected C44 processor.

DRAM memory is used primarily for program data and is tested using a walking one's write/read test. A write/read pattern test is considered essential is the case of program data since the memory is being used in a write/read manner (in contrast to the program text area, which is read-only). In order to run such a test, the memory to be tested is divided into small segments. The segments are small enough that each one can be tested in a short, determinate amount of time, allowing interrupts to be disabled during the critical save/write/read/restore operation.

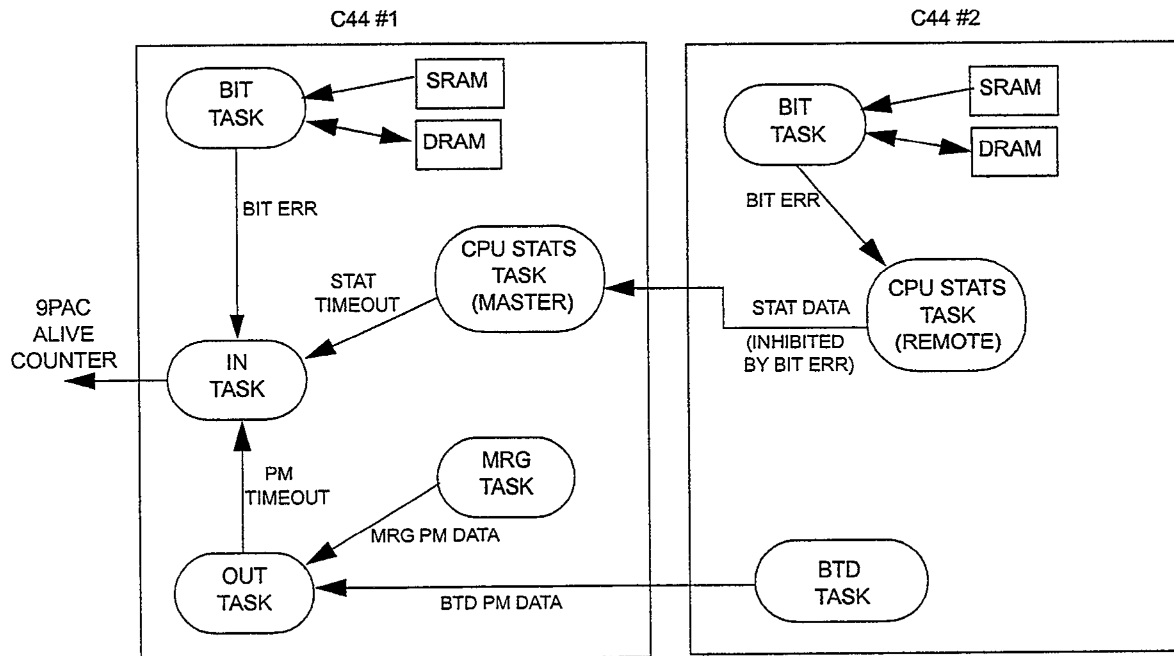


Figure 12. BIT Tasks and Communication Paths.

Note that a small region of DRAM memory is reserved for use by the ITC facilities. As with the SRAM equivalent, this region is not explicitly checked due to possible contention between the C44 and its DMA coprocessor.

Any error condition detected by the BIT memory testing task is handled in one of two ways. On a remote processor, an error status is communicated to the CPU statistics (CPU usage, etc.) task. The output of the CPU statistics task is inhibited, resulting in a timeout being detected on C44 #1 a short time later. This timeout, in turn, inhibits the 9PAC 'alive counter,' causing the radar to switch to the alternate channel via logic implemented within the ASR-9 ASP and MIP processors. This error path also serves to detect crashes of processor #2 since the CPU statistics data will be lost in such a case. The sharing of the CPU statistics data stream with the BIT functionality is a natural fit since a memory error can be considered to be a CPU statistic of sorts. The strategy also minimizes the use of ITC resources.

On processor #1, a BIT memory test error is communicated directly to the input task, and the alive counter is immediately inhibited (with the same result as above)

The second class of errors detected by the BIT subsystem are non-responsive processing tasks. In this case the hardware and system-level software are still functioning normally, but one of the major processing tasks is not. The most common causes of such a failure are infinite loops and unanticipated deadlocks. In a heavily tested system, failures of these types should be extremely rare, but they must be detected and handled nevertheless. Once again, advantage is taken of existing data streams to perform this task. Each major software module communicates performance monitoring data to an output task on processor #1 during normal operation. The basic update rate of the monitoring data packets is once per antenna scan (~4.6 sec), rapid enough to allow for timely detection of non-responsive tasks. As shown in the block diagram, a timeout of any of the performance monitoring data streams is communicated to the input task and the alive counter is again halted.

7. SOFTWARE APPLICATION ARCHITECTURE

The application software that runs on the 9PAC determines the positions of radar and beacon target reports, replacing software that previously ran on the ASR-9 Array Signal Processor (ASP). The deployment of 9PAC will occur in two phases. The Phase I application replaces only the beacon target detector (BTD) and radar/beacon target merge (MRG) functions of the ASP. The Phase II application includes the Phase I surveillance functions and also replaces the primary radar functions of the ASP, which are the radar correlation and interpolation (C&I) and radar scan-to-scan correlation (TRK) functions.

Phase I can be thought of as a subset of Phase II, and therefore there are many software libraries shared between the two applications. The many lower-level system services described in Sections 2 through 6 of this paper are shared, as are the BTD and MRG modules mentioned above.

The Phase I and II application programs are described in more detail in the following subsections.

7.1 The Phase I Application

The Phase I application consists of two executable programs since Phase I uses only two of the C44 processors on the 9PAC. A high-level block diagram of the Phase I application is shown in Figure 13. One processor, called the housekeeping processor, is responsible for all I/O functions, and in addition performs the radar/beacon target merge (MRG). The second processor, called the beacon processor, performs the BTD function, which entails generating beacon target reports from the beacon reply data input to 9PAC and identifying "false" target reports resulting from signal reflections.

The Phase I application interacts with the ASP and HSIB via a 128 kilobyte multi-port RAM. The multi-port RAM replaces the internal dual-port RAM used by the ASP in an ASR-9 without the 9PAC modification. The HSIB provides the beacon reply data. The ASP provides the primary radar target report data, variable site parameter data entered by an operator at the Remote Monitoring System (RMS) terminal, and also antenna position. The 9PAC outputs the target reports resulting from the MRG function. The 9PAC also outputs performance monitoring data once per scan, which the ASP forwards to the RMS for operator review. The ASP and 9PAC exchange various status information (e.g., alive counters, etc.).

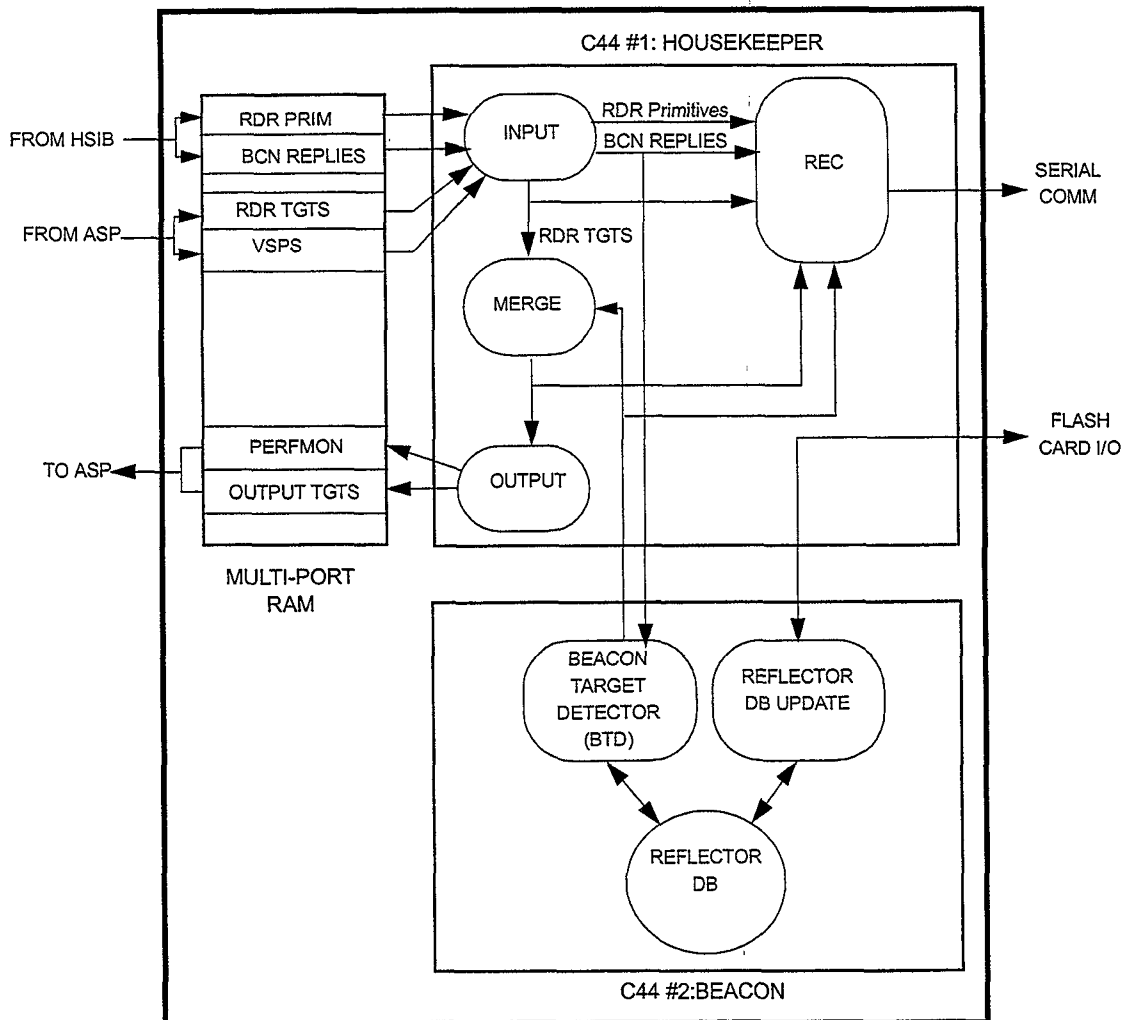


Figure 13. Phase I Application Software Block Diagram.

7.1.1 The Housekeeping Processor

The 9PAC housekeeping processor runs several software tasks. The major tasks, shown in the block diagram, include input, output, MRG, and data recording. A complete list is provided in Table 1, including the appropriate section in this document where low-level system services are described, where appropriate. The housekeeping processor is the only one that can access the multi-port RAM, serial ports, and flash card file system. As such, in Phase I, the housekeeping processor provides tasks to serve requests from the beacon processor to access these system resources.

Table 1.
Housekeeping Processor Tasks

Input	
Output	
MRG	
MRG Reflection Map Update	
Data Recording	Section 4
TTY Server	Section 4
File Server	Section 5
File MUX	Section 5
Built In Test (BIT)	Section 6
CPU Statistics	Section 6

The input task runs with the highest priority on the housekeeping processor. This design choice was made so that incoming data can be processed in a timely manner. In order for 9PAC to function properly, it must be fast enough to keep ahead of the incoming data stream. The input task fetches all incoming data from the ASP and HSIB and sends it to the appropriate 9PAC tasks using the PAC/OS interprocess communication facility described in Section 3. Beacon replies, antenna position, and VSP data are sent to the BTD task on the beacon processor. The beacon reply data are also sent to the data recording task. The primary radar primitive data are also sent to data recording even though these data are not processed by the Phase I application (since radar processing is performed by the ASP in Phase I). Radar target reports, antenna position, and VSP data are sent to the MRG task. The input task is also responsible for advancing the alive counter (approximately every 16 ms) that the ASP uses to determine whether 9PAC is functioning properly. Finally, the input task performs system integrity tests as follows: it checks the multi-port RAM to detect access errors and it looks for CPU statistics from the beacon processor to ensure that the beacon processor is alive. Whenever possible, the input task sleeps (for 2 ms) so that the other housekeeping tasks can run.

The primary responsibility of the output task is to copy target reports and performance monitor data into the multi-port RAM within certain time constraints. The output task is idle until data arrive in its input queues and the input tasks give it a chance to run. Target reports received from the MRG task are copied into the multi-port RAM for the ASP and also are sent to the data recording task. Beacon test reply data received from the BTD task are copied into the multi-port RAM. This can be thought of as part of the built-in-test capability. The output task receives messages from the MRG on a regular basis, even when no target report data is ready. This allows the output task to check for the arrival of performance monitor data from the BTD and MRG tasks. When both performance monitor data streams have arrived, usually near or at the north mark, the output task places this data in the multi-port RAM and also sends them to the data recording task.

The MRG task wakes up when it receives beacon target reports from the BTD or radar target reports from the input task. As was the case with the output task, the MRG task receives packets of data on a regular basis, even if there are no new target reports to process. Once awake, the MRG task inputs any new reports and goes through its beacon and radar target report lists looking for radar and beacon reports that appear to be from the same aircraft. The MRG task has a limit on how long it can hold reports. After dealing with any incoming reports and going through its report lists, the MRG task outputs any reports that it is done with to the output task. Radar and beacon target reports are also sent to the data recording task. Once per scan the MRG task sends performance monitoring data to the output task. Before going back to sleep, the MRG task checks for incoming VSP data from the input task.

As part of the MRG process, a dynamic radar/beacon reflection map is used to determine whether or not beacon reports flagged as false by the BTD can be deleted if they are merged with a primary radar report. A separate MRG reflection map update task is run, at a lower priority than the MRG task, to periodically copy this map to the 9PAC flash memory card.

Data recording is performed by connecting an external Sun workstation or suitably equipped PC to the high-speed serial high-level data link control (HDLC) port on the 9PAC. Data extraction services are provided by the 9PAC software in the data recording task running on the housekeeping processor. The data recording task has direct access to the serial I/O server process as discussed in Section 4. Data extraction from the beacon processor requires the formulation of data messages that are sent to the data recording task by the recording MUX task on the beacon processor. Figure 14 illustrates the various data message paths handled by the data recording task.

7.1.2 The Beacon Processor

The beacon processing is divided into two major tasks: the BTD and the reflector file update task. The BTD is the higher priority task that actually generates the beacon target reports and maintains the dynamic reflector file. The lower priority reflector file update task is responsible for periodically copying the reflector file to the 9PAC flash memory card. When the lower priority task runs, updates to the reflector file by the BTD are prevented via the semaphore mechanism provided by the PAC/OS kernel (Section 2.3).

The BTD task receives beacon replies, antenna position, and VSP data from the input task on the housekeeping processor. It outputs beacon target reports to the MRG task and beacon test targets and performance monitoring data to the output task. BTD also sends a variety of tracking and reflection data to the recording MUX, which forwards the data to the housekeeping data recording task as shown in Figure 14.

A complete task list for the beacon processor is provided in Table 2. The antenna position task runs periodically for a short time to receive the antenna position from the input task. The recording MUX and TTY MUX tasks are part of the serial I/O function (Section 4). The File MUX forwards requests for flash memory card operations to the File Server task on the housekeeping processor as discussed in Section 5. BIT and CPU Statistics tasks are part of the built-in-test facility (Section 6).

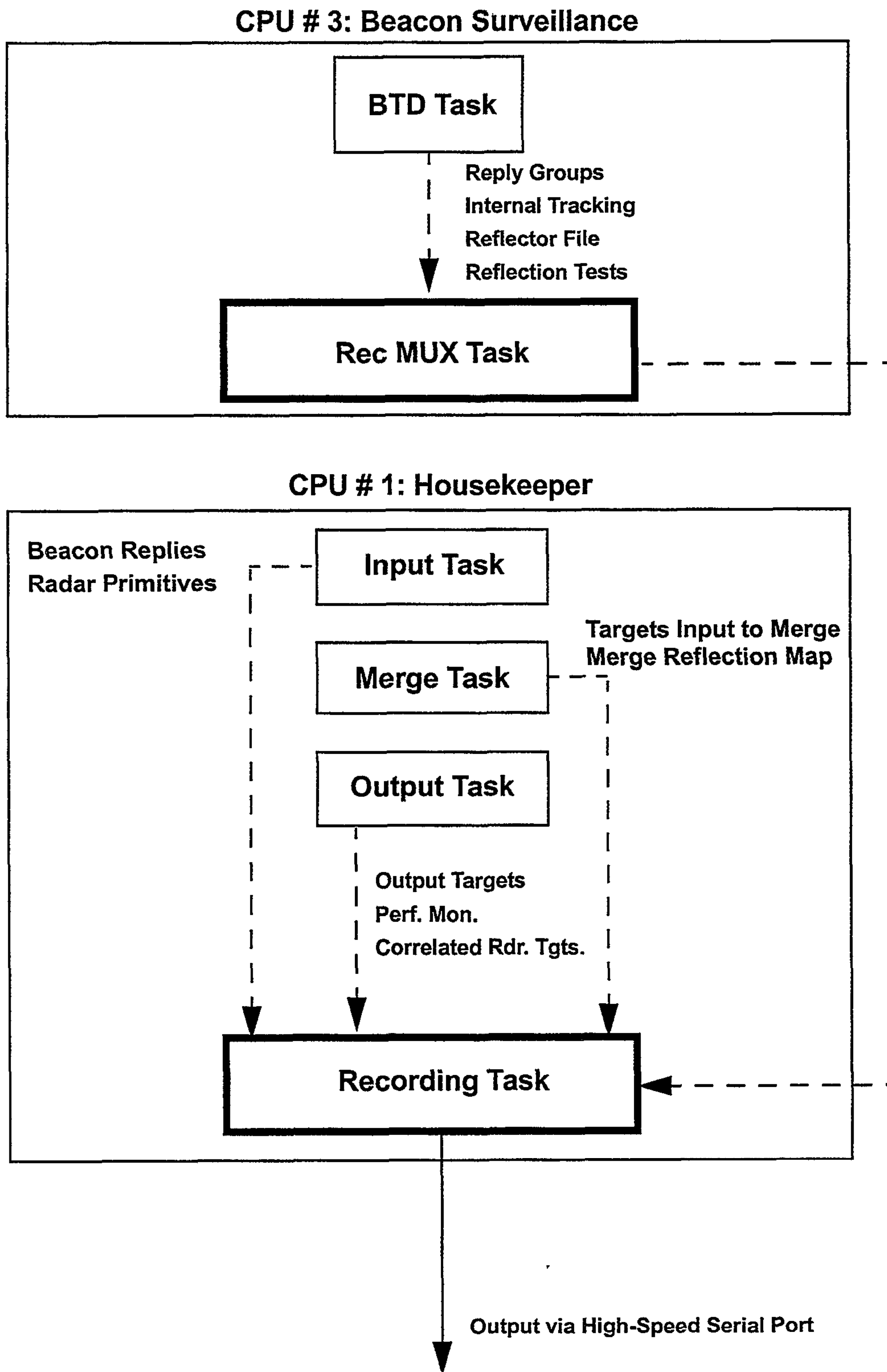


Figure 14. Phase I Data Extraction Block Diagram.

Table 2.
Beacon Processor Tasks

BTD	
Reflector File Update	
Antenna Position	
Recording MUX	Section 4
TTY MUX	Section 4
File MUX	Section 5
Built In Test (BIT)	Section 6
CPU Statistics	Section 6

7.2 The Phase II Application

The Phase II application performs much of the Phase I functionality and adds primary radar processing. Phase II consists of three executable programs, one for each of the C44 processors on the 9PAC. A high-level block diagram of the Phase II application is shown in Figure 15. The housekeeping and beacon processors perform similar tasks to their Phase I counterparts. The beacon processing executable is moved to the third processor, and the radar processing lives on the second processor. This was done because the second processor has twice as much dynamic RAM as the other two–the extra DRAM is required by the radar processing algorithms.

The Phase II application interacts with the MIP and HSIB via the multi-port RAM on the 9PAC and also on the external dual-port RAM where the MIP looks for target report data. The ASP is completely removed, and the 9PAC assumes control of the interface. The MIP is unable to access the multi-port RAM on the 9PAC, so all communications must take place via the external dual-port RAM. The Phase II application also takes over direct communication with the HSIB. The HSIB provides both beacon replies and radar primitives. The MIP provides the VSP data and antenna position. The 9PAC generates both the radar and beacon target reports, performs the radar/beacon merge, and outputs the target and performance monitoring data to the external dual-port RAM. The MIP and 9PAC exchange various status information (e.g., alive counters, etc.).

7.2.1 The Housekeeping Processor

The Phase II housekeeping processor runs an executable program very similar to the Phase I program, with a few exceptions. This section will discuss only the differences. Refer to Appendix A for a description of the basic housekeeping functions.

The Phase II input task does not receive radar targets; instead, it fetches radar primitives from the HSIB and sends them to the radar correlation and interpolation (C&I) task, which resides on the radar processor. VSP data are sent to the radar processing tasks (C&I and TRK).

The Phase II output task has additional performance monitoring streams to handle (from the C&I and TRK tasks). The output task also has to process correlated radar reports sent by the TRK task on the radar processor. These reports are copied into the external dual-port RAM in the same manner as are the target reports output by the MRG. The correlated reports are also sent to the data recording task.

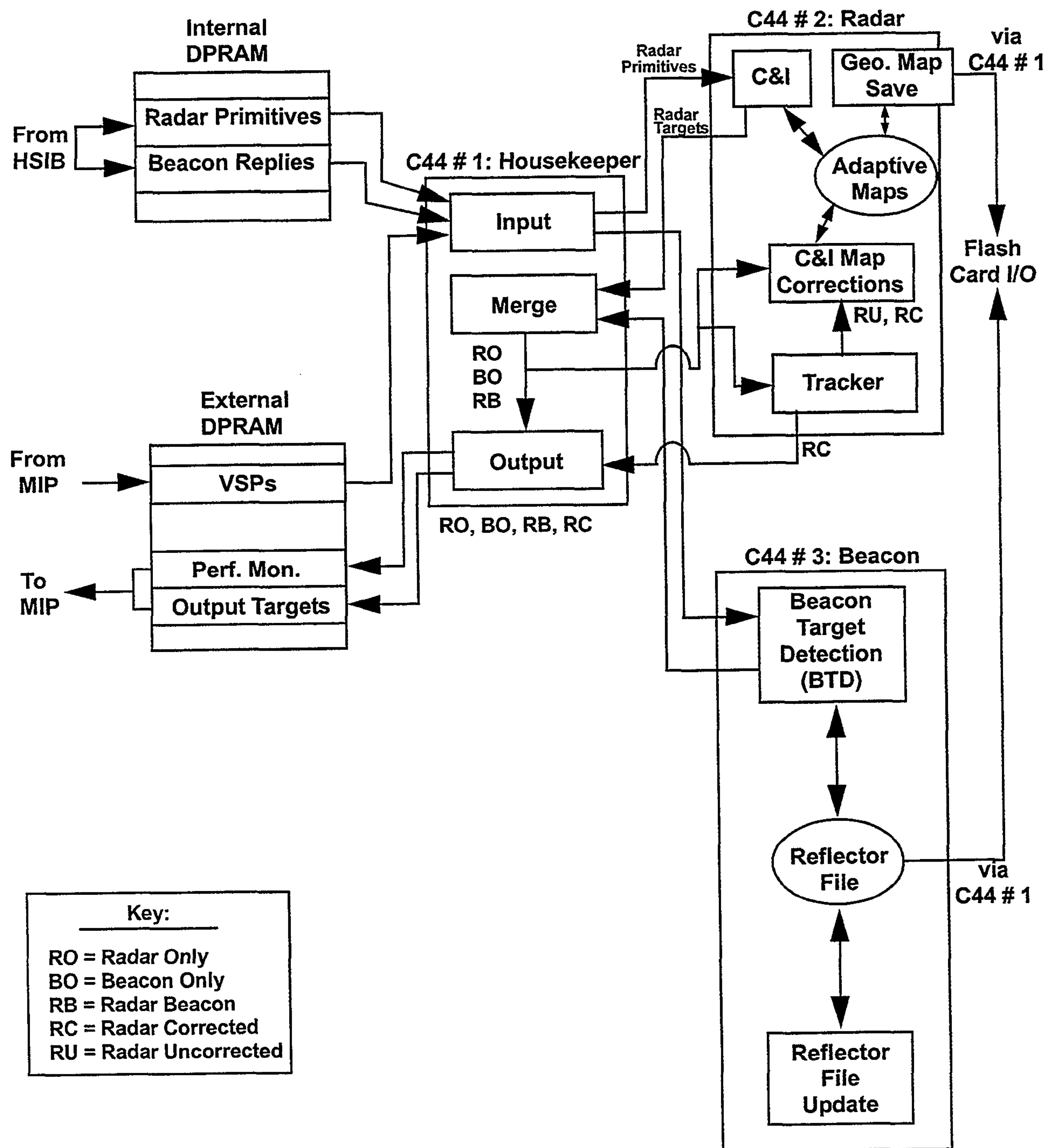


Figure 15. Phase II Application Software IBI Mode Block Diagram.

The block diagram in Figure 15 applies only when Phase II 9PAC is performing the beacon and merge functions. When the ASR-9 is co-located with a Mode S beacon system, the Phase II application must modify its behavior when Mode S is performing the beacon and merge functions. A block diagram of the Phase II software interactions in monopulse mode is illustrated in Figure 16. As shown in the diagram, there are two differences. The first is that the MRG processing is bypassed to allow radar targets to go from the C&I task to the output task as quickly as possible. The second difference is that the input task has to retrieve from the external dual-port RAM target reports resulting from the Mode S merge. It then forwards these reports directly to the TRK task.

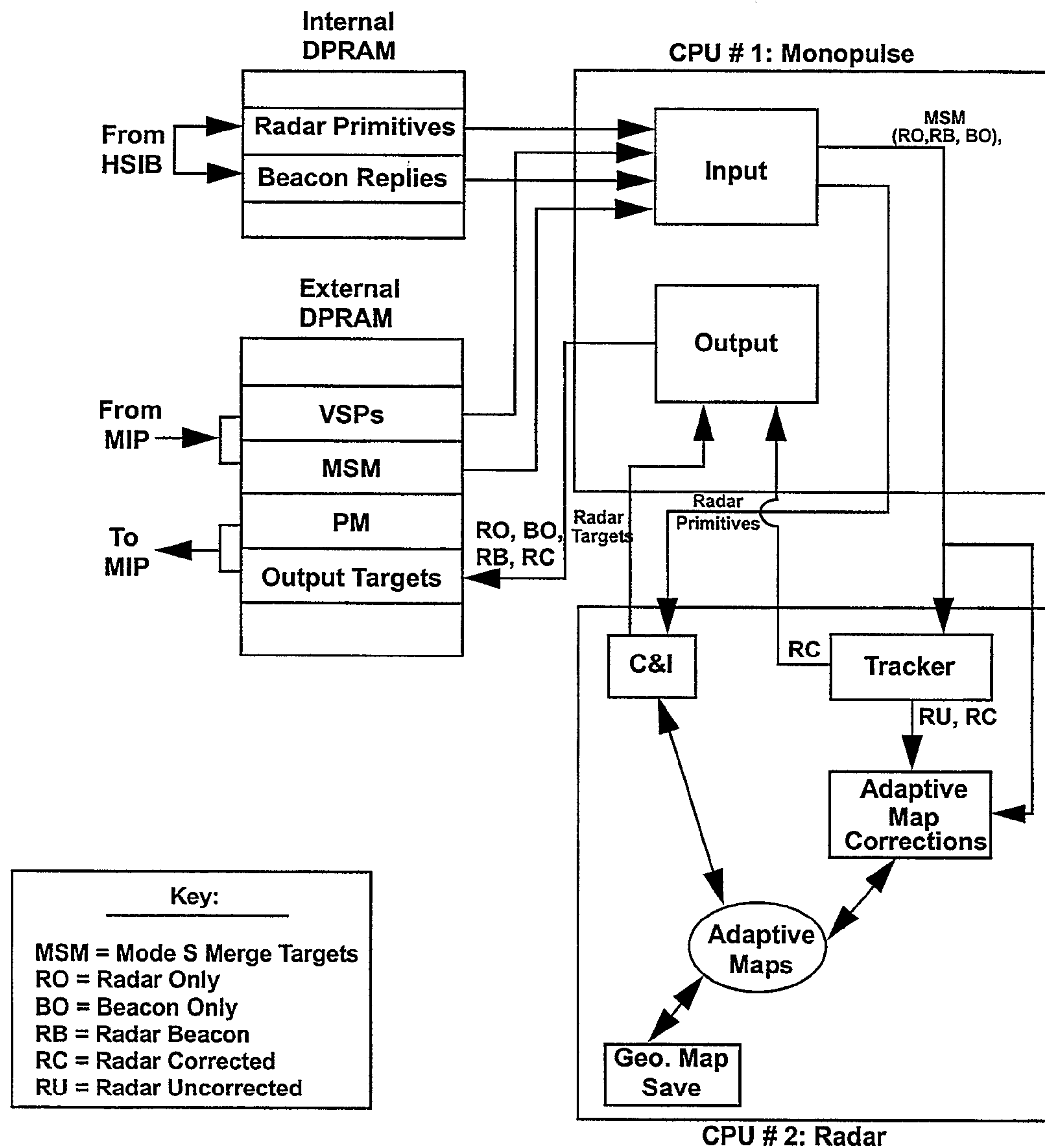


Figure 16. Phase II Application Software Monopulse Mode Block Diagram.

The Phase II data recording task is identical to that of Phase I. There are, however, additional data streams in Phase II that are not present in Phase I. These new data messages to be extracted from the 9PAC include primary radar adaptive map data, as well as track status information from the radar TRK function. The Phase II data extraction architecture is illustrated in Figure 17.

7.2.2 The Radar Processor

The radar processor of the Phase II 9PAC performs two major functions previously handled by the ASP: radar correlation and interpolation (C&I) and radar scan-to-scan correlation (TRK).

The C&I function itself performs three different sub-functions. The first job, correlation, consists of grouping together clusters of individual radar primitive input data by range and azimuth. The second job, interpolation, consists of determining the range, azimuth, and Doppler estimate for each radar primitive cluster. This results in a radar target report. There are many sources of false target reports, including ground clutter, vehicular traffic, weather fronts, and bird flocks. The third and final job of the C&I, adaptive thresholding and geocensoring, is responsible for setting amplitude thresholds in range/azimuth/Doppler space to regulate the probability of aircraft detection and false-alarm generation.

The C&I function is separated into three software tasks. The C&I task performs the bulk of the work, which includes correlation, interpolation, adjusting the various adaptive map thresholds, and applying these thresholds to each target report. It also outputs to the MRG task those reports that pass through the adaptive thresholding and geocensoring filters. The C&I Adaptive Update task makes corrections to the adaptive thresholds based on the results of the radar/beacon MRG and TRK functions. This prevents returns from aircraft from being used to erroneously increase the adaptive thresholds. Finally, the Geocensor Map Save task periodically copies the geocensor map data to the 9PAC flash file system on the flash memory card. This task is quite similar to the update tasks used to save the beacon dynamic reflector file (Section 7.1.2) and dynamic false target merge map (Section 7.1.1).

The TRK function performs scan-to-scan correlation of reports output by the MRG task. The TRK task tracks radar-only, beacon-only, and radar-beacon reports and maintains a track file. However, the ASR-9 is required to output only correlated radar reports, so the tracker outputs only those radar reports that correlate to a track. Beacon-only and radar-beacon reports are not output. Correlated radar reports are sent to the output task on the housekeeping processor.

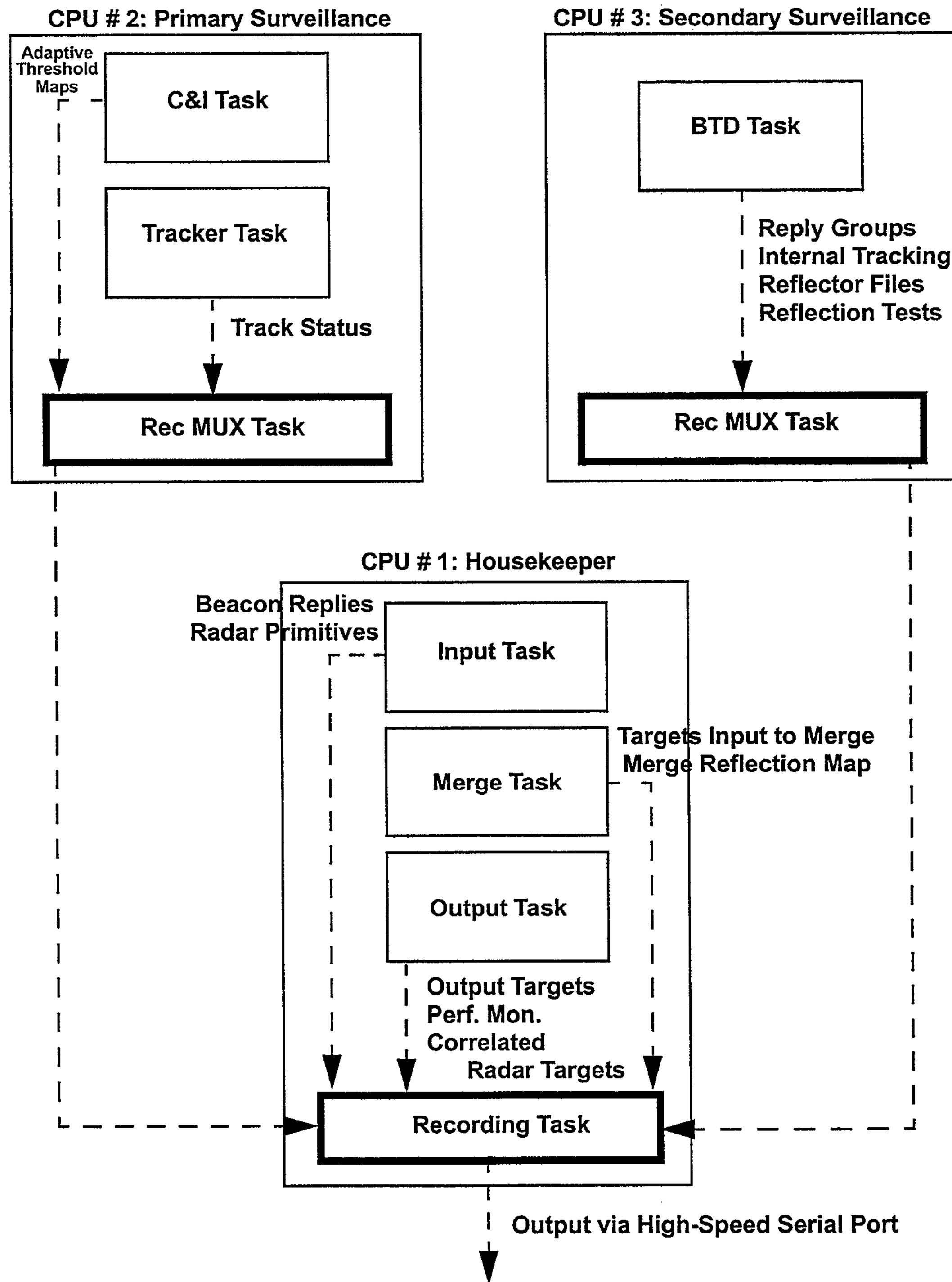


Figure 17. Phase II Data Extraction Block Diagram.

7.2.3 The Beacon Processor

The beacon processing in Phase II is identical to that of Phase I. See Section 7.1.2 for more information.

APPENDIX A
PAC/OS FUNCTION REFERENCE

This section provides detailed information regarding the PAC/OS functions available to application tasks. The functions are grouped into the following categories:

- Flash Filesystem
- MailBoxes
- Memory Allocation
- Queues
- Lists
- Stacks
- Semaphores
- Signals
- Register Access
- Task Management

A.1 Flash Filesystem Functions

fl_format()
fl_getFFS()
fl_checkFFS()
fl_delete()
fl_dir()
fl_move()
fl_putFFS()
fl_open()
fl_close()
fl_read()
fl_write()
fl_flush()
fl_seek()
fl_tell()
fl_size()

fl_format()

SYNOPSIS

int fl_format(unsigned int sectors_per_cluster);

sectors_per_cluster Number of (512 byte) sectors managed by each file system cluster

RETURNS

Returns 0 on success; returns -1 on failure.

DESCRIPTION

fl_format() creates a standard DOS FAT16 file system on a PCMCIA ATA flash card with the specified number of sectors per cluster. A sector is a contiguous block of 512 bytes; a cluster is a contiguous block of sectors. A cluster is the smallest unit of storage managed by the file system. Note that the time it takes to read/write the file system from/to the flash card is heavily dependent upon the sector per cluster value chosen and that writes take about ten times longer than reads. At one sector per cluster, it takes about eight seconds to write the file

system. At four, it takes about two seconds. Recommended is 8 or 16 sectors per cluster. Most laptop/notebook PCs use 2 or 4, reflecting their use with lots of small files. Some, but not all, can read file systems with values other than their design value. For a flash card to be accessible by a laptop/notebook, it must first be formatted by that machine. This puts code on the card that is executed by the laptop/notebook each time the card is inserted. The **fl_format()** function may then be used to reformat the card to change its sector-per-cluster value. The **fl_format()** function does not write over the code placed on the card by the laptop/notebook.

fl_getFFS()

SYNOPSIS

int fl_getFFS(unsigned int do_chksum);

do_chksum Checksum checks the file system as it is read

RETURNS

Returns 0 on success; returns -1 on failure.

DESCRIPTION

fl_getFFS() reads a DOS FAT16 file system from a PCMCIA ATA flash card into an in-memory image. The file system may have been created by **fl_format()** or by a compatible laptop/notebook format command. THIS FUNCTION MUST BE SUCCESSFULLY CALLED BEFORE ALL OTHER FILE SYSTEM FUNCTIONS (except **fl_format()**) to load a copy of the file system data structures into RAM memory. The other functions access these structures. If the file system should be checksum checked as it is read, **do_chksum** should be TRUE, otherwise set it to FALSE. This feature was included to allow file systems that were alternatively created or changed (e.g., via a laptop/notebook) to be read. Checksums are computed and stored on the card when **fl_putFFS()** is run.

fl_checkFFS()

SYNOPSIS

int fl_checkFFS(void);

RETURNS

Returns TRUE if the file system has been loaded, FALSE if it has not.

DESCRIPTION

fl_checkFFS() checks to see if the flash file system has been loaded (i.e., **fl_getFFS()** has been run successfully). It returns TRUE if it has, FALSE if it has not.

fl_delete()

SYNOPSIS

int fl_delete(char *filename);

filename Name of file (in DOS 8.3 format) to be deleted

RETURNS

Returns 0 if successful, -1 on error, and -2 if file was not present.

DESCRIPTION

fl_delete() deletes the specified file from the file system. Note that a filename must have a DOS 8.3 format. That is, it must have up to eight characters for the primary name and up to three characters for the name extension. Note also that THE FFS FILE SYSTEM SUPPORTS ALPHA, NUMERIC AND '_' CHARACTERS ONLY in a filename. Any other characters or format will result in a -1 error return.

fl_dir()

SYNOPSIS

unsigned int fl_dir(FILENAME_T ** fn_ptr[ ]);

fn_ptr[ ]	An array of pointers to FILENAME_T structures

RETURNS

Number of valid FILENAME_T data structures pointed to by fn_ptr.
Pointer fn_ptr, which points to the data structures.

DESCRIPTION

fl_dir() collects the filenames and (8 bit byte) sizes of files in the file system. FILENAME_T data structures containing file name and size data are constructed in an internal static array. This function returns the number of valid entries in that array and its address. The function collects the names and sizes of files for display by a command shell program. A file system may have up to 255 entries. The data structure is shown below:

```
typedef struct {            /* file info data structure */

    char *file_name_ptr; /* pointer to file name */
    char *file_ext_ptr;  /* pointer to file extension */
    unsigned int file_size;/* file size in 8 bit bytes */

}FILENAME_T;
```

Call/use syntax:

```
FILENAME_T **fnp;

num_entires = fl_dir(&fnp);
char_ptr1 = fnp[0]->file_name_ptr;
char_ptr2 = fnp[0]->file_ext_ptr;
uint_val = fnp[0]->file_size;
etc.
```

fl_move()

SYNOPSIS

int fl_move(char *srcfile, char *destfile);

srcfile	Source filename
destfile	Destination filename

RETURNS

Returns 0 on success, -1 on error, -2 if source file is not present, and -3 if the destination filename is in error or already exists.

DESCRIPTION

fl_move() changes the name of a file. Note that if a file with the destination filename already exists, it will not be overwritten. It is necessary to delete the file before this function will succeed. Both filenames must conform to the DOS 8.3 format and contain alpha, numeric and "_" characters only.

fl_putFFS()

SYNOPSIS

int fl_putFFS(void);

RETURNS

Returns 0 on success; returns -1 on failure.

DESCRIPTION

fl_putFFS() writes the DOS FAT16 file system from the in-memory image to the PCMCIA ATA flash card. This function MUST BE CALLED TO RECORD ANY CHANGES MADE BY THE FILE SYSTEM ACCESS FUNCTIONS. Note that a file's data sectors are written to the file system as they are changed. However, the in-memory file system data structures that manage access to these sectors are not written to the card until this function is called. This function "synchronizes" the card's file system data structures with its data. As a result, this operation should be carried out periodically, when time allows. Note also that this operation requires a fair amount of execution time. However, most of that time is spent polling the card for completion status. Therefore, the best place to call this function is in a low-priority task that may be preempted by other higher priority tasks.

fl_open()

SYNOPSIS

int fl_open(char *filename, unsigned int flags);

filename	Name of file to be opened (in DOS 8.3 format)
flags	File access attributes. Selected from the following: FL_RDONLYOpen for reading an existing file (returns -1 if file does not exist) FL_WRONLYOpen for writing only (if file exists, it is truncated to zero length) FL_WRONLY | FL_APPEND Open file for writing only (if file exists, it writes at its end)

RETURNS

On success, returns a file descriptor; on failure, returns -1.

DESCRIPTION

fl_open() opens the named file with the specified access attributes. In all cases except FL_RDONLY, if the named file does not already exist, it is created. The file's current file index is set to the beginning of the file in all cases except FL_WRONLY | FL_APPEND. In that case, the current file index equals the current length of the file—so an **fl_write()** operation will append to the file. The current file index may be moved to another location in the file using the **fl_seek()** function. Writes and reads will then begin at that location.

fl_close()

SYNOPSIS

int fl_close(unsigned int fd_idx);

fd_idx	File descriptor (an index, returned by a previously successful fl_open() call)

RETURNS

On success, returns 0; on failure, returns -1.

DESCRIPTION

fl_close() closes the file indicated by its argument file descriptor.

fl_read()

SYNOPSIS

int fl_read(unsigned int fd_idx, void *buff, unsigned int num_ints);

fd_idx	File descriptor
buff	Pointer to destination buffer
num_ints	Number of integers (32 bits) to read

RETURNS

On success, returns number of integers actually read; on failure, returns -1.

DESCRIPTION

fl_read() attempts to read the specified number of (32 bit) integers from the specified file starting at the current file index. The current file index points to the beginning of the file if the read is the first since the file was opened, or to the next integer in the file after the last read/write operation, or to the location specified in a previous **fl_seek()** operation. If the number of integers requested exceeds the number remaining in the file, the function reads those remaining and returns the number read. If the file descriptor argument is invalid (file is not open) or if a read operation fails (flash card problem), the function returns -1.

fl_write()

SYNOPSIS

int fl_write(unsigned int fd_idx, void *buff, unsigned int num_ints);

int fd_idx	File descriptor

buff	Pointer to source buffer
num_ints	Number of integers (32 bits) to write

RETURNS

On success, returns number of integers actually written; on failure, returns -1.

DESCRIPTION

fl_write() attempts to write the specified number of (32 bit) integers to the specified file starting at the file's current file index. The current file index points to the beginning of the file if the write is the first since the file was opened, or to the next integer in the file after the last read/write operation, or to the location specified in a previous **fl_seek()** operation. If the file descriptor argument is invalid (file is not open) or if a write operation fails (flash card problem or card is full), the function returns -1.

fl_flush()

SYNOPSIS

int fl_flush(unsigned int fd_idx);

fd_idx	File descriptor

RETURNS

On success, returns 0; on failure, returns -1.

DESCRIPTION

fl_flush() writes the current in-memory file sector buffer to the flash card. A program may wish to do this to assure that all data written to the file have actually been written to the flash card (similar to the UNIX **sync** operation). The function does not modify the file's logical contents.

fl_seek()

SYNOPSIS

int fl_seek(unsigned int fd_idx, unsigned int offset);

fd_idx	File descriptor
offset	File index

RETURNS

On success, returns 0; on failure, returns -1.

DESCRIPTION

fl_seek() relocates the current file index to the location specified by **offset.** The value of **offset** is the (32 bit) integer file index relative to the beginning of the file. It may be set to **FL_EOF**, in which case a subsequent **fl_write()** operation will append data to the end of the file. The value of **offset** may not exceed the length of the file (see **fl_size()** for length). The function will return -1 if it does, or if a card access problem occurs. The function may be used on any successfully opened file.

fl_tell()

SYNOPSIS

int fl_tell(unsigned int fd_idx);

fd_idx	File descriptor

RETURNS

Returns the file's current file index.

DESCRIPTION

fl_tell() returns the specified file's current file index. The current file index is the index (offset) to the next integer to be read or written. Its range is zero to the length of the file in integers. As a result, it may point to the next integer after the last integer written (as yet unoccupied). A file's current file index is advanced automatically as a file is read or written. Alternatively, it may be set by function **fl_seek()**.

fl_size()

SYNOPSIS

int fl_size(unsigned int fd_idx);

fd_idx	File descriptor

RETURNS

Returns the file's size in integers.

DESCRIPTION

fl_size() returns the specified file's size in (32 bit) integers (i.e., the number of integers in the file). If a file's (8 bit) byte size is not an integral number of integers, this function returns one more than the number of complete integers in the file. The last integer will contain less than four valid 8 bit bytes. Note that if the file was created by a laptop/notebook DOS file system, it may not be an integral number of integers in length.

A.2 Mailbox Functions

Function summary:

Mailbox *MailboxCreate()
Bool MailboxPend()
Bool MailboxPost()

MailboxCreate()

SYNOPSIS

Mailbox *MailboxCreate(Uns maxMsgSize, Uns mbxSlots, MailboxAttrs *attrs)

maxMsgSize	Maximum size for each message, in bytes.
mbxSlots	Total available slots in the mailbox
attrs	Other mailbox attributes (not yet used)

RETURNS

Pointer to an initialized mailbox.

DESCRIPTION

MailboxCreate() creates and initializes a new mailbox. The user specifies what the largest permitted message is, and also the number of available slots. No mailbox attributes are currently defined at this time, so the attrs argument should always be specified as NULL.

MailboxPend()

SYNOPSIS

Bool *MailboxPend(Mailbox *mbx, void *msg, int *length, int timeout)

mbx	Mailbox Pointer
msg	Buffer for received message
length	Length of received message
timeout	Timeout in system clock ticks

RETURNS

TRUE if a message was found, FALSE if a timeout occurred

DESCRIPTION

MailboxPend() Waits for a message to appear in the mailbox. If the timeout argument is non-zero, the routine will return FALSE if a timeout occurs before a message arrives. If the timeout argument is 0, the routine waits for a message indefinitely. Upon receipt of a message, the length of the message is return in the **length** argument.

MailboxPost()

SYNOPSIS

Bool *MailboxPost(Mailbox *mbx, void *msg, int length, int timeout)

mbx	Mailbox Pointer
msg	Buffer pointer to transmit message
length	Length of transmit message
timeout	Timeout in system clock ticks

RETURNS

TRUE if a message was sent, FALSE if a timeout occurred

DESCRIPTION

MailboxPost() Waits for a free slot to open in the mailbox, then copies the message to the mailbox. If the timeout argument is non-zero, the routine will return FALSE if a timeout occurs before a free slot becomes available. If the timeout argument is 0, the routine waits indefinitely for a free slot. The length of the transmitted message is specified by the caller in the **length** argument.

A.3 Memory Functions

Definition and Function Summary

```
#define PAC_SRAM 0
#define PAC_DRAM 1

typedef struct
{
 Uns size ;
 Uns used ;
 Uns length ;
 Uns arg ;
} MemSegStat ;

void *MemAlloc();
void *MemAllocVerboseErr()
void  MemInit();
void  MemStat() ;
```

MemAlloc(), MemAllocVerboseErr()

SYNOPSIS

void *MemAlloc(int segId, int size, int align);

void *MemAllocVerboseErr(int segId, int size, int align, char *errMsg);

segId	Memory segment Id. Valid values for the 9PAC are PAC_SRAM or PAC_DRAM
size	Requested memory size in bytes.
align	Buffer alignment. Must be a power of 2 (or 0)
errMsg	Verbose version Error message.

RETURNS

A pointer to a memory buffer of the requested size. These function do not currently return if an error occurs, but instead halt the processor after leaving an error message in the trace buffer. If a call to **MemAlloc()** fails, a message similar to the following will be put in the trace buffer:

MemAlloc: segId=X, size=XXXX ** Out of Memory! ******

If **MemAllocVerboseErr()** fails, it also writes the specified extra message to the trace buffer before halting. Usually the extra message consists of the function name that is calling **MemAllocVerboseErr()**, enabling the user to quickly tell which **MemAlloc()** is causing the problem.

MemInit()

SYNOPSIS

void MemInit(MemConfigSeg memConfigTable[]);

DESCRIPTION

MemInit() is called by main() at system startup to initialize the memory segments. Unless PAC/OS itself is being modified, the user will never call this routine.

MemStat()

SYNOPSIS

void MemStat(int segId, MemSegStat *segStat)

segId	Memory segment Id. Valid values for the 9PAC are PAC_SRAM or PAC_DRAM
segStat	Pointer to structure for retrieved status information

DESCRIPTION

MemStat() allows the caller to retrieve the current status of a memory segment. The returned structure is laid out as follows:

```
typedef struct
{
    Uns size ;
    Uns used ;
    Uns length ;
    Uns arg ;
} MemSegStat ;
```

A.4 Queue Functions

Definition and Function Summary

```
typedef struct QueueElemStruct QueueElem ;
struct QueueElemStruct
{
 QueueElem *next ;
 QueueElem *prev ;
}

typedef struct
{
 QueueElem *first ;
 QueueElem *last ;
} Queue ;

Queue       *QueueCreate()
Bool        *QueueEmpty()
QueueElem   *QueueGet()
QueueElem   *QueueGetWait()
void         QueuePut()
void         QueuePutAndSignal()
```

QueueCreate()

SYNOPSIS

Queue *QueueCreate(QueueAttrs *attrs)

attrs	Queue attributes (not yet used)

RETURNS

Pointer to an initialized queue.

DESCRIPTION

QueueCreate() creates a new queue and initializes it to be empty. No queue attributes are currently defined at this time, so the **attrs** argument should always be specified as NULL.

QueueEmpty()

SYNOPSIS

Boot *QueueEmpty(Queue *q)

q	Queue pointer.

RETURNS

TRUE if the queue is empty, otherwise FALSE.

QueueGet()

SYNOPSIS

QueueElem *QueueGet(Queue *q)

q	Queue pointer.

RETURNS

Element from front of queue, or NULL if the queue is empty.

QueueGetWait()

SYNOPSIS

QueueElem *QueueGetWait(Queue *q, int sigMask, int timeout)

q	Queue pointer.

RETURNS

Element from front of queue, or NULL if a timeout occurs.

DESCRIPTION

If the queue is non-empty, **QueueGetWait()** acts like **QueueGet()** and immediately returns the item at the front of the queue. If the queue is empty, this routine blocks waiting for a signal or timeout. If a timeout occurs before a signal, NULL is returned; otherwise, a pointer to the first item in the queue is returned. This routine is typically used in conjunction with **QueuePutAndSignal()** (see below)

QueuePut()

SYNOPSIS

void QueuePut(Queue *q, QueueElem *elem)

q	Queue pointer.
elem	Pointer to a data structure containing a QueueElem as its first member

DESCRIPTION

QueuePut() places the specified queue element on the tail of the queue.

QueuePutAndSignal()

SYNOPSIS

void QueuePutAndSignal(Queue *q, QueueElem *elem, Task *task, int sigNum)

q	Queue pointer.

elem	Pointer to a data structure containing a QueueElem as its first member
task	Task pointer
sigNum	Task signal number

DESCRIPTION

QueuePutAndSignal() places the specified queue element on the tail of the queue and then signals the specified task that it has done so. This routine is typically used in conjunction with **QueueGetWait()**.

A.5 List Functions

Function summary

Linklist *ListAlloc()
Linklist *ListCreate()
void ListAddItem()
void ListInit()
void ListAppendItem()

void *ListGetItem()
void *ListGetFirst()
void *ListGetNext()
void *ListDeleteItem()

int ListEmpty()
void *ListSeek()
void *ListGetItemNum()
void ListFreeItemsToStack()
int ListPack()
int ListUnpack()

void *ListGetFirstNode()
void *ListGetLastNode()
void *ListGetPrevNode()
void *ListGetNextNode()
void ListInsertNode()
void ListAppendNode()
void *ListDeleteNode()

ListAlloc()

SYNOPSIS

Linklist *ListAlloc()

RETURNS

A pointer to an initialized (empty) linked list, if successful; aborts program on a memory allocation failure.

DESCRIPTION

ListAlloc() allocates memory for a Linklist structure and initializes it (by calling **ListInit()**). The list name defaults to “noname”.

ListCreate()

SYNOPSIS

Linklist *ListCreate(ListAttrs *attrs)

attrs	Pointer to a list attributes structure

RETURNS

A pointer to an initialized (empty) linked list, if successful; aborts the program on a memory allocation failure.

DESCRIPTION

ListCreate() allocates memory for a Linklist structure and initializes it (by calling **ListInit()**). The list name is set to the string pointed to by attrs->name.

ListAddItem()

SYNOPSIS

void ListAddItem(Linklist *list, void *item, int pos)

list	Pointer to an existing linked list
item	Pointer to item to be added to the list
pos	Position of insertion:
APPEND_ITEM	Add at end
INSERT_ITEM_BEFORE	Insert before "current item"
INSERT_ITEM_AFTER	Insert item after "current item"
PREPEND_ITEM	Add at beginning

RETURNS

void; Aborts program if **pos** is invalid.

DESCRIPTION

ListAddItem() adds an item to an existing list at the position indicated by **pos**. The list "current item" (**list->curr_node**) is the position of the last list operation. All but the last seven list functions listed above maintain the "current item" pointer, **list->curr_node**, for future reference.

ListInit()

SYNOPSIS

void ListInit(Linklist *list)

list	Pointer to a Linklist structure to be initialized

RETURNS

void

DESCRIPTION

ListInit() initializes a Linklist structure to the "empty" state.

ListAppendItem()

SYNOPSIS

void ListAppendItem(Linklist *list, void *item)

list	Pointer to an existing linked list
item	Pointer to item to be appended

RETURNS

void

DESCRIPTION

ListAppendItem() is provided to make function inlining more likely for this frequently used operation. It appends an item to the end of an existing list.

ListGetItem()

SYNOPSIS

void *ListGetItem(Linklist *list, int option)

list	Pointer to an existing linked list option identifier of item to get:
NEXT_ITEM	item after "current item"
FIRST_ITEM	first item in list
LAST_ITEM	last item in list
PREV_ITEM	item before "current item"
CURR_ITEM	"current item"

RETURNS

Pointer to requested item; NULL if requested item is not present.

DESCRIPTION

ListGetItem() returns a pointer to the requested item on an existing linked list. It DOES NOT remove the item from the list. It repositions "current item" to the item whose pointer is returned.

ListGetFirst()

SYNOPSIS

void *ListGetFirst(Linklist *list)

list	Pointer to an existing linked list

RETURNS

Pointer to first item in the list.

DESCRIPTION

ListGetFirst() returns a pointer to the first item on an existing linked list. It DOES NOT remove the item from the list. The "current item" is repositioned to the first item.

ListGetNext()

SYNOPSIS

void *ListGetNext(Linklist *list)

list   Pointer to an existing linked list

RETURNS

Pointer to next item in list; NULL if "current item" is at end of list or list is empty.

DESCRIPTION

ListGetNext() returns a pointer to the next item on an existing linked list. It DOES NOT remove the item from the list. The next item is the one after the current "current item." The "current item" is repositioned to the item whose pointer is returned.

ListDeleteItem()

SYNOPSIS

void *ListDeleteItem(Linklist *list, Node_header *node)

list   Pointer to an existing linked list

node   Pointer to list item to be deleted

RETURNS

Pointer to list item after the item deleted; NULL if item deleted is the last node on the list.

DESCRIPTION

ListDeleteItem() repositions "current item" to the item past the item to be deleted and then removes the item from the list. If the item to be removed is the last item on the list, "current item" is set to NULL. Note that it is the calling function's responsibility to return the deleted item to the list's associated free stack.

ListEmpty()

SYNOPSIS

int ListEmpty(Linklist *list)

list   Pointer to an existing linked list

RETURNS

TRUE if list is empty, FALSE otherwise.

DESCRIPTION

ListEmpty() tests a list to see if it is empty or not and returns an indicator.

ListSeek()

SYNOPSIS

void *ListSeek(Linklist *list, Node_header *seek_node)

list   Pointer to an existing linked list

seek_node   Pointer to a node item

RETURNS

seek_node

DESCRIPTION

ListSeek() repositions "current item" to the item described by seek_node.

ListGetItemNum()

SYNOPSIS

void *ListGetItemNum(Linklist *list, int num)

list	pointer to an existing linked list
num	Number of the item desired

RETURNS

Pointer to requested item; NULL if item is not present.

DESCRIPTION

ListGetItemNum() searches for the n+1'th item in a list (n = 0 for first item, etc.). It DOES NOT remove the item from the list. The "current item" is repositioned to the requested item.

ListFreeItemsToStack()

SYNOPSIS

void ListFreeItemsToStack(Linklist *list, STACK_T *free_stack)

list	Pointer to an existing linked list
free_stack	Pointer to list's associated free stack (see Section A.6)

RETURNS

void

DESCRIPTION

ListFreeItemsToStack() removes all items from the list and deposits them in the indicated stack. A list usually has an associated stack for storage of unused list items.

ListPack()

SYNOPSIS

int ListPack(Linklist *list, char *buf, int maxbufbytes, int itemsize)

list	Pointer to an existing linked list
buf	Pointer to destination buffer
maxbufbytes	Size of destination buffer
itemsize	Size of individual list items

RETURNS

Number of items packed.

DESCRIPTION

ListPack() packs a linked list into a buffer so it can be transmitted to other processes. It packs

as many items as will fit in the buffer (this may be less than all on list!) It packs entire items, including the list node header. It DOES NOT remove items from the list.

ListUnpack()

SYNOPSIS

int ListUnpack(Linklist *list, STACK_T *freestack, char *buf, int bufbytes, int itemsize)

list	Pointer to an existing linked list
freestack	Pointer to a stack of free list items
buf	Pointer to a buffer containing the packed items
bufbytes	Size of buf buffer
itemsize	Size of items in buffer buf

RETURNS

Number of items unpacked.

DESCRIPTION

ListUnpack() unpacks an array of items into a linked list. Items are copied into list items popped from the list's associated free stack (the calling function must assure that sufficient items are present on the stack). Items copied contain the list node header. If **buf** was packed using **ListPack()**, this will be true.

The following functions do not use the concept of an implicit list "current item" (current node).

ListGetFirstNode()

SYNOPSIS

void *ListGetFirstNode(Linklist *list)

list	Pointer to an existing linked list

RETURNS

Pointer to first item (node) on the list.

DESCRIPTION

ListGetFirstNode() returns a pointer to the first item in an existing list. It DOES NOT remove the item from the list.

ListGetLastNode()

SYNOPSIS

void *ListGetLastNode(Linklist *list)

list	Pointer to an existing linked list

RETURNS

Pointer to last item (node) on the list.

DESCRIPTION

ListGetLastNode() returns a pointer to the last item in an existing list. It DOES NOT remove the item from the list.

ListGetPrevNode()

SYNOPSIS

void *ListGetPrevNode(void *item)

item   Pointer to reference item

RETURNS

Pointer to list item before reference item; NULL if no previous item is present.

DESCRIPTION

ListGetPrevNode() returns a pointer to the list item previous (before) the one specified by the function argument. It DOES NOT remove the item from the list.

ListGetNextNode()

SYNOPSIS

void *ListGetNextNode(void *item)

item   Pointer to reference item

RETURNS

Pointer to list item before reference item; NULL if no next item is present.

DESCRIPTION

ListGetNextNode() returns a pointer to the list item next (before) the one specified by the function argument. It DOES NOT remove the item from the list.

ListInsertNode()

SYNOPSIS

void ListInsertNode(Linklist *list, void *new_item, void *curr_item, int pos)

list	Pointer to an existing linked list
new_item	Pointer to list item to be inserted
curr_item	Pointer to desired list "current item"
pos	Insertion specification:
INSERT_ITEM_BEFORE	insert before curr_item
INSERT_ITEM_AFTER	insert after curr_item

RETURNS

void; aborts program if **pos** is unrecognized.

DESCRIPTION

ListInsertNode() inserts a new item into an existing list at a point before or after the specified "current item."

ListAppendNode()

SYNOPSIS

void ListAppendNode(Linklist *list, void *item)

list	Pointer to an existing linked list
item	Pointer to item to be appended

RETURNS

void

DESCRIPTION

ListAppendNode() appends a list item to the end of an existing list.

ListDeleteNode()

SYNOPSIS

void *ListDeleteNode(Linklist *list, void *voidNode)

list	Pointer to an existing linked list
voidNode	Pointer to item (node) to be deleted.

RETURNS

Pointer to next item (node) on the list; NULL if the item deleted was the last. Aborts the program if the specified node pointer is NULL.

DESCRIPTION

ListDeleteNode() deletes the specified item from an existing list. Note that it is the calling function's responsibility to return the deleted items to the lists associated free stack.

A.6 Stack Functions

Function summary

STACK_T *StackAlloc()
int StackFill()
STACK_T *MemStackCreate()
STACK_T *MemStackContigCreate()
STACK_T *MemStackContigReset()
int StackPush()
void *StackPop()
int StackSpace()

StackAlloc()

SYNOPSIS

STACK_T *StackAlloc(int max_items)

max_items	Stack capacity

RETURNS

Pointer to the allocated STACK_T structure; NULL if memory allocation failure.

DESCRIPTION

StackAlloc() allocates memory for a STACK_T structure and initializes it. Initialization includes allocating memory for **max_items** pointers to stack entries. Depreciated: use **MemStackCreate()** for new code.

StackFill())

SYNOPSIS

int StackFill(STACK_T *stack, int itemsize)

stack	Pointer to STACK_T structure returned by StackAlloc()
itemsize	Size of item to be stacked

RETURNS

One if successful; NULL if a memory allocation failure.

DESCRIPTION

StackFill() allocates memory for **max_items** stack items, each of size **itemsize**, and attaches them to the stack created by **StackAlloc()**. Depreciated: use **MemStackCreate()** in new code.

MemStackCreate()

SYNOPSIS

STACK_T *MemStackCreate(int memSeg, int nItems, int itemSize)

memSeg	Memory segment to use nItemsStack capacity
itemSize	Size of item to be stacked

RETURNS

Pointer to the allocated STACK_T structure; NULL if memory allocation failure.

DESCRIPTION

MemStackCreate() combines the operations of **StackAlloc()** and **StackFill()**. In addition it allows the calling function to select the memory segment within which the stack will be allocated. This function should be used in place of **StackAlloc()** and **StackFill()** in new code.

MemStackContigCreate()

SYNOPSIS

STACK_T *MemStackContigCreate(int memSeg, int nItems, int itemSize, char **baseAddr)

memSeg	Memory segment to use
nItems	Stack capacity
itemSize	Size of item to be stacked
baseAddr	Address of a pointer to the memory allocated for the contiguous stack items.

RETURNS

Pointer to the allocated STACK_T structure; NULL if a memory allocation failed.

DESCRIPTION

MemStackContigCreate() allocates memory for a stack in a manner similar to **MemStack-Create()**. However, memory allocated for stack items is continuous, and a pointer to that memory is returned in **baseAddr**.

MemStackContigReset()

SYNOPSIS

STACK_T *MemStackContigReset(STACK_T *stack, int nItems, int itemsize, void *baseAddr)

stack	Pointer to an existing stack
nItems	Stack capacity
itemsize	Size of item to be stacked
baseAddr	Address of a pointer to the memory allocated for the contiguous stack items.

RETURNS

NULL (return specification is not used).

DESCRIPTION

MemStackContigReset() re-initializes a stack created by **MemStackContigCreate()**. The stack is left in the "full" state.

StackPush()

SYNOPSIS

int StackPush(STACK_T *stack, void *item)

stack	Pointer to an existing stack.
item	Pointer to an item to be pushed onto the stack.

RETURNS

One if item is successfully pushed; NULL if stack is full.

DESCRIPTION

StackPush() pushes an item onto an existing stack.

StackPop()

SYNOPSIS

void *StackPop(STACK_T *stack)

stack	Pointer to an existing stack

RETURNS

A pointer to the removed stack item if successful; NULL if stack is empty.

DESCRIPTION

StackPop() pops an item from an existing stack.

StackSpace()

SYNOPSIS

int StackSpace(STACK_T *stack)

stack Pointer to an existing stack

RETURNS

The number of empty entries on the stack.

DESCRIPTION

StackSpace() returns the number of item spaces available on an existing stack.

A.7 Register Access Functions

C40 Register Access Functions. See the TI C40 User's Guide for more information regarding these registers:

GetST(), SetST(), OrST()
GetIIE(), SetIIE(), OrIIE()
GetIIF(), SetIIF(), OrIIF()
GetDIE(), SetDIE(), OrDIE()

IntDisable()
IntEnable()
IntRestore()

GetST(), GetIIE(), GetIIF(), GetDIE()

SYNOPSIS

Uns GetST(), Uns GetIIE(), Uns GetIIF(), Uns GetDIE()

RETURNS

The current value of the register.

SetST(), SetIIE(), SetIIF(), SetDIE()

SYNOPSIS

void SetST(Uns val), void SetIIE(Uns val), void SetIIF(Uns val), void SetDIE(Uns val)

val New value for register

OrST(), OrIIE(), OrIIF(), OrDIE()

SYNOPSIS

void OrST(Uns val), void OrIIE(Uns val), void OrIIF(Uns val), void OrDIE(Uns val)

val Value to OR with current value of register

IntDisable(), IntEnable(), IntRestore()

SYNOPSIS

Uns IntDisable(), void IntEnable(), void IntRestore(Uns oldGIE)

oldGIE Old Global Interrupt Enable Setting (0x2000 or 0)

RETURNS

IntDisable() returns the previous state of the GIE bit in the status register

DESCRIPTION

IntDisable() disables interrupts by setting the GIE bit in the status register to 0. The previous value of the GIE bit (0x2000 if interrupts were enabled, or 0 if they were disabled) is returned. **IntRestore()** allows the restoration of the previous GIE value returned by **IntDisable()** **IntEnable()** simply sets the GIE bit in the status register by doing an OR operation.

A.8 Semaphore Functions

Function summary:

SemCreate()
SemPend()
SemPost()

SemCreate()

SYNOPSIS

Sem *SemCreate(int initialCount, SemAttrs *attrs)

initialCount	Initial count for semaphore
attrs	Semaphore attributes

RETURNS

Pointer to an initialized semaphore.

DESCRIPTION

SemCreate() creates a new semaphore and initializes its value to initialCount. If the semaphore is being used for mutual exclusion, **initialCount** is usually specified as 1. When used for synchronizing queue access, the semaphore is usually initialized to the number of items that are initially in the queue.

No semaphore attributes are currently defined at this time, so the second argument should always be specified as NULL.

SemPend()

SYNOPSIS

int SemPend(Sem *sem, int timeout)

sem	Semaphore pointer
timeout	Timeout in system clock ticks

RETURNS

TRUE if successful, FALSE if timeout.

DESCRIPTION

If the semaphore count is greater than zero, **SemPend()** decrements the count by 1 and returns TRUE. Otherwise, the calling task is suspended until another task calls **SemPost()** or a time-out occurs. If the **timeout** argument is 0, the task will remain suspended indefinitely waiting

for the semaphore.

Note: **SemPend()** cannot be called by interrupt handler routines!

SemPost()

SYNOPSIS

SemPost(Sem *sem)

sem Semaphore pointer

DESCRIPTION

SemPost() wakes up the highest priority task waiting on the semaphore. If the awakened task is of higher priority than the current task, a context switch is performed. If **SemPost()** is being called from an interrupt handler, the context switch is delayed until just prior to the return from interrupt, so the entire C interrupt handler will always be completed before the switch. If no task is waiting, **SemPost()** simply increments the semaphore count by 1 and returns.

A.9 Signal Functions

Function summary:

```
Uns     SignalAlloc( ) ;
void    SignalSend();
int     SignalWait();
```

SignalAlloc()

SYNOPSIS

Uns SignalAlloc(Task *task)

task Task handle

RETURNS

Available signal number for the specified task, or 0 if no more signals are available.

DESCRIPTION

SignalAlloc() returns an available signal number for a given task. Signals are managed internally using a 32-bit bitmask, and the signal number returned from this routine ranges from 1<<0 to 1<<31. A zero return value signifies that no more signals are available for the given task.

SignalSend()

SYNOPSIS

Uns SignalSend(Task *task, int sigNum)

task Task handle

sigNum Signal number

DESCRIPTION

SignalSend() Sends a signal to a task. If the task is waiting for the signal, the task is put into the TASK_READY state. If the signalled task is of higher priority than the running task, a

context switch is performed. If called from within an interrupt handler routine, the context switch is not actually performed until just before the return from interrupt occurs, so the entire C interrupt handler is always completed before the context switch.

Note: Current implementation limits an interrupt handler to 1 **SignalSend()** call per interrupt. This is all that is currently required, but if the need arises, the signal code will be have to be modified.

SignalWait()

SYNOPSIS

Uns SignalWait(int sigMask, int timeout)

sigMask Bitmask of signals to wait for

timeout Timeout in clock ticks

RETURNS

The signal number that awoke the task, or 0 if a timeout occurred.

DESCRIPTION

Wait for the occurrence of a signal matching any one of those specified in **sigMask**. This allows a task to wait for any one of several events to occur.

Note: **SignalWait()** cannot be called by interrupt handler routines!

A.10 Task Management Functions

Definition and function summary:

```
#define OS_MINPRI 1
#define OS_MAXPRI 15

typedef struct
{
 int  priority ;      /* Default = 4 */
 int  stacksize ;     /* Default = 8192 */
 int  stackseg ;      /* Default = PAC_DRAM */
 char name[32] ;      /* Default = 'noname' */
} TaskAttrs ;

Task  *TaskCreate( ) ;

int    TaskGetId( ) ;
int    TaskGetPri( ) ;
Task  *TaskSelf( ) ;
void   TaskSetPri( ) ;
void   TaskSleep( ) ;
void   TaskYield( );
```

TaskCreate()

SYNOPSIS

Task *TaskCreate(Fxn entry, TaskAttrs *attrs, void *arg1, void *arg2, void *arg3, void *arg4)

entry	Entry point (Function address) for newly created task
attrs	Pointer to TaskAttrs structure containing the task initialization information
arg1-arg4	Arguments to be passed to new task entry point function

RETURNS

Task handle for new task.

TaskGetId(), TaskGetPri()

SYNOPSIS

int TaskGetId(Task *task)

int TaskGetPri(Task *task)

task	Task handle

RETURNS

An integer value representing the Task Id (0 to (OS_MAX_TASKS-1)) or the task priority.

TaskSelf()

SYNOPSIS

Task *TaskSelf( ) ;

RETURNS

The task handle for the current task.

TaskSetPri()

SYNOPSIS

void TaskSetPri(Task *task, int newPri)

task	Task handle
newPri	New priority value

DESCRIPTION

TaskSetPri() is used to dynamically adjust the priority of a task. It is most often used to adjust a task's own priority, using '**TaskSetPri(TaskSelf(), newPri)**'

TaskSleep()

SYNOPSIS

void TaskSleep(int sleepTicks)

sleepTicks	Number of clock ticks to sleep.

DESCRIPTION

TaskSleep() puts the task to sleep for **sleepTicks** timer ticks. The timer on the 9PAC is currently being run at 1000 Hz, so a **TaskSleep(1000)** will sleep for 1 second.

TaskYield()

SYNOPSIS

```
void   TaskYield( ) ;
```

DESCRIPTION

TaskYield() yields to another task of equal priority, if there is such a task in the TASK_READY state. This allows round-robin type scheduling to be easily implemented.

APPENDIX B
9PAC MEMORY MAP

Base Address	Top Address	Unique Size	Comments
Local Bus:			
0x00000000	0x002FFFFF	0x00300000	C44 internal ROM, registers, etc.
0x00300000	0x003FFFFF	0x00080000	EPROM (512K bytes)
0x00400000	0x004007FF	0x00000800	Flash Card Common Memory
0x00400800	0x0040FFFF		Empty
0x00410000	0x004107FF	0x00000800	Flash Card Attribute Memory
0x00410800	0x004FFFFF		Empty
0x00500000	0x0050FFFF	0x00000080	Serial #1 CPU access
0x00506000	0x00506FFF	0x00000080	Serial #1 channel B DMA access
0x00510000	0x0051FFFF	0x00000080	Serial #2 CPU access
0x00514000	0x00515FFF	0x00000080	Serial #2 channel A DMA access
0x00516000	0x00516FFF	0x00000080	Serial #2 channel B DMA access
0x00520000	0x00520FFF	0x00000001	Peripheral interrupts CSR
0x00521000	0x00521FFF	0x00000001	External DMA requester control
0x00522000	0x00522FFF	0x00000001	Flash memory CSR
0x00523000	0x00523FFF	0x00000001	Hardware version number/MPRAM fault
0x00600000	0x0061FFFF	0x00010000	Multi-port RAM (HSIB/ASP)
0x00620000	0x0062FFFF	0x00000001	Last HSIB write address #1 (Radar)
0x00630000	0x0063FFFF	0x00000001	Last HSIB write address #2 (Beacon)
0x00640000	0x0064FFFF	0x00010000	External dual port RAM (MIP)
0x00650000	0x0065FFA9		Empty
0x0065FFAA	0x0065FFAF	0x00000006	HSIB control/status registers
0x0065FFB0	0x007FFFFF		Empty
0x00800000	0x00FFFFFF	0x00040000	Zero wait state static RAM

Base Address	Top Address	Unique Size	Comments
0x01000000	0x7FFFFFFF		Empty
Global Bus:			
0x80000000	0x803FFFFF	0x00400000	Dynamic RAM bank 0
0x80400000	0x807FFFFF	0x00400000	Node 2 - Dynamic Ram bank 1 Node 1 and 3 - Empty
0x80800000	0xFFFFFFFF		Empty

GLOSSARY

9PAC	ASR-9 Processor Augmentation Card
ASCII	American Standard Code for Information Interchange
ASP	Array Signal Processor
ASR	Airport Surveillance Radar
ATA	Advanced Technology Attachment
BIT	Build-In Test
BTD	Beacon Target Detector
C&I	Correlation and Interpolation
CPU	Central Processing Unit
DMA	Direct Memory Access
DOS	Disk Operating System
DRAM	Dynamic RAM
DSP	Digital Signal Processor
EPROM	Electronically Programmable Read-Only Memory
FAT	File Allocation Table
FFS	Flash File System
FIFO	First In, First Out
JTAG	Joint Test Action Group
HDLC	High-level Data Link Control
HSIB	High-Speed Interface Buffer
I/O	Input/Output
ITC	InterTask Communication
PAC_CRAM	C44 on-chip RAM
RAM	Random Access Memory
MB	Megabyte
MIP	Message Interface Processor
MRG	radar/beacon Merge
MUX	Multiplexer
NMI	Non-Maskable Interrupt
PCMCIA	Personal Computer Memory Card International Association
PROM	Programmable Read-Only Memory
RMS	Remote Monitoring System
SRAM	Static RAM
TRK	scan-to-scan correlation
TTY	Teletype
VSP	Variable Signal Processor

REFERENCES

Pieronek, J.V., "The ASR-9 Processor Augmentation Card (9-PAC)," MIT Lincoln Laboratory Project Report ATC-232, October 1995.