ARTISAN®
TECHNOLOGY GROUP

Your **definitive** source for quality pre-owned equipment.

Artisan Technology Group
(217) 352-9330 | sales@artisantg.com | artisantg.com

Full-service, independent repair center
with experienced engineers and technicians on staff.

We buy your excess, underutilized, and idle equipment
along with credit for buybacks and trade-ins.

Custom engineering
so your equipment works exactly as you specify.

- Critical and expedited services
- In stock / Ready-to-ship
- Leasing / Rentals / Demos
- ITAR-certified secure asset solutions

Expert team | Trust guarantee | 100% satisfaction

All trademarks, brand names, and brands appearing herein are the property of their respective owners.

Find the *Intel iSBC CSM/001* at our website: ***Click HERE***

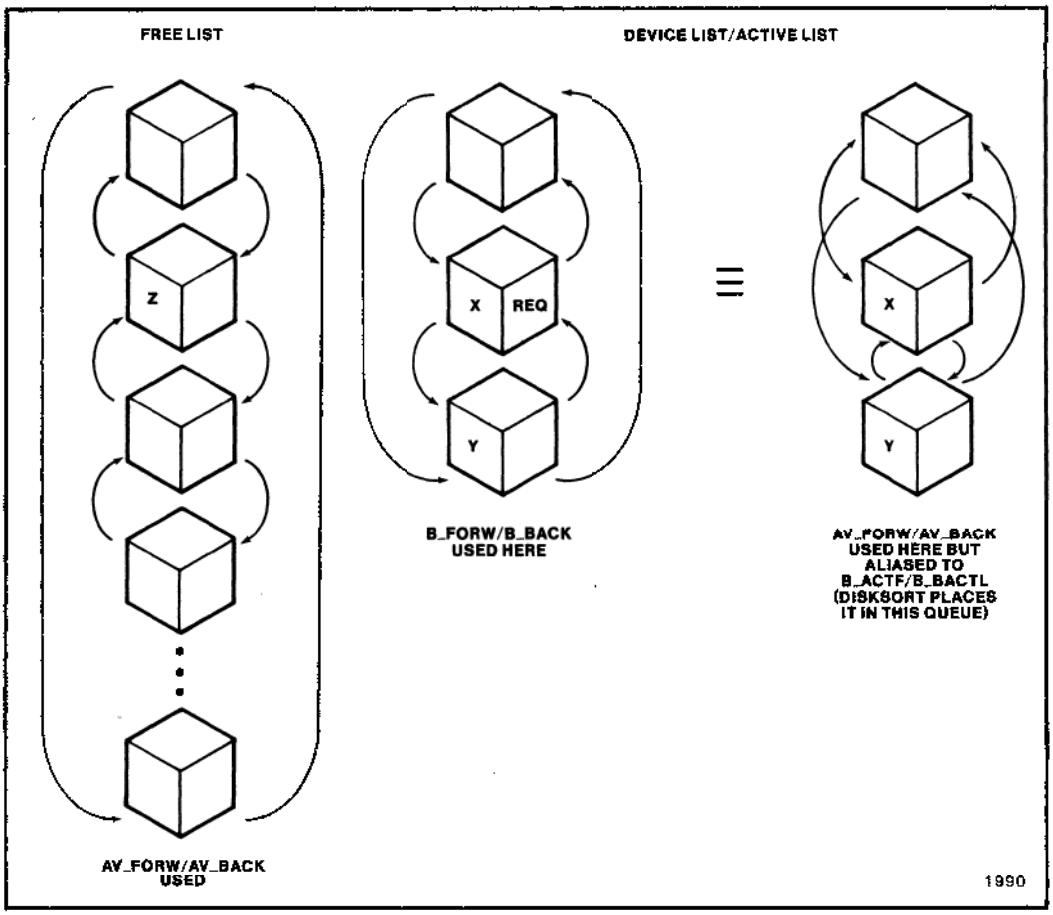


Figure 7a. Another Request Placed in Device List and also Placed in Active List by Driver

of the driver may be calling **disksort()** that orders the device list or when the kernel is placing the buffer header with a request onto the same list.

On completion, the buffer header is marked as IO complete and this header is released from the device specific queue and placed at the end of the freelist using the av_pointers. Note that the link to the active list is broken (reusing the av_pointers). Remember also, that **b_forw/b_back** pointers are still maintaining membership with the device specific list even when the buffer is now a member of the freelist pool. Here is where the buffer is placed in cache.

As shown in figure 7b,the buffer request X is placed on the freelist using the av_pointers but is still a leading member of the device list. On the next request, the kernel will search the device queue (not the active list) from the beginning. As you can see in figure 7b, the list is diverted into another list of available but recently used buffers by following **b_forw/b_back** pointers. Here cacheing occurs as most recent transactions can be checked for repetition. Note that the X buffer will bubble up the freelist until it will be re-used for other transactions. Until then it is cached.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

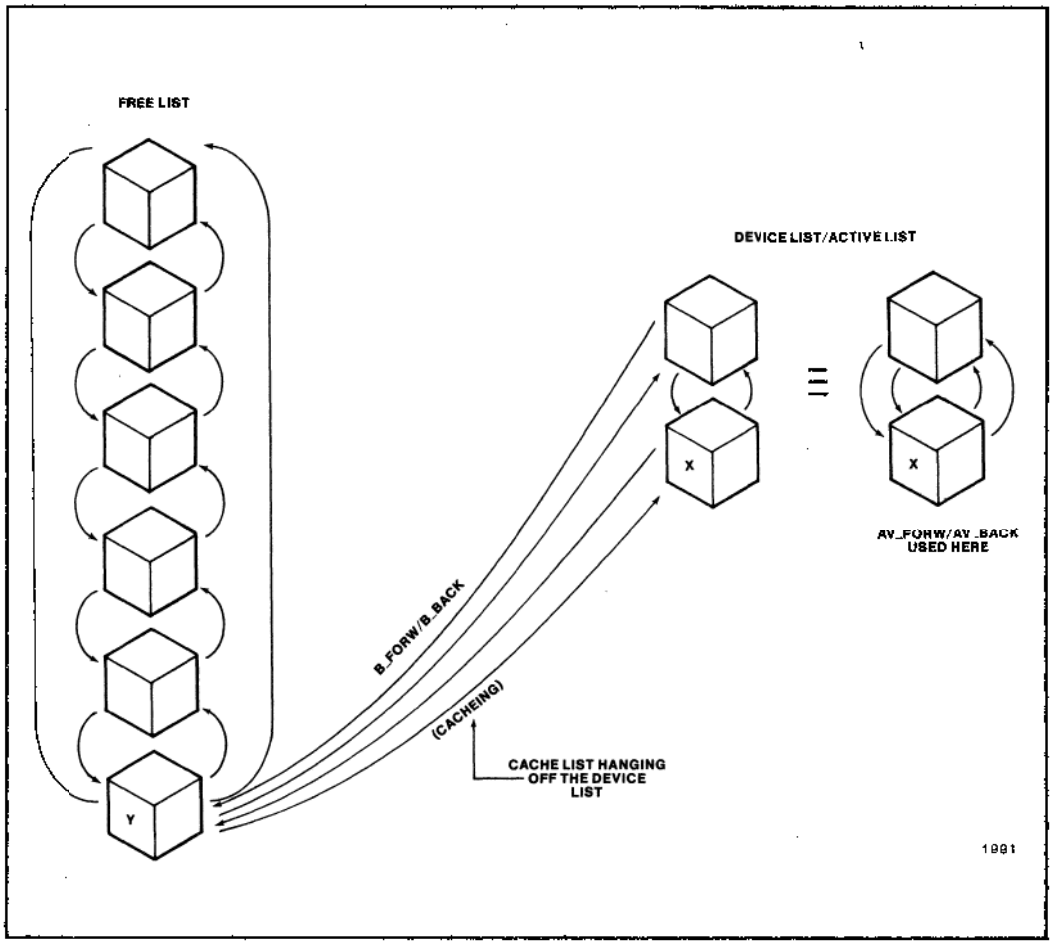


Figure 7b. When Request Granted the Buffer is Returned to Free List but is Still a Member of the Device List and is cached. No Longer in Active List.

For a review of the block drivers duties, in an I/O request, consider a read request and the following point-by-point discussion:

```
user calls read
READ
{
    . maps file fd to inode
    . calls readi
readi (read inode)
    . determines block to be read (bmap()
      called)
    . searches buffer list i.e follows av_pointers
      matching block # to last read/write
    . if cached, copy into user space and return
    . if not cached, flush first buffer from freelist
        .fill out buffer header
            . with device #
            . block #
        . call task-time portion of device driver
        . on return ,
        . sleep on buffer header waiting for
          wakeup
        . copy into user space
}
The device driver does
{
issues or queues request to the device
interrupt handler wakes processes when on I/O
  complete
when awakened return to readi
}
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

NOTE: the above is only a specific example and a brief one. For example, details of the support routines checking whether it is block/character read was not mentioned.

So far, elements of the I/O system anatomy discussed are invisible to the device driver writer. He/She need not know all of the intricacies of buffer management but there is a need to fully comprehend the routines to be written, the system calls available and the operating environment of device drivers. In accordance with this methodology, block drivers have, in their grasp, many powerful and consistent system calls available from the kernel. They can be called "driver support routines" because some of them are available to character drivers also.

7.1.2 DRIVER SUPPORT ROUTINES

The following list is an informal collection of possible support routines used in block I/O drivers and by the kernel. The kernel deals with the declarations for the arguments and on many situations, places values into these arguments. This is because the kernel is allowed access to most of these arguments and knows their values. Some of these calls are also used in character I/O transfers and will be referenced in the character interface description:

physio(strat,bp,dev,rw)

where **strat** is the address of the strategy routine(a driver procedure) which is the routine that performs read, writes and starts up the device. This routine will be discussed in section 7.1.3. It takes, as an argument, a pointer to a buffer-header (bp) which holds detailed information about the transfer.

where **dev** is the relevant device that character I/O is to occur to (the <major,minor>) pair.

where **rw** is a flag indicating the nature of the transfer(B_READ, B_WRITE in param.h)

Physio is used by block devices which can be treated as character devices. In this case, transfer between user space and device space is done directly(direct I/O) with no intervening buffers. Remember that for this to be successful, physical I/O must occur when the instigating user process is in memory and active (and not swapped out). **physio** is a routine called by a driver for physical I/O on a device. Among other functions, **physio** checks the validity of the transfer request. The buffer header pointer that is passed to **physio** does not hold a buffer address but the address of the physical location in memory or the device, depending on the direction of the transfer. Physical I/O is a contiguous transfer feature that is used in **tar**, **fsck** and **dd**, among other utilities.

disksort(&xxxtab, bp);

```
struct buff xxxtab /* static buffer header */
struct buf *bp /* new buff header to be inserted */
```

disksort() is the assist routine ingredient to the buffering/cacheing protocol as it manages the active request queue. It takes, as arguments, the address of the pointer to the static buffer header for the specific device. The active request queue holds all requests for the device. **bp** is the pointer to the new buffer header that holds another request on the device. **disksort()** inserts this request in the queue of requests in cylinder/block order to minimize disk accesses.

iodone(bp)

```
struct buf *bp /* header of completed request */
```

is a clean-up routine that informs the process that the request made is complete. **iodone()** is called by an interrupt service routine and issues a **wakeup()** on the relevant event i.e. the buffer pointer **bp**. The routine pulls the request off the device specific queue and places it onto the free list.

sleep(bp,pri), wakeup(bp) and iowait(bp)

```
struct buf *bp
```

Process/task synchronization is a required feature in the multi-user/multi-tasking XENIX Operating System. Processes have to be informed when to wait for the system's shared resources. The XENIX kernel provides two routines, **sleep()/wakeup()**, for this purpose. **Sleep()** takes, as argument, a key or event that the calling process waits on. This key or event is nothing more that a bit-pattern. The key or event in this case is conveniently the buffer-header pointer that the task-time portion of a driver is using for the transfer request. To re-iterate, the task-time portion of the driver, when making an I/O request, may have to wait after the request is made until the actual I/O is completed. The waiting is begun by a system routine called **iowait()** which is called by the kernel and **physio()** which is called by a driver in direct physical I/O (for magnetic tape drivers when driver calls strat() and waits for I/O to complete).

PRI, the second argument, is the priority at which the process is to sleep. The sleep priority is higher than what a user process can acquire. When the **wakeup()** occurs, the process continues at the sleep priority thus giving it a higher probability of being scheduled earlier.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Priorities range from 0 to 127. Priorities are not bound by rules but the priority PZERO is used to differentiate two main situations that may occur. If a priority < PZERO is set for the sleep, no signal can wake up the process. Hence, the process will be awakened with an **iodone()** in the future. With a priority > PZERO, signals will awaken the process even before **iodone()** Also, smaller numerical priorities mean higher priority levels. The safer technique is therefore to place 'sleep' in a loop that tests if the buffer is available for continuation. Hence, if I/O is complete and the buffer freed, the process is awakened legally. Otherwise, continue to sleep.

Also, when the device returns an interrupt **iodone()** is called which calls **wakeup()** that sets the event(buffer header) and induces life to all processes waiting on that event — not just the "first" one. Processes must therefore ensure that they are awake for the correct reason. One way to do this is for the task-time portion to check a predetermined static memory location for instructions left by the interrupt-time portion of the driver on completion.

timeout(func,arg,time)

```
int (*func)();    /* function called as argument */
int arg;
int time;
```

Arranges for **func** to be called with argument **arg** in **time** clock-ticks. **timeout()** is a facility that runs a procedure after n clockticks. The procedure is called at clock interrupt time and ,hence, conforms to the interrupt-time rules. Used for character I/O also.

iomove(addr, count, flag)

Used for large data transfers. **addr** gives you information on where in kernel the transfer is to occur. **count** signifies the size of the transfer in bytes. **Flag** tells us if it is a B_READ/B_WRITE. The other transfer address is found in the processes' **u_structure** as **u.u_base.**

7.1.3 BLOCK DEVICE DRIVER ROUTINES

Briefly, a block device driver is composed of one or more of the following routines:

init() is a routine called very early during system initialization (at boot time) to initialize the device. It is called with no parameters and returns no values. Interrupts are disabled at this time and the existence of the device is verified. It prints appropriate messages stating that the device is/is not found and remembers if the device is alive (sets a flag). This routine is called once.

open() is a routine that opens the device. Prepare it for activity and is called on every open of the device.

```
Its parameters are
  dev_t dev;
  int flag;
```

open(dev,flag) is the calling sequence where **dev** is <major,minor> device number and **flag** is either B_READ or B_WRITE. The program validates the device number and sets-up initial parameters. Any errors detected is recorded in the u.u_error. The presence of the device is verified before the open occurs.

close(dev)

is called on the final close of the device. The close routine flushes pending transfers in device specific queue and sets flags that remember that the device is closed.

strat(bp)

```
struct buf *bp; /* pointer to buffer header */
```

Called by the kernel in response to the user program instigating a read/write request. **strat** is "strategy." Inserts a request on the queue of device requests. The kernel provides a buffer header to the routine and it validates the header to ensure that it has all the necessary information (e.g B_READ). The driver routine calls **start() and disksort()**

intr(level)

```
int level;
```

The interrupt routine is called by the kernel when the device interrupts. This routine is called when the device is moving from an active state to an idle state. If the device is active on entry to the interrupt service routine, the interrupt service routine was awakened by a spurious interrupt. If not, the device state is changed to idle. In this state, the previous request was satisfied and an iodone should be called. The momentun is continued to keep the device busy by calling the **start()** routine if other requests are pending on the request queue (device specific queue).

start()

This routine functions to move the device from an idle state to an active state i.e. it talks with device. It is called when the device is idle or a request is queued. It interprets the information on the buffer header at the beginning of the queue of device requests and sends commands to the controller.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

These routines form a file called ixxx.c where xxx is the numerical representation of the device (see Naming Conventions - Appendix G). This file should reside in **/sys/io** directory. Conventionally, **cxxx.c** an adjoining file is also created to identify any data structure relevant for the main program. **cxxx.c** resides in the **/sys/cfg** directory. Finally, constants and **#defines** are found in a header file, generally "included" in **cxxx.c**, called **ixxx.h**. This file is created in **/sys/h**. Hence, a driver for the iSBC 254 Bubble Memory board should be composed of:

i254.c main driver routines
i254.h the header file
c254.c the configuration data structures

With this brief description of the driver routines, a casual discussion of how these routines interact with each other and the kernel is a natural follow-up. Some reiteration of previous details is necessary to give an overall consistent discussion.

7.1.4 REVIEW

User requests to be performed on a device (usually on a file living on the device) are converted by the kernel to simple requests for I/O which are passed to the driver. The kernel does any blocking/deblocking and cacheing to minimize device accesses. **Strat()** and **Intr()** are the main routines required of a block device driver. The request which is passed to the **strat()** routine is passed in the form of a pointer to a buffer header. This header contains all the information necessary to perform the operation - B_READ, B_WRITE, device address to use e.g. which track and sector, and the address of the kernel buffer from which the data should be taken or into which the data should be placed. The buffer header points to BSIZE'd buffers and the request will always be for BSIZE operations.

Be sure to keep in perspective the level of software being discussed - the driver itself sees only requests for transfers to or from a physical block of the device - entities like file-structure, disk space allocation, or blocking/deblocking of small or large requests are all performed by higher level kernel software. All the device driver needs to do is examine a request, determine whether it is a read or write and perform the operation between the indicated memory address and the indicated block device.

7.1.5 STEPS TAKEN TO SATISFY REQUESTS

This discussion centers around the **strat()** and **intr()** routines. All requests are passed to the driver by the kernel by calling the **strat()** routine, with a single parameter - a pointer to a buffer header. As before, the header specifies the type of operation that is to be performed, the memory and device addresses to be used, and a field for recording the result of the operation after it has completed. The **strat()** routine places the incoming request on the linked list of active requests to be performed. If the device is currently busy performing a previously queued request, the **strat()** routine has finished its job and returns. If the device is idle (i.e. the request is the only one on the active list), the **strat()** routine must initialize the operation for the request. This typically involves loading parameters into a peripheral controller and initiating a command. At this point, the **strat()** routine has completed and returns.

After a command is started, it is typically a long-time (by cpu standards) until the request is completed and an interrupt occurs. The interrupt routine must field this interrupt and determine the reason for it. If it is the expected "operation-complete" interrupt, the interrupt routine should perform any operation needed to complete the transfer, then call the **iodone()** routine with a single parameter — the pointer to the buffer header of the request just completed. The **iodone()** performs some clean-up, notably waking up the process which was waiting for the I/O to complete. At this point, the interrupt routine may determine if other requests are waiting in the active request queue for the device, and if so, initiate the next one by calling the **start()** routine. Once done, the interrupt routine returns with its job done. The interrupt routine is a trigger that keeps firing-up new requests as they are discovered on the queue. Once the list is exhausted, the **intr()** routine returns without starting another request (none there to start) and the seed is lost and the sequence stops. The **strat()** routine must start another request to "prime the pump" and start the momentum again.

With this understanding of driver routines, a pseudocode example of the iSBC 254 Bubble Memory board driver will complete the discussion of block I/O device drivers. As mentioned, this section of block I/O is not the main thrust of the application note as emphasis has been placed on the character interface. This discussion will culminate in a pseudocode walkthrough.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

7.1.6 iSBC® 254 BUBBLE MEMORY BOARD WALKTHROUGH

```
 1 /*
 2 * SBC 254 Bubble Memory board device driver. (Pseudo-code)
 3 *
 4 * - implements block and raw interfaces for an SBC 254 -1, -2, or -4.
 5 * - always accesses all bubbles in parallel, meaning that there are
 6 * always 2048 pages on the board, and the page size can be 64, 128,
 7 * or 256 bytes (see c254.c)
 8 * - will handle only one 254
 9 * - uses DMA mode for bubble accesses
10 * - I/O base address and number of bubbles configurable in c254.c
11 *
12 *
13 */
14
15 #include "../h/param.h"
16 #include "../h/systm.h"
17 #include "../h/buf.h"
18 #include "../h/conf.h"
19 #include "../h/dir.h"
20 #include "../h/user.h"
21 #include "../h/i254.h"
22
23
24 extern      struct      i254cfg     i254cfg;/* see c254.c
25                          * for values, i254.h for definition
26                         */
27 struct      buf   i254tab;          /* static buffer header */
28 struct      buf   i254rbuf;         /* static buffer header for
29                                     rawinterface */
30 short i254alive, i254isopen;        /* device existence, open flags */
31
32 /*
33 * i254init - called early in the system initialization - probes for
34 *            254 by resetting it and watching for appropriate reaction
35 */
36 i254init()
37 {
38
39     /*
40      * this is the first routine of the driver that will be called,
41      * it's a good time to clear the i254isopen flag
42      */
43
44      i254isopen = 0;     /* 254 is closed */
45
46     /*
47      * more stuff to init the board and check status
48      */
49 }
50
51 /*
52 * i254open - checks for correct minor number(0),and existence of the
53 *            board, and either allows or disallows the open
54 */
55
56 i254open(device, flag)
57 dev_t  device;          /* device number */
58 int    flag;          /* what kind of open (for reading, writing, etc.)
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
59          we'll ignore this */
60 {
61      if ((minor(dev) == 0) && (i254alive)) {
62              i254isopen = 1;         /* mark 254 as open */
63              return;
64          }
65          else{
66              u.u_error = ENXIO;
67              return;
68          }
69 }
70
71 /*
72 *i254strat - queues the I/O request and starts it if the device is idle
73 */
74 i254strat(bp)
75 struct      buf   *bp;
76
77 {
78      int     x, startpage, numpages, ppb;
79
80      /*
81      * first thing to do is check device is open; otherwise, allow
82      * no I/O
83      */
84
85      if (~ i254isopen) {
86              bp->b_flags |= B_ERROR;
87              bp->b_error = ENXIO;
88              iodone(bp);             /* mark it done */
89              return;
90      }
91
92 /*
93 * convert the block number to a page number, and the number of
94 * blocks to number of pages,and the starting block to the starting
95 * page - these could be sped up with some appropriate shifts instead
96 * of * and / */
97 */
98
99      ppb = BSIZE / i254cfg.c_page_size;    /* pages per block */
100         numpages = bp->b_bcount * ppb;          * number of pages */
101         startpage = bp->b_blkno * ppb;      /* 1st page of transfer */
102
103  /*
104   * Now check the requested operation for validity in terms of the
105   * the number of pages on the device.
106   * Here's the thinking:
107   * if request is READ on 1st block after the last block -> EOF
108   * " " " WRITE " " " " " " -> error
109   * " " " READ or WRITE 2 or more pages past the end -> error
110   * if request starts on valid page, but runs off end -> EOF
111   */
112
113         if (startpage > BUBPAGES) {
114                 /* 2 or more after last page, so error */
115                 bp->b_flags |= B_ERROR;
116                 bp->b_flags = ENXIO;
117                 iodone(bp);
118                 return;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
119         }
120         if (startpage == BUBPAGES) {
121                 /* 1st block after last one */
122                 if (bp->b_flags & B_READ)
123                         bp->b_resid = bp->b_bcount; /* read, so just EOF */
124                 else {
125                         * write, so error */
126                         bp->b_flags |= B_ERROR;
127                         bp->b_error = ENXIO;
128                 }
129                 iodone(bp);
130                 return;
131         }
132         if ((startpage + numpages) > (BUBPAGES -1)) {
133                 /* starting ok, but running off end */
134                 bp->b_resid = bp->b_bcount;
135                 iodone(bp);
136                 return;
137         }
138
139         /* if we're here, request looks OK, so queue it and
140          * (if necessary) start it
141          */
142
143         bp->b_cylin = startpage; /* use page number as sort field */
144
145         /*
146          * Since we're about to play with the queue, and this can be
147          * accessed at any time via the interrupt handler, we need to
148          * shut down this interrupt level to provide mutex
149          */
150
151         x = SPL();
152         disksort(&i254tab, bp);
153         if (i254tab.b_active == IO_IDLE) i254start(bp); /* start if idle */
154         splx(x); /* reenable this interrupt level */
155 }
156 i254start(bp)
157 struct buf *bp;
158
159 /* this routine is the most device specific of all others.
160    The routine is called by both strat() and intr() at task-time
161    and interrupt time. It starts up the device if it is idle and
162    keeps the momentum going with other requests.
163  */
164
165 /*
166  *i254intr - interrupt handler-checks status of operation just completed
167  * and starts new operation if one is queued
168  */
169 i254intr(level)
170 int level;
171 {
172         short stat;
173
174         /*
175          * point to first buffer header in the active queue, making sure
176          * that it's actually pointing to a buffer header
177          */
178
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
179         if ((bp = i254tab.b_actf) == NULL) {
180                 printf("No active buffer header, i254intr, level %d0,
181                         level);
182                 return;
183         }
184
185         /*
186          * clear the interrupt source and disable DMA
187          */
188
189         outb(i254cfg.c_base_port + BMCCMD, CLRINT);
190         outb(i254cfg.c_base_port + DMAMODE, DMADIS);
191
192         /*
193          * Now look at the status of the BMC to determine if this is the
194          * successful end of an operation. Report any errors encountered
195          */
196
197         stat = inb(i254cfg.c_base_port + STATUSPORT);
198         if (stat & BMCBUSY) {
199                 printf("BMC still busy, i254intr, status = %d0, stat);
200                 return;
201         }
202         if (stat & (BMCOPFAIL|BMCTIMERR|BMCUNCERR)) {
203 #ifdef VERBOSE
204                 printf("Error, i254intr, status = %d0, stat);
205 #endif
206                 /* call deverror here? */
207                 bp->b_flags|= B_ERROR;
208                 bp->b_error = EIO;
209         }
210         else if (stat & BMCCORERR)
211                 printf("Cor error, i254intr, status = %d0, stat);
212
213 /*
214  * At this point we have determined that a legitimate bubble interrupt
215  * has occurred, and if there has been an error, it's recorded. Now
216  * we need to mark the operation as complete and start the next request
217  * in the queue (if there's one there).
218  */
219
220         i254tab.b_actf = bp->av_forw;
221         iodone(bp);
222         if ((bp = i254tab.b_actf) == NULL) {
223                 i254tab.b_active = IO_IDLE;
224                 return;
225         }
226
227         /*
228          * At this point, bp is pointing to the next request in the
229          * queue, so start it.
230          */
231
232         i254start(bp);
233 }
234
235 /*
236  * i254close - clears i254isopen flag
237  */
238
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
239   i254close(device)
240   dev_t device;
241   {
242         i254isopen = 0;
243   }
244
245   /*
246    * i254read - RAW interface read routine - calls physio
247    */
248
249   i254read(device)
250   dev_t device;
251   {
252         physio(i254strat, &i254rbuf, device, B_READ);
253   }
254
255   i254write(device)
256   dev_t device;
257   {
258         physio(i254strat, &i254rbuf, device, B_WRITE);
259 }
```

The **i254init()** routine (line 36) reports on the devices it expects to find and the ones it actually finds. The normal procedure is to give the device some command (usually a reset or initialization), then do a busy wait loop, waiting for a "sign-of-life" from the device. of course, the busy wait loop should have some other form of termination, so that a non-responsive device does not cause the system to hang at the point. Usually, a simple counter to limit the time spent in the wait loop is used. The **i254init()** routine (line 36) writes a message to the console, such as "SBC 254, port 0x040 found" or "SBC 254, port 0x040 NOT found." Also, it is usual to let the **i254init()** routine set a flag which indicates the presence or absence of the device. The **i254open()** routine (line 56) then checks this flag and will return an error if an open is attempted on a non-existent device.

The **i254open()** routine (line 56) is called upon each open on a device and provides an opportunity for doing any device set-up that is required. This could be the routine that spins-up a Winchester disk, or turns on the motor of a mini-floppy. As before, the **i254open()** routine should check for device existence flag, set by the **i254init()** routine, and return an error if necessary.

The **i254strat()** routine (line 72) is the routine which actually performs the bulk of the work in any block device driver. The **strat()** routine queues the new request onto the active queue (of the device specific queue). This queue is a linked list of buffer headers which contains data transfer parameters and are awaiting service. At the head of this list is a static buffer header that is called ixxxtab, where xxx is the device handler prefix that is used on all the procedures composing the device driver (see Naming Conventions - Appendix G) In this case "xxx" is "254." This header contains pointers which make forward and backward links to other buffer headers in the list (remember **b_forw/b_back**). Besides the list of pending device requests, the buffer headers can also be linked onto a free-list, which contains all the buffer headers which are not set-up for an operation using **av_forw/av_back**. A third list contains the buffer headers that are currently available or have been used(cache list). This list is the cache mechanism which the kernel uses by looking down to see if perhaps it can re-use the data from another request and avoid physically accessing the device again. It is managed with the **av_forw/av_back** pointers redeclared. Details of these buffers and how they are managed are found in Block I/O Interface (Section 7.1).

The actual manipulation of the device list and the free list is handled completely by the kernel with its higher level routines and does not need to be the concern of the device driver. The active list (device specific list with av_pointers) is managed by the device driver, but usually can be handled by a couple of provided subroutines, so that the driver does not have to explicitly deal with the pointers which structure the list. **Disksort()** is the routine normally used to place a buffer header onto the active list. Remember the active list is a reordered version of the device list. Besides the actual mechanics of updating the list, **disksort()** also provides a sorting facility which orders the active requests by cylinder number, to minimize seek time between requests. Since the **disksort()** routine is general in nature, and does not know of the specific physical characteristics of particular devices, its algorithm for sorting is to simply order requests based on the cylinder field of the

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

buffer header. By filling in this field with an appropriate number **i254strat()** can effectively implement sorting routines which correspond to the device being handled. For example, on a bubble memory board, there is no such thing as a cylinder. By ordering requests based on the page numbers, seek time can be minimized between successive bubble accesses. So, the **i254strat()** routine may just drop the page number into the cylinder field, and let **disksort()** do the rest. If a more complex algorithm is desired, the **i254strat()** routine can implement it and handle the insertion onto the active list itself, bypassing use of **disksort()** completely. When a transfer is complete, the buffer header is removed from the active list (**av_forw/av_back** removed) by the routine **iodone()**. This is typically called by the **i254intr()** routine (line 169).

After **i254strat()** has queued a request onto the active list, it must check and see if the device is currently in the process of performing an operation. If so, the **1254intr()** routine will start pending requests, and **i254strat()**'s work is done. If the device is not busy, **i254strat()** must start the operation to satisfy the request. Since the responsibility of starting an operation rests with both **i254strat()** and **1254intr()** routines, depending on the circumstances, the code used to start a request is usually placed into a procedure call **i254start()** (line 156), which is then called by **i254strat()/1254intr()**.

The **i254start()** routine is typically the most device specific routine in the set composing the device driver. It examines the next request on the active lists, sets-up the operation accordingly and starts it. Thus **i254start()** is the routine which does output to I/O ports and is generally cognizant of the details of performing a function with the device. Once **i254start()** is called to initiate a transfer, **i254strat()** is complete and returns.

The **1254intr()** routine (line 88) is called by the kernel when an interrupt from the device occurs. Details of manipulating the interrupt controller have already been handled by the kernel, and the only thing **1254intr()** needs to be concerned about is handling the device itself. The interrupt typically occurs to indicate that the transfer is complete. Once **1254intr()** has determineed that this is the case, it should do anything necessary to finish up the operation as far as the device is concerned, set the status field in the buffer header to indicate successful completion of the operation, and then call **iodone()** (line 221) to finish up as far as the kernel is concerned. **iodone()** performs several functions, among them, awakening the process which was waiting for this operation to complete, removing the buffer header from the active list and placing the buffer header onto the free list. If the interrupt occurred as a result of the operation terminating unsuccessfully, the status field (**u.u_error**) should be set and **iodone()** should be called. Once done, **1254intr()** must then check and see if there are other requests waiting in the active queue. If so, the **i254start()** routine should be called to start the next request.

Obviously, **i254strat()/1254intr()** manipulate common data structures. Because the **i254strat()** routine can be interrupted at any time by the occurrence of an interrupt (and the subsequent execution of the interrupt handler), care must be taken such that the interrupt routine does not "clobber" something that **i254strat()** was in the process of doing(e.g. like adding a request to the queue, checking to see if the device is active etc.). The only mechanism available to achieve this mutual exclusion(mutex) is to shut down the interrupt level used by the device while **i254strat()** is in the critical region(s) of code. The **slpN** routines are used to do this (see Interrupt Mapping for details - Appendix D).

The **i254close()** routine (line 239) is called only once by the kernel This routine should also clean-up(e.g. turn off a minifloppy motor) for whatever is appropriate for the device.

Finally, the **i254start()** (line 156) routine is called by both the interrupt routine and the strategy routine i.e. at task-time and interrupt-time. As described before, the routine checks to see if more requests are on the device queue and outputs the appropriate commands to the ports.

7.1.7 FINAL LOOK AT BLOCK DRIVERS

Note that the above discussion on block I/O devices and interfaces is as detailed a discussion as possible given the fact that the next step is to look at existing code and work from it. Device drivers are seldom written from scratch. They are usually based on existing examples. However, there will be problems if every detail about the device is not understood. Knowing the device and its intricacies is of utmost importance to developing a device driver for it.

Kernel overhead is high for block I/O devices i.e. the kernel works hard for these drivers. But not all of driverideology can be viewed through such structured notions as buffers, blocks and the like. Drivers have an unstructured side to them. This side is the character I/O interface.

7.1.8 THE RAW (CHARACTER INTERFACE) TO A BLOCK DEVICE

A block I/O device like a disk can have a character I/O interface. Note that a new set of routines, namely **ixxxread(), ixxxwrite() and ixxxioctl() etc.** have to be created for a block device. These routines implement the character interface for a block device driver. This character interface permits a block device, like a disk, to have direct I/O or byte I/O transfer capability. The data structure **cdevsw[ ]**

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

in c.c shows that the disk has both character and block interfaces. Notice that certain locations unused in **cdevswll** are either titled **nulldev/nodev. Nodev** is a macro that implies a position in the array that is invalid i.e. printers cannot read. **Nulldev** is nothing more than a no-op, that implies it is legal but not implemented.

Moving away from these details, the respective character I/O reads and writes merely call **physio()** (see Device Driver Support Routines Section 7.1.2). With this call,

1) the user process is locked in memory
2) **strat()** is called with the buffer header already updated with the transfer information. However, the address of the transfer is not a buffer header address but a physical location.
3) **iowait()** is called which will wait for I/O completion.

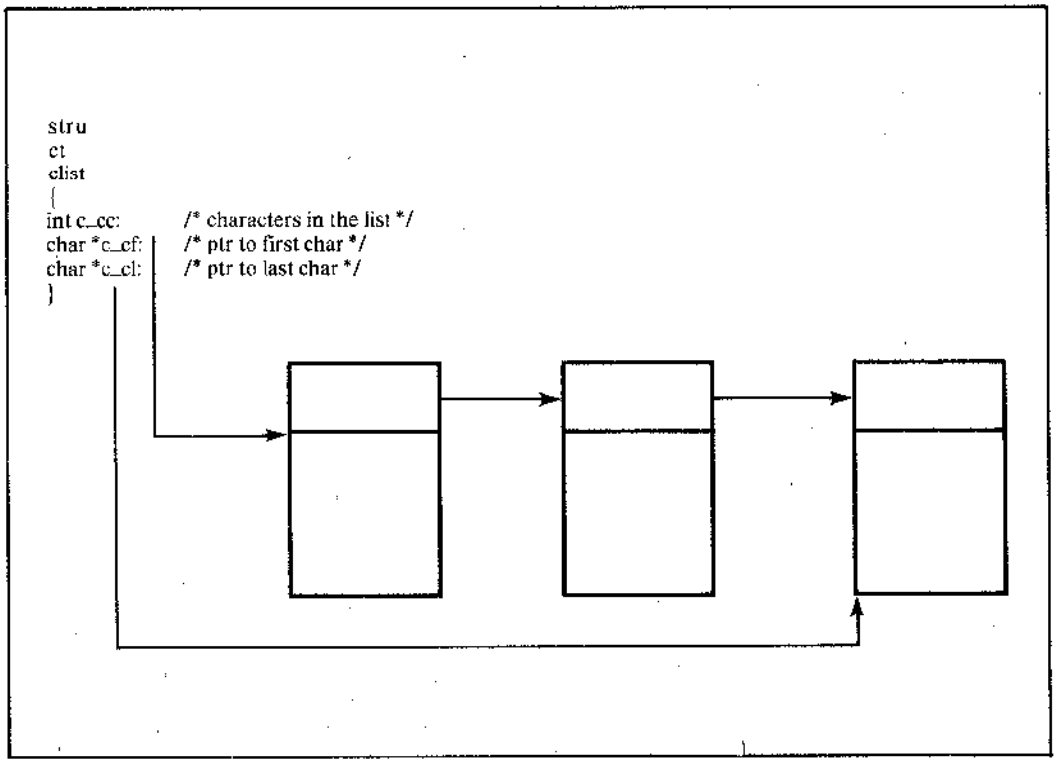
The character I/O interface is often called the raw interface and is used in general utilities like **dd** and **tar**. This interface is fast, unstructured and does not go through the kernel buffer scheme. Further details can only surface by investigating the character I/O interface.

7.2 The Character Interface

Character I/O is synonymous to "byte" I/O or direct I/O. Terminals and printers fall under this category of relatively unstructured I/O mediums between device and user-space. In general, to access character I/O devices, user processes must be locked in memory and hence, for overall system throughput, these devices must be fast. For relatively slower devices (guess who? ==> terminals), a data buffering mechanism is employed. These queues are character-based and, hence, are used only for small datatransfers.

7.2.1 CLISTS

Each driver that wishes to use these buffers declares a static buffer header. These buffers are called clists and are linked-lists of buffers. The buffer header is declared in **tty.h** (Appendix H). The structure of the header is:



280041-001

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

These clist-headers point to character-holding links that contain four-word blocks of characters(pointer and sixcharacters). Each driver program declares its own clists and this structure accumulates clist-structure-elements (as data transfer prevails) from a "freelist" of buffers. The buffer mechanism is simple compared to the block interface as only a few routines manipulate the clist structure.

Two routines, **getc()** and **putc()** manipulate the clist.

1) **getc()** removes a character from the list and moves **c_cf** forward if it not in a block boundary i.e. it is not in the end of a six character boundary. If it is, the pointer **c_cf** is set to the beginning of the next block and the block that the last character was read from is placed in the freelist (see figure 8a and b).

2) Consequently, **putc()** places a character in the list and obtains a new block when necessary (see Useful Routines).

If precautions are not taken, the freelist may be exhausted by one process. This is alleviated by defining two marks - a low and high -water mark in **tty.h.** These marks are maintained for each clist by routines that manage them. When a process requests more than its high-water mark, it puts itself to sleep until there are more free to use i.e. when the clist is flushed and it hits its low-water mark). This is an output feature (cannot suspend keyboard input!!). This will alleviate the problem of any process dominating the freelist. Low speed character devices are assisted by the clist structure declared by the relevant driver. Each driver that deals with byte I/O must declare clist structures required for each operation i.e. an input clist and an output clist.

7.2.2 TERMINAL I/O

Each terminal line is associated with line characteristics called the tty structure. Details of these are found in the manual section of **tty(4)** in the XENIX Operating System Documentation (173258-001). The file **tty.h** describes the tty structure for each line.

There exists a set of routines that manipulate this tty structure.These routines are found in tty.c (not attached) and are termed "line discipline routines." In **/sys/conf/c.c** (Appendix A) these routines are outlined under the data structure **lineswl.** These

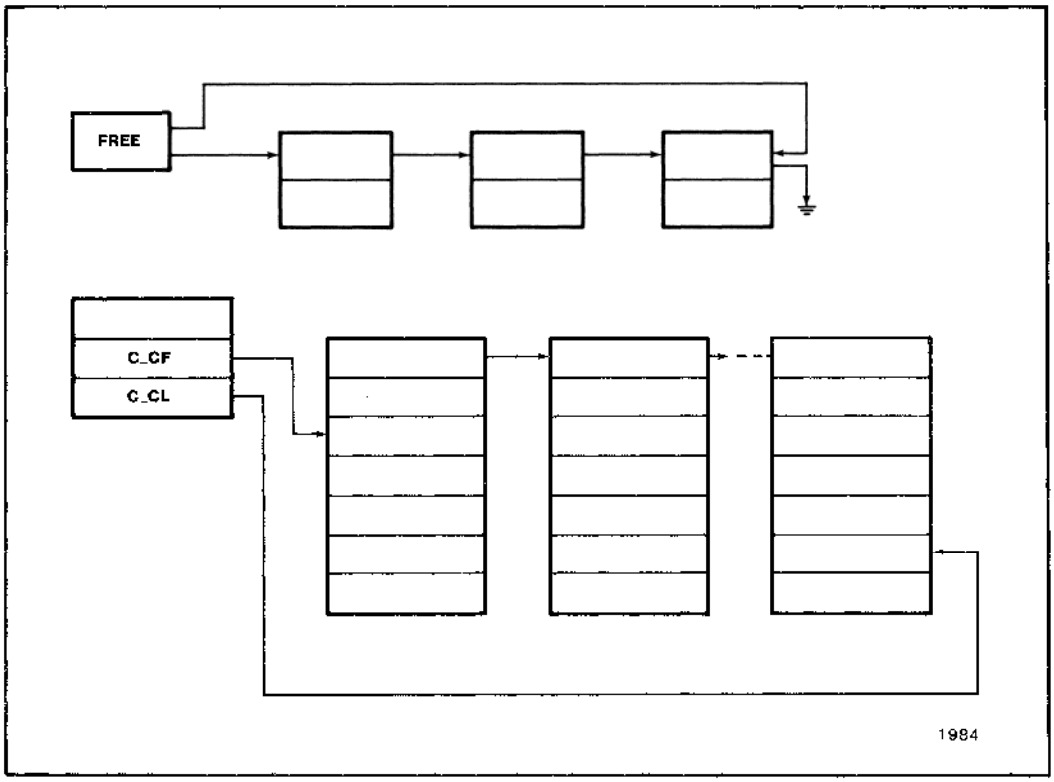


Figure 8a. clists — free list and clist for a device notice they point not to first/last character.

280041-001

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

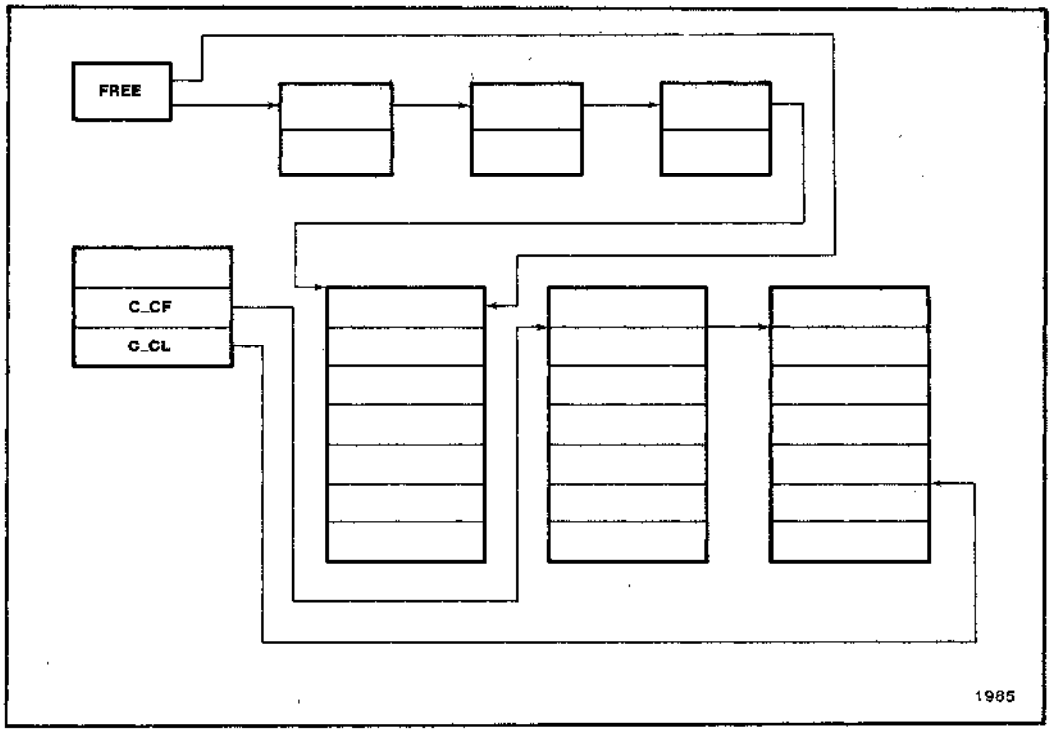


Figure 8b. When getc() takes last char on first block, the clist structure is returned to FREE LIST.

routines are preceded by "tty" or "tt" (Naming Conventions). A summary of these routines is found in **tty.help** (section 7.2.4).

Coming back to the tty structure(Appendix H), three queues are identified for use by each terminal interface. These structures are clists and are:

. the output queue
. the input queues
  . the canonical queue(cooked)
  . the raw queue

Three static buffer headers (clist headers) are established in the tty structure for each line. The output queue facilitates output to the terminal using the high/low-water marks as gauge. The raw queue is used as the first input vehicle where all characters input are placed. It is from here that the terminal echo occurs. Notice that echo response is generally quick and is irrespective of whether the requesting process is in or out of memory. The canonical queue is that queue maintained for each line that respects all characters especially those that are dependent on surrounding characters i.e. backspace, delete, character expansion etc. This queue is also called the cooked queue to symbolize a — raw (not raw) queue to further establish that XENIX is not lacking in humor! Two queues live to serve the goal of quick and consistent character treatment.

The tty.c (line discipline) routines are invisible to the driver-writer. Once these routines are called, they do most of the work. The queue manipulation and I/O functions are purely interface functions that are not in the driver code.

As mentioned before (Section 6.2), the function of any driver can be partitioned into task vs. interrupt-time execution. For character device drivers, the task-time portion deals with moving data to/from the user space and the clist queues. This movement (transfer) uses data derived from the users **u_structure** e.g. **u.u_count** is decremented on data transfer (no device driver involvement) that is performed by the **tty.c** routines.

The interrupt-time portion of the driver manages the transfer of data between the device and the clist queues (namely, the raw and output queues).

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

7.2.3 USEFUL ROUTINES

The following are useful kernel routines used by the line discipline routines. Further routines are found in Device Driver Support Routines (section 7.1.2).

```
int getc(queue)
struct clist *queue;
```

Returns a character from the clist queue or -1 if the queue is empty. The queues are either the raw, canonical or output queues.

```
int putc(c,queue)
struct clist *queue;
```

Puts a character "c" on the queue. Returns "0" if the character is placed and "−1" if unable to place in queue. The "−1" returns if gone beyond the high-water mark of the respective clist.

7.2.4 TTY.HELP — THE LINE DISCIPLINE ROUTINES

XENIX currently supports one line discipline routines i.e. how to interpret characters on I/O on the line. This line discipline is made up of several routines. They are:

ttyopen(dev, tp)

```
dev_t dev;
struct tty *tp;
```

This routine is called by the device driver's open routine. It is given an address of the line's device number and tty structure. Relevant fields in the tty structure are updated and the raw, canonical and output queues are initialized.

ttyclose(tp)

```
struct tty *tp;
```

All character queues with respect to the respective tty structure are flushed. Relevant fields in the structure are set to "closed."

ttread(tp)

```
struct tty *tp;
```

Handles a read request i.e a system call. Details on the input target addresses are found in the u_structure i.e **u.u_base, u.u_count** etc. This routine obtains data from the canonical queue and waits, if necessary, for more input. It also calls **canon()** , a routine that transfers characters from the raw input list to the canonical list after processing these lines. **Canon()** basically waits until a full line has been typed when in cooked mode whereas, in raw, it transfers data immediately.

ttread() also waits on **ttyinput()** to function in the interrupt time portion of the device driver.

ttwrite(tp)

```
struct tty *tp
```

Handles a write request. **ttwrite()** outputs **u.u_count** characters into the output queue (outq in **tty.h**) guarding the high/low-water marks. Calls **ttyoutput()** which places character in output queue adding delays, expanding tabs etc. Calls **ttstart()** to begin transmitting the character.

ttyinput(c, tp)

```
struct tty *tp;
char c;
```

Places a character on the raw queue and echoes it if required. This is how input characters are given to the read request. **ttyinput()** is run at interrupt-time to add "c" to the raw queue identified by "tp." The echo is done by a call to **ttstart()** to begin character transmittal and a call to **ttyoutput()** that basically transports characters from the raw queue to the output queue and prepares them for output.

ttstart(tp)

```
struct tty *tp;
```

is called to cause the next byte to be output if the device is idle. It is called by the task-time portion of the driver as well as the interrupt-time portion. **ttstart()** calls the "xxxstart()" routine in the driver.

ttiocomm(cmd, tp, addr, dev)

```
int cmd;
struct tty *tp;
caddr_t addr;
dev_t dev;
```

Handles common I/O control functions like line-editing, setting line characteristics except baud rates. Consider this call a transfer of input/output control and line characteristic functions to the relevant data structure that holds that information. This routine is called from the **xxxioctl()** routine of the driver.

As mentioned in the discussion on driver interfaces, the file c.c holds all kernel interface data-structures to the main driver routines. The data structure **cdevsw[]** is the link to the character I/O drivers. This structure maps the main special devices like **/dev/ttya0** to the actual driver routines.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

7.2.5 TERMINAL I/O DEVICE DRIVER ROUTINES

The following routines are typical of a character device driver. The **tty.c** routines are used as most terminal I/O device drivers rely on them. Other drivers for line printers (output only) do not use the **tty.c** routine as less processing of output is necessary and is simpler. The terminal I/O driver-routines are:

ixxxinit()

This routine initializes the device. It checks to see if the device is alive and sets a flag to remember this. It then prints messages which tells the user interface that it is/is not alive.

ixxxopen(dev, flag)

```
dev_t dev;
int flag;
```

dev is <maj,min> device number. **Flag** is B_READ, B_WRITE. Flag may be ignored for tty drivers. Called every time the device is opened. Checks for validity of open, fills out tty structure for the device line. The fields it fills are:

```
t_addr - set the device's I/O address
t_oproc - set to the address of the device's
          output start routine
t_iproc - set to address of start routine for output
t_state - device's state
```

Calls **ttyopen**. Calling sequence:

```
ixxxopen() ======> ttyopen
```

ixxxclose(dev, flag)

```
dev_t dev;
```

dev and **flag** are described for ixxxopen(). The routine is called when the last file attached to the device is closed. Calls **ttyclose** and performs clean-up e.g. flushes pending clists. Calling sequence:

```
xenix ====> ixxxclose() =====> ttyclose
```

ixxxread(dev)

```
dev_t dev;
```

This routine implements the read system call for the line and calls **ttread()** passing it the tty structure for the line. Calling sequence:

```
xenix ====> ixxxread() ======>
ttread
```

ixxxwrite(dev)

Like the read(). Calling sequence:

```
xenix ====> ixxxwrite() ====> ttwrite
```

ixxxioctl(dev, cmd, addr, flag)

```
dev_t dev;
int cmd;
caddr_t addr;
int flag;
```

Implements the ioctl system call for this line. I/O control is used for special functions such as rewinding tapes and the like. In terminal drivers, the ioctl routine is used to get/set various characteristics of the line. A common **tty.c** routine used is **ttioccomm()**. Calling sequence:

```
xenix ====> ixxxioctl() ===>
  ttioccomm
```

ixxxstart(tp)

```
struct tty *tp;          /* ptr to the tty structure */
```

the **start()** routine is called by the common tty support routines to start output on the line. The address of this routine is set in the **open()** routine and this address is kept in the line's tty-structure. This address is picked up by the tty.c routine to initiate action on the device. Typically,if the device is idle, a character is grabbed from the output queue and sent to the device. If the device is busy or no characters are available, the procedure is exited. Calling sequence:

at task time:

```
xenix===>ixxxwrite()===>ttwrite()
  ===>ttstart()===>ixxxstart()
```

at intr time:

```
device===>ixxxintr()===>ttyinput()
  ===>ttstart()===>ixxxstart()
```

on input,

```
device===>ixxxintr()===>ttstart()
  ===>ixxxstart()
```

on output

ixxxintr(level)

```
int type;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

This is the interrupt procedure. Level may be ignored. An input interrupt service routine will call **ttinput()** while an output interrupt will call **ttstart()** to begin output of the next character. The ttstart() routine will then call the driver's start() routine. As in block device, the interrupt service routine is called when the device is returning from the busy state to the idle state. **ttstart()** is called to bring the device back to the busy state if further output is necessary. Calling sequence:

HW/input intr ===> xenix ===> ixxxintr() ===> ttyinput()

HW/output intr ===> xenix ===>ixxxintr() ===> ttstart()

Note: These routines are simpler to write due to the **tty.c** routines. Terminal I/O is a special case of character I/O. Other character I/O devices require the same routines i.e. ixxxopen() etc. but cannot rely on the functionality provided by the line discipline routines.

The following diagram (figure 9) illustrates how I/O occurs in terminal I/O. Before any further detail is tackled, a brief diagram of the calling sequences for the main driver routines is shown also. **ixxxstart()** (figure 10) is called from both the **ixxxintr()** /**ixxxwrite()** routines.

7.2.6 EXAMPLES OF CHARACTER I/O (TERMINAL) DRIVERS

There is no real substitute for actual code. However, in XENIX drivers, code can be a challenge to read and understand. To alleviate long hours, a walk-through of a roughlywritten driver follows. Another example is found in Appendix I.

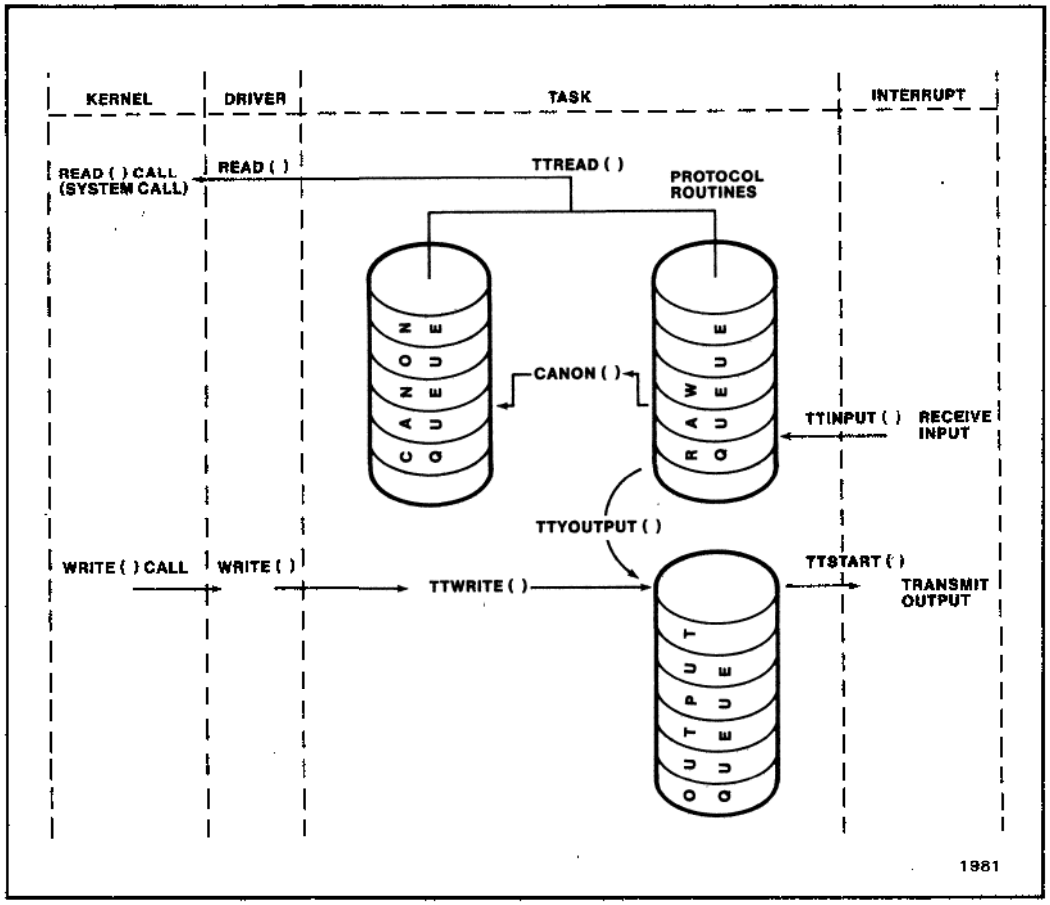


Figure 9. Architecture of Lists and Hold the Work.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

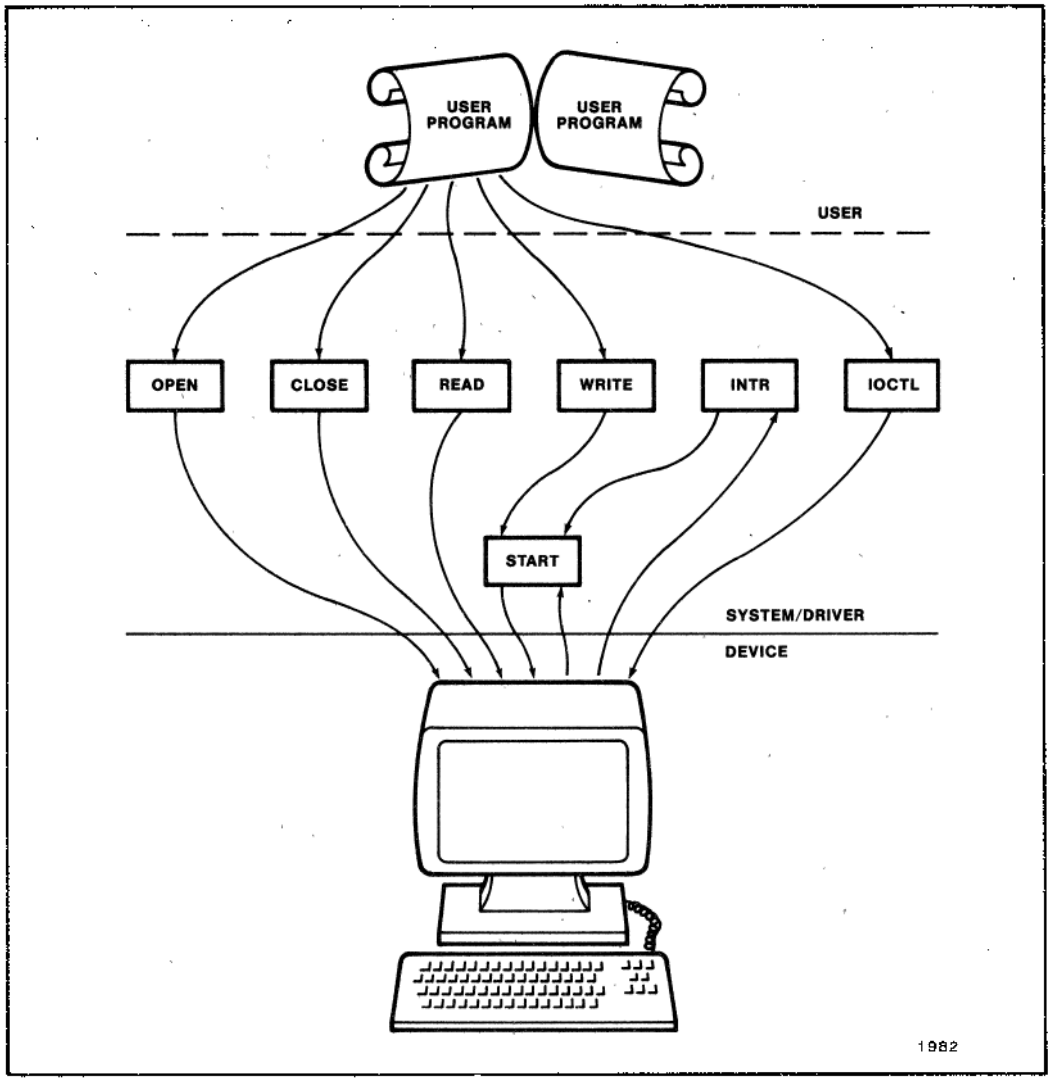


Figure 10. Terminal I/O Driver Routines.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

7.2.6.1 iSBX™ 270 Walkthrough

```
 1 /*
 2 * i270.c - iSBX270 device driver
 3 *
 4 * - implements terminal device driver for the iSBX270 character
 5 *   graphics video display controller
 6 *
 7 * See also: c270.c - i270 configuration
 8 *       i270.h - i270 include files
 9 *
10 * Notes on this driver:
11 *       (1) The driver supports the keyboard interface and display
12 *           in scroll mode or page mode
13 *       (2) All manual references in the comments are to the iSBX270
14 *           Video Display Terminal Controller Board Hardware Reference
15 *           Manual, order number 143444-001.
16 */
17
18 #include "../h/param.h"
19 #include "../h/user.h"
20 #include "../h/tty.h"
21 #include "../h/i270.h"
22
23 extern struct       i270cfg       i270cfg;       /* configuration structure */
24
25 short i270_alive;                 /* board alive flag */
26 struct tty i270tty;               /* tty structure */
27 int c_state;                      /* state variable used for escape
28                                   sequences */
29 /*
30 * i270init()
31 *       - tests for presence of the iSBX270, and reports its presence
32 *         or absence
33 *       - initializes 270 for the configured modes of operation
34 *       - this routine is called very early in the system initialization
35 */
36
37 i270init()
38 {
39   short mode;
40 int     rststat;
41
42 /*
43 * The sequence below sends a reset command to the 270. The
44 * algorithm is based on the flowchart, page 3-8 of the manual.
45 * We perform the additional task of determining if the board is
46 * present or not.
47 */
48 c_state = 0;
49 rststat = rst270();      /* reset 270 */
50 if (rststat == RSTERR)   {
51
52       /*
53       * Board not found.
54       */
55
56       printf("iSBX 270 board port %x NOT found.0, i270cfg.c_data);
57       i270_alive = I270DEAD;
58       return;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
59 }
60 else {
61       printf("iSBX 270 board port %x found.0, i270cfg.c_data);
62       i270_alive = I270LIVES;
63 }
64
65 /*
66  * Board lives, so set it up in the configuration specified.
67  * NIMASK is anded with everything to clear any bits which
68  * aren't implemented.
69  * Once we reach this point, we'll assume that the board is
70  * alive to some extent, so we'll just concern ourselves with
71  * getting through the initialization; but we can't afford to
72  * get hung up if the firmware is acting funny. Our approach
73  * will be to protect ourselves against infinite loops, but not
74  * check for error conditions or worry about reporting them.
75  */
76
77  mode = (i270cfg.c_keybrd|i270cfg.c_lpen|i270cfg.c_dma|
78          i270cfg.c_mode|IBEINT|(i270cfg.c_cursor & CURMSK))
79          & NIMASK;
80  mode270(mode);
81 }
82
83 /*
84 * rst270() - resets the 270 board
85 */
86
87 rst270()
88 {
89 unsigned   i;
90
91 /*
92  * Clear out any garbage in input and output buffers.
93  * i is a safety valve in case the board is not here and
94  * we read a 1 in the IBF bit - we'll take care of the
95  * presence or absence of the board later - for now we just
96  * need to get out of this loop.
97  */
98
99 i = 0;
100 while ((in_270(i270cfg.c_stat) & I270IBF) && (i++ < 30000)) {
101       if (in_270(i270cfg.c_stat) & I270OBF)
102             in_270(i270cfg.c_data);       /* dummy data */
103 }
104 out_270(i270cfg.c_stat, I270RST);       /* reset it */
105
106 /*
107  * We'll look for some sign of life from the 270. If
108  * it's not there, we'll either blow right through the
109  * loop, or get stuck in it forever. We'll limit
110  * forever to 30000 iterations, and check for a count of
111  * 0 or > 30000 upon terminating the loop - either one is
112  * then interpreted as a sign that the board isn't here.
113  */
114
115 i = 0;
116 while ((in_270(i270cfg.c_stat) &
117       (I270OBF|I270IBF|I270BUS)) && (i++ < 30000)) {
118       if (in_270(i270cfg.c_stat) & I270OBF)
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
119             in_270(i270cfg.c_data);       /* dummy input */
120 }
121 if ((i == 0) | (i > 30000)) return RSTERR;
122 else return RSTOK;
123}
124
125 /*
126  * mode270(mode) - sets VDTC mode for 270 board to specified mode
127  */
128
129 mode270(mode)
130 short mode;
131 {
132  unsigned   i;
133
134 /*
135  * First, go into null busy wait until we can stick another
136  * command into the input buffer - once again, i is used as
137  * an escape in case the firmware on the 270 is acting goofy.
138  */
139 i = 0;
140 while ((in_270(i270cfg.c_stat) & I270IBF) && (i++ < 30000));
141
142 /*
143  * Now we can issue a new command - we do a set VDTC mode.
144  */
145
146 out_270(i270cfg.c_stat, I270SM);
147
148 /*
149  * Wait until the 270 can accept the parameter.
150  */
151
152 i = 0;
153 while (((in_270(i270cfg.c_stat) &
154         (I270OBF|I270IBF|I270BUS)) != I270BUS) && (i++ < 30000)) {
155         if (in_270(i270cfg.c_stat) & I270OBF)
156                 in_270(i270cfg.c_data);       /* dummy input if OBF set */
157 }
158 out_270(i270cfg.c_data, mode);
159 }
160
161 /*
162 * i270open(dev) - opens dev
163 */
164
165 i270open(dev)
166 dev_t dev;
167 {
168 int    unit, i270start();
169 struct      tty     *tp;
170
171 /*
172  * First check to make sure that board is alive - if not,
173  * mark error and return.
174  */
175
176 if (i270_alive == I270DEAD) {
177         u.u_error = ENXIO; /* no such device or address */
178         return;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
179 }
180
181 /*
182  * tp will point to the tty structure
183  */
184
185 tp = &i270tty;
186
187 /*
188  * Check for 270 already being exclusively opened
189  */
190
191 if ((tp->t_state & XCLUDE) && u.u_uid) {
192         u.u_error = EBUSY;
193         return;
194 }
195
196 /*
197  * Set up fields in tty structure.
198  */
199
200 tp->t_addr = (caddr_t)i270cfg.c_data; /* io base address */
201 tp->t_oproc = i270start;     /* routine to start output */
202
203 /*
204  * If this is first open...
205  */
206
207 if ((tp->t_state & ISOPEN) == 0) {
208         ttychars(tp);       /* sets special characters */
209         tp->t_ispeed = tp->t_ospeed = 9600; /* baud rate meaningless */
210         tp->t_flags = ODDP|EVENP|ECHO|CRMOD; /* should add no
211                                                 tab expansion here */
212 }
213 tp->t_state|= CARR_ON;
214 ttyopen(dev, tp);
215 }
216
217 /*
218  * i270close(dev, flag)
219  */
220
221 i270close(dev)
222 dev_t dev;
223 {
224  struct         tty      *tp;
225
226 tp = &i270tty;
227 ttyclose(tp);
228 tp->t_addr = (caddr_t) 0; /* forget base address */
229 }
230
231 i270read(dev)
232 dev_t dev;
233 {
234  struct         tty *tp;
235
236 tp = &i270tty;
237 ttread(tp);
238 }
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
239
240 i270write(dev)
241 dev_t dev;
242 {
243  struct tty    *tp;
244
245 tp = &i270tty;
246 ttwrite(tp);
247 }
248
249 i270iintr(level)
250 int     level;
251 {
252  struct        tty *tp;
253 short status, chr;
254
255 tp = &i270tty;
256 status = inb(i270cfg.c_stat);
257 chr = inb(i270cfg.c_data);
258 if (status & I270KDR)
259         ttyinput(chr, tp);      /* only if a valid keyboard hit */
260 }
261
262 i270ointr(level)
263 int     level;
264 {
265  struct        tty     *tp;
266 .
267 tp = &i270tty;
268 if (tp->t_state & BUSY) {
269         tp->t_state &= ~BUSY;
270         ttstart(tp);
271         if ((tp->t_state & ASLEEP) &&
272             (tp->t_outq.c_cc <= TTLOWAT))
273                 wakeup((caddr_t)&tp->t_outq);
274  }
275 }
276 int     ttrstrt();
277 extern  char    partab[];
278
279 i270start(tp)
280 struct  tty     *tp;
281 {
282  int    c,s;
283  short mode;
284
285  s = spl5();
286  if (tp->t_state & (TIMEOUT|BUSY)) {
287         splx(s);
288         return;
289  }
290  if ((c=getc(&tp->t_outq)) >= 0) {
291         tp->t_state |= BUSY;
292         splx(s);
293         switch (c_state)    {
294         case 0:
295                 if (c == 0x1b) {
296 #ifdef DEBUG
297                         printf("O %x0, c);
298 #endif
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
299                     outb(i270cfg.c_data, c);
300                     c_state = 1;
301                 }
302                 else if (c == 0x11) {
303                     outb(i270cfg.c_data, c);
304                     c_state = 8; /* graphics leadin */
305                 }
306                 else if ((tp->t_flags & RAW) || (c <= 0x7f))
307                     outb(i270cfg.c_data, c);
308                 else {
309                     tp->t_state |= TIMEOUT;
310                     tp->t_state &= ~BUSY;
311                     timeout(ttrstrt, (caddr_t)tp, (c & 0x7f));
312                     return;
313                 }
314                 break;
315             case 1:
316                 switch (c)  {
317                 case '=':
318                     /* cursor address sequence */
319 #ifdef DEBUG
320                     printf("O %x0, c);
321 #endif
322                     outb(i270cfg.c_data, 0);
323                     c_state = 2;
324                     break;
325                 case 'G':
326                     /* visual attribute sequence */
327 #ifdef DEBUG
328                     printf("O 00000);
329 #endif
330                     outb(i270cfg.c_data, 0);
331                     c_state = 3;
332                     break;
333                 case 'M':
334                     /* change mode sequence */
335                     outb(i270cfg.c_data, 0);
336                     c_state = 6;
337                     break;
338                 default:
339                     /* regular escape sequence */
340 #ifdef DEBUG
341                     printf("O %x0, c);
342 #endif
343                     outb(i270cfg.c_data, c);
344                     c_state = 0;
345                     break;
346                 }
347                 break;
348             case 2:
349 #ifdef DEBUG
350                     printf("O %x0, c);
351 #endif
352                 outb(i270cfg.c_stat, I270SCP);
353                 c_state = 4;
354                 break;
355             case 3:
356 #ifdef DEBUG
357                 printf("O %x0, c|0x80);
358 #endif
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
359                 outb(i270cfg.c_data, ((c & 0x3f) |0x80));
360                 c_state = 0;
361                 break;
362         case 4:
363                 outb(i270cfg.c_data, (c - 0x30));
364                 c_state = 5;
365                 break;
366         case 5:
367                 outb(i270cfg.c_data, (c - 0x30));
368                 c_state = 0;
369                 break;
370         case 6:
371                 outb(i270cfg.c_stat, I270SM);
372                 c_state = 7;
373                 break;
374         case 7:
375                 mode = (i270cfg.c_keybrd | i270cfg.c_lpen |
376                     i270cfg.c_dma | i270cfg.c_mode |
377                     IBEINT | (i270cfg.c_cursor & CURMSK))
378                      & NIMASK;
379                 mode = (mode & 0x3f) | ((c - 0x30) < < 6);
380                 outb(i270cfg.c_data, mode);
381                 c_state = 0;
382                 break;
383         case 8:
384 /*
385                   * just output character and go back to state 0 -
386                   * nothing special, just didn't want to flip into a
387                   * special mode as a result of this graphics character
388                   */
389                  outb(i270cfg.c_data, c);
390                  c_state = 0;
391                  break;
392     }
393   }
394   else splx(s);
395 }
396
397 i270ioctl(dev, cmd, addr, flag)
398 caddr_t         addr;
399 {
400   struct        tty     *tp;
401   tp = &i270tty;
402   if (ttioccomm(cmd, tp, addr, dev)) {
403
404   /*
405    * No ioctl functions supported
406    */
407
408   }
409   else u.u_error = ENOTTY;
410 }
411
412 in_270(port)
413 unsigned        port;
414 {
415   unsigned      i;
416
417{ for (i=0; i < DELAY270; i++);
418   return (short) inb(port);
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
419}
420 out_270(port, val)
421 unsigned      port;
422 short   val;
423 {
424  unsigned    i;
425
426  for (i=0; i < DELAY270; i++);
427  outb(port, val);
428 }
```

7.2.6.2 Low Level Routines

The following routines are not required but were built for clarity and in accordance with structured methodology. The routines are:

in_270-line 412
out_270-line 420
rst270-line 87
mode270-line 129

Details of the calling sequences for these routines and others are found in figure 11. Consider the procedures **in_270** (412) and **out_270** (420). These routines are just the in and out routines of the kernel with a constant delay in front, and are used only at boot time. This implies that DELAY270 times the execution time of a for loop is lost for every byte transfer (at boot time) to/from the 270 controller, and that the constant DELAY270 must be fixed every time the kernel is ported to a new cpu. These routines would make sense if the 270 required requests to be a certain time interval apart, and the author of the driver could not guarantee that the commands to the 270 would be far enough apart any other way. It would also need to be checked that DELAY270 iterations took less than 100 microseconds, to insure interrupt latency was not adversely affected when these routines were called at a high cpu priority during interrupt service.

rst270 (87) resets the 270 controller. It begins by reading the status register of the controller. If the input buffer is full (I270IBF), the 270 cannot accept another command (even the reset), so if the data buffer is full, a data byte is read to make room in the 270 hardware's buffer. Once the input buffer of the 270 is no longer full, it is sent a reset command. The variable 'i' is used to insure that if the 270 is not present, this loop will execute at most 30,000 times. Once the reset command is sent, it should be acted upon by the controller. We enter another loop, similar to the first , which is designed to check the existence of the 270 controller. The controller exists if it is now unable to receive input (270IBF), ready to send to the cpu (I270OBF), or busy doing something (perhaps, but not necessarily, the reset). If it is none of the above, the loop will execute zero times. Note that it is possible that the 270 is not there and that garbage is being read from where the status register would have been. There is code here that assumes that if after 30,000 tries, the controller does not return to an idle state, garbage is being read. Note again that the use of a constant number of iterations to represent an interval of time is bad practice since it implies assumptions about the execution speed of the cpu, which will vary.

Mode270 (129) is a routine similar to **rst270**, but designed to set the initial mode of the controller after the controller is reset. First the driver waits until the controller is ready to receive the command (or 30,000 tries). Note that this means the controller must become ready within 30,000 executions of an **in_270** to accept the mode change command. This is why there is a delay in **in_270**, but this is still bad practice, since it is cpu dependent code. The 'set mode' command is sent, and a wait is done until the controller can accept the new mode, then the new mode byte is sent to the controller. The code to do this wait ensures that 'while the controller is busy and has input or output flush any input from it'. It should be noted that this is acceptable because this routine is called only by **i270init** at boot time. If **mode270** were called during normal system operation, this input would be meaningful and therefore would need to be processed. It is not clear that this loop will succeed in flushing all input generated while the 270 was in the previous mode, raising the possibility that garbage typed before a reboot will remain in the 270's buffers.

7.2.6.3 Required Routines

i270init (37) is called at boot time to be sure the 270 controller is ready to use, and is the only reference to the routines described above. First, **rst270** is called to see if a 270 exists (49, 50). If not, a diagnostic is printed, and the fact there is no 270 controller is remembered (56, 57). If the 270 controller exists, it is set for a standard mode modulo NIMASK, which insures the controller is not placed in an unimplemented mode (77-80). This is also the place where interrupts from the 270 are enabled (78).

i270open (165) is called for every open of the 270 device. In line 176 ensures that if the 270 controller was not found at boot time, all opens will fail with a

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

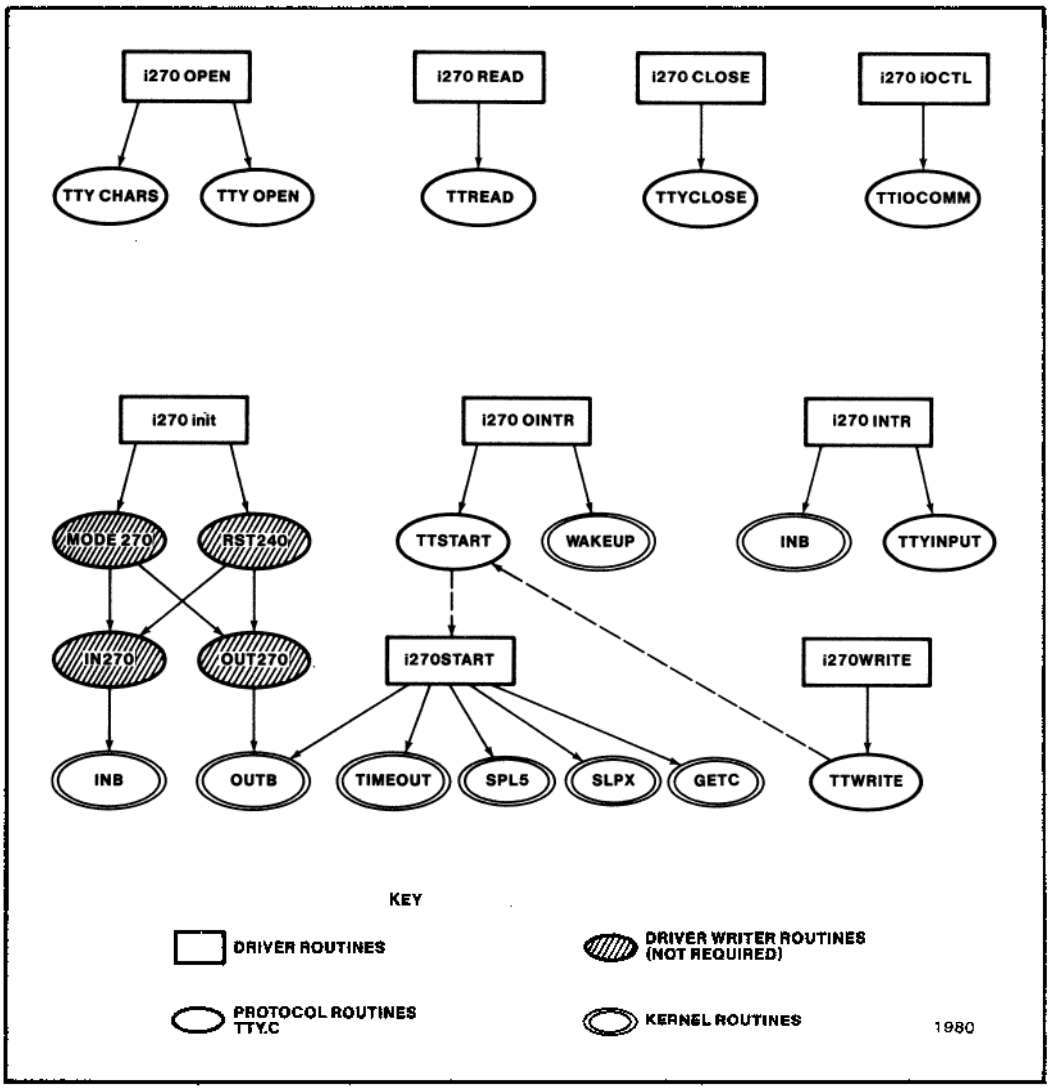


Figure 11. ISBX™ 270 Video Terminal Controller Board Driver Routine Dependancies Calling Sequences.

'nonexistent device' (ENXIO). Line 185 sets up a pointer to the status record for the device. If the device has an active exclusive open, line 191 arranges an open failure with a busy return. Line 201 sets a pointer for use by the protocol routines to start output. Line 207 arranges that the 270 looks to the kernel like a 9600 baud terminal with characteristics suited to the device. Line 213 sets a flag to note the device is open for later. Note that ISOPEN is set by ttyopen.

i270close (221) is called on the last close of the device. It just calls the protocol routine ttyclose, then removes the I/O address of the 270 from the tty structure.

i270read (231) and **i270write** (240) respond to reads and writes by simply calling the protocol routines 'ttread' and 'ttwrite', respectively.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

i270iintr (249) responds to an input interrupt by simply getting the character, and calling the protocol routine 'ttyinput' with it if it came from the keyboard.

i270ointr (262) responds to an output interrupt by checking if there was output in progress. If so, line 269 clears the BUSY flag (in case there are no more characters to output). The protocol routine 'ttstart' will send the next character and set BUSY if there is more output to do. If a process is sleeping because the output queue was at the high water mark, and the queue is now below the low water mark, all such processes are awakened(271-273).

i270start (279) is called after output interrupts and by the protocol routines to initiate output to the 270. This routine sets a cpu priority high enough to prevent reentrancy problems caused by the fact it can be called by interrupts (285). The high cpu priority is used to insure the BUSY bit does not change, so that it may be used as a lock to prevent more than one flow of control from getting into the switch statement of line 293. Once this lock is obtained, the cpu priority is lowered. The interrupt priority should remain raised for no more than the 100 microsecond. If the driver is doing output, this routine will return(line 286). It is necessary that all paths out of this routine lower the cpu priority (note lines 287, 292, and 394) within the time constraint.

Line 290 uses the 'getc' routine to get the next character from the output clist. The way interrupt driven output to tty devices is performed is by maintaining the output portion of the terminal in one of two states (idle or BUSY). In the idle state the device is ready to receive a character for output, and in the BUSY state a character has been sent to the device and the device has not yet signaled (via an output interrupt) that it is finished initiating the output. Output interrupts do not necessarily signal that the character has been output, only that the output has been started and the output device is ready to accept the next output character. A call to **i270start**, then should force the terminal into the BUSY state except in the case that the output clist is empty (lines 290 and 394). At the next output interrupt the terminal goes from BUSY to idle (line 269), and protocol routine ttstart calls the appropriate start routine (set by line 201 to be i270start) to attempt to put the device back into the BUSY state.

A major reason for the bulk of this routine is that it maintains a finite state machine which controls the disposition of characters sent to the 270. This machine implements certain escape sequences which do device control operations. Some of these device control operations cannot be done by writing to the data port, and so require special code. There is also code implementing an escape for graphic attribute bytes which prevents them from being mistaken for device control escape sequence leadins. A state transition diagram, shown in figure 12, details the control character sequences and other input sequences expected by the device. The code for the default treatment for most characters is in state 0 at lines 306 and 307. Characters with the high bit set (non-ASCII characters) are used to implement a timed delay at lines 308 through 313. If this state is not cleared, the driver will hang. The code at lines 295 through 301 implement a transition to state one in the event an escape character is written to the 270. The graphic attribute escape is in state 0 at lines 302 through 305 and in state 8 at lines 383 through 392.

An escape is followed by a character which determines which escape sequence is involved. At state 1 lines 317 through 324 we see a transition to state 2 if a cursor motion sequence (<ESC> =) is found, after sending an ASCII NUL to the 270 controller to abort the escape sequence. State 2 sends a set cursor position (I270SCP) command to the status port and sends us to state 4 (lines 352 and 353). State 4 then sends a data byte minus an ASCII '0' (the X axis position), and sends us to state 5 (lines 363 and 364). State 5 then sends another data byte minus an ASCII '0' (the Y axis position), and sends us back to state 0 (lines 367 and 368). State 1 lines 325 through 332 abort the escape sequence and move to state 3 to implement a 'set attributes' (<ESC> G) command. State 3 sends the character after masking it to insure a valid attribute byte (line 359), the returns to state 0. State 1 lines 333 through 337 abort the escape sequence and move to state 6 to initiate a 'set mode' command. State 6 sends a 'set mode' to the controller's status port and moves to state 7 (lines 371 and 372). State 7 shifts the output byte's low bit into the 'page mode' bit, sends the new mode to the data port and returns to state 0 (lines 375 through 381). Note that an <ESC> M sequence followed by a character other than an ASCII '0' or '1' is not guaranteed to result in a valid mode. The remaining state 1 code (lines 338 through 347) implement the default action, which is to send the escape sequence on to the device and return to state zero. Note that this works because no 270 hardware escape sequence is over two characters (the escape and the command). Longer escape sequences would probably need a protection similar the protection by state 8 of attribute bytes, to prevent them from being mistaken for lead-in characters. If all escape sequence support were implemented by the device via the data port, as is the case for terminals on a serial line, none of this state machine logic would be appropriate.

i270ioctl (379) is called to respond to every ioctl issued to the 270. It responds by using the protocol routine 'ttioccomm' to process all terminal ioctls, and rejecting all others since the board does not support baud rates.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

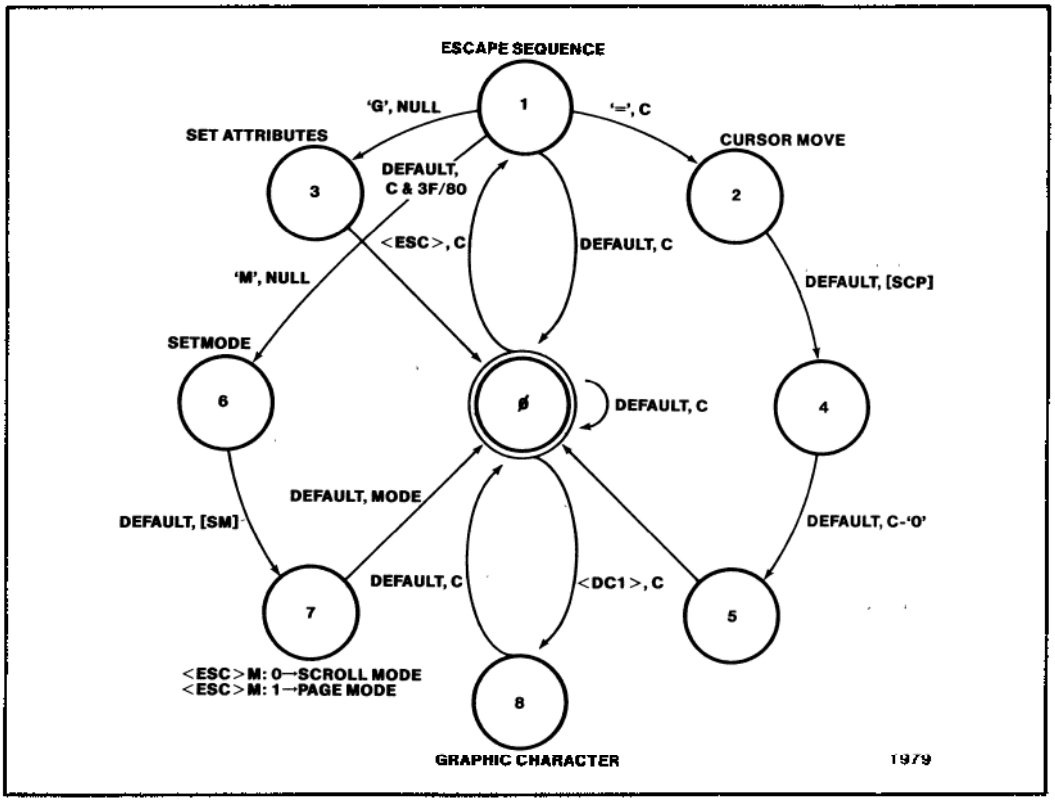


Figure 12. State Transition Diagram for iSBX™ 270 Terminal Controller Board. Used by the Driver.

7.2.6.4 A Final Note

This walkthrough has detailed the required routines, their subtleties as well as their interactions with the line discipline routines (protocol routines). The iSBX 270 Video Terminal Controller Board driver is only in draft form (it works!) and is not an example of Intel Corporation's coding methodology.

CONCLUSIONS

Now the clouds should have cleared around the XENIX Operating System and device driver writing. Insight is but one aspect of the challenge of writing device drivers. It is premature to call it a trivial task. The task of writing one is simple but the task of debugging, testing and completing one demands respect. Device driver writers have the power to do almost anything with the operating system especially crash it! This power can be extremely difficult to adjust to at the tender age of a "XENIX user." However, this discussion has briefly covered definitions, the I/O environment, the representative driver routines and code walkthroughs.

In attacking the device-driver writing issue, there is difficulty in approaching the subject with any step-by-step logical flow with a top-down methodology. Hence, imagine the entire learning environment of device drivers to be a circle. This discussion joins the circle at any one point, follows it and embraces the entire concept of device driver writing for the XENIX 286 and 86 Operating System. With this circle relatively filled, by this discussion, the task of writing device drivers may still be more a task than trivia but will be more a challenge than a chore!

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX A: THE c.c FILE CREATED BY MASTER AND XENIXCONF .....

APPENDIX B: XENIXCONF .....

APPENDIX C: THE CONFIGURATION FILE MASTER .....

APPENDIX D: INTERRUPT MAPPING .....

APPENDIX E: param.h FILE THAT LISTS THE SYSTEM CONSTANTS .....

APPENDIX F: THE buf.h FILE DESCRIBING THE BUFFER-HEADER STRUCTURE .....

APPENDIX G: NAMING CONVENTIONS .....

APPENDIX H: THE tty.h FILE DESCRIBING THE TTY STRUCTURE .....

APPENDIX I: THE c254.c AND i254.h FILES .....

APPENDIX J: THE iSBC® 534 .....

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX A:

THE C.C FILE CREATED BY MASTER AND XENIXCONF

```
/*
 *  Configuration information
 */

#define   NBUF 29
#define   NINODE    120
#define   NFILE     120
#define   NMOUNT    8
#define   SMAPSIZ   (NPROC/2)
#define   NCALL     25
#define   NPROC     100
#define   NTEXT     40
#define   NCLIST    150
#define   NFLOCKS   100
#define   MAXUPRC   15
#define   TIMEZONE  (8*60)
#define   NCOREL    1
#define   DSTFLAG   1
#define   GENBOOT   0
#define   CMASK     0
#define   MTOP 512

#include  "../h/param.h"
#include  "../h/buf.h"
#include  "../h/tty.h"
#include  "../h/conf.h"
#include  "../h/proc.h"
#include  "../h/text.h"
#include  "../h/dir.h"
#include  "../h/a.out.h"
#include  "../h/user.h"
#include  "../h/file.h"
#include  "../h/inode.h"
#include  "../h/acct.h"
#include  "../h/mmu.h"
#include  "../h/map.h"
#include  "../h/callo.h"
#include  "../h/mount.h"
#include  "../h/var.h"
#include  "../h/clist.h"

extern nodev(), nulldev(), novec();

int  clock();
int  dbgintr();
int  i544intr();
int  i215intr();
int  i74intr();
int  lpintr();

int  (*vecintsw[])() =
{
     clock,
     dbgintr,
     novec,
     i544intr,
     novec,
     i215intr,
     i74intr,
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    novec,
    lpintr,
};
extern i215open(), i215close(), i215init(), i215read(), i215write(), i215ioctl(),
  i215strategy(), i215tab;
extern i544open(), i544close(), i544init(), i544read(), i544write(), i544ioctl();
extern i74open(), i74close(), i74init(), i74read(), i74write(), i74ioctl();
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
extern lpopen(), lpclose(), lpinit(), lpwrite();
extern mmread(), mmwrite();
extern syopen(), syread(), sywrite(), syioctl();

struct    bdevsw bdevsw[]=
{
/* 0*/    1215open,  1215close, 1215strategy, &1215tab,
};

struct    cdevsw cdevsw[]=
{
/* 0*/  1215open,  1215close, 1215read,  1215write, 1215ioctl, nulldev,   0,
/* 1*/  syopen,    nulldev,   syread,    sywrite,   syioctl,   nulldev,   0,
/* 2*/  nulldev,   nulldev,   mmread,    mmwrite,   nodev,     nulldev,   0,
/* 3*/  1544open,  1544close, 1544read,  1544write, 1544ioctl, nulldev,   0,
/* 4*/  174open,   174close,  174read,   174write,  1741octl,  nulldev,   0,
/* 5*/  lpopen,    lpclose,   nodev,     lpwrite,   nodev,     nulldev,   0,
};

int  nblkdev=  1;
int  nchrdev=  6;

dev_t     rootdev=  makedev(0,1);
dev_t     pipedev=  makedev(0,1);
dev_t     swapdev=  makedev(0,2);
daddr_t   swplo=    1;
int  nswap=    1180;


int  (*initsw[])()=
{
     1215init,
     1544init,
     1741init,
     lpinit,
     (int (*)())0
};

int  ttyopen(), ttyclose(), ttread(), ttyinput(), ttstart();
char *ttwrite();

struct linesw linesw[]=
{
/*0*/     ttyopen, ttyclose, ttread, ttwrite, nodev, ttyinput, nulldev,
          nulldev, ttstart, nulldev,
        0
};

int  nldisp = 1;

int Timezone=TIMEZONE;
int Dstflag=DSTFLAG;
int Genboot=GENBOOT;
int Cmask=CMASK;

struct buf buf[NBUF];
char buffers[NBUF][BSIZE+BSLOP];
struct file file[NFILE];
struct inode inode[NINODE];
struct locklist locklist[NFLOCKS];
struct proc proc[NPROC];
struct text text[NTEXT];
struct map swapmap[SMAPSIZ];
struct callo callout[NCALL];
struct cblock cfree[NCLIST];
struct mount mount[NMOUNT];

struct var v=
{
     NBUF,
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
        NCALL,
        NINODE,
        (char *)(&inode[NINODE]),
        NFILE,
        (char *)(&file[NFILE]),
        NMOUNT,
        (char *)(&mount[NMOUNT]),
        NPROC,
        (char *)(&proc[NPROC]),
        NTEXT,
        (char *)(&text[NTEXT]),
        NCLIST,
        MAXUPRC,
        NFLOCKS
};

short mm_free = 0;
short mm_nfree = 0;
short mem_top = MTOP;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX B:

XENIXCONF

```
     The following file establishes the devices to be
selected and configured into the system.
*
*    Devices
*
i215     1
*1534     1
1544     1
i74      1
*i270     1
lp       1
*sm  1
debug    1
*fd       1
*i287     1
root     1215 1
pipe     1215 1
swap     1215 2 1 1180
*
*    Local parameters
*
timezone (8*60)
daylight 1
cmask     0
*
*    Tunable Parameters
*
*    Dont change them unless you're sure you know what you're doing!
*
buffers 29
procs   100
mounts  8
inodes  120
files   120
clists  150
locks   100
maxproc 15
mem_top 512
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX C:

The configuration file-Master

```
*
* The following devices are those that can be specified in the system
* description file.  The name specified must agree with the name shown
*name vsiz msk  typ  hndlr na bmaj cmaj #  na  vec1 vec2 vec3 vec4
*  1    2    3   4      5  6   7    8   9  10   11    12    13   14
1215    2  0137 014   i215  0   0    0   2  -1 0005     0     0   0a
fd      0  0137 014     fd  0   6    6   1  -1    0     0     0   0a
1534    2  0137 004   i534  0   0    1   1  -1 0002     0     0   0a
i544    2  0137 004   i544  0   0    3   1  -1 0003     0     0   0a
i74     2  0137 004    i74  0   0    4   1  -1 0006     0     0   0a
1270 2  0133 004   i270  0   0    7   1  -1 0106o    0     0   0a
sm      0   036 010     sm  0   1    0   1  -1    0     0     0   0a
lp      2  0132 004     lp  0   0    5   1  -1 0107     0     0   0a
debug   2     0   0    dbg  0   0    0   1  -1 0001     0     0   0a
slave7    2     0   0     sl   0   0   0   1  -1 0007     0     0   0a
i287    2     0 300   i287  0   0    0   1  -1 0008     0     0   0a
*
* The following devices must not be specified in the system description
* file.  They are here to supply information to the config program.
*
memory  0   06  0324    mm  0  -1   2   1   0   0   0   0   0
tty     0   027 0324    sy  0  -1   1   1   0   0   0   0   0
$$$
*
* The following is the line discipline table
*
tty     ttyopen ttyclose ttread  ttwrite nodev   ttyinput nulldev nulldev ttstart nulldev
$$$$$
*
* The following entries form the alias table.
*
1215    disk
1534    serial
sm      sim
$$$
*
* The following entries form the tunable parameter table.
*
buffers NBUF     50
inodes  NINODE  100
files   NFILE   100
mounts  NMOUNT    8
swapmap SMAPSIZ  (NPROC/2)
calls   NCALL  25
procs   NPROC  60
texts   NTEXT  40
clists  NCLIST 150
locks   NFLOCKS 200
maxproc MAXUPRC 15

timezone TIMEZONE (8*60)
pages   NCOREL 1
daylight DSTFLAG 1
genboot GENBOOT 0
cmask   CMASK  0
mem_top   MTOP 512
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX D:

Interrupt Mapping

Interrupts are not vectored directly to the interrupt routine procedure of a driver. Rather, the interrupt is vectored inot part of the Xenix kernel. The kernel code takes care of playing with the 8259A PIC, setting up an appropriate interrupt mask, switching to the kernel map and stack for the process, saving and restoring registers and handline scheduling semantics The outcome to this in that the interrupt routine can be written in C. Xenix handles the other details

The interrupt model is one of multiple levels of priorities. An interrupt is unique in priority and can be served only if it higher(smaller in numerical form) than the current interrupt level

The splN() command is used to lock out other interrupts which are lower in priority (i.e. <N). Spl() (set priority level) is a privileged operation and not one any process can use. All interrupt driven routines need a method to interlock data access. Data items such as buffer pools and private data. The calls splbuf()/splcli() are these features that permit routines to interlock their allocation and de-allocation of buffers. Spl only raises the current CPU interrupt level, it never changes the priority level to a lower level with as with an splx() The system buffers/clists can also be mutually excluded during access by splcli() and splbuf() which is an splN() with N high enough to lock out device interrupts that affect them. In mutual exclusion, all relevant interrupt levels are locked out from access by.

```
short s
  .
  .
  .
s = spl6()    /* returns 16 bit value, not-decipherable */
  .
  .
  .
splx()        /* must accompany each spl call to return pri level */
              /* note that spl7 is most restrictive and spl0 is the least */
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX E:

Param.h file that lists the system constants

```
#define MAXMEM 0x180          /* max core per process - 750K */
#define SSIZE  4096       /* initial stack size (bytes) */
#define SINCR  1024       /* increment of stack (bytes) */
#define NOFILE 20         /* max open files per process */
#define CANBSIZ 256       /* max size of typewriter line */
#define HZ     20         /* Ticks/second of the clock */
#define DHZ    20         /* Ticks/second of the clock */
#define MSGBUFS 128       /* Characters saved from error messages */
#define NCARGS 5120       /* # characters in exec arglist */
#define   USTK SIZE 4096 /* default size of user stack */
#define   MAXTTYS   16        /* Max # open ttys */
#define NIOSTAT 50              /* max number of bufs to keep stats for */

/*
 * priorities
 * probably should not be
 * altered too much
 */

#define PSWP   0
#define PINOD  10
#define PRIBIO 20
#define PZERO  25
#define NZERO  20
#define PPIPE  26
#define PWAIT  30
#define PSLEP  40
#define PUSER  50

/*
 * signals
 * dont change
 */

#define NSIG   17
/*
 * No more than 16 signals (1-16) because they are
 * stored in bits in a word.
 */
#define SIGHUP 1    /* hangup */
#define SIGINT 2    /* interrupt (rubout) */
#define SIGQUIT 3   /* quit (FS) */
#define SIGINS 4    /* illegal instruction */
#define SIGTRC 5    /* trace or breakpoint */
#define SIGIOT 6    /* iot */
#define SIGEMT 7    /* emt */
#define SIGFPT 8    /* floating exception */
#define SIGKIL 9    /* kill, uncatchable termination */
#define SIGBUS 10   /* bus error */
#define SIGSEG 11   /* segmentation violation */
#define SIGSYS 12   /* bad system call */
#define SIGPIPE 13  /* end of pipe */
#define SIGCLK 14   /* alarm clock */
#define SIGTRM 15   /* Catchable termination */
#define   SIGFN     16   /* function key */

/*
 * fundamental constants of the implementation--
 * cannot be changed easily
 */
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
#define NBPW   sizeof(int)    /* number of bytes in an integer */
#define BSIZE  1024       /* size of secondary block (bytes) */
/* BSLOP can be 0 unless you have a TIU/Spider */
#define BSLOP  4          /* In case some device needs bigger buffers */
#define NINDIR (BSIZE/sizeof(daddr_t))
#define BMASK  01777          /* BSIZE-1 */
#define BSHIFT 10         /* LOG2(BSIZE) */
#define NMASK  0377       /* NINDIR-1 */
#define NSHIFT 8          /* LOG2(NINDIR) */
#define   INOPB     (BSIZE/sizeof(struct dinode)) /* # inodes per block */
#define   LINOPB    4         /* LOG2(INOPB) */
#define NULL   0
#define DCMASK  0               /* default mask for file creation */
#define NODEV  (dev_t)(-1)
#define ROOTINO ((ino_t)2)    /* i number of all roots */
#define SUPERB ((daddr_t)1)   /* block number of the super block */
#define DIRSIZ 14         /* max characters per directory */
#define NICINOD 100       /* number of superblock inodes */
#define NICFREE 100       /* number of superblock free blocks */
/* #define INFSIZE 138          /* size of per-proc info for users */
#define CBSIZE 6          /* number of chars in a clist block */
#define CROUND 07         /* clist rounding: sizeof(int *) + CBSIZE - 1*/

/*
 * MMU parameters.
 */

#define MMPGSZ 2048       /* bytes/page in the MMU */
#define LMMPGSZ 11        /* log2(MMPGSZ) */
#define NPAGEPS 32        /* There are 32 pages in a segment */
#define NSEG   0          /* max seg / user (see user.h) */
#define   MMFRAGMENTS    256  /* mamimum number of free segments */

/*
 * Some macros for units conversion
 */

/* pages to disk blocks */
#define ptod(x) ((x)*(MMPGSZ/BSIZE))
/* bytes to disk blocks */
#define btod(x)     (((x)+(BSIZE-1))>>BSHIFT)

/* inumber to disk address */
#define itod(x) (daddr_t)(((unsigned)(x)+(INOPB+INOPB-1))>>LINOPB)

/* inumber to disk offset */
#define itoo(x) (int)(((x)+(INOPB+INOPB-1))&(INOPB-1))

/* pages to bytes */
#define ptob(x) ((x)<<LMMPGSZ)

/* bytes to pages */
#define btop(x) (((unsigned)(x)+(MMPGSZ-1))>>LMMPGSZ)

/* bytes to page number */
#define btopn(x) (((unsigned)(x))>>LMMPGSZ)

/* page to address */
#define ptoa(x)  ( ((long)(x) << LMMPGSZ) )

/* address (long (32 bit)) to page number (int)*/
#define atopn(x) ((int)(((long)(x))>>LMMPGSZ))

/* address (long (32 bit)) to page count (int)*/
#define atop(x)  ((int)(((long)(x)+(MMPGSZ-1))>>LMMPGSZ))

/* address (long (32 bit)) to offset (int) get bits LMMPGSZ-1 - 0 */
#define atoo(x) ((int)((x)&(MMPGSZ-1)))

/* long address to short address (get low 16 bits of long address */
#define atos(x) ((int)( (x) & 0x0000FFFF))
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
/* long address to short address (get low 16 bits of long address */
#define atoh(x) ((int)( (x) >> 16))

/* page number to long */
#define ptol(x) ((long)((int)(x))<<LMMPGSZ)

/* major part of a device */
#define major(x)    (int)(((unsigned)(x)>>8))

/* minor part of a device */
#define minor(x)    (int)((x)&0377)

/* make a device number */
#define makedev(x,y)     (dev_t)((x)<<8 | (y))

/* extract low word of long */
#define   LOWWORD(x)     ((int)x)

/* extract high word of long */
#define   HIGHWORD(x)    ((int)((long)x >> 16))
/* 8086 base from an absolute physical address */
#define   base86(x) ((short)(x>>4))

typedef struct { int r[1]; } *     physadr;
typedef struct { unsigned short off;
          unsigned short seg; }  segadr;
typedef long        daddr_t;
typedef char *      caddr_t;
typedef unsigned short   ino_t;
typedef long        time_t;
typedef int         label_t[5];    /* return, sp, si, di, bp */
typedef int         dev_t;
typedef long        off_t;

/*
 * Machine-dependent bits and macros
 */

#define   SPLOMASK  0x00      /* 0xC0 ==> SM on On-Board USART */
#define USERMODE(ps)     ((ps)&03 == 03)
#define   CLKONLY(ps)    ((ps)&0x8000)  /* I0i0 --- PLB */
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX F:

The buf.h file describing the buffer-header structure

```
/*
 * Each buffer in the pool is usually doubly linked into 2 lists:
 * the device with which it is currently associated (always)
 * and also on a list of blocks available for allocation
 * for other use (usually).
 * The latter list is kept in last-used order, and the two
 * lists are doubly linked to make it easy to remove
 * a buffer from one list when it was found by
 * looking through the other.
 * A buffer is on the available list, and is liable
 * to be reassigned to another disk block, if and only
 * if it is not marked BUSY.  When a buffer is busy, the
 * available-list pointers can be used for other purposes.
 * Most drivers use the forward ptr as a link in their I/O
 * active queue.
 * A buffer header contains all the information required
 * to perform I/O.
 * Most of the routines which manipulate these things
 * are in bio.c.
 */
struct buf
{
     int  b_flags;       /* see defines below */
     struct    buf *b_forw;        /* headed by d_tab of conf c */
     struct    buf *b_back;        /*  "  */
     struct    buf *av_forw;       /* position on free list, */
     struct    buf *av_back;       /*     if not BUSY*/
     dev_t     b_dev;              /* major+minor device name */
     unsigned b_bcount;       /* transfer count */
     union {                 /* always points to buffer area */
         caddr_t b_addr;             /* as low order core address */
         int *b_words;               /* as words for clearing */
         struct filsys *b_filsys;    /* as superblocks */
         struct dinode *b_dino;      /* as ilist */
         daddr_t *b_daddr;           /* as indirect block */
     } b_un;
     daddr_t   b_blkno;       /* block # on device */
     char b_xmem;             /* high order core address */
     char b_error;       /* returned after I/O */
     unsigned int b_resid;         /* bytes not transferred after error */
     unsigned int b_cylin;         /* cylinder number for disk i/o queue */
};

extern struct buf buf[];      /* The buffer pool itself */
extern struct buf bfreelist;       /* head of available list */

/*

 * These flags are kept in b_flags.
 */
#define   B_WRITE   0    /* non-read pseudo-flag */
#define   B_READ    01   /* read when I/O occurs */
#define   B_DONE    02   /* transaction finished */
#define   B_ERROR   04   /* transaction aborted */
#define   B_BUSY    010  /* not on av_forw/back list */
#define   B_PHYS    020  /* Physical IO potentially using UNIBUS map */
#define   B_MAP     040  /* This block has the UNIBUS map allocated */
#define   B_WANTED 0100  /* issue wakeup when BUSY goes off */
#define   B_AGE     0200 /* delayed write for correct aging */
#define   B_ASYNC   0400 /* don't wait for I/O completion */
#define   B_DELWRI 01000 /* don't write till block leaves available list */
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
#define   B_TAPE 02000   /* this is magtape (no bdwrite, raw i/o at any loc) */
#define   B_PBUSY   04000
#define   B_PACK    010000
#define   B_PURGE   020000    /* bpurge() in progress--invalidate buf when releas

/*
 * special redeclarations for
 * the head of the queue per
 * device driver.
 */
#define   b_actf    av_forw
#define   b_actl    av_back
#define   b_active b_bcount
#define   b_errcnt b_resid

/*
 * collect io statistics
 */
#define   DISKMON   1

#ifdef    DISKMON
struct {
     int  nbuf;
     long nread;
     long nreada;
     long ncache;
     long nwrite;
     long    bufcount[NIOSTAT];
     long    nswapb;
} io_info;
#endif
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX G:

Naming Conventions

The convention followed is:
ixxxppp

where
xxx is the number of the device (i.e 534,524)
ppp is the procedure name i.e. init, open

Thus, the iSBX 270 Video Terminal Controller Board driver has the interface procedures:

```
i270init()
i270open()
i270start() etc.
```

This naming convention allows the kernel procedures to understand unique driver interfaces. Usually, data structures also follow this convention to identify variable names and symbols.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX H:

The tty.h file describing the tty structure

```
/*
 * A clist structure is the head
 * of a linked list queue of characters
 * The characters are stored in 4-word
 * blocks containing a link and several characters.
 * The routines getc and putc
 * manipulate these structures.
 */
struct clist
{
     int  c_cc;          /* character count */
     char *c_cf;         /* pointer to first char */
     char *c_cl;         /* pointer to last char */
};

/*
 * A tty structure is needed for
 * each UNIX character device that
 * is used for normal terminal IO.
 * The routines in tty.c handle the
 * common code associated with
 * these structures.
 * The definition and device dependent
 * code is in each driver. (kl.c dc.c dh.c)
 */

struct tc {
     char t_intrc;  /* interrupt */
     char t_quitc;  /* quit */
     char t_startc; /* start output */
     char t_stopc;  /* stop output */
     char t_eofc;        /* end-of-file */
     char t_brkc;        /* input delimiter (like nl) */
};

struct tty
{
     struct    clist t_rawq;  /* input chars right off device */
     struct    clist t_canq;  /* input chars after erase and kill */
     struct    clist t_outq;  /* output list to device */
     int  (* t_oproc)(); /* routine to start output */
     int  (* t_iproc)(); /* routine to start input */
     struct chan *t_chan;     /* destination channel */
     caddr_t   t_linep;  /* aux line discipline pointer */
     caddr_t   t_addr;        /* device address */
     dev_t     t_dev;         /* device number */
     short     t_flags;  /* mode, settable by ioctl call */
     short     t_state;  /* internal state  not visible externally */
     short     t_2state; /* continuation of state, driver specific */
     short     t_pgrp;        /* process group name */
     char t_delct;  /* number of delimiters in raw q */
     char t_line;        /* line discipline */
     char t_col;         /* printing column of device */
     char t_erase;  /* erase character */
     char t_kill,        /* kill character */
     char t_char;        /* character temporary */
     char t_ispeed; /* input speed */
     char t_ospeed; /* output speed */
     union {
          struct tc t_tc;
          struct clist t_ctlq;
     } t_un;
};
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
#define   tun  tp->t un

/*
 * structure of arg for ioctl/
 */                          *
struct    ttiocb {
     char ioc_ispeed;        d
     char ioc_ospeed;        e
     char ioc_erase,         f
     char ioc_kill;          a
     int  ioc_flags;         u
};                           l
                             t
#define   TTIPRI    28
#define   TTOPRI    29       s
                             p
#define   CERASE    '        e
                             c#define  CEOT 004
#define   CKILL     '@'      i
#define   CQUIT     034      a/* FS, cntl shift L */
#define   CINTR     0177     l/* DEL */
#define   CSTOP     023       /* Stop output: ctl-s */
#define   CSTART    021      c/* Start output: ctl-q */
#define   CBRK 0377          h
                             a
/* limits */                 r
#define   TTHIWAT   100      a
#define TTLOWAT 70           c
#define   TTYHOG    256      t
                             e
/* modes */                  r
#define   TANDEM    01       s
#define   CBREAK    02
#define   LCASE     04       *
#define   ECHO 010           /
#define   CRMOD     020
#define   RAW  040
#define   ODDP 0100
#define   EVENP     0200
#define   NLDELAY   001400
#define   TBDELAY   006000
#define   XTABS     006000
#define   CRDELAY   030000
#define   VTDELAY   040000

/* Hardware bits */
#define   DONE 0200
#define   IENABLE   0100

/* Internal state bits */
#define   TIMEOUT   01        /* Delay timeout in progress */
#define   WOPEN     02        /* Waiting for open to complete */
#define   ISOPEN    04        /* Device is open */
#define   FLUSH     010       /* outq has been flushed during DMA */
#define   CARR_ON   020       /* Software copy of carrier-present */
#define   BUSY 040       /* Output in progress */
#define   ASLEEP    0100      /* Wakeup when output done */
#define   XCLUDE    0200      /* exclusive-use flag against open */
#define   TTSTOP    0400      /* Output stopped by ctl-s */
#define   HUPCLS    01000          /* Hang up upon last close */
#define   TBLOCK    02000          /* tandem queue blocked */
#define   DKCMD     04000          /* datakit command channel */
#define   DKMPX     010000         /* datakit user-multiplexed mode */
#define   DKCALL    020000         /* datakit dial mode */
#define   DKLINGR   040000         /* datakit lingering close mode */
#define   CNTLQ     0100000        /* interpret t_un as clist */


/* Driver specific state bits */
#define INBUSY 01         /* Input in progress */
#define INSTOP  02        /* Stop input interrupts */
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
/*
 * tty ioctl commands
 */
#define   TIOCGETD  (('t'<<8)|0)
#define   TIOCSETD  (('t'<<8)|1)
#define   TIOCHPCL  (('t'<<8)|2)
#define   TIOCMODG  (('t'<<8)|3)
#define   TIOCMODS  (('t'<<8)|4)
#define   TIOCGETP  (('t'<<8)|8)
#define   TIOCSETP  (('t'<<8)|9)
#define   TIOCSETN  (('t'<<8)|10)
#define   TIOCEXCL  (('t'<<8)|13)
#define   TIOCNXCL  (('t'<<8)|14)
#define   TIOCFLUSH (('t'<<8)|16)
#define   TIOCSETC  (('t'<<8)|17)
#define   TIOCGETC  (('t'<<8)|18)
#define   TIOCGETS  (('t'<<8)|19)
#define   DIOCLSTN  (('d'<<8)|1)
#define   DIOCNTRL  (('d'<<8)|2)
#define   DIOCMPX       (('d'<<8)|3)
#define   DIOCNMPX  (('d'<<8)|4)
#define   DIOCSCALL (('d'<<8)|5)
#define   DIOCRCALL (('d'<<8)|6)
#define   DIOCPGRP  (('d'<<8)|7)
#define   DIOCGETP  (('d'<<8)|8)
#define   DIOCSETP  (('d'<<8)|9)
#define   DIOCLOSE  (('d'<<8)|10)
#define   DIOCTIME  (('d'<<8)|11)
#define   DIOCRESET (('d'<<8)|12)
#define   FIOCLEX       (('f'<<8)|1)
#define   FIONCLEX  (('f'<<8)|2)
#define FIORDCHK        (('f'<<8)|3)
#define   MXLSTN        (('x'<<8)|1)
#define   MXNBLK        (('x'<<8)|2)

/*
 * tty ioctl commands (extension)
 */
#define   MLCRESET  (('m'<<8)|0)
#define   MLCBOOT       (('m'<<8)|1)
#define   MLCREAD       (('m'<<8)|2)
#define   MLCWRITE  (('m'<<8)|3)
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX I:

The c254.c and i254.h files

```
/*
 * include file for 254 driver ... this is i254.h
 *
 * mask constants for BMC status:
 */

#define   BMCBUSY   0x80
#define   BMC


/*
 * configuration structure for 254
 */
struct    i254cfg {unsigned c_base_port;
          unsigned c_page_size;
          };

/* this is c254.c
*/

#include "../h/i254.h"
struct    i254cfg   i254cfg={0x40, /* I/O base port address */
                256} /* bubble page size
                      - 64 for 1 bubble,
                        128 for 2 bubbles,
                        256 for 4 bubbles */
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX J:

The iSBC 534

```
/*
 * INTEL CORPORATION PROPRIETARY INFORMATION.  THIS LISTING IS
 * SUPPLIED UNDER THE TERMS OF A LICENSE AGREEMENT WITH INTEL
 * CORPORATION AND MAY NOT BE COPIED NOR DISCLOSED EXCEPT IN
 * ACCORDANCE WITH THE TERMS OF THAT AGREEMENT.
 */

/*
 * isbc534 device driver.
 *
 * This is the set of procedures that make up the isbc534 device driver.
 * The procedures provided include i534open, i534close, i534intr, i534start,
 * i534ioctl which are the interfaces between xenix and the hardware.
 * The subroutines used are i534init, i534param which are used to program the
 * hardware. The isbc534 hardware consists of 4 usarts, 2 pic's, 2 pit's and
 * a ppi.
 *
 * Multiple isbc534 minor number structure:
 * bits 0-4:
 * Minor #:    Board:
 * 0-3    usarts    1st Board lowest intr level
 * 4-7    usarts    2nd Board intr level
 *  .
 *  .
 * 20-23 usarts 6th Board last intr level(7)
 * bits 5-6 reserved for future use.
 *
 * NOTES: The base address of the board MUST be non-zero!!!
 *     The isbc86/12 board must have the fail safe timer installed.(default)
 *
 *     The isbc534 REQUIRES a HARDWARE MODIFICATION for MODEM SUPPORT
 *        The isbc534 requires a default jumper removed from
 *             pin 105-106
 *        and add a jumper from
 *             pin 105-104
 *        This modification cascades timer bdg4 to bdg5 to allow a
 *        2 second timer used in detecting carrier from a modem.
 *        The carrier loss signal is generated via a separate interrupt.
 *     The above modification is ONLY NEEDED to FOR MODEM SUPPORT but should
 *     be done for consistency.
 *
 *
 *     Debug switches are: DEBUG for isbc534 support.
 *                   i534debug: output control
 *                        0 == no output except spurious intrs
 *                        1 == special currently same as 0
 *                        2 == little but useful output
 *                        3 == all output
 *
 * Written by Jim Chorn
 *      on 12/29/81
 *
 * History: modified 1/15/82 for multiple board support.
 *       modified 1/29/82 for console support.
 *       modified 3/29/82 for addition of modem support
 *        mods affect i534open,i534close,i534intr.
 *       modefied 4/22/82     moved console support out to support isbx351
 *       modified 6/22/82 added OR tie'ng of 534's on the same interrupt
 *        level.
 *        Changed the modem support bit to 0xC0 meaning configure
 *        the line for detection of aquisition AND loss of carrier detect
 *        signal. Bit 0x40 means detection of aquisition and bit 0x80
 *        means detection of loss of carrier detect signal.
 *        The detection of aquisition of carrier without detection of
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
*       loss of carrier is meaningless and is not mentioned in the
*       manual entry.
*
*/
#include "../h/i534.h"       /* hardware structure and local commands */
#include "../h/param.h"
#include "../h/systm.h"      /* system */
#include "../h/conf.h"       /* system configuration */
#include "../h/dir.h"        /* system directory structures */
#include "../h/a.out.h"         /* needed for user.h */
#include "../h/user.h"       /* user structures (system)   */
#include "../h/tty.h"        /* device structures (system) */
#include "../h/usart.h"      /* baud rates */
#include "../h/intr.h"       /* some pic commands from system */

#ifdef DEBUG
int  i534debug = 1;      /* debug output control */
#endif

int       i534wakeup;          /* wakeup variable for modems */
struct    tty  i534tty[N534*4];    /* 4 USARTs per 534 */
struct    i534cfg   i534cfg[N534];      /* board software addresses von conf*/
int       i534base[8];         /* board number -> board base addr */
int       i534alive[N534];     /* does it live ?? */

/*
 * This procedure verifies that a isbc534 board is presently
 * configured by putting the board into test mode and
 * then checking if the board actually is in test mode.
 * This test mode check is a one bit test. If the board configured is not
 * present an array variable for each board called i534alive is set to false.
 *
 * TITLE: i534probe
 *
 * CALL:  i534probe();
 *
 * INTERFACES: i534open, i534intr (thru the variable i534cfg[])
 *
 * CALLS: none
 *
 * History:
 *
 */

i534probe()
{
     register  board;
     register struct i534cfg *cf;
     struct    db534     *DBbase;  /* set up the i/o boards base address */
     int       alive;

     for (board=0; board<N534; board++){
          cf = &i534cfg[board];
          if(cf->c_base != 0) {
               alive = 1;               /* assume it lives */
               DBbase =cf->c_base;
               outb(&DBbase->stestmd, 1);     /* select test mode */
               if((inb(&DBbase->stestmd) & 1) == 0 )   /* is test mode selected?  */
                    alive = 0;          /* trash base addr for intr() */
               outb(&DBbase->stestmd, 0xff);
               if((inb(&DBbase->stestmd) & 1) == 0)
                    alive = 0;
               outb(&DBbase->stestmd, 0);     /* deselect test mode */
               printf("iSBC 534 Based %x board %d %s.0,
                    cf->c_base, board,
                    alive? "found" : "NOT found" );
               i534base[board] = cf->c_base; /* associate board & tty struc*/
               i534alive[board] = alive;
          }
     }
}
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
/*
 * This procedure initializes the isbc534 when the call to dinit is
 * made. This procedure is done ONCE ONLY in the following sequence:
 *   initialize the isbc534 structures to point at the board,
 *   reset the board,
 *   initialize the usarts with a special hardware sequence,
 *   initialize the ppi port for input,
 *   initialize and mask the on-board pic's.
 * After this has been accomplished there is no reason to reinitialize the
 * isbc534 except when hardware failure occurs.
 * NOTE: The baud rate clocks are not programmed here; this
 * is done on the first device open in the call to i534param;see i534open.
 *
 * TITLE: i534init
 *
 * CALL:  i534init();
 *
 * INTERFACES: dinit
 *
 * CALLS: delay
 *
 * History: 1/11/82 Shortened the delay time from 100 to 10 to speed things
 *                  up a bit.
 *          1/15/82 Added probing for boards.
 *
 */

i534init()
{
     struct    db534     *DBbase;            /* set up the i/o boards base address */
     struct    cb534     *CBbase;
     register int  i, board;

#ifdef DEBUG
     if(i534debug>=2)
          printf("i534 init, ");
#endif
     i534probe();
     for(board=0; board<N534; board++) {
          if(i534alive[board] == 0)
               continue;              /* Board not there! */
          CBbase = DBbase = i534cfg[board].c_base;
          outb(&DBbase->reset, 0);
          outb(&DBbase->seldata, 0);
          for (i=0;i<4;i++){              /* init each usart */
               i51uinit(&DBbase->USART[i].cntrl);
          }
          outb(&DBbase->PIC[0].csr, PICICW1);
          outb(&DBbase->PIC[0].msr, PICICW2);
          outb(&DBbase->PIC[1].csr, PICICW1);
          outb(&DBbase->PIC[1].msr, PICICW2);
          outb(&DBbase->PIC[0].msr, MASKINT);
          outb(&DBbase->PIC[1].msr, MASKINT);
          outb(&CBbase->selcntr,1);
          i53tprog(&CBbase->PIT[1].timer[1],      /* timer bdg4*/
               &CBbase->PIT[1].pcr,               /* pcr */
               (RATEMD0|0x40),                    /* mode */
               U534SPEED);                   /* 2 sec */

          i53tprog(&CBbase->PIT[1].timer[2],      /* timer bdg5*/
               &CBbase->PIT[1].pcr,               /* pcr */
               (RATEMD0|0x80),                    /* mode */
               U534SPEED );                  /* 2 sec */
          outb(&CBbase->seldata,1);
     }
}


/*
 * This procedure sets up a usart timer for a load operation.
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
 * The code depends on having the ttystructure filled out before a call is made
 * to i534param. This is the sequence of events;
 *   check for valid speed
 *   program timer       (using i53tprog)
 * This procedure will program bdg0 to bdg4 as a baud rate generator.
 *
 *
 * TITLE: i534param
 *
 * CALL:  i534param(dev);
 *
 * INTERFACES: i534init,
 *
 * CALLS: i53tprog
 *
 * History:    1/20/82 : removed bdg4, bdg5 programming options.
 *               These timers aren't used.
 *        1/29/82 : added console programming
 *        4/7/82    : added i53tprog to handle pit programming
 *        4/22/82 : removed console programming
 *
 */

#define MAXBAUDS 15 /* maximum indexes into i534baud[] */
int i534baud[] = {
     US_B0 ,        US_B50,   US_B75 ,  US_B110,  0,
     US_B150,  US_B200,  US_B300,  US_B600, US_B1200,
     0,        US_B2400,       US_B4800,       US_B9600,       0,
     0
};
int  i534speed[N534*4];       /* track record */

i534param(dev)
dev_t dev;

{
     struct    cb534     *CBbase;  /* set up the i/o boards base address */
     register struct tty *tp;
     int unit, s, speed,mode,pit;

     unit = minor(dev) & MINORMSK;
     tp = &i534tty[unit];
     CBbase = tp->t_addr & 0xf0;
     s = (int)tp->t_ospeed;
     if(s==0)  {         /* hangup signal via stty */
          outb(tp->t_addr+1,SHANGUP);
          return;
     }
     if(s == i534speed[unit])
          return;             /* already that fast */
     else
          i534speed[unit] = s;
     unit %= 4;               /*  which usart? */
     speed = i534baud[s];
     if ((s > MAXBAUDS) || ((s != 0) && (speed == 0))) {
          u.u_error = EINVAL; /* invalid baud rate */
          return;
     }
     if (unit == 3){
          pit = &CBbase->PIT[1].timer[0];
          mode = RATEMD0;
     }else{
          pit = &CBbase->PIT[0].timer[unit];
          mode = RATEMD0 | (unit <<6);
     }
     s = SPL();
     outb(&CBbase->selcntr,1);
     i53tprog(pit,(pit|0x03),mode,speed);
     outb(&CBbase->seldata,1);
     splx(s);
}
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
/*
 * This procedure opens one of the 4 lines on the isbc534 board for
 * exclusive use by a user.  The file structure is initialized
 * and control is passed to ttyread which does the actual open.
 * Not supported is the fifth device which is the parallel port.
 *
 *
 * TITLE: i534open
 *
 * CALL:  i534open(dev, flag);
 *
 * INTERFACES: xenix
 *
 * CALLS: i534init, ttyopen
 *
 * History: 1/15/82:     Modifed code for multiple i534's to:index a
 *             configuration table to get the board base address.
 *
 */
int i534start();

i534open(dev)
dev_t     dev;

{
     struct    db534     *DBbase;  /* set up the i/o boards base address */
     register struct tty *tp;
     register int unit;
     int modem;               /* modem bit in minor dev numb */

     unit = minor(dev) & MINORMSK;
     if (unit >= (N534*4)) {            /* not enough tp's */
          u.u_error = ENXIO;
          return;
     }
     tp = &i534tty[unit];
     if (i534alive[unit/4] == 0) {      /* Board not there! */
          u.u_error = ENXIO;
          return;
     }
     DBbase = (struct db534 *)i534cfg[unit/4].c_base;
     unit %= 4;
     tp->t_addr = (caddr_t)&DBbase->USART[unit].data;
     modem = minor(dev) & MODEMMSK;
     tp->t_oproc = i534start;
     if ((tp->t_state & ISOPEN) == 0 ) {
          ttychars(tp);
          tp->t_ispeed =  tp->t_ospeed = ISPEED;  /* channel speed */
          tp->t_flags = ODDP | EVENP | ECHO | CRMOD;
          i534param(dev);               /* load baud clock */
          outb((tp->t_addr +1), SANSWER);    /* turn usart (dtr) on */
          if(modem) {
               if(modem & MODEMWAIT)    /* mask detect leaving aqua */
                    while((inb((tp->t_addr +1)) & DTRON) == 0)
                         sleep((caddr_t)&i534wakeup,TTIPRI);
               outb(&DBbase->PIC[1].msr ,((inb(&DBbase->PIC[1].msr)) &
               (~(0x10<< unit)) &TIMERGO));
                              /*unmask carrier/detect */
          }
          outb(&DBbase->PIC[0].msr ,((inb(&DBbase->PIC[0].msr)) &(~(3<< unit * 2))));
                              /* unmask txrdy, rxrdy */
     }
     if (tp->t_state & XCLUDE && u.u_uid != 0) {
          u.u_error = EBUSY;
          return;
     }
     tp->t_state |= CARR_ON;
     (*linesw[tp->t_line].l_open)(dev, tp);
}
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
/*
 * This procedure performs the close operation on one of the devices of the
 * isbc534. A close masks the device on board; reinstalls the flags that
 * state the device is closed; calls ttyclose the do the operation.
 * Not implemented yet is device 4 which is the parallel port; it is
 * unknown device at this minute.
 *
 * TITLE: i534close
 *
 * CALL:  i534close(dev, flag);
 *
 * INTERFACES: xenix
 *
 * CALLS: ttyclose
 *
 * History:
 *
 */


i534close(dev)
dev_t     dev;


{
     struct    db534     *DBbase;  /* set up the i/o boards base address */
     register struct tty *tp;
     register unit;
     register mask;
     int  s,

     unit = minor(dev) & MINORMSK;
     tp = &i534tty[unit],
     DBbase =  (tp->t_addr & 0xf0);
     if (unit < N534*4) {
          if(tp->t_state & HUPCLS) {
               tp->t_state &= ~CARR_ON;
               outb(tp->t_addr +1,SHANGUP);  /* dtr off */
          }
          (*linesw[tp->t_line].l_close)(tp);
          ttyclose(tp);
          unit%=4;
          s = SPL();
          mask = inb(&DBbase->PIC[0].msr) | (3 << (unit * 2));
          outb(&DBbase->PIC[0] msr, mask);   /* RxRDY, TxRDY off */
          splx(s);
     }
     tp->t_addr = (caddr_t) 0,
}


/*
 * This procedure interfaces the read request with the system read operation
 * to obtain a byte from the usart. The usart's character is read after an
 * interrupt so this procedure calls the system to wait for the interrupt
 * procedure to pass the character on to the input character queue.
 *
 * TITLE: i534read
 *
 * CALL:  i534read(dev)
 *
 * INTERFACES: xenix
 *
 * CALLS: ttread
 *
 * History:
 *
 */
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
i534read(dev)
dev_t     dev;

{
     register struct tty *tp;
     register int unit;

     unit=minor(dev) & MINORMSK;
     tp = &i534tty[unit];
      (*linesw[tp->t_line] l_read)(tp);
}


/*
 * This procedure is the compliment of the i534read routine. A call is
 * made to ttwrite which watches the output queue for characters and
 * gets the characters in the queue out to the device
 *
 * TITLE: i534write
 *
 * CALL:  i534write(dev),
 *
 * INTERFACES: xenix
 *
 * CALLS: ttwrite
 *
 * History:
 *
 */

i534write(dev)
dev_t     dev;

{
     register struct tty *tp;
     register int unit;

     unit=minor(dev) & MINORMSK;
     tp = &i534tty[unit];
     (*linesw[tp->t_line].l_write)(tp);
}


/*
 * This procedure is called by xenix with interrupts off (spl5) when the
 * isbc534 interrupts. The interrupt process polles the 8259's on the isbc534
 * to find out which device ; (if the device is a usart receiving it gets the
 * character)  then sends the character to ttyinput or restarts output by
 * calling ttstart depending on which interrupt was set off. Ttystart calls
 * i534start to make sure that no more characters need to be transmitted and
 * to let every body know a character has been transmitted.  The carrier detect,
 * ring indicator, present next digit and pit interrupt signals are not
 * implemented yet. The present next digit signal comes from the external
 * source on line 4.
 *
 * NOTE : all carrier detect signals both interrupt and latch on the 8255 ppi.
 *    Refer to the H/W manual for possible uses of these signals
 *    (ie ACU | printer applications).
 *    The rxrdy/txrdy lines from the older usarts (8251A/s2657 & older) cause
 *    giltches on the pic interrupt lines. This is a problem with the USART.
 *    If possible replace usart with a newer version.
 *
 *
 * TITLE: i534intr
 *
 * CALL:  i534intr(level);
 *
 * INTERFACES: xenix
 *
 * CALLS: ttyinput, ttstart
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
 *
 * History:    1/13/82: Condensed the usart Rxrdy/txrdy intr switch to
 *              run more efficiently using an if.. ; Added the
 *              unset of busy flag which gets set in 1534start.
 *         1/15/82: changed variable type to level which was incorrect.
 *              added multiple isbc634 support.
 *
 */
int  wakeup();

i534intr(level)
int  level;

{
     struct     db534    *DBbase;  /* set up the i/o boards base address */
     register struct tty *tp;
     register char  c;
     int   status,mask;       /* mask & status to/from PIC */
     int   gotone,board;

     do {
          gotone=0;
          for(board=0;board<N534;board++) {
               if(i534alive[board]) {
                    DBbase = i534base[board];
                    outb(&DBbase->PIC[0].csr, GETINT);
                    status = inb(&DBbase->PIC[0].csr);
                    if ((status & GOODINT) == GOODINT) {    /* check bit 8 for an int */
                         gotone++;
                         outb(&DBbase->PIC[0].csr, PIC_EOI);
                         status &= 0x07;                 /* mask off garbage bits */
                         tp = &i534tty[board*4] + (status >> 1);
                         if ((status & 0x01) == 0){          /*  Rxrdy intr */
                              c = inb(tp->t_addr);
                              (*linesw[tp->t_line].l_rint)(c, tp);
                         }else{                         /*  Txrdy intr */
                              tp->t_state &= ~BUSY;     /* the character is out */
                              (*linesw[tp->t_line].l_start)(tp); /* do the next one */
                              if((tp->t_state & ASLEEP) && (tp->t_outq.c_cc <= TTLOWAT)) {
                                   tp->t_state &= ~ASLEEP;
                                   wakeup((caddr_t)&tp->t_outq);
                              }
                         }
                    }
                    outb(&DBbase->PIC[1].csr, GETINT);
                    status = inb(&DBbase->PIC[1].csr);
                    if ((status & GOODINT) == GOODINT) {    /* check bit 8 for an int */
                         gotone++;
                         outb(&DBbase->PIC[1].csr, PIC_EOI);
                         status &= 0x07;                 /* mask off garbage bits */
                         if (status >= 4)
                              tp = &i534tty[board*4] + (status -4);
                         switch(status) {
                         case 0 :       /* pit 1 cntr 4 */
                              break;
                         case 1 :       /* pit 1 cntr 5 */
                              wakeup((caddr_t)&i534wakeup);
                              break;
                         case 2 :       /* ring ind all */
                              break;
                         case 3 :       /* present next */
                              break;
                         case 4 :       /* port 0 detect*/
                         case 5 :       /* port 1 detect*/
                         case 6 :       /* port 2 detect*/
                         case 7 :       /* port 3 detect*/
                              if((tp->t_state & (CARR_ON|ISOPEN))
                              == (CARR_ON|ISOPEN)) {
                                   signal(tp->t_pgrp, SIGHUP);
                                   tp->t_state &= ~CARR_ON;
                                   /* flick dtr off to cause
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                                        * hardware hang up on
                                        * modem
                                        */
                                       mask =    inb(&DBbase->PIC[i].msr)
                                                 | (1<<status);
                                       outb(&DBbase->PIC[i].msr, mask);
                                            /* carrier detect off */
                                       outb((tp->t_addr +i),SHANGUP);
                                  }
                                  break;

                             }
                        }
#ifdef DEBUG
     /* else  printf(''i534: Spurious Int level %d0, level); */
#endif
                        /* no interrupt from this device
                         * a call should be made to handle
                         * some form of accounting as this
                         * interrupt is probably caused by
                         * an out of date usart 8251A/s2657
                         * or older.(glitches occasionally
                         * the rxrdy/txrdy lines)
                         */
                   }
              }
     } while(gotone);

 * This procedure starts output on a usart if needed. i534start gets a
 * character from the character queue, outputs the character to the usart,
 * and sets the BUSY flag. The busy flag gets unset when the character
 * has been transmitted by i534intr().
 *
 * TITLE: i534start
 *
 * CALL:  i534start(tp)
 *
 * INTERFACES: ttystart
 *
 * CALLS: none
 *
 * History:    1/13/82: Removed the hardware probing for txrdy and added
 *              a set of the busy flag which gets unset on txrdy
 *              interrupt.
 *
 */
int ttrstrt();
char partab[];

i534start(tp)
register struct tty *tp;
{
     register c;
     register s;

#ifdef DEBUG
     if(i534debug>=3)
          printf(''i534start: called on unit at %x0, tp->t_addr);
#endif
     s = spl6();
     if (tp->t_state&(TIMEOUT|BUSY)) {
          splx(s);
          return;
     }
     splx(s);
     if ((c=getc(&tp->t_outq)) >= 0) {
          if (tp->t_flags & RAW) {
               outb(tp->t_addr, c);
          }else{
               if (c<=0x7f) {
                    outb(tp->t_addr, c | (partab[c]&0200));
               }else{
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                tp->t_state |= TIMEOUT;
                timeout(ttrstrt, (caddr_t)tp, (c&0x7f));
                return;         /* i'm timed out & !BUSY */
            }
        }
        tp->t_state |= BUSY;
}

/*
 * This procedure handles the ioctl system calls for such things as baud rate
 * changes and various hardware control changes from the initial set up.
 * Currently only baud rate changes are supported.
 *
 * TITLE: i5341octl
 *
 * CALL:  i5341octl(dev, cmd, addr, flag)
 *
 * INTERFACES: ioctl
 *
 * CALLS: i534param, ttioccomm
 *
 * History:
 *
 */

i5341octl(dev, cmd, addr, flag)
caddr_t addr;
{
     register struct tty *tp;
     register int unit;

     unit = minor(dev) & MINORMSK;
     tp = &i534tty[unit];
     if (ttioccomm(cmd, tp, addr, dev)) {
          if (cmd==TIOCSETP || cmd ==TIOCSETN)     /*if baud change do it*/
               i534param(dev);
     }else
          u.u_error = ENOTTY;
}
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

REFERENCES

1) Ritchie, Dennis M., *The Unix I/O System* , undated.

2) Scheulen, Bob, *Microsoft Device Driver Guide*, unpublished '82.

3) Letwin, Gordon, *Interrupt Structure*, unpublished (MICROSOFT) '82.

4) Short, Antony, *The XENIX I/O System*, unpublished (MICROSOFT) '82.

5) Beck, Bob, *The Anatomy of XENIX Device Drivers*, unpublished.

6) Byrant, McNamara, Vaish, *Writing Device Drivers*, UNIFORUM '84.

ACKNOWLEDGEMENTS

1) Jim Emmons, for hours of shared discussion on device drivers and for his iSBC 254 Bubble Memory Board Pseudo-Code.

2) Dilip Ratnam, Phil Barret, Jean McNamara Rick Byrant and other members of the XENIX team for sharing their ideas.

3) Vince Slyngstad, for his iSBX 270 Video Terminal Controller walkthrough.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

intel

APPLICATION NOTE

AP-221

October 1984

An Introduction to Task Management in the iRMX™ 86 Operating System

CATHERINE J. LUNDBERG
APPLICATIONS ENGINEERING

© INTEL CORPORATION, 1984

Order Number 280047-001

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

An Introduction to Task Management in the iRMX™ 86 Operating System

Contents

Introduction .....

iRMX™ 86 Operating System Nucleus Architecture .....
Memory Management .....
Task Scheduling .....

Single Task Application Example .....
Inittask .....
Onetask .....
Creating Tasks .....
Mailboxes .....
Using the System Debugger (SDB) ..
Exception Handling .....

Multiple Task Application Example ....
Main$task .....
Supervisor$task .....
Widget$task .....
IO$task .....
Mailboxes .....
Deadlock .....
Initialization .....
Debugging .....

Configuration .....
Configuring the iRMX™ 86 Operating System .....
Linking and Locating the Application .....

Conclusion .....

Appendix A: Inittask Code .....

Appendix B: Onetask Code .....

Appendix C: Tasks Code .....

Appendix D: Submit file .....

Appendix E: iRMX 86 Operating System Definition File .....

Appendix F: Related Publications ....

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

INTRODUCTION

The purpose of this application note is to help users understand the nucleus of the iRMX™ 86 Operating System, and how to use the nucleus. The other layers of the Operating System are not discussed. It is assumed that the reader has a basic understanding of the iRMX 86 Operating System, which can be gained by reading the product documentation. This application note does not discuss all areas of the nucleus with equal depth, so readers wishing to understand areas other than task scheduling should refer to the iRMX 86 Nucleus Reference Manual for more information. Related areas are covered in enough detail to provide the necessary background for understanding how to use tasks in the iRMX 86 Operating System.

This application note focuses on the nucleus and its task scheduling and resource management functions. The nucleus of the iRMX 86 Operating System must handle two major functions: task scheduling, which also involves interrupt handling; and resource management, in particular memory management. The iRMX 86 Operating System has other layers, which increase the functions provided by the Operating System and which rely on the nucleus as their base.

The iRMX 86 Operating System is a real time operating system which can have multiple jobs and tasks. It is a preemptive, priority based operating system. Since only one task can be executing on the central processor at any time, the task scheduler is the heart of the operating system.

The application note has two examples. The first example creates a single task to show what is involved in creating a task. The second example shows multitasking, creating three tasks. It demonstrates how the relative priorities of the tasks affect the way the application behaves, and is used as the basis for discussing deadlock between tasks.

iRMX™ 86 OPERATING SYSTEM NUCLEUS ARCHITECTURE

The nucleus of the iRMX 86 Operating System is essentially a resource manager. There are many resources which an operating system must handle. These can be divided into three areas for the iRMX 86 Operating System: processor time, objects and memory. The main concern in this application note is control of processor time, but the reader should remember that in controlling processor time, the nucleus also manages each of the other two areas.

Processor time is the key resource the iRMX 86 Operating System manages. The Operating System should use the processor as efficiently as possible. When a task requires the processor, that task is placed in the ready state. The processor always executes the highest priority ready task first. If more than one task of that priority is ready, the processor is allocated to the task that has been ready the longest. Once a task gains control of the processor, that task retains control until preempted by a higher priority task or interrupt, or the task gives up control.

Objects are the building blocks of the iRMX 86 Operating System. There are several types of objects: tasks, jobs, segments, mailboxes, semaphores, regions, extension objects, and composite objects. Tasks are the primary focus of this application note. However, the use of segments and mailboxes will also be discussed since they are used to communicate between the tasks in the example programs.

Memory usage is usually a critical factor in an application. It is desirable to use as little memory as possible, while still allowing the application to run efficiently. Insufficient memory can slow down the execution of a task or cause an error when executing the application.

Some other functions of the iRMX 86 Operating System that are implemented at the nucleus level are exception handling, interrupt management, and hardware manipulation of devices such as the programmable interrupt controller, the system clock, and the numeric data processor. These functions are not specifically discussed in this application note.

Memory Management

In the iRMX 86 Operating System, memory is partitioned into pools. That portion of the nucleus that allocates memory is called the free space manager. The root job, which is the first job in the Operating System and the ancestor of all other jobs, has a memory pool consisting of all available memory when the Operating System initializes. As jobs are initialized, they are allocated memory from the root job's pool for their own use.

Jobs which are created during system initialization, such as those that make up the iRMX 86 Operating System layers, are allocated memory based on configuration parameters. There are two parameters used in requesting memory. The first is the minimum memory pool size. This parameter specifies the minimum amount of memory that the job requires in order to run. The second parameter is maximum memory pool size. This parameter can either have the same value as the minimum pool size, or it can have some larger value. If it has a larger value, the job will be allowed to borrow memory dynamically from its parent or some more

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

distant ancestor. If the maximum memory pool is set to a smaller value than the minimum memory pool, an error will result. If the minimum and maximum memory pool sizes are the same, the job will be allocated that amount of memory and will not be allowed to borrow memory.

If there is sufficient memory to fulfill the request when a job is created, it is given the minimum amount of memory requested. If there is not enough memory, the job is not created and an error (E$MEM) will be returned to the creator. If the job is created and later needs more memory, the request will be honored up to the maximum memory pool size by borrowing. The memory is borrowed from the job's ancestors. A job with the same minimum and maximum memory pool size cannot borrow memory from its parent.

Usually, only one job created by the root job should be allowed to borrow memory. This prevents more than one job borrowing memory from the same memory pool. Multiple jobs borrowing from the same memory pool can cause deadlock between the jobs, so borrowing should be used with great caution. In an iRMX 86 Operating System with all the layers, the Human Interface is usually chosen to be the layer allowed to borrow memory.

Memory is allocated by the free space manager on a first come, first served basis. The job that is created first will receive the memory it requested if there is sufficient memory available to satisfy the minimum memory pool request. Then the next job created will request the memory it needs. If a job is not able to get as much memory as it needs, the operating system will return an E$MEM error to the creating task and the job will fail to be created. The free space manager will continue to try to allocate memory to the next job that is created.

The free space manager for Release 6 of the iRMX 86 Operating System keeps a sequential doubly linked list of the available segments of memory within each job pool. Each block of memory has a header which contains two links: one forward, and one backward. A pointer called the rover always points to the next entry of the linked list's unallocated memory.

When a memory request is made, the next memory entry in the linked list is checked to see if it is large enough. The first segment found which is large enough is allocated and removed from the free space manager's list. The rover points to the remainder of the segment just allocated. Memory is always allocated in contiguous segments, including allocating minimum memory pools. The rover keeps the lower portion of memory from becoming more fragmented than the upper portions. Using the rover and a first fit algorithm means that the average number of segments that must be checked is also decreased. (See Knuth: The Art of Computer Programming: Vol. 1 pp 435-453. In particular, refer to exercise 6 on p. 452 and its answer on p. 597.) When memory is returned to a pool, it is merged with existing segments when possible.

The Application Loader allocates memory to jobs at run time under control of the iRMX 86 Operating System. The OMF (Object Module Format) for the iAPX 86 processor has two areas in which minimum and maximum memory pools are specified. First are the program's minimum and maximum static memory requirements. These static memory requirements correspond to the code and data size of the job. The second area in which memory pools are specified are a minimum and maximum dynamic memory pool which are specified in the LINK86 process.

When the Application Loader allocates memory for a program, the Application Loader calls the Nucleus to create a memory pool as large as possible within the specified bounds from the user's memory pool. In Releases 5 and 6 of the iRMX 86 Operating System, with each failed attempt to allocate memory, the application loader decrements the size of the memory pool requested by 3% of the difference between the current size attempted and the minimum size. The Application Loader then tries again to allocate the memory. This approach is used so that the application is given the largest amount of memory possible in as few tries as possible, and so that the loading time is decreased.

Task Scheduling

Tasks are the active objects in an iRMX 86 Operating System, and they do all the work. They run inside of jobs, which provide the environment the tasks need, such as the memory pools. There are five possible execution states for an iRMX 86 task. These states are running, ready, asleep, suspended, and asleep-suspended. Nucleus system calls can change the state of a task. External events can also affect the state of the task. Figure 1 shows the state transition diagram for tasks.

Tasks can have different priorities. A numerically lower priority is a logically higher priority task. A task which has a logically higher priority will execute first if it is in the ready state. Tasks will be put on the ready list in priority order, and within a priority, the task which has been ready the longest will execute first.

Normal tasks are assigned priorities between 80H and 0FFH so that they can be serviced with minimum delay. Interrupts are usually at a higher priority than normal tasks, and will always interrupt

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

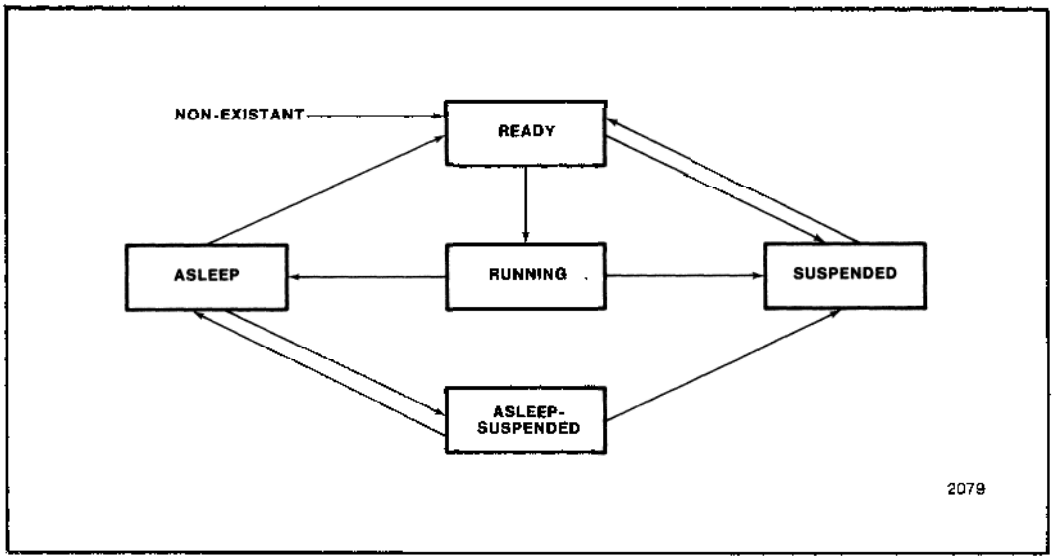


Figure 1. Task State Transition Diagram

the processor when they occur. The interrupt handler may be able to handle the interrupt directly, or it may invoke an interrupt task to handle the interrupt. The interrupt handler will retain control of the processor until the handler exits or a higher priority event occurs.

When mailboxes are used, queues of either tasks and objects can form at the mailbox. Task queues can form at semaphores. Task queues can be priority ordered or FIFO (First In First Out) ordered. This order is specified when the mailbox is created. A FIFO queue on a mailbox can cause a task with a lower priority to execute before a task with higher priority. If both tasks are waiting on the mailbox before continuing execution and the lower priority task is first on a FIFO queue, the lower priority task will execute first. However, when the higher priority task receives the object for which it was waiting, the task now becomes the ready task with the highest priority and can take control of the processor.

SINGLE TASK APPLICATION EXAMPLE

This section explains how to create a task, how to use mailboxes, and how to use the System Debugger. It also covers exception handling, as well as how to configure the iRMX 86 Operating System, and how to link and locate the application job.

The single task example shows how to create a single task that writes to the terminal. The structure of the operating system used for this application example is shown in Figure 2. The code has an initial module, called Inittask (Appendix A), which is used to provide a stable entry point for the application code. The entry point of Inittask is the start address for the task. In the User Job screen of the ICU, this value must be supplied for the task start address parameter. Inittask calls Onetask (Appendix B) which does all the work of the application. The submit file which links and locates the user job is given in Appendix D.

INITTASK

The initial module, shown in Appendix A, is very simple. It illustrates how to set up a stable entry point for a user job. There is no data in this module, and there are only two calls. Inittask is never changed, and it is linked first, so its entry point is stable no matter what changes are made to the rest of the application code. This approach allows the user job's entry point to be set up only once in the configuration of the operating system, and removes the need to generate a new operating system whenever the application code is changed.

The .MP2 file generated by the LOC86 utility shows that the module has the entry point Inittask at 1500:0002H. This address is used as the task start address in the definition file, in the User Job screen of the ICU. Inittask calls RQENDINIT$TASK, which is a requirement for any job created by the root job in the iRMX 86 Operating System. Calling RQENDINIT$TASK allows the root task to resume execution and create another first level job. Once RQENDINIT$TASK has been called,

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

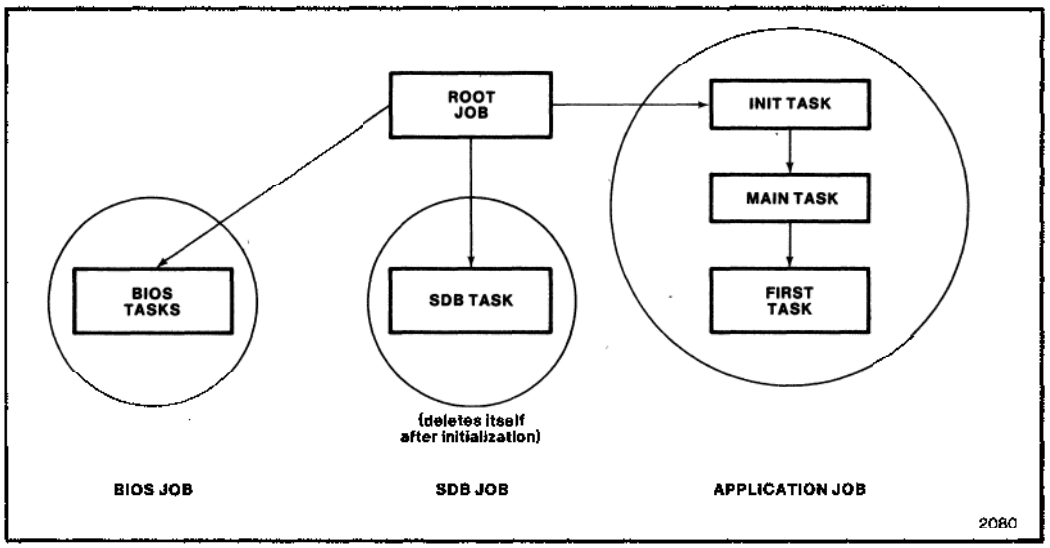


Figure 2. Single Task System

Inittask calls an external procedure called Main$task, which is the actual code that creates the example task. The same initial module is used with both application examples.

ONETASK

In Appendix B, the code for Onetask is shown. There are two parts to Onetask. First is the main module, called Main$task. Second is the task which is created by Main$task, called First$task. Note that there are really two tasks, only one of which is created specifically in the example code.

Main$task creates a mailbox and catalogs it in the user job's directory under the name DONEMBX. It uses this mailbox to synchronize Main$task and the task which is created. It creates First$task and then waits at the mailbox to receive a message from First$task to indicate that the task has finished. Main$task then deletes the task and deletes the mailbox.

Main$task deletes segments received from the mailbox, and then deletes the mailbox. In the code shown in Appendix B, there is a loop around creating and deleting the task, with the PL/M 86 call CAUSE$INTERRUPT (3) at each end of the loop. This code as used in this example was for debugging purposes, but could have also allowed stopwatch timing of the routine. Note also that using CAUSE$INTERRUPT (3) calls in your code will result in all other tasks in the system halting, including those of other users.

At the point where Main$task calls RQ$RECEIVE$-MESSAGE to wait at the mailbox, First$task begins to run. Main$task has a higher priority than First$task, so First$task cannot run until Main$task either suspends itself or goes to sleep. In this case, Main$task is waiting on the mailbox, which puts it in the sleeping state, and Main$task cannot continue until an object is received from the mailbox. This situation gives First$task a chance to run, since it is now the highest priority ready task.

First$task creates some mailboxes and segments, and does a lookup to find the mailbox DONEMBX which it must use to communicate with Main$task. First$task then physically attaches the terminal, opens a file connection, and writes a buffer to the terminal. Then it closes the connection, deletes the file connection, and detaches the device. It cleans up by deleting the segments and mailboxes it created, and signals Main$task that it is done by sending a message to the mailbox DONEMBX.

Main$task receives control of the processor after First$task sends a message to DONEMBX to indicate completion. Main$task then deletes the segments and mailboxes which are left, and then deletes the application job. All memory allocated to the application job will then be returned to the root job's memory pool.

If there was an error while running this application, the task would end up looping in one of the 'error' routines. If the application completed successfully and exited, the nucleus idle task for Release 6 of the iRMX 86 Operating System would begin executing.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

The Basic I/O System (BIOS) was used in this application to provide immediately visible results. The section which involves using the BIOS is the most complex part of the example code. Many applications will have no need of the BIOS.

These applications were done in the LARGE model of compilation to provide simpler examples. The COMPACT model can be used if the application's code and data are less than 64K each. COMPACT code can usually execute faster because calls will be within the same segment, so won't require changing as many registers to execute the call.

Creating Tasks

To illustrate all the areas that are involved in creating a task, let's go through each of the parameters of the RQ$CREATE$TASK system call. The call looks like this.

```
task$token = RQ$CREATE$TASK
                    (priority,
                     start$address$pointer,
                     data$segment,
                     stack$pointer,
                     stack$size,
                     task$flags,
                     exception$pointer);
```

The parameters in the RQ$CREATE$TASK call are explained below.

Priority: Task scheduling involves setting relative priorities of tasks. Unless a task is involved in processing interrupts, its priority should be between 129 and 255. When a task having a priority in the range 0 to 128 is running, certain external interrupt levels are disabled, depending on the priority. The task for this application used a priority of 202. The initial task itself was given a priority of 82H, or 130, at configuration time.

Start$address$pointer: The start address pointer is used to point to the beginning of the task which is being created. In the PL/M 86 LARGE and COMPACT models, the pointer points to the label of the procedure containing the task. The task was a procedure within the same main module for these examples. If the task had been compiled separately, it would have to be defined as an external procedure within the main module which created the task. The actual locations are resolved when the application is linked and located.

Data$segment: In the PL/M 86 LARGE model, the data segment is set equal to zero when creating the task. Setting the data segment to zero allows the task to set up its own data segment. In other models of PL/M 86, the user task must explicitly set up its own data segment or the value of the data segment must be obtained from the locate map and used in the call. Refer to the iRMX 86 Configuration Guide, which is part of the *iRMX 86™ Installation and Configuration Guide for Release 6* for more information on how to set up the data segment of a task.

Stack$pointer: The stack pointer is also set to zero to allow the iRMX 86 Operating System to automatically allocate a stack of size stack$size. While the task is running, the SS register will show which stack segment is being used for the application task.

Stack$size: The stack size will need to vary with stack requirements of the task. If the task is reentrant, or makes calls to subroutines with many parameters, or if the task makes iRMX 86 Operating System calls, the amount of stack must be larger than if the task only keeps local variables on the stack.

There are two ways to determine the stack size needed. The first method involves arithmetically determining the stack size needed, based on three things: the number of bytes required for interrupts, the number of bytes required for system calls, and the amount of stack required by the task's code segment. This method is explained in iRMX 86 Programming Techniques, which is part of the *iRMX™ 86 Programmer's Reference Manual for Release 6, Part II*. The other method involves choosing a relatively large stack size and reducing it through empirical methods. To use the empirical method, display the stack with a debugger. If there are "C7"s on the stack when the application has completed, that part of the stack hasn't been used. You can also watch the stack pointer, kept in the SP register, to see how low it goes. It will grow toward zero from the value given as the stack pointer when the task is created.

While testing the example application code, the stack size was set to 2000 (7D0H) which was much too large. A stack size of 300 was sufficient for this task.

Task$flags: Task flags are used in the iRMX 86 Operating System Releases 5 and 6 to tell the nucleus whether the task contains floating point instructions. This task did not, so task$flags was set to 0. Setting bit 0 of task$flags to 1 to indicate the use of floating point instructions will result in memory being reserved for the NPX registers. The other bits of task$flags are reserved. The iAPX 8087 or iAPX 80287 must

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

be included in the system if floating point instructions are used.

Exception$pointer: This pointer gives the location of the word where the status of this call will be returned. The result is checked after the call to make sure that an E$OK was returned. For convenience while debugging, the condition code can also be found in the CX register. The nucleus manual (Chapter 7 for iRMX 86 Release 6) contains a table of exceptional conditions that can be returned and their numeric codes.

Mailboxes

There are two object queues associated with every mailbox. One is a fast queue, which has a fixed length determined when the mailbox is created. The other object queue is an overflow queue, and memory must be allocated for that queue each time it is used. This example used one mailbox to communicate between the original task and the task it created. The fast queue has a length of 4, which is the default value, indicated by the 0 as the first parameter of the create$mailbox call. If this mailbox frequently had large numbers of objects on its queue, it might have been useful to use a larger fast queue. This approach, of course, means that more memory would be allocated to the mailbox when it is created. In this application, the minimum fast queue length was used.

The DONEMBX mailbox is being used as if it was a semaphore. It is used only to tell the parent task that the child task has completed. First$task also creates some mailboxes which are used to send information. For instance, when a physical connection is made to the terminal, the mailbox is used to receive a token for the physical connection. Similarly, when a file connection is made, the mailbox receives a file connection token.

Different protocols can be set up to handle objects which are received at a mailbox. In this application, after an object is received from a mailbox, that object is deleted after it has been used in an appropriate manner (such as to extract the file connection token). A protocol can also be set up so that objects are reused. A response can be sent to the task that sent the object. Only Operating System created objects can be sent to a mailbox. Objects should be deleted when they are no longer needed, because they take up memory. Creating extra objects and neglecting to delete them can eventually cause a task to use up all its available memory. The objects which are sent to mailboxes in this example are segments. Examples of how to check for the presence of objects are shown in the following section on using the System Debugger.

Using the System Debugger (SDB)

The System Debugger is a job which is added to the iRMX 86 Operating System at configuration time. The only configurable parameter in the SDB is its interrupt level, which in this example should be the default value of 018H, master interrupt level one. The SDB knows most of the data structures of the iRMX 86 Operating System, and can be used to determine what is happening within the Operating System while an application job is running.

The first thing that must be done when using the SDB is to activate it. If you are writing an application which can be run from the Human Interface, the DEBUG cusp can be used. The exampte application wasn't run from the Human Interface. Instead, it is built in as a user job. The SDB can be invoked by pressing the front panel interrupt button, or by inserting a 'CAUSE$INTERRUPT (3)' call into the source code. While bootloading an application, the 'CAUSE$INTERRUPT (3)' call is a more useful tool, since pressing the interrupt button cannot stop the application at a specific point. The monitor won't get control immediately as it would if the job were loaded from a development system. The 'CAUSE$INTERRUPT (3)' call can be taken out of the application when the code is debugged. Any use of the SDB should be done only in a single-user system, since the SDB will stop all jobs.

CAUSE$INTERRUPT (3) is used in three places in the single task application code. The first occurrence is as soon as the module Main$task is entered, but before any of its code has executed. The second place is before the loop to create and delete the task. The third place is after the loop is completed. During the debugging phase of developing the code, there were also CAUSE$INTERRUPT (3) calls within the task, to help determine what was happening. Breakpoints can also be used once the code has been stopped so that the monitor has control. A breakpoint is set by the monitor command 'g, address'. The code will execute until it gets to that address, and then it will break to the monitor.

The job tree is found by using the SDB command 'vj' for view job. An example follows for the application task.

```
.vj
BFDD
    B68A
    BE9E
```

BFDD is the token for the root job. By knowing the memory pools of each of the layers, and looking at the memory pools of each of the other job tokens, a user can determine that B68A is the user job, (the application code), and BE9E is the BIOS. The 'vt' command will show the current state of each token.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
.vt BE9E
Object type = 1      Job          (NOTE: This token is for the BIOS.)

Current tasks        0003    Max tasks     FFFF    Max priority    00
Current objects      000A    Max objects   FFFF    Parameter obj   BFB8
Directory size       0000    Entries used  0000    Job flags       0000
Except handler  0EE4:01C0    Except mode     00    Parent job      BFDD
Pool min             0800    Pool max      0800    Initial size    0800
Pool size            0800    Allocated     0077    Largest seg     0764

.vt B68A
Oject type = 1       Job          (NOTE: This token is for the application job.)

Current tasks        0001    Max tasks     0010    Max priority    00
Current objects      0001    Max objects   0020    Parameter obj   BFB8
Directory size       0010    Entries used  0001    Job flags       0001
Except handler  0EE4:01C0    Except mode     00    Parent job      BFDD
Pool min             0500    Pool max      0500    Initial size    0500
Pool size            0500    Allocated     01FD    Largest seg     0303
```

Notice from these tables that the application job may have been created with too much memory. It is only using 01FD of memory (Allocated), and has a minimum and a maximum pool size of 0500H (Pool min, Pool max). A pool size of 0250H would probably have been sufficient. If this command was executed early in the application, all the objects might not be created yet. So the job might require more memory than is currently being used at some point in its existence.

The command 'vo job-token' shows all the objects that have been created in a job. The token for each object contained by that job will be shown, listed after a designation for which type of object it is. You can check for the presence of leftover segments at the end of a task's execution with this command. Before executing the application code, the following list shows what the command and its results look like.

```
.vo B68A
Child Jobs:
Tasks:        B491
Mailboxes:
Semaphores:
Regions:
Segments:
Extensions:
Composites:
```

By using the SDB command 'vk', the tasks that are ready or sleeping can be seen. After running the user job to completion, the result of the 'vk' command looks like this:

```
.vk
Ready Tasks:
Sleeping Tasks:  BE64  BE2E  BE06  1CCF
```

If the code hadn't completed executing, results like this might indicate deadlock. The tasks would have to be examined to see how long they were asleep. If they are all asleep forever, nothing further will happen without an external event.

Exception Handling

In this example, the default system exception handler is used, and the exception mode is set to 'never'. This setting means that the application code either wishes to handle exceptions in-line or through a call to an exception handler. The system exception handler will not be invoked for errors. If the handler were invoked, it would simply delete the task containing the call which caused the exception.

This application handles exceptions in line. After each system call, the task checks the status word for E$OK. If an exception is detected, the task will jump to ERROR and loop there forever. This technique is to help identify where an error occurred, and is useful for debugging the application. The CX register contains the status returned from the call, so it is possible to find out which error occurred by using the monitor command 'x' to display the registers. The problem them becomes the following: to find out where the error occurred. This procedure usually involves stepping through the code, or setting several breakpoints (using the monitor commands to break at given points).

It is useful to insert 'CAUSE$INTERRUPT (3)' calls at points when the task is likely to transfer control (such as after sending or receiving messages). When the application is running properly, remove the 'CAUSE$INTERRUPT (3)' calls to allow the code to execute unattended. Breakpoints can also be used to monitor the code.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

If you write your own exception handler, you have to decide upon which conditions it will be invoked. You must compile, link and locate the exception handling code, and determine the starting address of the exception handler. This value must be configured into the operating system as the User Job's exception handler address. Each task can also have its own exception handler by using the call RQSETEXCEPTION$HANDLER.

An exception handler is the preferred method of handling exceptions, but exception handlers are beyond the scope of this application note. An exception handler would eliminate the need for GOTOs in the code. GOTOs are considered bad programming practice in most structured languages.

MULTIPLE TASK APPLICATION EXAMPLE

For the second example, the same initial code, Inittask, was used. The main module is called Main$task as it was for the single task example. The same configuration of the iRMX 86 Operating System is used, since the start address is the same for both applications. The code for this example, called Tasks, is shown in Appendix C. The main module for this example creates three tasks, and lets them do the work. A diagram of the system is shown in Figure 3.

The three tasks are set up according to their function. This very simple example of a machine control system makes the classic product, widgets. One task is the supervisor and controls the other two tasks. It sends messages to the other tasks to tell them what to write or how many widgets to make. The second task, called IO$task, outputs messages which it has received from Supervisor$task to the terminal. The third task, called Widget$task, makes widgets. In this example, Widget$task is essentially a no-op task, but it could easily be replaced with code that implemented a real application.

MAIN$TASK

The code in the second example has an initial module which creates three tasks, and then waits at a mailbox for them to complete execution. The initial module then deletes the three tasks and the mailboxes it has created. When everything is cleaned up, it deletes the application job. The initial task not only creates the mailbox it needs for signaling when the tasks are done, but it also creates the other four mailboxes that are used by the three tasks to communicate with each other. The mailboxes are cataloged in the user job's directory, and each task must look up the mailboxes it needs to use.

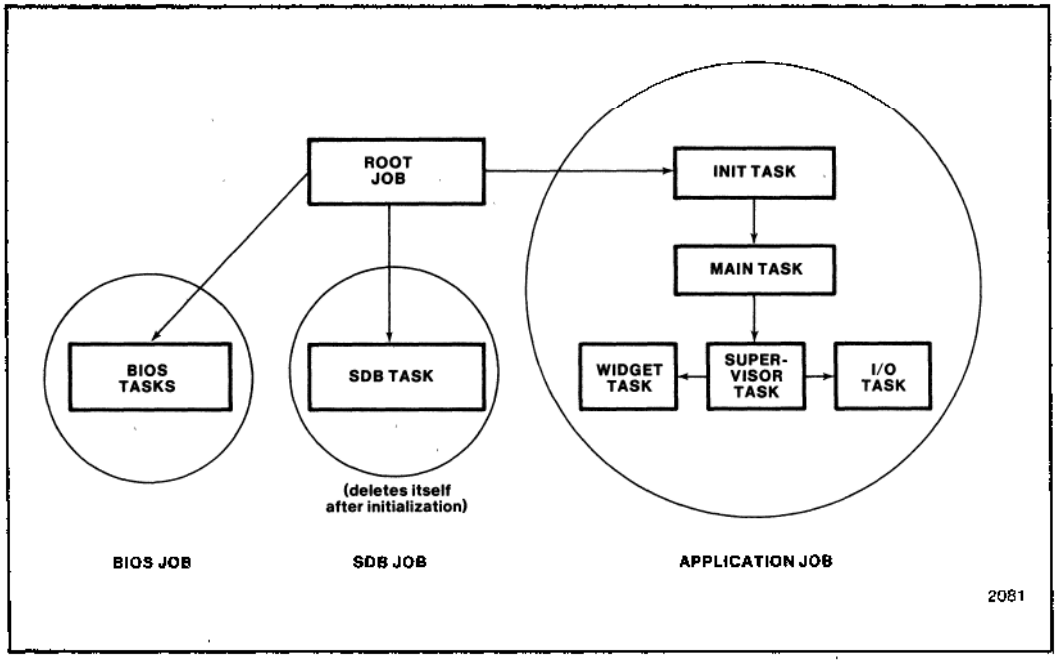


Figure 3. Multiple Task System

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Pseudocode for Main$task

```
create mailboxes
catalog mailboxes in job's directory
create Supervisor$task
create Widget$task
create IO$task
receive message from Supervisor$task (done)
delete tasks
delete mailboxes
delete myself
```

SUPERVISOR$TASK

Supervisor$task is created with the highest priority of the three tasks. It happens to be created first, but because it has the highest priority it would execute first regardless of when it was created. Supervisor$task controls the other two tasks. The other two tasks are created with priorities lower than both Supervisor$task and the creating task. If one of them had been created with a priority higher than the creating task and had been the first one created, it would have begun executing as soon as it was created, preventing the creator task from creating Supervisor$task.

Pseudocode for Supervisor$task

```
lookup mailboxes in job's directory
do 1 to 10
   send message to Widget$task (make widget)
   send message to IO$task (making widget, message)
   receive message from Widget$task (done)
   receive message from IO$task (done)
end
send message to Widget$task (cleanup)
send message to IO$task (cleanup)
receive message from Widget$task (done)
receive message from IO$task (done)
cleanup by deleting segments
send message to Main$task (done)
```

The supervisor allows Widget$task and IO$task to work as independently as possible. It sends messages to both of them, and then waits for both to reply before it repeats the loop. This approach allows the tasks to execute when they can get the processor, completely independently of each other. We can also look at the ready list at various points in the code and see which task is executing. Tasks will not always execute in the same order with this method. Because each task is required to wait for a message indicating that it may run, IO$task cannot inform the console that a widget is being made any sooner than Widget$task begins to make the widget. If the sends and receives had instead been paired (sending then receiving from the same task) Supervisor$task could have guaranteed which task would be executing at any given point in the code.

Supervisor$task is sending information in the mailboxes to each task. In the single task example, a simple segment with no information content was sent between the creating task and the created task to signal that the created task was done executing. A semaphore could have been used just as well in that example. In this multitasking example a semaphore would not work. Information is being passed to each of the tasks to tell them whether or not to continue executing, or that they should clean up their environments. In addition, Supervisor$task is passing to IO$task a message that will be printed out. To pass the message, a structure is used. The structure contains values, rather than tokens for objects. The values are moved into the structure with the MOVB PL/M call prior to sending the structure's token through the mailbox. Structures which contain tokens should be avoided, especially when using mailboxes which communicate between jobs. On more advanced processors than the iAPX 86, operating systems may be implemented in which jobs have disjoint address spaces. In that case, a token may have different values in different jobs. The iRMX 86 Operating System will expect this convention to be followed but will not enforce it. Future operating systems may enforce the convention. This example is completely contained within one job, so it isn't quite as restricted.

WIDGET$TASK

Widget$task is extremely simple for this example. This portion in a real application would probably be the most complex, since it would involve the machine interfaces for a control process. It could also include any mathematical calculations which need to be done.

Pseudocode for Widget$task

```
look up mailboxes
receive message from Supervisor$task (make
    widget)
do while make widget is true
   send message to Supervisor$task (done)
   receive message from Supervisor$task (make
      widget)
end
cleanup the environment
send message to Supervisor$task (done)
```

IO$TASK

This task is very similar to First$task in the first example. The main difference between the two is the way the information for the messages is given to the tasks. In Onetask, the message was defined within the task. In IO$task, the information for the message is passed to IO$task via a mailbox from Supervisor$task. This task illustrates how to do simple I/O with just the BIOS.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Pseudocode for IO$task

```
look up mailboxes
receive message from supervisor (making
     widget, message)
create user
physically attach terminal
get device connection
create file
get file connection
open file
receive message from Supervisor$task (making
     widget, message)
do while making widget is true
   write message (Making widget)
   delete segments
   send message to Supervisor$task (done)
   receive message from Supervisor$task (making
     widget, message)
end
close file
delete file connection
detach device
delete user
cleanup environment by deleting segments
send message to Supervisor$task (done)
```

Mailboxes

In this application, mailboxes are used for several purposes. Their most obvious purpose is to send information between tasks. Less obvious but more important is the role they play in allowing mutual exclusion and synchronization between tasks.

The simplest messages in the application return an empty segment to the mailbox to indicate that the task has completed some portion of work, and the receiving task can continue. This empty segment is the kind of message that Widget$task and IO$task send to Supervisor$task, and the kind of message that Supervisor$task sends to Main$task when it has completed execution. This exchange could be accomplished just as well by using a semaphore rather than a mailbox.

The more complex messages contain some information. In this example, the message contains information indicating that the task should continue executing a loop, or that it is time to clean up the environment and exit. The message that was sent between Supervisor$task and IO$task also contained the message that Supervisor$task wanted IO$task to print out. This example illustrates the kind of information that can be passed between tasks using mailboxes.

Mailboxes are also used in this application to implement mutual exclusion and synchronization between the tasks. One alternative implementation which is not shown here keeps the three tasks all at the same priority, and uses mailboxes to allow the tasks to execute in a strictly defined order. The implementation shown in this example is quite general purpose, and doesn't use as many mailboxes as the alternative implementation would. This implementation also allows the tasks to have more freedom in when they can run, and uses the processor more efficiently if one of the tasks is blocked while doing I/O.

Deadlock

As more tasks are used, and as more mailboxes are used to communicate between the tasks, the possibility of deadlock increases. Deadlock usually is caused by faulty design, and may appear when debugging of the code begins. Evidence of possible deadlock occurs when all the tasks are sleeping, and no tasks are ready when you use the SDB command 'vk' to check what's going on. This situation can be caused by sending a message to the wrong mailbox, or by not creating segments to send within a loop that is sending messages. An example of deadlock can be obtained by changing the code in Supervisor$task in some minor ways.

Executes correctly:

```
Do i = 1 to 10
   create segments
   send message to Widget$task (make widget)
   send message to IO$task (making widget, message)
   receive message from Widget$task (done)
   delete segment
   receive message from IO$task (done)
   delete segment
end
```

Causes deadlock:

```
create segments
Do i = 1 to 10
   send message to Widget$task (make widget)
   delete segment
   send message to IO$task (making widget, message)
   delete segment
   receive message from Widget$task (done)
   receive message from IO$task (done)
end
```

The second code example results in deadlock because the object which is sent to the mailbox is created outside the loop. Once it is sent, there is no longer an object to send, and the receiving task can't continue unless it has a timeout specified because it never receives another object. With a different synchronization scheme, if the receiving task hadn't deleted the message, that object could have been sent again.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Key places to watch for deadlock in the code are where some communication occurs between the tasks. Other situations that can cause deadlock are tasks needing the same resources, such as a unit from a semaphore, or a region. Insufficient memory will cause an E$MEM error rather than deadlock.

Initialization

Supervisor$task controls the other two tasks. This control is necessary since the tasks cannot execute more than a few lines of code without receiving a message from the supervisor. The tasks execute a few lines of code to lookup the mailbox which they will use to communicate with the other tasks. Then they can make the RQ$RECEIVE$MESSAGE call. The task with the highest priority which was created first, and as a result, has been ready the longest, will execute first. Since Supervisor$task was created with a higher priority than the other two tasks, it will execute first. By the time it gives up control of the processor, it has already sent messages to both of the other tasks. The task that has been ready the longest at this point will execute first. In this example, the first executed task happens to be Widget$task, since it was created before IO$task was.

Debugging

The same techniques are used to debug a multiple task application as were discussed in the single task example. Look at the .MP2 file before beginning debugging, and find the entry points to each task. The .MP2 file is produced as a result of the LOC86 step in building the application. Use 'CAUSE$INTERRUPT (3)' calls at the beginning of each task, and keep track of which task is executing at a given time. One technique that was used in this application to make it easier to debug was to send all errors to an error routine within each task. The error routine was different in each task (outputting A1H for the first task, A2H for the second task, etc.). It was immediately obvious by looking at the disassembled code containing the call which task caused the exception.

As a second debug alternative, the iRMX 86 Dynamic Debugger is also useful in a multiple task application. Rather than halting the entire system like the SDB does, the Dynamic Debugger allows users to examine vital system objects while the system is running. The Dynamic Debugger must be configured into the Operating System with its own terminal handler and its own terminal if the BIOS is used.

CONFIGURATION

There are two steps involved in configuration: configuring the operating system; and compiling, linking and locating the application code. The same definition file was used for both applications, and the same submit file was used to link and locate the applications.

Configuration of the iRMX™ 86 Operating System

For both examples in this application note, the same operating system configuration was used. The layers used are the Nucleus (for scheduling and intertask communication) and the BIOS (so that I/O could be done). The SDB, for debugging the code, was also configured into the operating system as a user job. The only device driver required is the terminal driver. The listing of the operating system definition file is shown in Appendix E.

User Jobs

This application code is configured into the operating system as a user job. The Interactive Configuration Utility (ICU) requires information to be given about the user job, and sets up a %JOB macro for the job. However, the ICU does not set aside memory for the user job, and it does not link and locate the job as it does for the layers of the operating system.

An application that uses only nucleus and BIOS calls is a user job. If the application uses the EIOS or the Human Interface, it is an I/O job and must be configured as a child job of the EIOS. Applications can also be run from the Human Interface level, as jobs under the Human Interface. In a real-time application, treating the application as a user job or user jobs is usually most appropriate. During development, however, the application could be run under the Human Interface. This technique would eliminate rebooting after each code change or trial run to test the application.

The following parameters appear in the User Jobs screen in the ICU. Each parameter is defined and explained in the context of the example application.

JOB NAME (NAM)

The first question in the User Jobs screen for the iRMX 86 Release 6 ICU is Job Name. This question is optional, and is just used for the user to keep track of which user job is being configured in the screen. It is not used by the ICU.

OBJECT DIRECTORY SIZE (ODS)

Object directory size refers to how many objects can be cataloged in the job's directory. In the first example, the only object that is being cataloged is

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

the mailbox DONEMBX which is used to let the main task know that First$task has completed. In the second example, five mailboxes are cataloged, so the object directory size doesn't have to be very big for this application. The default value for this parameter is 10H, which is large enough for this application. If you have a large application and many objects are cataloged, this number would have to be increased. For small applications, a small value can be used to conserve memory. Memory is allocated for the object directory of a job when the operating system is initialized.

POOL MINIMUM, POOL MAXIMUM (PMI, PMA)

Pool minimum and pool maximum are closely related. They specify the amount of memory that the job requests from its parent. For this example, a pool minimum and pool maximum size of 500H was chosen. The minimum and maximum should be set to the same value in the user job to avoid memory fragmentation. The Human Interface is usually the only exception to this rule.

MAXIMUM OBJECTS (MOB)

This parameter defines how many objects can be created by the application. For this application, the maximum objects was set to 20H, but only 10 objects existed at any given time. In a larger application, of course, 20H could easily become insufficient. To determine how many objects are being used at a given point in time, use the SDB command 'vo user-job-token'.

MAXIMUM TASKS (MTK)

The next parameter specifies the maximum number of tasks which can be created by tasks in this user job. This application used the default value of 10H, but could have used 4H, since only three tasks were created (one task was created by the root job to be the user job task). The definition file was being used by both the single task example and the multiple task example, so the default value is appropriate.

MAXIMUM PRIORITY (MPR)

The maximum priority parameter refers to the maximum priority allowed of any task in this job which is created. This parameter was set to 0H where 0H indicates that the priority of the root job is the maximum allowable priority of the tasks. The root job's priority in this operating system is 00H, which doesn't limit the maximum allowable priority.

ADDRESS OF EXCEPTION HANDLER (AEH)

This application is using the system exception handler, so the address of exception handler used is 0000H:0000H. If the user job had its own exception handler, the correct address of that exception handler would have to be found by first linking and locating the application code and the exception handler, and then looking in the .MP2 file for the address of the exception handler.

EXCEPTION MODE (EM)

The exception mode is set to 'never' for this user job, indicating that the exception handler won't be invoked for any kind of error condition. Instead, exceptions will be handled in-line in the example code.

PARAMETER VALIDATION (PV)

Parameter validation is used by the nucleus to determine if the parameters passed in system calls are valid. This question should be answered yes until the application code is debugged. If the BIOS or other upper layers are being used in the operating system, parameter validation should be enabled even when the application code is debugged, or the Operating System will not work. If parameter validation is turned off, the nucleus calls will execute faster, so setting parameter validation to 'no' can improve performance.

TASK PRIORITY (TP)

Task priority sets up the static priority of the initial task which is created for the user job. For this application, the priority of the task is set to 82H, or 130.

TASK START ADDRESS (TSA)

The Task Start Address is the start address of the job's initialization task. This address is determined from the .MP2 file after locating the application code. For both example applications this address was 1500H:0002H.

DATA SEGMENT BASE (DSB)

The data segment base is set to 0000H, which allows the task to set up the data segment base for the initial task of the user job. Since the LARGE model of compilation was used, the parameter can be set to zero.

STACK SEGMENT ADDRESS (SSA)

The stack segment address is also set to 0000H:0000H to allow the nucleus to allocate a stack segment to the task and take care of initializing the SS and SP registers. This setting permits dynamic stack allocation and deallocation.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

STACK SIZE (SS)

The stack size for the task is set to 300, which is the amount that is considered necessary to make any nucleus system calls. Since this application was very small, it didn't need a very large stack. If a job used a lot of subroutines and nested procedures with many parameters, or if the job was recursive, the amount of stack needed could increase.

NUMERIC PROCESSOR EXTENSION USED (NPX)

The 8087 Numeric coprocessor was not used in this application task. If any floating point functionality is needed within a task, this parameter should be set to yes in the configuration process.

Ram

The last parameter which must be considered when creating the definition file for the application is the amount of RAM required and where it is located. This is a parameter in the memory screen of the ICU. Remember that the ICU does not locate the user job for you, so memory must be specifically set aside to be used by the user job. In this application, the RAM that was used was from 0104H to 1500H, and from 1800H to F7FFH. The user job was allowed to use RAM from 1500H to 1800H (these numbers are specified in paragraphs of 16 bytes each). The operating system itself was put into the memory from 104H to just under 1500H. This location can be determined by looking at the .MP2 files for each of the layers of the operating system after completing the configuration.

Linking and Locating the Application

The submit file that was used for this application is shown in Appendix D. Note first of all that instead of using the name of the application code program, a %0 was used. This convention allows you to invoke the submit file with a parameter which is the name of the application code program. The same submit file was used for both examples. Note that Inittask is linked with the INITCODE option. This is necessary with LINK86 v 2.0 and LOC86. The INITCODE option is not necessary with other versions of LINK86.

While the submit file is running, some warnings will be generated. The following errors are normal and should be ignored.

WARNING 12: UNRESOLVED SYMBOLS

WARNING 26: DECREASING SIZE OF SEGMENT
SEGMENT: STACK

WARNING 66: START ADDRESS NOT SPECIFIED IN OUTPUT MODULE

CONCLUSION

This application note is an introduction to the basic functions of the iRMX 86 Operating System Nucleus. Task scheduling and memory management functions were covered in detail. Two applications were discussed, using some pieces of code in common. The functions involved in developing and testing real-time code were explained while using the application code for examples and reference. Configuration of the application operating system was also covered in detail.

The examples shown in this application note illustrate how to develop a real-time application. The first example shows how to use the nucleus and BIOS to do simple I/O at the lowest, most optimizable level. The second example builds on the concepts developed in the first example, expanding the application to a more realistic process control situation.

Both examples shown earlier are fairly simple. They illustrate what has to be done to create and use a task. They are good examples for a user who has written a limited amount of PL/M 86 code using the nucleus system calls. The multiple task example would be a good foundation for a process control application. Each of the tasks in the second example shows a major function of real time code, demonstrating control, I/O, and supervisory functions.

The iRMX 86 Operating System is ideally suited to multi-tasking, real-time applications. The ability to use the same system for both development and as the target system is a great benefit to the development engineer and the company which is developing real-time applications. The modularity provided by the iRMX 86 Operating System and the PL/M 86 language make it easier to develop code for one application, then modify it for another application. The ultimate benefits to users are reduced development time, added cost savings, and shorter time-to-market for new products.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX A: .....
APPENDIX B: .....
APPENDIX C: .....
APPENDIX D: .....
APPENDIX E: .....
APPENDIX F: .....

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX A

```
PL/M-86 COMPILER    single task creation; for ap note

iRMX 86 PL/M-86 V2.1 COMPILATION OF MODULE INITTASK
OBJECT MODULE PLACED IN INITTASK.OBJ
COMPILER INVOKED BY:  :LANG:plm86 INITTASK.P86


                $large rom debug
                $title('single task creation; for ap note')

                /*********************************************************
                 *  This is an example to be used for an ap note on task *
                 *  scheduling.                                          *
                 *  Cathy Lundberg 03/22/84                              *
                 *********************************************************/
   1            inittask: do;
                $INCLUDE (/RMX86/INC/NEINIT.EXT)
          =     $SAVE NOLIST

   4    1       main$task: PROCEDURE EXTERNAL;
   5    2       END main$task;

                /*********************************************************
                 *  This separate module is used to keep the user job's  *
                 *  start address constant while changing the code.      *
                 *  This module has no data or constants in it, all it   *
                 *  does is call the main routine, main$task, after      *
                 *  calling rq$end$init$task.                            *
                 *********************************************************/

   6    1       begin: PROCEDURE PUBLIC;

   7    2       CALL rq$end$init$task;
   8    2       CALL main$task;
   9    2       END begin;

  10    1       END inittask;


MODULE INFORMATION:

     CODE AREA SIZE     = 0018H     24D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0008H      8D
     36 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
DICTIONARY SUMMARY:

     112KB MEMORY AVAILABLE
     3KB MEMORY USED   (2%)
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX B

```
PL/M-86 COMPILER     single task creation; for ap note

iRMX 86 PL/M-86 V2.1 COMPILATION OF MODULE ONETASK
OBJECT MODULE PLACED IN ONETASK.OBJ
COMPILER INVOKED BY:  :LANG:PLM86 ONETASK.P86


                $large rom debug
                $title('single task creation; for ap note')
   1            onetask: DO;

                /*************************************************************
                 *  This is an example to be used for an ap note on task *
                 *  scheduling.                                          *
                 *  Cathy Lundberg 03/22/84                              *
                *************************************************************/

  /* The include for LTKSEL.LIT must be done before any other includes,
   * because anything that uses TOKENS must have the data type defined
   * for a token.  */

                $INCLUDE (/RMX86/INC/LTKSEL.LIT)
           =    $SAVE NOLIST
                $INCLUDE (/RMX86/INC/NEXCEP.LIT)
           =    $save nolist
                $INCLUDE (/RMX86/INC/NUC.EXT)
           =    $SAVE NOLIST
                $INCLUDE (/RMX86/INC/BIOS.EXT)
           =    $SAVE NOLIST

/*****************************************************************************
 *  main$task is the procedure that is called by inittask to create the  *
 *  task 'first$task'.  It creates a mailbox, creates the task, and then *
 *  waits at the mailbox, allowing first$task to execute. When first$task*
 *  finishes and sends a message to the mailbox, control is returned to  *
 *  main$task, and it deletes first$task and the mailbox.                *
 *****************************************************************************/

280    1       main$task:  PROCEDURE REENTRANT PUBLIC;

281    2       DECLARE job                 TOKEN,
                       data$seg            TOKEN,
                       user$token          TOKEN,
                       taska               TOKEN,
                       done$writing$mbx    TOKEN,
                       resp                TOKEN;

282    2       DECLARE task$flags          WORD,
                       status              WORD,
                       i                   WORD;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
283   2      DECLARE seg$pointer          POINTER,
                     start$address         POINTER,
                     taska$ptr             POINTER,
                     stack$pointer         POINTER;
284   2      DECLARE stack$size$300       LITERALLY   '300';
285   2      DECLARE priority$level$202   LITERALLY   '202';
286   2      DECLARE iors$token           SELECTOR;
287   2      DECLARE done$obj     TOKEN   AT(@iors$token);
288   2      DECLARE iors         BASED   iors$token     STRUCTURE
                        (status           WORD,
                         unit$status      WORD,
                         actual           WORD);

289   2         data$seg      = 0;    /* task sets up own data segment */
290   2         stack$pointer = 0;    /* automatic stack allocation */
291   2         task$flags    = 0;    /* no floating point */
292   2         job           = 0;    /* catalog object in containing job */

293   2      CAUSE$INTERRUPT (3);
             /* create a mailbox to use for letting this task know
              * that the task it created is done writing. */
294   2         done$writing$mbx = rq$create$mailbox (0, @status);
295   2         IF status <> E$OK THEN GOTO error;
297   2         CALL rq$catalog$object (0, done$writing$mbx, @(7,'DONEMBX'),
                                          @status);
298   2         IF status <> E$OK THEN GOTO error;
300   2         CAUSE$INTERRUPT (3);
301   2         DO I = 1 TO 1000;
                     /* Now create taska */
302   3            taska = rq$create$task (priority$level$202, @first$task,
                                          data$seg, stack$pointer,
                                          stack$size$300, task$flags, @status);
303   3            IF status <> E$OK THEN GOTO error;

305   3            done$obj = rq$receive$message (done$writing$mbx, 0ffffh,
0,
                                          @status);
306   3            IF status <> E$OK THEN GOTO error;

308   3            CALL rq$delete$segment (done$obj, @status);
309   3            IF status <> E$OK THEN GOTO error;
311   3            CALL rq$delete$task (taska, @status);
312   3            IF status <> E$OK THEN GOTO error;
314   3         END; /* of DO WHILE loop */
315   2         CAUSE$INTERRUPT (3);
316   2         CALL rq$delete$mailbox (done$writing$mbx, @status);
317   2         IF status <> E$OK THEN GOTO error;
319   2         GOTO ok;

320   2      error:                  /* output to usart to determine which */
                  DO WHILE 1;        /* task had the error.  For debugging.*/
321   3            OUTPUT(9CH) = 0AAH;
322   3         END;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
323   2       ok:

                  CALL rq$delete$job (0, @status); /* delete myself */

324   2       END main$task;


/*************************************************************************
*  FIRST$TASK is the task which is created by main$task.  It creates     *
*  the necessary mailboxes and segments, and then attaches the terminal  *
*  physically.  It then creates a file so that it has a file connection.*
*  It opens the file connection, and writes the contents of a buffer to  *
*  the terminal.  Then it closes the file connection, deletes the        *
*  device connection, and detaches the device.  Last, it looks up the    *
*  mailbox created in main$task and sends a message to the mailbox.      *
*  This allows control to return to main$task.                           *
*************************************************************************/


325   1       first$task: PROCEDURE REENTRANT PUBLIC;

326   2       DECLARE  job                TOKEN,
                       mbx$token          TOKEN,
                       seg$token          TOKEN,
                       user$token         TOKEN,
                       file$connection    TOKEN,
                       device$connection  TOKEN,
                       done$token         TOKEN,
                       done$writing$mbx   TOKEN;
327   2       DECLARE  status             WORD;
328   2       DECLARE  hard               BYTE;
329   2       DECLARE  iors$token         TOKEN;
330   2       DECLARE  dev$conn$object    TOKEN  AT (@iors$token),
                       file$conn$object   TOKEN  AT (@iors$token),
                       object             TOKEN  AT (@iors$token);
331   2       DECLARE  iors     BASED     iors$token     STRUCTURE
                      (status             WORD,
                       unit$status        WORD,
                       actual             WORD);
332   2       DECLARE  buffer             BASED          seg$token (1)  BYTE;

333   2       DECLARE  user$object        STRUCTURE
                      (length             WORD,
                       count              WORD,
                       id (1)             WORD);
334   2       DECLARE message(*)          BYTE    DATA   ('SINGLE TASK TEST');

335   2          user$object.length = 1;
336   2          user$object.count  = 1;
337   2          user$object.id(0)  = OFFFFH;

338   2          job = 0;       /* catalog object in containing job */
339   2          hard = OffH;   /* request a hard detach of the device */
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
340   2         user$token = rq$create$user (@user$object, @status);
341   2         IF status <> E$OK THEN GOTO error;
343   2         mbx$token = rq$create$mailbox (0, @status);
344   2         IF status <> E$OK THEN GOTO error;
346   2         seg$token = rq$create$segment ( 48, @status);
347   2         IF status <> E$OK THEN GOTO error;

349   2         CALL rq$a$physical$attach$device ( @(2,'T0'), 1, mbx$token,
                                @status);
350   2         IF status <> E$OK THEN GOTO error;
352   2         dev$conn$object = rq$receive$message (mbx$token, OFFFFH, 0,
                                @status);
353   2         IF status <> E$OK THEN GOTO error;
355   2         device$connection = dev$conn$object;

356   2         CALL rq$a$create$file (user$token, device$connection,
                                  0, 0, 0, 0, 0, mbx$token, @status);
357   2         IF status <> E$OK THEN GOTO error;
359   2         file$conn$object = rq$receive$message (mbx$token, OFFFFH, 0,
                                    @status);
360   2         IF status <> E$OK THEN GOTO error;
362   2         file$connection = file$conn$object;

363   2         CALL rq$a$open (file$connection, 2, 0, mbx$token, @status);
364   2         IF status <> E$OK THEN GOTO error;
366   2         object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
367   2         IF status <> E$OK THEN GOTO error;
369   2         CALL rq$delete$segment (object, @status);
370   2         IF status <> E$OK THEN GOTO error;

372   2         CALL movb( @message, @buffer, SIZE(message));
373   2         CALL rq$a$write (file$connection, @buffer, size(message),
                                mbx$token, @status);
374   2         IF status <> E$OK THEN GOTO error;
376   2         object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
377   2         IF status <> E$OK THEN GOTO error;
379   2         CALL rq$delete$segment (object, @status);
380   2         IF status <> E$OK THEN GOTO error;

382   2         CALL rq$a$close (file$connection, mbx$token, @status);
383   2         IF status <> E$OK THEN GOTO error;
385   2         object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
386   2         IF status <> E$OK THEN GOTO error;
388   2         CALL rq$delete$segment (object, @status);
389   2         IF status <> E$OK THEN GOTO error;

391   2         CALL rq$a$delete$connection (file$connection, mbx$token,
                                    @status);
392   2         IF status <> E$OK THEN GOTO error;
394   2         object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
395   2         IF status <> E$OK THEN GOTO error;
397   2         CALL rq$delete$segment (object, @status);
398   2         IF status <> E$OK THEN GOTO error;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
400   2          CALL rq$a$physical$detach$device (device$connection, hard,
                                                   mbx$token, @status);
401   2          IF status <> E$OK THEN GOTO error;
403   2          object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
404   2          IF status <> E$OK THEN GOTO error;
406   2          CALL rq$delete$segment (object, @status);
407   2          IF status <> E$OK THEN GOTO error;

409   2          CALL rq$delete$mailbox (mbx$token, @status);
410   2          IF status <> E$OK THEN GOTO error;

412   2          CALL rq$delete$user (user$token, @status);
413   2          IF status <> E$OK THEN GOTO error;
415   2          CALL rq$delete$segment (seg$token, @status);
416   2          IF status <> E$OK THEN GOTO error;
418   2          done$token = rq$create$segment ( 16, @status);
419   2          IF status <> E$OK THEN GOTO error;
421   2          done$writing$mbx = rq$lookup$object (0, @(7,'DONEMBX'), 500,
                                                     @status);
422   2          IF status <> E$OK THEN GOTO error;
424   2          CALL rq$send$message (done$writing$mbx, done$token, 0,
                                                       @status);
425   2          IF status <> E$OK THEN GOTO error;
427   2          CALL rq$delete$segment (done$token, @status);  /* this code */
428   2          IF status <> E$OK THEN GOTO error;           /* shouldn't be*/
430   2          CALL rq$delete$mailbox (done$writing$mbx, @status);
431   2          IF status <> E$OK THEN GOTO error;             /* executed  */
433   2          GOTO ok;

434   2        error:                    /* output to usart for debugging */
              DO WHILE 1;
435   3          OUTPUT(9CH) = 0AAH;
436   3        END;

437   2        ok:
              DO;
438   3          CALL rq$suspend$task (0, @status );  /* suspend myself */
439   3          END;
440   2        END first$task;
441   1        END onetask;
```

```
MODULE INFORMATION:

     CODE AREA SIZE     = 0554H   1364D
     CONSTANT AREA SIZE = 0023H     35D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0040H     64D
     1520 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
DICTIONARY SUMMARY:

     84KB MEMORY AVAILABLE
     20KB MEMORY USED    (23%)
     0KB DISK SPACE USED


END OF PL/M-86 COMPILATION
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX C

```
PL/M-86 COMPILER     three task creation; for ap note

iRMX 86 PL/M-86 V2.1 COMPILATION OF MODULE TASKS
OBJECT MODULE PLACED IN TASKS.OBJ
COMPILER INVOKED BY:  :LANG:PLM86 TASKS.P86


                $large rom debug
                $title('three task creation; for ap note')
   1            tasks: DO;

                /*************************************************
                 *  This is an example to be used for an ap note on task *
                 *  scheduling.                                  *
                 *  Cathy Lundberg 04/23/84                      *
                 *************************************************/

                /* The include for LTKSEL.LIT must be done before any other
                 * includes, because anything that uses TOKENS must have the
                 * data type defined for a token.  */

                $INCLUDE (/RMX86/INC/LTKSEL.LIT)
            =   $SAVE NOLIST
                $INCLUDE (/RMX86/INC/NEXCEP.LIT)
            =   $save nolist
                $INCLUDE (/RMX86/INC/NUC.EXT)
            =   $SAVE NOLIST
               $INCLUDE (/RMX86/INC/BIOS.EXT)
            =   $SAVE NOLIST


/*************************************************************************
 *  MAIN$TASK is the procedure that is called by inittask to create the  *
 *  three tasks.  IO$task outputs to the screen.  Widget$task makes      *
 *  widgets.  Supervisor$task controls what IO$task and widget$task are  *
 *  doing.  When widget$task has made X widgets, supervisor$task sends   *
 *  a message to IO$task saying that X widgets have been made, and the   *
 *  message is printed by IO$task.  Then it sends a message to           *
 *  main$task to tell it that the tasks are done executing.  Main$task   *
 *  has been waiting at a mailbox for that welcome news, and when it     *
 *  receives the message, it deletes all three tasks.                    *
 *************************************************************************/


280    1      main$task:  PROCEDURE REENTRANT PUBLIC;

281    2      DECLARE data$seg              TOKEN,
                      IO$task$token         TOKEN,
                      widget$task$token     TOKEN,
                      supervisor$task$token TOKEN,
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                        main$super$mbx          TOKEN,
                        write$msg$mbx           TOKEN,
                        start$write$mbx         TOKEN,
                        done$write$mbx          TOKEN,
                        start$widget$mbx        TOKEN,
                        done$widget$mbx         TOKEN;
282   2       DECLARE   task$flags              WORD,
                        status                  WORD,
                        i                       WORD;
283   2       DECLARE   stack$pointer           POINTER;
284   2       DECLARE   stack$size$300          LITERALLY   '300';
285   2       DECLARE   priority$level$202      LITERALLY   '202';
286   2       DECLARE   priority$level$190      LITERALLY   '190';
287   2       DECLARE   iors$token              SELECTOR;
288   2       DECLARE   done$obj      TOKEN     AT(@iors$token);
289   2       DECLARE   iors          BASED     iors$token     STRUCTURE
                        (status                 WORD,
                         unit$status            WORD,
                         actual                 WORD);

290   2           data$seg      = 0;            /* task sets up own data segment */
291   2           stack$pointer = 0;            /* automatic stack allocation */
292   2           task$flags    = 0;            /* no floating point */

293   2           CAUSE$INTERRUPT (3);
                  /* create a mailbox to use for letting this job know
                   * that the tasks are done executing. */
294   2           main$super$mbx = rq$create$mailbox (0, @status);
295   2           IF status <> E$OK THEN GOTO error;
297   2           start$write$mbx = rq$create$mailbox (0, @status);
298   2           IF status <> E$OK THEN GOTO error;
300   2           done$write$mbx = rq$create$mailbox (0, @status);
301   2           IF status <> E$OK THEN GOTO error;
303   2           start$widget$mbx = rq$create$mailbox (0, @status);
304   2           IF status <> E$OK THEN GOTO error;
306   2           done$widget$mbx = rq$create$mailbox (0, @status);
307   2           IF status <> E$OK THEN GOTO error;
309   2           CALL rq$catalog$object (0, main$super$mbx, @(9,'MAINSUPER'),
                                            @status);
310   2           IF status <> E$OK THEN GOTO error;
312   2           CALL rq$catalog$object (0, start$write$mbx,
                                       @(10,'STARTWRITE'), @status);
313   2           IF status <> E$OK THEN GOTO error;
315   2           CALL rq$catalog$object (0, done$write$mbx, @(9,'DONEWRITE'),
                                            @status);
316   2           IF status <> E$OK THEN GOTO error;

318   2           CALL rq$catalog$object (0, start$widget$mbx,
                                      @(11,'STARTWIDGET'), @status);
319   2           IF status <> E$OK THEN GOTO error;
321   2           CALL rq$catalog$object (0, done$widget$mbx,
                                            @(10,'DONEWIDGET'), @status);
322   2           IF status <> E$OK THEN GOTO error;
324   2           CAUSE$INTERRUPT (3);
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                /* Now create the tasks. */
325    2           supervisor$task$token = rq$create$task (priority$level$190,
                        @supervisor$task, data$seg, stack$pointer,
                        stack$size$300, task$flags, @status);
326    2           IF status <> E$OK THEN GOTO error;
328    2           widget$task$token = rq$create$task (priority$level$202,
                                  @widget$task, data$seg, stack$pointer,
                                  stack$size$300, task$flags, @status);
329    2           IF status <> E$OK THEN GOTO error;
331    2           IO$task$token = rq$create$task (priority$level$202, @IO$task,
                                    data$seg, stack$pointer,
                                    stack$size$300, task$flags, @status);
332    2           IF status <> E$OK THEN GOTO error;

334    2           done$obj = rq$receive$message (main$super$mbx, OFFFFH, 0,
                                                  @status);
335    2           IF status <> E$OK THEN GOTO error;
337    2           CAUSE$INTERRUPT (3);

338    2           CALL rq$delete$segment (done$obj, @status);
339    2           IF status <> E$OK THEN GOTO error;
341    2           CALL rq$delete$task (IO$task$token, @status);
342    2           IF status <> E$OK THEN GOTO error;
344    2           CALL rq$delete$task (widget$task$token, @status);
345    2           IF status <> E$OK THEN GOTO error;
347    2           CALL rq$delete$task (supervisor$task$token, @status);
348    2           IF status <> E$OK THEN GOTO error;
350    2           CAUSE$INTERRUPT (3);
351    2           CALL rq$delete$mailbox (main$super$mbx, @status);
352    2           IF status <> E$OK THEN GOTO error;
354    2           CALL rq$delete$mailbox (start$write$mbx, @status);
355    2           IF status <> E$OK THEN GOTO error;
357    2           CALL rq$delete$mailbox (done$write$mbx, @status);
358    2           IF status <> E$OK THEN GOTO error;
360    2           CALL rq$delete$mailbox (start$widget$mbx, @status);
361    2           IF status <> E$OK THEN GOTO error;
363    2           CALL rq$delete$mailbox (done$widget$mbx, @status);
364    2           IF status <> E$OK THEN GOTO error;
366    2           GOTO ok;
367    2        error:            /* output to usart for debugging */
                    DO WHILE 1;
368    3               OUTPUT(9CH) = 0AAH;
369    3           END;

370    2        ok:

                    CALL rq$delete$job (0, @status);   /* myself */

371    2        END main$task;
/*************************************************************************
 *  This is the task which is controlling the other two tasks.  It has *
 *  higher priority than them now, but could have the same priority    *
 *  since it uses the mailboxes to synchronize the tasks.              *
 *************************************************************************/
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
372   1     supervisor$task:  PROCEDURE REENTRANT PUBLIC;

373   2     DECLARE done$token            TOKEN,
                    main$super$mbx        TOKEN,
                    start$write$mbx       TOKEN,
                    done$write$mbx        TOKEN,
                    start$widget$mbx      TOKEN,
                    done$widget$mbx       TOKEN,
                    cleanup$token         TOKEN,
                    make$widget$token     TOKEN;
374   2     DECLARE status                WORD,
                    i                     WORD;
375   2     DECLARE cleanup$ptr           BASED     cleanup$token (1)  BYTE,
                    make$widget$ptr       BASED make$widget$token (1)  BYTE;
376   2     DECLARE cleanup$d             BYTE      DATA(0),
                    make$widget$d         BYTE      DATA(1),
                    message$d(*)          BYTE      DATA('Making widget'),
                    making$widget$d       BYTE      DATA(1);
377   2     DECLARE send$to$io$token      TOKEN;
378   2     DECLARE send$to$io    BASED      send$to$io$token  STRUCTURE(
                                             making$widget (1) BYTE,
                                             message      (13) BYTE);

379   2       CAUSE$INTERRUPT (3);
380   2       main$super$mbx = rq$lookup$object (0, @(9,'MAINSUPER'),
                                  OFFFFH, @status);
381   2       IF status <> E$OK THEN GOTO error;
383   2       start$write$mbx = rq$lookup$object (0, @(10,'STARTWRITE'),
                                    OFFFFH, @status);
384   2       IF status <> E$OK THEN GOTO error;

386   2       done$write$mbx = rq$lookup$object (0, @(9,'DONEWRITE'),
                                  OFFFFH, @status);
387   2       IF status <> E$OK THEN GOTO error;
389   2       start$widget$mbx = rq$lookup$object (0, @(11,'STARTWIDGET'),
                                    OFFFFH, @status);
390   2       IF status <> E$OK THEN GOTO error;
392   2       done$widget$mbx = rq$lookup$object (0, @(10,'DONEWIDGET'),
                                    OFFFFH, @status);
393   2       IF status <> E$OK THEN GOTO error;

395   2       DO i = 1 TO 10;
396   3          CAUSE$INTERRUPT (3);
                   /*  Send message to widget task, mailbox STARTWIDGET */
397   3          make$widget$token = rq$create$segment (SIZE(make$widget$d),
                                                  @status);
398   3          IF status <> E$OK THEN GOTO error;
400   3          CALL MOVB (@make$widget$d, @make$widget$ptr,
                         SIZE(make$widget$d));
401   3          CALL rq$send$message (start$widget$mbx, make$widget$token,
                               0, @status);
402   3          IF status <> E$OK THEN GOTO error;
404   3          CAUSE$INTERRUPT (3);
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
                    /*  Send message to IO task, mailbox STARTWRITE */
405   3              send$to$io$token = rq$create$segment ( 32, @status);
406   3              IF status <> E$OK THEN GOTO error;
408   3              CALL MOVB (@message$d, @send$to$io.message,
                         SIZE(message$d));
409   3              CALL MOVB (@making$widget$d, @send$to$io.making$widget,
                           SIZE(making$widget$d));
410   3              CALL rq$send$message (start$write$mbx, send$to$io$token, 0,
                                       @status);
411   3              IF status <> E$OK THEN GOTO error;
413   3              CAUSE$INTERRUPT (3);

                    /*  Receive message from widget task, mailbox DONEWIDGET  */
414   3              done$token = rq$receive$message (done$widget$mbx, OFFFFH,
                                                    0, @status);
415   3              IF status <> E$OK THEN GOTO error;
417   3              CAUSE$INTERRUPT (3);
418   3              CALL rq$delete$segment (done$token, @status);
419   3              IF status <> E$OK THEN GOTO error;

                    /*  Receive message from IO task, mailbox DONEWRITE  */
421   3              done$token = rq$receive$message (done$write$mbx, OFFFFH, 0,
                                                    @status);
422   3              IF status <> E$OK THEN GOTO error;
424   3              CAUSE$INTERRUPT (3);
425   3              CALL rq$delete$segment (done$token, @status);
426   3              IF status <> E$OK THEN GOTO error;
                  /* receiving messages from the tasks, they are done and we  *
                   * are ready to tell them to clean up.                      */
428   3        END;

                     /*  Send message to widget task, mailbox STARTWIDGET */
429   2          cleanup$token = rq$create$segment ( SIZE(cleanup$d), @status);
430   2          IF status <> E$OK THEN GOTO error;
432   2          CALL MOVB (@cleanup$d, @cleanup$ptr, SIZE(cleanup$d));
433   2          CALL rq$send$message (start$widget$mbx, cleanup$token, 0,
                                      @status);
434   2          IF status <> E$OK THEN GOTO error;
436   2          CAUSE$INTERRUPT (3);

                   /*  Send message to IO task, mailbox STARTWRITE */
437   2           send$to$io$token = rq$create$segment ( 32, @status);
438   2           IF status <> E$OK THEN GOTO error;
440   2           CALL MOVB (@message$d, @send$to$io.message, SIZE(message$d));
441   2           CALL MOVB (@cleanup$d, @send$to$io.making$widget,
                          SIZE(cleanup$d));
442   2           CALL rq$send$message (start$write$mbx, send$to$io$token, 0,
                            @status);
443   2           IF status <> E$OK THEN GOTO error;

                    /*  Receive message from widget task, mailbox DONEWIDGET  */
445   2           done$token = rq$receive$message (done$widget$mbx, OFFFFH, 0,
                                                  @status);
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
446   2         IF status <> E$OK THEN GOTO error;
448   2         CAUSE$INTERRUPT (3);
449   2         CALL rq$delete$segment (done$token, @status);
450   2         IF status <> E$OK THEN GOTO error;

                /*  Receive message from IO task, mailbox DONEWRITE  */
452   2         done$token = rq$receive$message (done$write$mbx, OFFFFH, 0,
                                                 @status);
453   2         IF status <> E$OK THEN GOTO error;
455   2         CAUSE$INTERRUPT (3);
456   2         CALL rq$delete$segment (done$token, @status);
457   2         IF status <> E$OK THEN GOTO error;

459   2         CAUSE$INTERRUPT (3);

                /*  Send message to main task, mailbox MAINSUPER  */
460   2         done$token = rq$create$segment ( 16, @status);
461   2         IF status <> E$OK THEN GOTO error;
463   2         CALL rq$send$message (main$super$mbx, done$token, 0, @status);
464   2         IF status <> E$OK THEN GOTO error;

466   2         CALL rq$suspend$task (0, @status);  /* suspend myself */
467   2         IF status <> E$OK THEN GOTO error;
469   2         GOTO ok;

470   2       error:                  /* output to usart for debugging */
              DO WHILE 1;
471   3         OUTPUT(9CH) = 0A3H;
472   3         END;

473   2       ok:
              CALL rq$suspend$task (0, @status) /* myself */
            END supervisor$task;

/*************************************************************************
 *  IO$task is a task which is created by main$task.  It creates         *
 *  the necessary mailboxes and segments, and then attaches the terminal *
 *  physically.  It then creates a file so that it has a file connection.*
 *  It opens the file connection, and writes the contents of a buffer to *
 *  the terminal.  Then it closes the file connection, deletes the       *
 *  device connection, and detaches the device.                          *
 *************************************************************************/
474   1       IO$task: PROCEDURE REENTRANT PUBLIC;

475   2       DECLARE  mbx$token          TOKEN,
                       seg$token          TOKEN,
                       user$token         TOKEN,
                       file$connection    TOKEN,
                       device$connection  TOKEN,
                       cleanup$token      TOKEN,
                       done$token         TOKEN,
                       make$widget$token  TOKEN,
                       continue$token     TOKEN,
                       done$write$mbx     TOKEN,
                       start$write$mbx    TOKEN;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
476   2       DECLARE  status              WORD;
477   2       DECLARE  hard                BYTE;
478   2       DECLARE  iors$token          TOKEN;
479   2       DECLARE  cleanup$ptr           BASED     cleanup$token (1)  BYTE;
480   2       DECLARE  send$to$io$token      TOKEN;
481   2       DECLARE  send$to$io     BASED      send$to$io$token  STRUCTURE(
                                                 making$widget (1) BYTE,
                                                 message      (13) BYTE);
482   2       DECLARE  dev$conn$object     TOKEN  AT (@iors$token),
                       file$conn$object    TOKEN  AT (@iors$token),
                       msg$received$obj    TOKEN  AT (@iors$token),
                       object              TOKEN  AT (@iors$token);
483   2       DECLARE  iors      BASED     iors$token     STRUCTURE
                      (status              WORD,
                       unit$status         WORD,
                       actual              WORD);
484   2       DECLARE  buffer              BASED          seg$token (9)  BYTE;
485   2       DECLARE  user$object         STRUCTURE
                      (length              WORD,
                       count               WORD,
                       id (1)              WORD);


486   2          user$object.length = 1;
487   2          user$object.count  = 1;
488   2          user$object.id(0)  = OFFFFH;

489   2          hard = OffH;        /* request a hard detach of the device */

490   2          CAUSE$INTERRUPT (3);
                   /*  Set up the mailboxes for this task to use  */
491   2          start$write$mbx = rq$lookup$object (0, @(10,'STARTWRITE'),
                                    OFFFFH, @status);
492   2          IF status <> E$OK THEN GOTO error;
494   2          done$write$mbx = rq$lookup$object (0, @(9,'DONEWRITE'),
                                    OFFFFH, @status);
495   2          IF status <> E$OK THEN GOTO error;

                   /*  Receive message from supervisor task, mailbox STARTWRITE
*/
497   2          send$to$io$token = rq$receive$message (start$write$mbx,
                                    OFFFFH, 0, @status);
498   2          IF status <> E$OK THEN GOTO error;
500   2          CAUSE$INTERRUPT (3);

501   2          user$token = rq$create$user (@user$object, @status);
502   2          IF status <> E$OK THEN GOTO error;
504   2          mbx$token = rq$create$mailbox (0, @status);
505   2          IF status <> E$OK THEN GOTO error;
507   2          seg$token = rq$create$segment ( 48, @status);
508   2          IF status <> E$OK THEN GOTO error;

510   2          CALL rq$a$physical$attach$device ( @(2,'T0'), 1, mbx$token,
                                                   @status);
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
511   2         IF status <> E$OK THEN GOTO error;
513   2         dev$conn$object = rq$receive$message (mbx$token, OFFFFH, 0,
                                                      @status);
514   2         IF status <> E$OK THEN GOTO error;
516   2         device$connection = dev$conn$object;

517   2         CALL rq$a$create$file (user$token, device$connection,
                                    0, 0, 0, 0, 0, mbx$token, @status);
518   2         IF status <> E$OK THEN GOTO error;
520   2         file$conn$object = rq$receive$message (mbx$token, OFFFFH, 0,
                                                     @status);
521   2         IF status <> E$OK THEN GOTO error;
523   2         file$connection = file$conn$object;

524   2         CALL rq$a$open (file$connection, 2, 0, mbx$token, @status);
525   2         IF status <> E$OK THEN GOTO error;
527   2         object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
528   2         IF status <> E$OK THEN GOTO error;
530   2         CALL rq$delete$segment (object, @status);
531   2         IF status <> E$OK THEN GOTO error;

/************************************************************************
 *  This is the part of the code that will be repeated.  It receives    *
 *  the message from second task and writes it to the screen, then      *
 *  returns control to the second task.                                 */

533   2         DO WHILE send$to$io.making$widget(0) = 1;
534   3            CAUSE$INTERRUPT (3);
535   3            CALL rq$a$write (file$connection, @send$to$io.message,
                                  size(send$to$io.message),
                                  mbx$token, @status);
536   3            IF status <> E$OK THEN GOTO error;
538   3            object = rq$receive$message (mbx$token, OFFFFH, 0,
                                               @status);
539   3            IF status <> E$OK THEN GOTO error;
541   3            CALL rq$delete$segment (object, @status);
542   3            IF status <> E$OK THEN GOTO error;
544   3            CALL rq$delete$segment (send$to$io$token, @status);
545   3            IF status <> E$OK THEN GOTO error;

                   /*  Send message to supervisor task, mailbox DONEWRITE  */
547   3            done$token = rq$create$segment ( 16, @status);
548   3            IF status <> E$OK THEN GOTO error;
550   3            CALL rq$send$message (done$write$mbx, done$token, 0,
                                         @status);
551   3            IF status <> E$OK THEN GOTO error;
553   3            CAUSE$INTERRUPT(3);

                   /* Receive message from supervisor task, mailbox
                    * STARTWRITE  */
554   3            send$to$io$token = rq$receive$message (start$write$mbx,
                                                OFFFFH, 0, @status);
555   3            IF status <> E$OK THEN GOTO error;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
557   3           CAUSE$INTERRUPT (3);
558   3         END;

559   2         CALL rq$delete$segment (send$to$io$token, @status);
560   2         IF status <> E$OK THEN GOTO error;
562   2         CALL rq$a$close (file$connection, mbx$token, @status);
563   2         IF status <> E$OK THEN GOTO error;
565   2         object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
566   2         IF status <> E$OK THEN GOTO error;
568   2         CALL rq$delete$segment (object, @status);
569   2         IF status <> E$OK THEN GOTO error;
571   2         CALL rq$a$delete$connection (file$connection, mbx$token,
                                              @status);
572   2         IF status <> E$OK THEN GOTO error;
574   2         object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
575   2         IF status <> E$OK THEN GOTO error;
577   2         CALL rq$delete$segment (object, @status);
578   2         IF status <> E$OK THEN GOTO error;

580   2         CALL rq$a$physical$detach$device (device$connection, hard,
                                                    mbx$token, @status);
581   2         IF status <> E$OK THEN GOTO error;
583   2         object = rq$receive$message (mbx$token, OFFFFH, 0, @status);
584   2         IF status <> E$OK THEN GOTO error;
586   2         CALL rq$delete$segment (object, @status);
587   2         IF status <> E$OK THEN GOTO error;

589   2         CALL rq$delete$mailbox (mbx$token, @status);
590   2         IF status <> E$OK THEN GOTO error;

592   2         CALL rq$delete$user (user$token, @status);
593   2         IF status <> E$OK THEN GOTO error;
595   2         CALL rq$delete$segment (seg$token, @status);
596   2         IF status <> E$OK THEN GOTO error;

                /*  Send message to supervisor task, mailbox DONEWRITE  */
598   2         done$token = rq$create$segment ( 16, @status);
599   2         IF status <> E$OK THEN GOTO error;
601   2         CALL rq$send$message (done$write$mbx, done$token, 0, @status);
602   2         IF status <> E$OK THEN GOTO error;
604   2         CAUSE$INTERRUPT(3);
605   2         GOTO ok;

606   2       error:
              DO WHILE 1;
607   3           OUTPUT(9CH) = 0A1H;
608   3         END;

609   2       ok:
              CALL rq$suspend$task (0, @status);

610   2       END IO$task;

611   1       widget$task:  PROCEDURE REENTRANT PUBLIC;
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
612   2       DECLARE done$token          TOKEN,
                      continue$token      TOKEN,
                      cleanup$token       TOKEN,
                      start$widget$mbx    TOKEN,
                      done$widget$mbx     TOKEN,
                      make$widget$token    TOKEN;
613   2       DECLARE status              WORD;
614   2       DECLARE cleanup$ptr          BASED     cleanup$token (1) BYTE,
                      continue$ptr         BASED     continue$token (1) BYTE,
                      make$widget$ptr  BASED     make$widget$token (11) BYTE;

615   2        CAUSE$INTERRUPT (3);
616   2        start$widget$mbx = rq$lookup$object (0, @(11,'STARTWIDGET'),
                                                    OFFFFH, @status);
617   2        IF status <> E$OK THEN GOTO error;
619   2        done$widget$mbx = rq$lookup$object (0, @(10,'DONEWIDGET'),
                                                    OFFFFH, @status);
620   2        IF status <> E$OK THEN GOTO error;

               /* Receive message from supervisor task, mailbox STARTWIDGET */
622   2        make$widget$token = rq$receive$message (start$widget$mbx,
                                          OFFFFH, 0, @status);
623   2        IF status <> E$OK THEN GOTO error;

625   2        DO WHILE make$widget$ptr(0) = 1;  /*  Repeat endlessly.
                   *  Let supervisor task take care of controlling. */
626   3           CAUSE$INTERRUPT (3);
627   3           CALL rq$delete$segment (make$widget$token, @status);
628   3           IF status <> E$OK THEN GOTO error;

                  /*  Send message to supervisor task, mailbox DONEWIDGET  */
630   3           done$token = rq$create$segment ( 16, @status);
631   3           IF status <> E$OK THEN GOTO error;
633   3           CALL rq$send$message (done$widget$mbx, done$token, 0,
                                        @status);
634   3           IF status <> E$OK THEN GOTO error;
636   3           CAUSE$INTERRUPT(3);

              /* Receive message from supervisor task, mailbox STARTWIDGET */
637   3              make$widget$token = rq$receive$message (start$widget$mbx,
                                            OFFFFH, 0, @status);
638   3              IF status <> E$OK THEN GOTO error;

640   3        END;  /* do making the widgets */

641   2        CAUSE$INTERRUPT (3);
               /* cleanup the environment now.  */
642   2        CALL rq$delete$segment (make$widget$token, @status);
643   2        IF status <> E$OK THEN GOTO error;

               /*  Send message to supervisor task, mailbox DONEWIDGET  */
645   2        done$token = rq$create$segment ( 16, @status);
646   2        IF status <> E$OK THEN GOTO error;
648   2        CALL rq$send$message (done$widget$mbx, done$token, 0,
                                      @status);
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
649   2          IF status <> E$OK THEN GOTO error;
651   2          CAUSE$INTERRUPT(3);
652   2          GOTO ok;

653   2       error:    /* output to usart for debugging */
                   DO WHILE 1;
654   3                OUTPUT(9CH) = 0A2H;
655   3             END;

656   2       ok:
                   CALL rq$suspend$task (0, @status);

657   2       END widget$task;

658   1       END tasks;


MODULE INFORMATION:

     CODE AREA SIZE     = 0D1BH   3355D
     CONSTANT AREA SIZE = 00A9H    169D
     VARIABLE AREA SIZE = 0000H      0D
     MAXIMUM STACK SIZE = 0044H     68D
     1796 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     84KB MEMORY AVAILABLE
     22KB MEMORY USED   (26%)
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX D

```
PLM86 %0.P86
PLM86 inittask.P86
LINK86 inittask.OBJ to inittask.lnk initcode
LINK86 %0.OBJ to %0.lnk1 initcode
link86 inittask.LNK, &
       %0.LNK1, &
       /lib/ndp87/dcon87.lib, &
       /lib/ndp87/cel87.lib, &
       /lib/ndp87/eh87.lib, &
       /lib/ndp87/8087.lib, &
       /lib/rmx86/epifl.lib, &
       /lib/rmx86/ipifl.lib, &
       /LIB/RMX86/RPIFL.LIB, &
       /LIB/PLM86/PLM86.LIB &
      TO %0.LNK
LOC86 %0.LNK TO %0 &
 SEGSIZE(STACK(0)) ADDRESSES(CLASSES(CODE(015000H),DATA(017000H))) NOINITCODE
lib86
delete /boot/%0(inittask)
add %0 to /boot/%0
e
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

APPENDIX E

```
ICU86 V2.0  ONETASK.DEF                                   05/21/84 07:04:32

Hardware
(CPU) Processor used in the system                 8086
(OSP) 80130 Operating System Extension [Yes/No]    No
(TP)  8253/8254 Timer Port [0-0FFFFH]              00D0H
(CIL) Clock Interrupt Level [0-7]                  0002H
(CN)  Timer Counter Number [0,1,2]                 0000H
(CI)  Clock Interval [0-0FFFFH msec]               000AH
(CF)  Clock Frequency [0-0FFFFH khz]               04CDH
(TPS) Timer Port Separation [0-0FFH]               0002H
(NPX) Numeric Processor Extension [Yes/No]         No
(NIL) NPX Interrupt Level [Encoded]                0008H


Interrupts
(MP)  8259A Master Port [0-0FFFFH]                 00C0H
(MPS) Master PIC Port Separation [0-0FFH]          0002H
(SIL) Slave Interrupt Levels [0-7/None]            None
(LSS) Level Sensitive Slaves [0-7/None]            None


Memory
Type : RAM = low, high
Type : ROM = low, high
Type : RAM = 0104H, 1500H
Type : RAM = 1800H, F7FFH


Sub-systems
(UDI) Universal Development Interface [Yes/No]     No
(HI)  Human Interface [Yes/No]                     No
(AL)  Application Loader [Yes/No]                  No
(EIO) Extended I/O System [Yes/No]                 No
(BIO) Basic I/O System [Yes/No]                    Yes
(SDB) System Debugger [Yes/No]                     Yes
(DDB) Dynamic Debugger [Yes/No]                    No
(TH)  Terminal Handler [Yes/No]                    No
(CA)  Crash Analyzer [Yes/No]                      No


BIOS
(ASC) All Sys Calls in BIOS [Yes/No]               Yes
(ADP) Attach Device Task Priority [1-0FFH]         0081H
(TF)  Timing Facilities Required [Yes/No]          Yes
(TTP) Timer Task Priority [0-0FFH]                 0081H
(CON) Connection Job Delete Priority [0-0FFH]      0082H
(ACE) Ability to Create Existing Files [Yes/No]    Yes
(SMI) System Manager ID [Yes/No]                   Yes
(CUT) Common Update Timeout [0-0FFFFH]             03E8H
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
(CST) Control-Sequence Translation [Yes/No]      Yes
(OSC) Terminal OSC Controls [Yes/No]             Yes
(TS)  Tape Support for iSBC 215G [Yes/No]        No
(PMI) BIOS Pool Minimum [0-0FFFFH]               0800H
(PMA) BIOS Pool Maximum [0-0FFFFH]               0800H


8251A Driver
(IIL) Input Interrupt Level [Encoded]            0068H
(OIL) Output Interrupt Level [Encoded]           0078H
(UDP) USART Data Port [0-0FFFFH]                 00D8H
(USP) USART Status Port [0-0FFFFH]               00DAH
(IRP) 8253 Inrate Port [0-0FFFFH]                00D4H
(ICP) 8253 Input Control Port [0-0FFFFH]         00D6H
(IRC) 8253 Input Counter Number [0-2]            0002H
(IRF) Inrate Frequency [0-0FFFFFFFFH]            0012C000H
(ORP) 8253 Outrate Port [0-0FFFFH]               0000H
(OCP) 8253 Output Control Port [0-0FFFFH]        0000H
(ORC) 8253 Output Counter Number [0-2]           0000H
(ORF) Outrate Frequency [0-0FFFFFFFFH]           00000000H


8251A Unit Information
(NAM) Unit Info Name [1-17 Chars]                t0info
(LEM) Line Edit Mode [Trans/Normal/Flush]        Normal
(ECH) Echo Mode [Yes/No]                         Yes
(IPC) Input Parity Control [Yes/No]              Yes
(OPC) Output Parity Control [Yes/No]             Yes
(OCC) Output Control in Input [Yes/No]           Yes
(OSC) OSC Controls [Both/In/Out/Neither]         Both
(DUP) Duplex Mode [Full/Half]                    Full
(TRM) Terminal Type [CRT/Hard Copy]              CRT
(MC)  Modem Control [Yes/No]                     No
(RPC) Read Parity Checking [See Help/0-3]        0000H
(WPC) Write Parity Checking [See Help/0-4]       0000H
(BR)  Baud Rate [0-0FFFFH]                       2580H
(SN)  Scroll Number [0-0FFFFH]                   0017H


251A Device-Unit Information
(NAM) Device-Unit Name [1-13 chars]                    T0
(UN)  Unit Number on this Device [0-0FFH]              0000H
(UIN) Unit Info Name [1-17 Chars]                      t0info
(MB)  Max Buffers [0-0FFH]                             0000H


System Debugger
(SLV) SDB Interrupt Level [Encoded Level/None]   0018H


Nucleus
(ASC) All Sys Calls [Yes/No]                     Yes
(PV)  Parameter Validation [Yes/No]              Yes
(ROD) Root Object Directory Size [0 - 0FF0h]     0020H
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
(MTS) Minimum Transfer Size [0-0FFFFH]           0040H
(DEH) Default Exception Handler [Yes/No/Deb/Use] Yes
(NEH) Name of Ex Handler Object Module [1-32chs]
(EM)  Exception Mode [Never/Program/Environ/All] Never
(SRR) Start Root job from Reset [Yes/No]         No


User Jobs
(NAM) Job Name [0-14 characters]                 onetask
(ODS) Object Directory Size [0-0FF0H]            0010H
(PMI) Pool Minimum [20H - 0FFFFH]                0500H
(PMA) Pool Maximum [20H - 0FFFFH]                0500H
(MOB) Maximum Objects [1 - 0FFFFH]               0020H
(MTK) Maximum Tasks [1 - 0FFFFH]                 0010H
(MPR) Maximum Priority [0 - 0FFH]                0000H
(AEH) Address of Exception Handler [CS:IP]       0000H:0000H
(EM)  Exception Mode [Never/Prog/Environ/All]    Never
(PV)  Parameter Validation [Yes/No]              Yes
(TP)  Task Priority [0-0FFH]                     0082H
(TSA) Task Start Address [CS:IP]                 1500H:0002H
(DSB) Data Segment Base [0-0FFFFH]               0000H
(SSA) Stack Segment Address [SS:SP]              0000H:0000H
(SS)  Stack Size [0-0FFFFH]                      1F40H
(NPX) Numeric Processor Extension Used [Yes/No]  No


User Modules
Module : 1-55 characters


ROM code
(BIR) Basic I/O System in ROM [Yes/No]           No
(SIR) SDB in ROM [Yes/No]                        No
(NIR) Nucleus in ROM [Yes/No]                    No
(RIR) Root Job in ROM [Yes/No]                   No

Includes and Libraries
Path Name [1-45 Characters]
(UDF) UDI Includes and Libs
                          /rmx86/udi/
(HIF) Human Interface Includes and Libs
                          /rmx86/hi/
(EIF) Extended I/O System Includes and Libs
                          /rmx86/eios/
(ALF) Application Loader Includes and Libs
                          /rmx86/loader/
(BIF) Basic I/O System Includes and Libs
                          /rmx86/ios/
(SDF) System Debugger Includes and Libs
                          /rmx86/sdb/
(THF) Terminal Handler and Dynamic Debugger Includes and Libs
                          /rmx86/th/
(NUF) Nucleus and Root Job Includes and Libs
                          /rmx86/nucleus/
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

```
(ILF) Interface Libraries
                              /rmx86/lib/
(CAF) Crash Analyzer Includes and Libs
                              /rmx86/crash/
(DTF) Development Tools Path Names
                              :lang:


Generate File Names
File Name [1-55 Characters]
(ROP) ROM Code Prefix
                              none
(RAF) RAM Code File Name
                              /boot/onetask
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Appendix F: Related publications

Knuth, The Art of Computer Programming, Vol 1, pp 435-453, and exercise 6, p 452 with answer p 597. (c) 1973, 1978, Addison-Wesley Publishing Co., Redding, MA

iRMX™ 86 Introduction and Operator's Reference Manual For Release 6 (146194-001)

iRMX™ 86 Programmer's Reference Manual, Part I, For Release 6 (146195-001)

iRMX™ 86 Programmer's Reference Manual, Part II, For Release 6 (146196-001)

iRMX™ 86 Installation and Configuration Guide For Release 6 (146197-001)

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

ARTICLE
REPRINT

AR-286

June 1983

Software That Resides in Silicon

Ron Siamp
and Jim Person
Intel Corporation

Reprinted with permission from *VLSI Design* magazine, March/April 1983. Copyright© 1983.

210341-005

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Software That Resides in Silicon

Ron Slamp and Jim Person, Intel Corporation

Silicon software sounds like a contradiction in terms. The casting of software in silicon implies that the software cannot be changed; yet software does and must change. For example, it must be possible to alter a microprocessor operating system so that the system will support different hardware and software designs, as well as accommodate new hardware components and applications. And if the software has been committed to silicon, then a way must exist to overcome any bugs that are discovered later.

Design Considerations

Silicon software consists of two kinds of code: on-chip code and off-chip code (see Figure 1). In a typical case, some of the off-chip code works closely with the on-chip code, and is developed as part of the silicon software package. This special off-chip (or "support") code might contain initialization, interface, system, and version update codes. For silicon software to tolerate change and be usable in more than one system, the on-chip code must have three qualities: position independence, configuration independence and stepping independence.

Position Independence

Because the most advanced microprocessors address at least 1 megabyte of memory, system software that resides in silicon must work right regardless of its location in memory. Absolute addresses in the read-only, on-chip code or data restricts the configuration of the system. Because the on-chip code recognizes only offsets, absolute addresses are unacceptable. On-chip code cannot presume to know the location of any code or data, it can only presume to know the structure of the data which it accesses. It cannot know, except relatively, where in memory it (or any other code) resides. If the on-chip code is to be position independent, then any absolute addresses needed by the on-chip code must be obtained via the processor's registers.

Position independence is not a new concept; in fact, it is rather an obvious requirement for silicon software. Compilers and relocatable assemblers allow linking and locating, thus making it easier to produce position-independent code. But most of these tools can also produce code that is not position independent. Silicon software developers need to be aware of the position-independence requirement throughout the design, implementation and test phases for their products.

Configuration Independence

The second requirement for silicon-resident software is that the on-chip code must not depend on the underlying hardware and software configuration of the system. Instead, the on-chip code must have indirect access to other code or data, and must then check the run-time data to deduce the system configuration.

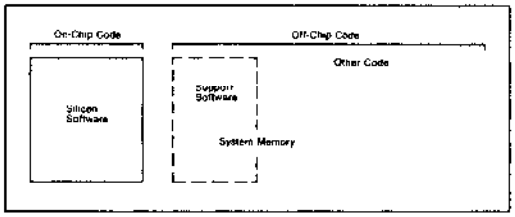


FIGURE 1. Silicon software is divided into on-chip code and off-chip code. The off-chip code either directly supports the on-chip code or contains other applications code.

Because of the read-only nature of silicon software, constants can cause problems when they are located within the on-chip code. Values representing a hardware device must not reside on-chip if that device can be located anywhere in the system, or when values support several devices having similar functions but different programming interfaces. Indirect access is necessary for all values that vary depending on the configuration of the system.

Stepping Independence

Stepping independence is an expansion of configuration independence, and is perhaps the most elusive of the requirements to be met by software intended for residence in silicon. A "step" is an updated version of the on-chip code. The on-chip code and the off-chip code must remain compatible, regardless of changes in either of them. Stepping independence exists when all versions of the on-chip code work with all versions of the off-chip code.

If stepping independence is taken into consideration when the silicon software is developed, then provisions can be made for the subsequent additions of options without changing the on-chip code. Otherwise, the static nature of the on-chip code might make it impossible to add options. Although configuration independence can be designed into software from the start, stepping independence can be achieved only if a system's existing silicon software does not include features that prevent it.

One type of data that is likely to change between steps is the value representing the size of a data area. If the software is to be stepping independent, it cannot know the sizes of the data areas accessed by on-chip code prior to run time. (No problems arise if on-chip and off-chip code agree on the size of the data area.)

But what happens if the on-chip code is not from the same version of the product as the off-chip code, and if the size of the data area has changed between versions? If the size of the data area is defined by a constant in the on-chip code, then that area might be smaller than the off-chip code expects it to be. This misunderstanding can lead to disaster as the off-chip code reads and writes beyond the data area.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

This problem is solved when the on-chip code ascertains the size of the data area from off-chip data. Thus, the size of the data areas for the system becomes a configuration option.

Getting the Bugs Out of Silicon Software

Every large program contains bugs. Designers usually remove bugs by modifying the program to correct the problem, and then discarding the old program. However, a program in silicon cannot be modified without stepping the component. And even so, it is undesirable to discard the outdated component.

Software designed for silicon should include a facility for fixing bugs in on-chip code. One way to fix an on-chip bug is to prevent access to the routine containing the bug. A correct version of the routine is provided off-chip, and program execution is forced to branch to the off-chip version whenever the routine is invoked. Modular programming practices during development help reduce the cost of such off-chip duplication.

This on-chip bug-fix works well over time. Each component step has an associated collection of bug-fix modules. The collection is updated for each new version of the product, as component steps fix known bugs. During system configuration, the user specifies which component step is being used; the fixes for that step are included automatically in the off-chip code. Because of this facility, one step looks just like another to the user.

Intel's OSF: A Software Component

The Operating System Firmware (OSF) component consists of several hardware modules (see Figure 2). These modules provide two functions that are essential to operating systems: interrupts and timers. The OSF modules include a Control Store (16K-bytes of fast ROM) to contain the silicon software, three programmable interval timers, an eight-input programmable interrupt controller, a bus interface, control logic, a data buffer, and address latch logic.

The 80130: The iRMX™ 86 Kernel in Silicon

Intel's first software-on-silicon product is the 80130. It provides a functional subset of the iRMX™ 86 Nucleus, which is the heart of the iRMX 86 operating system (OS). The iRMX 86 OS is a real-time, multi-tasking, multiprogramming operating system intended for 16-bit microprocessor designs. The iRMX 86 family of standard software modules includes a nucleus, a stand-along terminal handler, a stand-alone debugger, an asynchronous I/O system, a synchronous I/O system, a loader, a human interface, and options required for real-time applications. The nucleus manages the creation and dynamic deletion of all system architectural features (tasks, program environments, memory segments, data-communication managers, etc.). It also schedules tasks, based on priority, interrupt management, memory management, validation of parameters, management of exceptional conditions, and co-processor support.

How the 80130 Satisfies the Silicon Software Criteria

The iRMX 86 Nucleus provides both the on-chip and off-chip codes needed to implement the operating system. The on-chip code resides in the 16K-byte ROM space of the 80130. It is the main portion of the Nucleus code, and includes the kernel of the

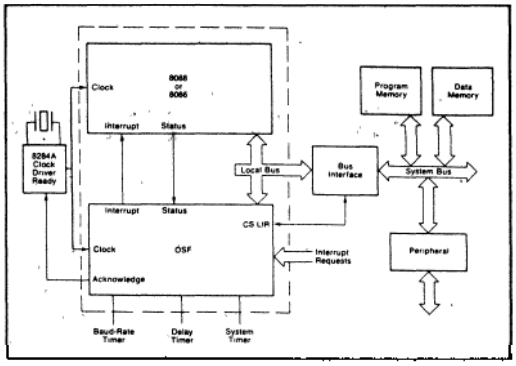


FIGURE 2. The OSF component works with systems that use the iAPX 86, 88, 186, or 188 microprocessor. Close coupling of the CPU and the OSF allows maximum zero-wait-state performance of the OSF software.

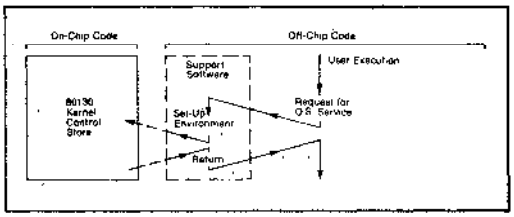


FIGURE 3. The position-independent interface supplies data location and run-time values, and starts on-chip execution of the software.

operating system and the primitives, which are present in the basic 80130 configuration. The off-chip code is stored in external RAM or ROM. It consists of initialization code, and code that either cannot be position independent or cannot be known before a given system is configured.

Position independence is guaranteed if entry to the on-chip code is possible only through an interface in the off-chip code that sets up the necessary registers. The off-chip position-independence interface (see Figure 3) provides an absolute data location and begins on-chip execution by the silicon-resident code. All run-time values can be determined based on the data location. On-chip execution gives the processor a location in the on-chip code from which other on-chip locations can be calculated.

It was relatively easy to make the 80130 configuration independent, because (like most operating-system kernels) it contains only general-purpose functions. The off-chip code contains all the drivers for particular peripheral chips. The Interactive Configuration Utility integrates the drivers with the 80130.

The interface between the off-chip and on-chip codes remains stable across component steps. The stepping-independence interface (see Figure 4) resides on the chip, and is a map of the on-chip code. This interface gives the off-chip code indirect access to all on-chip "publics" (*e.g.*, externally accessible routines, modules, and labels). It is also a chart that routes execution to the proper on-chip location. The off-chip code uses an index of this chart to specify which public should

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

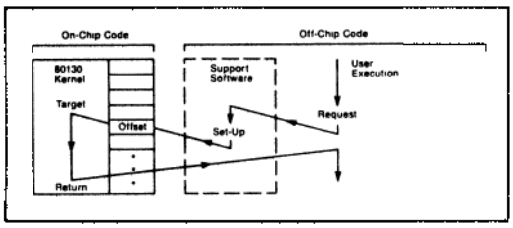


FIGURE 4. All on-chip accesses are routed through the on-chip stepping-independence interface, which provides compatibility between on-chip and off-chip code. Because the interface structure stays constant, the external reference also stays constant, while the on-chip OFFSET changes to point to the new location of the on-chip code.

be accessed. The index of a given routine remains the same across component steps, even though the actual address (offset into the component) of the public has changed. For different versions of the on-chip and off-chip codes to work correctly, all access from outside the component must be routed through the stepping-independence interface.

The 80150: CP/M-86* in Silicon

Intel's decision to implement CP/M-86 operating system in silicon (the 80150) raised a different design problem. With the 80130, Intel only had to deal with Intel-designed software. Code design, implementation, extensions, corrections, support, and the subsequent effect on the end user were all under Intel's control. The selection of an independent software system such as CP/M-86 (a product of Digital Research, Inc.) introduced new factors into the implementation.

The CP/M-86 Architecture

The CP/M-86 operating system consists of three modules. The Console Command Processor (CCP) handles command line processing, and executes built-in utilities. The Basic Disk Operating System (BDOS) performs logical disk I/O, including disk reading and writing, directory management, and sector allocation. The Basic Input/Output System (BIOS), which contains the configuration-dependent code and data, also provides I/O for specific peripheral chips.

CP/M-86 is a single-user, single-tasking operating system written in position-dependent code. The 80150 contains the *entire* CP/M-86 operating system; for many configurations, it requires no off-chip code. Intel's goal was to use the configuration-independent CCP and BDOS elements as a base, and add to them a BIOS that supported a variety of peripheral components but was still configuration independent.

The 80150 BIOS supports the following two functional configuration options:

1. *A preconfigured-mode system*, for which the system designer needs to do no operating-system code development or extension.
2. *A configurable-mode system*, for which the designer makes a selection from among the Intel drivers supplied, and makes changes as required to meet hardware needs.

The 80150 BIOS includes drivers for the following chips:

**CP/M-86 is a trademark of Digital Research, Inc.*

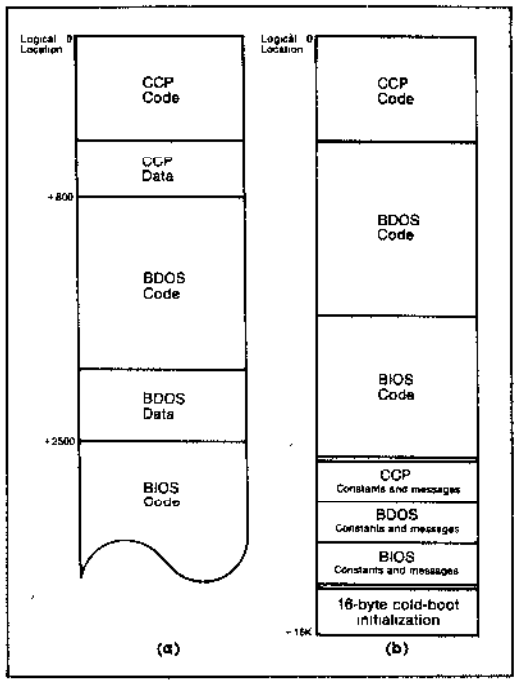


FIGURE 5. (a) The standard disk-based CP/M-86 module is one long structure containing both code and data. (b) Intel reorganized the basic CP/M-86 architecture to fit the operating system into the 80150 OS firmware component.

8251A Universal Asynchronous Receiver/Transmitter (UART)
8274 Multi-Protocol Serial Controller (MPSC)
8255A Programmable Parallel Interface (PPI)
8275 Floppy-Disk Controller
8237 Direct Memory Access (DMA) Controller

If the 80150 is used as a co-processor with the iAPX 186 or the 188, then the on-chip peripherals of these processors (DMA, timers, interrupt controller, chip-select logic) are also used.

Configuration independence is achieved via the Configuration Block (CB), with which whole BIOS drivers, data structures, and built-in utilities can be selected independently by the system integrator.

CP/M-86 Transformations

Intel and Digital Research together addressed the issues of position dependence and intermixed code, data, buffers, and stacks. The CCP and BDOS were reorganized to consolidate code and to use the 80150's ROM space efficiently.

CP/M-86 was originally developed using an 8080 model structure. The use of this structure implied that the code and data groups would overlap, as they do in the classical 8080-based CP/M design. Each module contained set-aside buffer areas, and included separate data stacks. Therefore, all variable areas

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

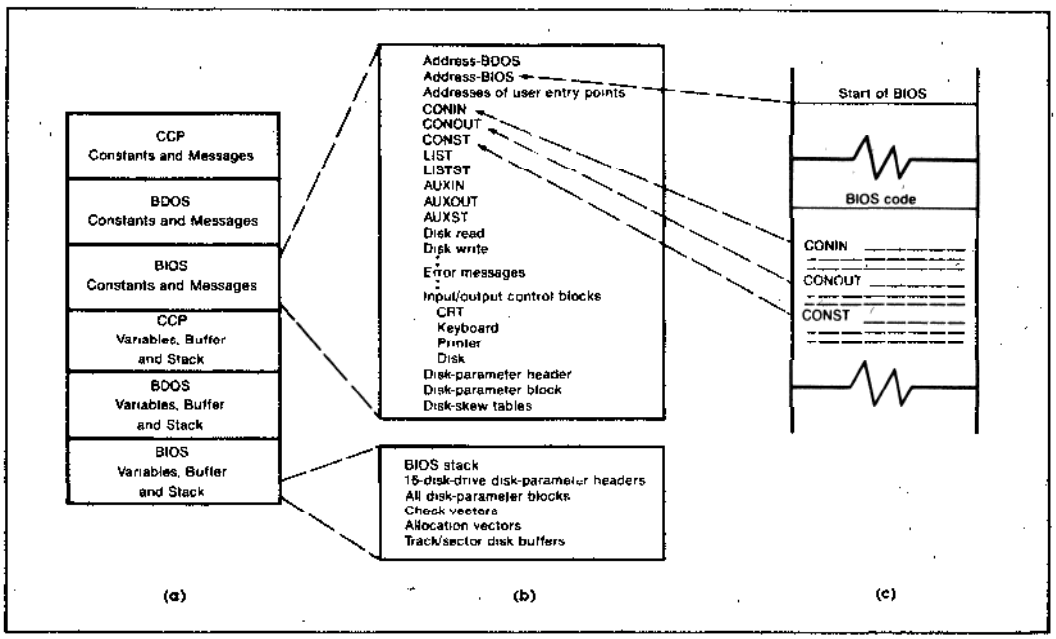


FIGURE 6. The Configuration Block (CB) reconfigures the 80150 for specific hardware systems. a) The CB constants read down from the 80150, and variables used at run-time. b) The BIOS portion of the CB contains configuration-dependent data. c) These addresses provide access to the 80150 on-chip code, to alter execution paths for different configurations and steppings.

and stack areas had to be removed from code that would reside in ROM.

Figure 5(a) shows the general structure of the original CCP and BDOS. Although a natural separation between code and data is clear, Digital Research did not distinguish between constants, literal messages, and pure scratch storage.

Intel's first step in the transformation of CP/M-86 was to group all variables within each module, including buffers and stacks. We then placed this data grouping at the end of the constants and literal messages for each of the CCP and BDOS modules.

The new structure (Figure 5(b)) includes all code, constants, and internal messages, as well as a 16-byte initial-program-load (IPL) boot resident in the 16K-byte OSF ROM. We removed all variables from the body of CP/M-86, and put them in an external RAM-based structure.

Second, the implementation of CP/M via the Intel 8086 "small model" (separate code and data segments) rather than via the 8080 model (intermixed code and data), meant that the necessary additional variable data space would be available at 80150 execution time. The segmented architecture of the iAPX 86 family made this implementation easy, because separate CPU registers were available for data and code addresses. As part of the BIOS initialization, we moved the constant data structures for the CCP, BDOS, and BIOS to the base of a RAM-resident Configuration Block (CB). An additional amount of RAM equivalent to the total variable space was also allocated and preset to zero. This 8086 "small-model" transformation not only made it easy to separate code and data, but also made the code more efficient and eliminated approximately 2100 bytes.

We achieved configuration and stepping independence via the off-chip RAM-based Configuration Block. Figure 6(a) shows the overall structure of the CB as constructed during BIOS initialization. During initialization, the 80150 BIOS copies the CCP, BDOS, and BIOS constant and literal structures into the Configuration Block, and appends additional space for variable and scratch-pad storage. Even the location of the CB is alterable, based on the address stored in locations 0:3FE-3FF.

Figure 6(b) shows expanded portions of the CB. The data area contains pointers that can be changed to select custom off-chip code instead of the standard on-chip code. The entire BIOS can be replaced. (The BIOS code insert in Figure 6(c) and the various code labels are reflected back to the CB.) Complete I/O control block structures are provided for each CP/M logical device, including CRT, keyboard, list, auxiliary, and disk. The control block includes port addresses, protocol support, and other default data needed to detect and control the status of each peripheral. Figure 6(b) also expands the systems tables and buffers created for disk support.

The addresses in Figure 6(b) indicate how stepping independence is achieved. Any off-chip routines changed by the user can be selected by altering the address of the CB. If Intel updates an on-chip routine, the address in the CB is updated automatically when the 80150 copies its constant structures into the CB. As explained above, full stepping independence is maintained, because any ROM changes can also be imple-

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

mented off-chip by having the address in the CB point to an off-chip patch. (The CB contains BDOS entry points (shown in Figure 6(b)) that make this change possible.)

The Configuration-Independent Interface

Use of the predefined configuration requires that the 80150 be installed at the top of the 8086 memory address space (FC00:0). The 16-byte internal hardware boot is activated at all POWER ON and hardware resets, and passes control to the 80150. The 80150 initialization sequence uses this positioning to indicate the default hardware configuration (floppy disk, printer port, serial console, or auxiliary port). Each device has predefined port addresses, interrupt assignments, and protocols. The iAPX 186 or 188 CPU supports programmable chip-selection and the on-chip DMA drives the floppy disk controller.

If the configuration must be altered, or if the BIOS code needs revision, the 80150 can be installed on any 16K code boundary except at the very top or bottom of memory. A PROM that contains off-chip code and data for a user's particular configuration is also installed at the top of memory.

The 80150 initializes the default system hardware tables, then calls an EPROM to complete or revise the existing data in the off-chip CB RAM area. At this point, the CB contains the addresses that select either on-chip or off-chip code. When the configuration is complete, control is returned to the 80150. The 80150 completes the CP/M initialization, displaying the familiar CP/M "A" sign-on.

Conclusion

Converting software to silicon is not new. But redesigning software to consist of on-chip ROM code and configurable RAM data is somewhat more innovative. One silicon-related specter that haunts software designers is the fear of "committing code before its time." But software designers can *never* expect to produce bug-free code the first time. And system designers cannot always predict the capabilities or the implementation requirements of peripheral devices that have yet to be built. Nevertheless, software designers who use the general silicon-implementation strategies of position independence and configuration independence, and who provide for stepping independence, can create standard silicon hardware without fear of component obsolescence. □

About the Authors

Ron Slamp received the A.S. degree in software technology from Portland Communnity College, and gained much of his skill in electronics at Clark Community College in Vancouver, Washington. He has worked in Intel's OEM Module Operation in Hawthorne, Oregon since 1978 and is currently the project leader for component software.

Jim Person received the B.S. degree in mathematics in 1962 from the University of Arizona. He was the engineering project manager at Intel for the 80150 "CP/M-on-a-chip."

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

June 1983

Putting Real-Time Operating Systems to Work

Stephen Evanczuk
Electronics Magazine

Reprinted with permission from *Electronics*, March 24, 1983 Copyright© 1983 McGraw-Hill Inc. All rights reserved

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

SPECIAL REPORT

real-time OS

Punching in for real-time jobs in industry, R&D, and offices, operating systems use special software structures to squeeze better-than-ever performance out of 16-bit microprocessors

by Stephen Evanczuk, *Software Editor*

☐ A special class of operating systems is hard at work in the 16-bit microsystem world. For controlling environmental processes, acquiring data at high speed, or even handling transactions at a commercial bank, these operating systems contain mechanisms that enable them to respond rapidly to external events and that differentiate them from the more familiar general-purpose operating systems.

In fact, all the operating systems for 16-bit microprocessors respond in a reasonable period of time. But the general-purpose, or developmental, operating systems like CP/M, Bell Laboratories' Unix, and MS-DOS are intended for standard programming activities like editing, compiling, and file management [*Electronics*, March 24, 1982, p. 113]. As such, they lack certain software structures needed for reliable control of processes producing data at a high speed.

Real-time operating systems tend to fall into two general categories—multipurpose and embedded, reflecting the type of hardware they run on. Multipurpose real-time systems are typically built around full-fledged microcomputer systems with terminal, keyboard, plenty of system memory, and mass storage. Furthermore, in process-control or data-acquisition applications, some special-purpose hardware is usually included in these systems to serve equipment or high-speed data input operations. Besides the familiar applications for research and development, transaction-processing environments are an example of situations needing multipurpose real-time systems.

No doubt the largest class in volume because of their growing use in consumer items, embedded systems are minimal hardware systems, often just one-chip microprocessors that control limited parts of a larger system. Programmers ordinarily employ a special development system to create the software, which is loaded into the target system for use and ideally is never seen again.

To meet the needs of these two classes of applications, real-time operating systems come in three flavors for 16-bit microprocessors. Serving multipurpose real-time systems, one type—discussed in the first part of this report (see p. 106)—includes all the software development support found in their general-purpose counterparts. Furthermore, many can be stripped of the layers needed in the developmental environment and placed in programmable read-only memory for use in an embedded system.

For those who swear by Unix, the group of Unix-based operating systems discussed in the second part (see p. 111) may mean no need to swear at it in real-time applications. A growing number of vendors are starting to convert this admittedly non–real-time operating system into versions that can be used to handle external processes. Although the industry is cautious, if not downright skeptical, of real-time versions of Unix, the fact that C—the language of Unix—is so highly regarded for use in real-time applications may help swing this group into the forefront.

The potential for distributed-control systems based on embedded microprocessors hinges largely on the availability of high-performance real-time operating systems that can be plugged into the application with the same ease as an integrated circuit. Called silicon software, these operating systems discussed in the last part (see p. 114) have been designed to be stored in read-only memory. Providing a fixed set of system calls, they present programmers with a consistent set of high-level commands to perform the low-level functions usually built from scratch.

Building system-level software from scratch has long been the hallmark of real-time programmers, even a mark of honor. Fortunately, however, the increased acceptance of ready-made operating systems using well-understood algorithms (described in the first part) is helping to replace this software "random logic" with rather more standardized packages.

On still another level, the unique responsiveness and throughput demonstrated by real-time operating systems is a truly user-friendly feature. For this reason, these systems should find their way into less obvious real-time applications, such as transaction processing, word processing, and personal work stations for office automation.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

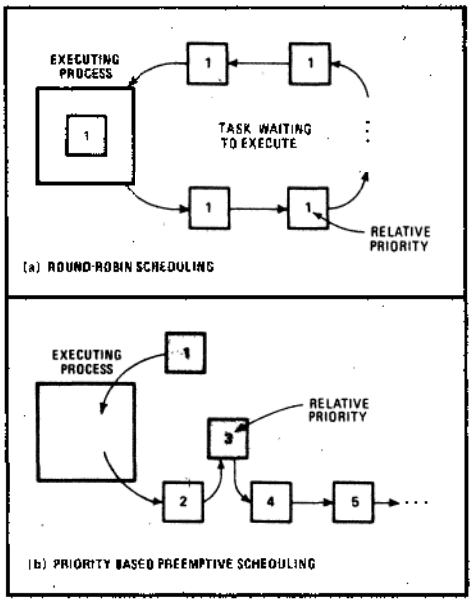
real-time OS

Algorithms star in multipurpose systems

☐ Whatever environment it finds itself in, the function of an operating system is the efficient management of shared resources by a number of users, whether these are human beings accessing a computer through terminals or programs vying for a single central processing unit. In fact, the degree of sophistication of an operating system is reflected by the number and types of physical resources it manages and by the fineness of control it exercises in their management. And operating systems targeted for control of the external environment must wrestle with the most demanding resource of all—time. The degree of care with which such software is designed to manage time is what determines its suitability for the real-time environment.

Schedulers and queues

Two critical aspects of the real-time environment are the random nature of physical events and the simultaneous occurrence of physical processes. Consequently, interrupt handling and multitasking are primary attributes of a real-time operating system. In fact, it might be argued that the mechanism for handling multitasking—the scheduler—is the heart of the operating system. The rest of the operating system lies atop this kernel and serves the specific demands of the application environment.



1. Priorities. In round-robin scheduling (a), tasks (or processes) take equal turns executing, while a higher-priority task will supersede a lower-priority one in priority-based preemptive scheduling (b). Most schedulers employ some combination of these techniques.

In particular, the lists, or queues, and their managers that surround the scheduler are constructed to deal with the different physical resources supported by the operating system. Thus, one queue may contain those tasks (processes, or programs in the course of being run) that are ready to execute on the processor, another queue may be tasks waiting for access to input/output hardware, and another queue may contain tasks waiting for some specified event to occur.

In any multitasking operating system, the scheduler uses the queues as input. Its output, on the other hand, is a single task that has been activated and allowed to execute on the central processing unit. The scheduling algorithm in large part defines the operating system.

In one system, the scheduler may simply select a task on a first-come, first-served basis, allowing it to run until completion or until some specified period of time has elapsed. This type of relatively primitive algorithm was commonly used in mainframe computers running simple batch-oriented operating systems.

In a slightly more sophisticated operating system that can be used interactively through terminals, the scheduler may select tasks on a round-robin basis and permit each of them to run for a specified period of time (Fig. 1). Once the task exceeds its time slice, it is placed at the end of the queue and forced to wait until all other tasks have had a chance to execute.

Round-robin scheduling with equal time slices is adequate if every task is no more important than any other task. However, if some are considered to possess a higher priority, then a more sophisticated scheduling algorithm must be used—one that recognizes that some tasks are more important, but that no task should be excluded from using the CPU.

One solution is the use of several queues, where the length of the time slice is related to the priority of elements in the queue. In this case, the scheduler would allow all tasks in each queue of a different priority to execute on the CPU, but lower-priority tasks would be given less time.

A further refinement permits higher-priority tasks to suspend a running task. This technique, called preemptive scheduling, is an important feature for real-time environments, in which the delayed execution of a high-priority task could have disastrous results, rather than simply disappointing the user.

In scheduling algorithms, tasks may exist in a number of logical states, depending on their readiness to run. In the Versatile Real-Time Executive (VRTX) from Hunter

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

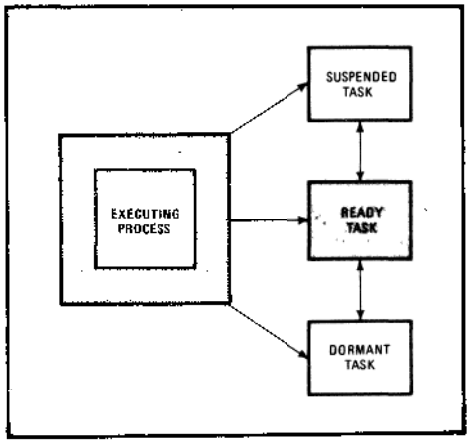
& Ready Inc., Palo Alto, Calif., for example, tasks are driven through four possible states by external events, by other tasks and system utilities, or by their own system calls (Fig. 2). For example, an executing task may delete itself—in which case it enters a dormant state—or may cause itself to be blocked either explicitly through a call to suspend itself or implicitly through a call to perform some I/O function. On the other hand, once suspended, a task may reschedule itself through a system call, or an external real-time event may bring the task back into the ready queue.

Recognizing the importance of scheduler design, at least one software vendor has made it easier for real-time users to build systems around a prepared kernel. United States Software of Portland, Ore., is offering a basic scheduler that assembles into less than 100 bytes of object code for the target microprocessor [*Electronics*, Nov. 17, 1982, p. 206]. Furthermore, in anticipation of real-time systems targeted for specific application areas, U. S. Software supplies a list of design notes detailing extensions to the basic kernel.

Another use for queues

In addition to having queues serving the scheduler directly, most systems use them as the preferred means of associating a task with a required resource. For example, one capability commonly found in real-time operating systems is the ability to suspend a task for a specified period of time. Typically, the operating system contains a special queue for this function. Each element in the queue is a task in a suspended state. Associated with each task is a counter that contains the number of clock ticks remaining until it should be reactivated.

For example, in iRMX-86 from Intel Corp., Santa Clara, Calif., the counters keep track of the incremental time remaining with respect to the previous element in the queue, rather than the total time remaining before the task may be reactivated. Thus at each clock tick only the counter in the element at the head of the queue need be decremented, rather than every counter in every queue element. This method takes longer to insert new elements into the queue and so requires slightly higher overhead for insertion than when the total time is maintained by each counter; however, that overhead is more than offset by the time saved by updating only a single counter.



2. Task states. As one task (or process) runs, others may be in various states of readiness. In Hunter & Ready's VRTX, for example, tasks can be ready (able to run immediately), suspended (waiting for a resource), or dormant (deleted by a system call).

Real-time environments pose a special set of problems for resource allocation. Besides all the more familiar problems of scheduling, a real-time operating system must maintain reliable behavior under extremes of load when it is driven by a high rate of external stimuli. From the system user's point of view, the system must maintain a predictable level of response and throughput.

In an interactive environment, users sitting at terminals measure response as the time the system needs to react to a keystroke. In general, system response is the time that the system needs to detect and collect data from some external stimulus. Throughput, in an interactive environment, is seen as the number of users able to utilize the installation simultaneously. In a more general real-time environment, throughput is the rate at which the system is able to collect, process, and store data.

In fact, although response and throughput share some common software elements, operating-system designers will invariably find themselves forced to make choices that will tend to optimize one at the expense of the other. Often, the interrupt-handling requirements of a real-time operating system force this choice.

Interrupt processing is hardware and software integration at its most demanding (see "Handling hardware interrupts," p. 108). To handle interrupts, operating systems often place layers of software between the user and the microprocessor in order to allow different levels of performance and capability.

Intel's RMX-86 is a typical example of distinct levels of software used to perform basic interrupt processing. At the lowest level, an interrupt handler works intimately with the hardware to execute some operation, such as sending a message character by character to a printer. Code for interrupt handlers is kept compact and simple, since system interrupts are disabled during their operation. The higher level, called the interrupt task, works at a priority associated with the particular hardware it services. Interrupt tasks act as interfaces between application tasks, working with specific interrupt handlers to complete execution of operations dealing with external devices. RMX makes this interrupt-handling mechanism available to application programs through a special set of system calls.

Protection and communication

Once the interrupt software has completed its function, tasks that use the data are indistinguishable from any other task in the system as far as the operating system is concerned. Unless special care is taken, conflicts could still arise between two separate tasks that might need to use the same resource, such as the same location in memory. MP/M-86, for example, employs a special queue, called a mutual exclusion queue, that contains a unique message representing the shared resource. In or-

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

real-time OS

der to use the resource, a task must first capture the message, much as a node in a token-passing network must first obtain the token before being at liberty to transmit.

Per Brinch Hansen[1] identified such shared resources as key elements in multitasking systems. Sections of code that access critical resources are called critical regions. The simple expedient of ensuring that only one task at a time is allowed in a critical region guarantees that multiple tasks may share the same critical resource without fear that its integrity may be compromised when two of them attempt to access it simultaneously (Fig. 3).

This concept of the mutual exclusion of tasks from critical regions is implemented in a structure called a monitor, in which critical regions are gathered in one section of code and protected from use by more than one task at a time. The MSP operating system from Hemenway Corp. of Boston [*Electronics*, Jan. 27, 1983, p. 119] explicitly supports mutual exclusion through monitors in its internal structure.

Furthermore, user-written routines needing monitor protection are provided with four functions in MSP that are implemented using hardware traps for rapid access: Entermon, Exitmon, Wait, and Signal. Entermon and Exitmon serve as monitor entry and exit points, respectively, performing required housekeeping functions. Entermon disables system interrupts and preserves all registers, while Exitmon reverses these actions. Wait and Signal, on the other hand, work in tandem to control access to a critical resource. Wait queues up tasks needing an unavailable resource. Signal releases them from the queue when the resource becomes available.

Wait and Signal are examples of an intertask communication mechanism, called semaphores, found in most real-time operating systems. As noted, these commands simply queue up and release tasks needing a critical resource. Such a resource may be an I/O device, a memory location, or simply a go-ahead signal that synchronizes a pair of tasks. For example, task A may execute only after task B has completed. In this case, task A would begin with a Wait (flag) command, where the flag is used as an associated variable. Task B, on the other hand, would end with a Signal (flag) command. In this way, task A would be blocked until task B had executed its Signal command at the end of its processing. But exchanging simple go–no-go signals is not sufficient for many multitasking environments.

For longer messages, real-time operating systems offer extensive intertask communication facilities called mailboxes. Mailboxes are essentially semaphores with storage. As such, tasks needing data from another task will wait until the other has loaded the mailbox with the information. Intel's object-oriented RMX-86 transfers any of the defined objects in the system through mailboxes. Hemenway's MSP, on the other hand, provides a buffer of fixed size that may be used without restriction on its contents, as long as the 256-byte buffer is not exceeded. With its Multibus message exchange (iMMX) extension to RMX for

Handling hardware interrupts

Underlying the special software of a real-time system is the assumption that the hardware itself can respond in a coordinated fashion to external events, or interrupts. In fact, microprocessors contain subsystems whose sole function is to deal with interrupts in a way that eases integration of the interrupt-handling software.

All modern computers integrate interrupt-handling hardware and software at a very low level of design. When a user accesses a microprocessor through a terminal, the same hardware interrupt facilities come into play as when, for example, an analog-to-digital converter sends data to the same type of microprocessor. The software response, on the other hand, depends on the type of operating system, but both real-time and general-purpose operating systems must take some action, like read in the data value or the character.

Examining the details of a simple keyboard task illustrates the complex nature of real-time processing. It also serves as a vehicle for introducing some of the basic vocabulary in this field.

A standard software subsystem in a microcomputer system, called the keyboard monitor, is responsible for working with the hardware interrupt system to detect a character, collect it, and effect some action based on the input character. When a key is struck on a terminal, the corresponding byte is converted into a serial stream of bits that are passed from the terminal to a universal asynchronous receiver-transmitter. Once it receives the full character, the UART generates a hardware signal, or interrupt, that notifies the processor. Since interrupt management is a common activity, processors contain special hardware to respond to this signal.

Although the details may vary from one particular microprocessor to the next, the result is the same for all. When its interrupt-request line is asserted, the processor ceases its current processing and places values from its internal registers into system memory. Typically, the processor status and instruction-address registers are saved in the system stack, a last-in, first-out buffer located in some portion of system memory. As the figure shows, the processor responds to the original interrupt-request signal by issuing a signal of its own, called an interrupt acknowledge.

The peripheral hardware that originated the interrupt detects the interrupt-acknowledge signal on the system bus and responds by returning the memory addresses of both the interrupt-handling subroutine and the new processor status. Typically, the new processor status will provide for disabling any further interrupts. This latter action is a simple precaution, preventing a single external stimulus from causing a continuous series of interrupts that will eventually result in an overflow of the system stack.

Such an interrupt mechanism, called a vectored interrupt, allows the speediest identification and reaction to an interrupt. (An alternative interrupt mechanism used by earlier processors, called a device-polling interrupt, simply forced the processor to switch to a defined address in memory containing software that polled each peripheral device until the device that generated the interrupt was discovered.) At

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

this point in the interrupt-handling task, all the activity was exclusively in hardware, but nevertheless resulted in extensive processor activity and bus traffic due to multiple accesses of system memory and the involved peripheral-device controller.

Consequently, it is not surprising that the time for hardware to set the processor to handle the interrupt—the hardware-interrupt latency—should be several processor cycle times in length. In general, hardware-interrupt latency is not a fixed number, but will lie within some range, since the processor will need a variable length of time to complete its current instruction and to initiate the interrupt-acknowledge signal. For example, if a processor is involved in a lengthy floating-point operation, several microseconds could elapse before the interrupt is acknowledged.

Once the processor has reached the interrupt-handling subroutine, the contents of only a minimal set of its internal registers have been preserved. However, before the real work of the subroutine may commence, the contents of other registers and variables shared by independent sections of the operating system must be preserved. The time needed to perform this action is called the context-switching time. Only after the software context is switched is the system ready to begin handling the special requirements of the device that originated the interrupt. The period of time between the occurrence of the external event and this state is the total interrupt-response latency.

In real-time operating systems, interrupt-response latency is usually a specified value—around 100 microseconds in very high-performance systems based on 16-bit microprocessors. Designers often bypass the constraints imposed by response latency by including special-purpose hardware to boost system response to external events.

Throughout all this time, system interrupts are still disabled. However, now that the context switch has taken place, the keyboard handler is free to transfer the character from the UART. Deciding where to put the character is important in terms of system throughput and overall efficiency. When it is put in some specified location in system memory, system interrupts must remain disabled; otherwise, if the handler attempted to service a subsequent interrupt, the new character would overwrite the character already in the location, but not yet fully processed.

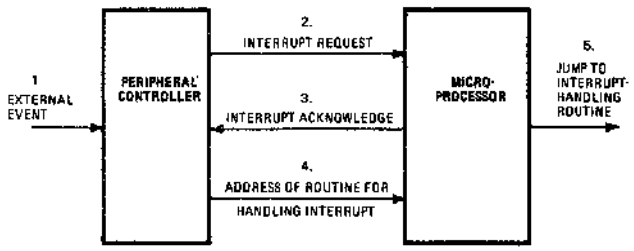
In general, there are two methods for handling this problem. In the first method, the character is simply placed on the system stack and referenced through the relevant-pointer. In an alternative method, the character is placed in a block of memory that has been reserved just for the handler and is called a context block. In this case, the character is referred to by using a specified offset from the base of the context block. Each time the keyboard handler is called in response to an interrupt, one of these context blocks is reserved from available system memory. Setting up a context block and switching the processor to it in a context switch accounts for a significant fraction of the time that is needed to respond to an interrupt.

Software code, such as the UART handler in this example, that does not contain any memory locations for variables is called reentrant because the processor may asynchronously enter it, be called away by an interrupt (even one that results in another call to the same piece of code), and return without loss of data or context. If the code is not already resident in system memory, another routine causes a copy of the code to be read from storage into memory. With reentrant code, only a single copy of the program or task need be resident at any time. Each context block, or logical copy of the task, is called an instance of the task.

Multiple instances of a task help explain some of the confusion associated with performance figures reported as a result of benchmarks. In examining benchmark figures, it should be clear just what the values are that are being reported. Total interrupt latency generally includes hardware interrupt latency, the time to create an instance of a task (plus the time to call in the task into memory if not already resident), the context-switch time, and an additional period needed to execute a variable amount of code that causes the data to be read from the peripheral registers. Creating a new task means either calling in a new task and creating a context block for an instance of it or just creating a new instance of a task already existing in memory.

Once the handler in the UART example reads in the character from the receiver buffer, it will reenable interrupts. The time between entry to the interrupt routine, when interrupts were disabled, until the time when interrupts are reenabled is an important factor in determining the effective latency of system response.

This dead time must be minimized, or the system will remain deaf to external stimuli for unacceptably long periods of time. In fact, the length of time that system interrupts are disabled is one of the criteria for determining the usefulness of an operating system for real-time applications. The longest period during which interrupts are disabled is a direct measure of the responsiveness of the system. Because of the weight of disabled interrupts on total system performance, modern microprocessors use a number of hardware-interrupt levels, or priorities, that disable interrupts at or below the priority level of the device originating the interrupt.



Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



multiprocessor-based systems, Intel replaces the concept of a mailbox with that of a software port connecting different tasks, whether they exist on the same or different physical processor.

Unlike memory-intensive software development systems, real-time environments find less need to support a virtual address space. In fact, the increased system overhead is less than desirable, because the designer seeks to minimize response latency. A useful feature, however, that can be found in some real-time operating systems is a set of system calls responsible for dynamically allocating and deallocating memory.

For example, in the ZRTS system from Zilog Corp., which comes in different versions for the Cupertino, Calif., firm's segmented Z8001 and nonsegmented Z8002, a set of three system calls provides for dynamic allocation and deallocation, as well as information on the status of memory allocation. The system call for memory allocation allows application programs to specify the attributes of the memory block to be allocated and returns a name referring to the created structure.

Besides similar system calls, Intel's RMX adds some calls suited to its context-based architecture. In RMX, each task lies within the context of a job environment that bounds the scope of tasks within it (Fig. 4). As such, each task is allowed to draw from the memory pool of its job. In case more memory is required than that initially allocated to the job, a pair of system calls provides for querying the system on the size of the job memory pool and for dynamically changing it.

Dynamic memory allocation and deallocation is a relatively advanced concept that exacts some overhead during runtime. However, the alternative—static allocation before runtime based on expected requirements—may be less suitable for applications in which the real-time environment is relatively unpredictable.

In real-time operating systems, disk-file management is treated as just another asynchronous task possessing a particular set of critical resources—mass-storage devices. In real-time environments, file-management utilities have to meet not only the requirements of general-purpose systems but some additional demands.

In terms of system response, a requirement of real-time operating systems in heavily loaded systems is the ability to conduct asynchronous I/O operations. In such an operation, the calling task simply queues up the I/O request, then immediately returns as if the task were completed in zero time. When the I/O request is fulfilled, the operating system switches the processor to a separate routine whose address is supplied when the original asynchronous request was made. This completion routine then may continue any processing that may be required following the I/O request.

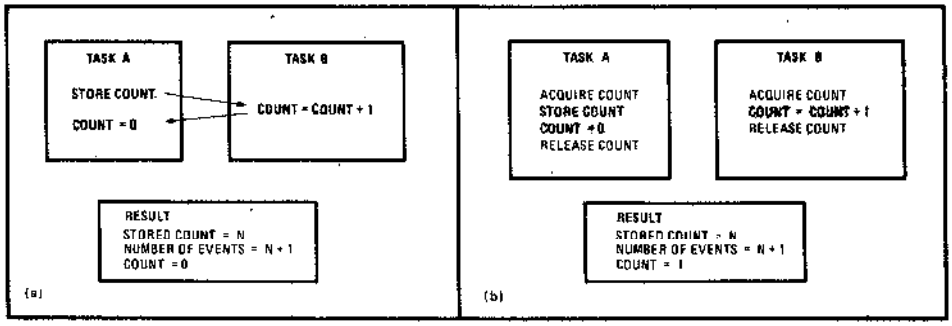
System throughput depends heavily on the efficiency and performance of the I/O subsystem. Peripheral controllers with direct memory access and the ability to move the disk's read-write head without necessarily performing data transfer can significantly reduce the overhead associated with data movement.

Reducing overhead

System software can also contribute to reduced overhead by providing a simple disk organization when high throughput is needed. One of the simplest structures is a file consisting of an unbroken series of disk sectors, such as the contiguous file in Hemenway's MSP or the physical file in Intel's RMX. By ensuring that the next block of data will be written to the next physical sector on a disk, the operating system can reduce the delay caused by head movement on the disk.

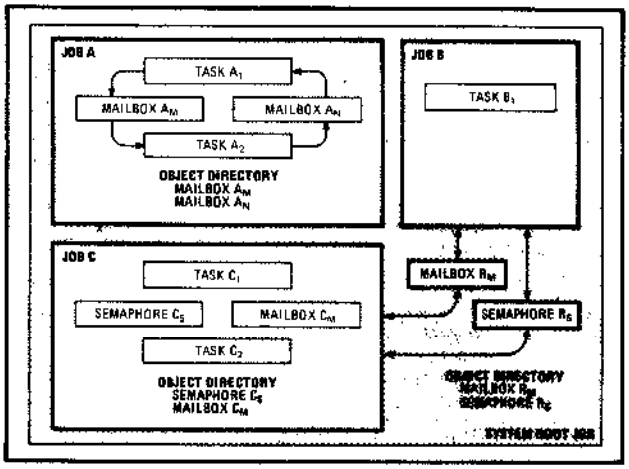
In their use of an I/O interface that is common to all system device drivers, MSP and RMX attack another important aspect of system design, though one not necessarily tied to their utility in real-time applications. In MSP, a basic I/O routine called Iohdlr serves for all operations by accessing a special block of information in memory. RMX, on the other hand, uses a number of device-independent system calls to handle communication with peripheral devices.

Next to multiprocessor-based software systems, real-time software systems are the most difficult to debug. Again, the cause is the distinguishing feature of real-time operating systems—precise management of time. Standard debugging tools for single-user general-purpose op-



3. Critical regions. If two asynchronous tasks use a counter, events can be miscounted if task B interrupts task A before the counter is reset (a). Forcing the tasks to acquire a counter before using it (b) ensures synchronization through the critical regions (tinted).

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com



4. Job context. In Intel's RMX, all jobs exist within the context of another job. A directory defines the objects that are known to other objects in the same context. For example, all three jobs may use mailbox R_M since it is in the system's root-job object directory.

erating systems generally disable all system interrupts in various phases of the debugging routines. Since the object of a real-time software system is asynchronous involvement with the task under control, this effect makes standard debugging tools useless.

Ideally, debugging real-time software would use performance-analysis tools and troubleshooting aids built into the operating system itself. Unfortunately, the processing overhead and additional memory requirements imposed by such a technique make this an unpopular notion in the design of an operating system. However, some systems do provide some means for run-time error handling. The exception handlers in RMX, for example, are procedures that are associated with each task when it is created. If a task attempts to use a system call but encounters an error, called an exception, the operating system invokes the associated exception handler to allow some graceful recovery from the error.

Although the technique in VRTX is not true exception handling, Hunter & Ready's silicon-software system does include a mechanism to build run-time debugging software. A special location in the VRTX configuration table (see p. 115) causes a user-defined routine to be called whenever a context switch is performed. By recording information about the task as well as the processor, such a routine can be used to create a list, called a trace, of the history of task execution.

Because real-time systems often include special-purpose hardware, the accepted technique for debugging user-written routines uses the classical approach of collecting data before and after passing through a suspect region, along with a logic analyzer to monitor timing of traffic through critical regions.

Intel offers some relief to this problem through the iRMX debugger. In particular, the debugger allows the user to work with individual tasks without interfering in the operation of other tasks, as well as to monitor the activity of the system as a whole without disturbing it. The debugger recognizes data structures in the RMX kernel, so the user may examine system objects. In addition, Intel's crash analyzer brings mainframe debugging power to microprocessor-based applications using RMX.

Zilog's ZRTS configuration language offers another level of support to the development of systems targeted to specific hardware complements. By defining the details of the hardware, a system designer can configure ZRTS to particular systems.

References
1. Per Brinch Hansen, "Operating System Principles," Prentice-Hall, Englewood Cliffs, N. J., 1973, p. 84

real-time OS

Designers tune Unix for real-time use

☐ With an eye on the growing momentum of Bell Laboratories' Unix, real-time system designers have endeavored to squeeze this complex operating system into the rigid confines imposed by the demands of real-time environments. Although Unix brought advanced system capability to mini- and microcomputers, the original intent was to provide a hospitable software-development environment, rather than to include the features considered necessary for real-time uses.

Until now, data-acquisition systems employing unmod-

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

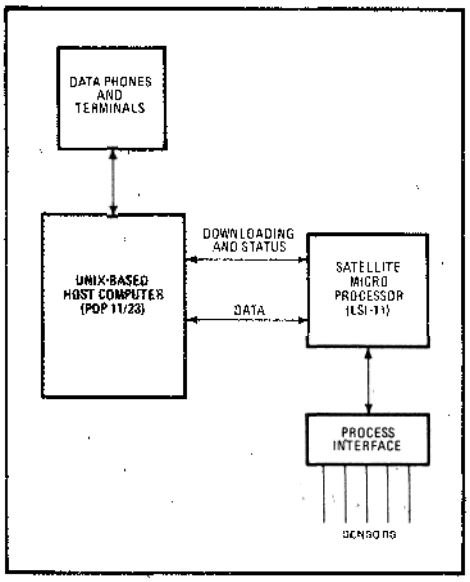
OS real-time

ified Unix typically used dedicated microprocessors to buffer a central computer from constant random activity caused by external events. For example, in the Conceps process-control system from Bell Laboratories, Murray Hill, N. J., a Unix-based host is linked with auxiliary microprocessors. In each microprocessor, software derived from Unix software handles the low-level details of real-time activity (Fig. 1).

Unix goes real-time

Appearing in all shapes and sizes, Unix-compatible executives, Unix lookalikes, and new Unix versions are bringing this popular environment into real-time applications. However, unlike their colleagues creating totally new operating systems (see pp. 106–111), designers of these second-generation systems are constrained by the boundaries set by the original. Caught between Unix's complex organization and the high-speed needs of some real-time applications, they have opted for preserving the basic architecture. Still, for intensive data-acquisition applications, vendors like VenturCom, Cambridge, Mass., and Masscomp, Littleton, Mass., add on dedicated hardware like high-speed peripheral controllers to link devices into the main system without losing the generality of the Unix software architecture.

For microprocessor-based dedicated systems, memory-resident kernels like the C Executive bring a measure of Unix compatibility to even dedicated systems. Offered by JMI Software Consultants of Roslyn, Pa., the C Executive combines support of an extensive C-language run-time library with many of the features considered important in real-time applications. Although not directly supporting shared data in its multitasking architecture, the executive's intertask-communication facilities include data exchange through a queuing mechanism. As befits a real-time executive, the task-scheduling algorithm allows higher-priority tasks to preempt lower-priority ones. Because it is intended primarily for embedded systems—that is, dedicated microsystems that do not have disks—the C Executive is totally contained in system memory and does not support the extensive Unix file-management subsystem.



1. Satellite processing. In Bell Labs' Conceps system, separate microprocessors handle low-level details of process control. Yet another processor—a host computer that runs the Unix operating system—is in charge of coordinating these satellite machines.

Controlling real-time tasks

Full-blown Unix lookalikes, on the other hand, find themselves forced to deal with some of the very internal structures that aided Unix's rise in popularity. For applications like program development where regular scheduling is more important that instant response, scheduling is aided by Unix's manipulation of the priority levels of tasks (or processes, in Unix's preferred terminology). For real-time applications, however, the slight uncertainties this feature introduces could destroy the synchrony of timed events controlled by the system.

Consequently, one enhancement commonly found in the real-time offshoots is the addition of some mechanism to ensure more precise control of real-time tasks. A technique that sits well within Unix's task-oriented (that is, process-oriented) design is the definition of a real-time class of tasks (or processes). This class earns special rights in the operating system, such as a guarantee that each task will not be swapped out of memory, but remain locked in and ready to respond more rapidly to events.

VenturCom's Venix, for example, defines a real-time priority level. The scheduler allows tasks running at this level to maintain control of the processor for as long as necessary. In contrast, Regulus from Alcyon Corp. of San Diego, Calif., speeds response to real-time events through the use of 32 user-defined priority signals.

Better I/O handling

In addition to its scheduling algorithm, Unix's method of handling input/output operations needs improvement to perform well in real-time applications. Aiding total system response, the asynchronous I/O procedure in Venix supplements the conventional synchronous procedure in Unix, in which the requesting task must be suspended until the I/O operation is completed (Fig. 2). By placing asynchronous requests at the head of the I/O request queue, Venix's manager lets real-time tasks issue a write request, for example, and immediately continue processing, assured that the request will be honored next.

Concentrating instead on improving what happens when I/O requests have been completed, Masscomp's enhanced version of Bell Labs' Unix System III adds a modified signal called an asynchronous signal trap. Similar to the concept of completion routines in other operat-

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Going Forth with alternatives

Few nightmares evoke the feelings of dread experienced by a programmer who must alter code that has been developed by another programmer—worse yet if the code is all assembly language for an embedded system. Fortunately, system developers are seeing the light of day and are specifying one of the commercially available real-time operating systems, so programmers now are dealing with a set of well-defined software calls for system functions. Still, for the true diehard who feels restricted by using someone else's system or the developer trying to eliminate all processing overhead caused by the operating system, alternatives do exist.

For straightforward, yet high-performance, process-control applications, the use of a finite-state machine as the controller is an easily implemented technique. A finite-state machine is simply some device that produces a defined output state based on its input state. For example, a microprocessor may read some input register, access a table in memory using this input as the address, and send out the value contained in the accessed location. In such a system, a value could be created with a single indirect move instruction in a microprocessor using a memory-mapped input/output scheme. Clearly, using a microprocessor this way would allow only a relatively small number of states.

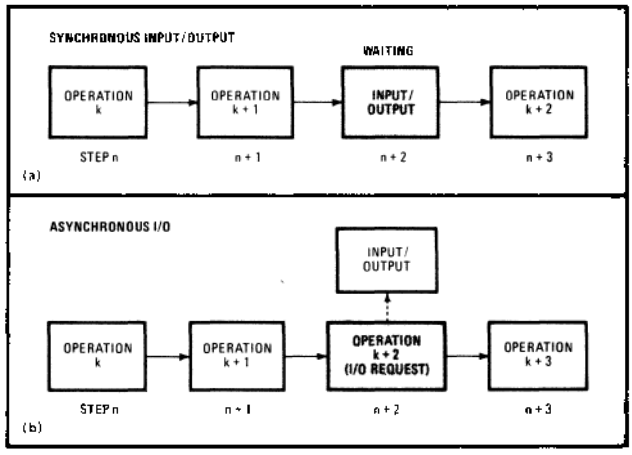
Besides this hardware approach, the software alternatives include the interpreters for high-level languages, such as Forth and concurrent versions of Pascal, that are appearing in the read-only memory of single-chip 8-bit microcomputers. For example, the CDP1804P complementary-MOS single-chip microcomputer from RCA Corp.'s Solid State division, Somerville, N. J. [*Electronics*, Nov. 30, 1982, p. 127], contains a core interpreter for Micro Concurrent Pascal (mCP) from Enertec Inc. of Lansdale, Pa. Based on Per Brinch Hansen's Concurrent Pascal, mCP contains all the constructs necessary for real-time applications, such as shared data, monitors, interrupt handling, and task queuing and switching. RCA also provides a ROM that extends the core interpreter to include full multitasking support. Software for this microsystem is developed using an RCA cross-compiler available on various host machines.

In parallel with the use of modified high-level languages like mCP, Forth interpreters are on the verge of appearing as single-chip microcomputers like the CDP1804 or the RF1/12 from Rockwell International Corp.'s Newport Beach, Calif., Electronic Devices division [*Electronics*, Jan. 13, 1983, p. 41]. After its development in the 1960s for real-time applications, Forth gained a slow acceptance among system developers. But with the inception of Forth standards committees and the spread of interpreters into more systems, this stack-oriented language is rapidly attracting the attention of larger houses.

Forth is a threaded language in which basic procedure calls, or words, are used to build up more complex words. Because of the threading, programs tend to be very compact. Once the programmer gets used to reverse-Polish notation, program development is simply a matter of building up the system dictionary with the words needed for the particular application.

ing systems, the AST mechanism allows tasks to perform operations that were contingent on the completion of a separate real-time operation. For example, by issuing an AST when it has completed its work, a read task is able to notify another task that a buffer has been filled. The other task is then free to initiate whatever calculation may be needed to make use of this new data.

Besides such modifications improving Unix's response to asynchronous events, Masscomp upgraded the system's throughput by adding support for contiguous files to the file-management system. In this way, large amounts of data may be written at a high speed to



2. No blocking. In synchronous I/O, execution of a task blocks, or waits (tinted), until the data transfer is completed (a). Since I/O is handled independently, a task need only request an I/O operation (shaded) and continue on to the next operation

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

consecutive disk sectors. Since other disk accesses are locked out in this mode, the disk head will be positioned correctly, thereby eliminating unnecessary and time-consuming movements.

In addition to these I/O add-ons, Masscomp boosted intertask communication capability by enlarging the Unix standard intertask communication mechanism, called pipes, to allow tasks to transfer buffers. In an alternative approach, Charles River Data Systems of Natick, Mass., allows tasks in its Unix-like Unos system to share data directly. A number of independently constructed software tasks may use a common set of locations in memory to transfer data between themselves or to perform some sequence of calculations. However, whenever asynchronous tasks share some common resource, their use of the resource could result in corrupted data—unless some mechanism coordinates their activities, such as the monitor concept described on page 108. Unos provides a mechanism called event counts to help avoid these conditions.

Event counts are integer values that are a nondecreasing count of the number of times some particular event has occurred. By using an event count associated with some task that produces shared data and another event count for a task that consumes the shared data, programmers may ensure the correct sequencing of asynchronous data-producing and -consuming tasks. Similarly, event counts serve as primitive operations for emulating the synchronization function that is provided by semaphores and the mutual exclusion that is furnished by monitors.

real-time OS

Chips come to aid of embedded systems

☐ Storing machine instructions in read-only memory is hardly a new concept in microprocessors. If supporting software totally breaks down, Digital Equipment Corp.'s LSI-11, for example, resorts to a basic keyboard monitor stored in a special ROM that is logically placed in the input/output address space. Using a primitive on-line debugging technique stored in the same ROM as the monitor, a software designer may read and alter memory locations and initiate a bootstrap loading operation from storage—a common provision in computer systems.

From these primitive beginnings, however, ROM-based software has evolved into complete operating systems in memory, engendering the term silicon software. Complementing hardware for distributed-processing architectures, such silicon-software systems signal a migration of application software into dedicated microcomputers previously considered unable to gain full systems capability. For developers of dedicated microcomputers embedded in some larger real-time system, silicon software spells the end of the need to reinvent the wheel to carry out the fundamental functions of a real-time operating system.

Extending the microprocessor

Functionally, silicon operating systems extend the microprocessor's instruction set to include system-level instructions that perform operations on software structures, like queues and tables, rather than on hardware registers. Application-program developers are then presented with a virtual machine—one that is perceived by the programmer as different from the actual host processor. In these virtual operating-system machines, their instruction set includes a well-defined set of system calls as well as the basic machine instructions of the host microprocessor. For example, with systems like VRTX and RMX, the virtual microprocessor has a special set of instructions for handling interrupts (see Table 1).

For system developers, however, the problems in developing reliable silicon software extend beyond resource protection, timing, and communication problems (see pp. 106–111). In fact, the development problems extend beyond the purely logistical exercise of maintaining a separate ROM-based instruction store and one for variables that need to be placed in system read-write memory. Treading a fine edge between the full function of a general operating system and the fine-tuned performance of special-purpose software, silicon systems need to balance the need for a wide range of system functions with the requirement that they squeeze into a minimal amount of ROM.

Flexibility for expansion

Still, once a system meets a reasonable compromise between capability and size, it should not irrevocably lock the user into accepting its choices. For example, many real-time applications require some custom peripheral-device drivers and system-level functions. Consequently, the program should provide a mechanism for logically incorporating user-written extensions to the operating system, such as the user-defined pointers in the VRTX system from Hunter & Ready, Palo Alto, Calif.

In VRTX, a configuration table (Table 2) in system random-access memory allows specification of a custom routine that is to be executed whenever the system is initialized. For even more delicate control of system operations by custom software, a trio of pointers in the table specifies user-written routines to be accessed whenever a task is created or deleted or whenever a context switch is performed. Hunter & Ready also includes a location in this baseline configuration table for its anticipated file-management extensions to VRTX.

The 80130, an RMX-86 kernel in silicon from Intel Corp., Santa Clara, Calif., generalizes this approach through an index table containing pointers to system routines. If circumstances require the replacement of an existing system routine, the index-table pointer is merely altered to indicate the address of the new routine. In an

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

TABLE 1: SYSTEM CALLS FOR HANDLING INTERRUPTS

Instruction	Description
Versatile Real Time Executive (VRTX)	
UI POST	deposit message from interrupt handler
UI EXIT	exit from interrupt handler
UI TIMER	timer interrupt
UI RXCHR	receiver ready interrupt
UI TXRDY	transmitter ready interrupt
iRMX-86	
RQ$SETSINTERRUPT	assign interrupt handler
RQ$RESETSINTERRUPT	deassign interrupt handler
RQ$GETSLEVEL	return number of highest priority interrupt level currently being processed
RQ$SIGNAL$INTERRUPT	signal from interrupt handler that event has occurred
RQ$WAITSINTERRUPT	wait for occurrence of event
RQ$EXITSINTERRUPT	relinquish control of the system
RQ$ENABLE	enable hardware to accept interrupts
RQ$DISABLE	disable hardware from accepting interrupts

TABLE 2: VRTX CONFIGURATION TABLE

Table Entry	Entry Description
sys RAM addr	system beginning address
sys RAM size	system memory size
sys stack size	system stack size
user RAM addr	starting address for available memory in initial partition
user RAM size	size of initial partition
user block size	size of memory block for dynamic allocation
user stack size	size of stack for user tasks
user task addr	address of first user task
user task count	maximum number of tasks
sys init addr	address of user supplied initialization routine
sys tcreate addr	address of user supplied routine accessed when a task is created
sys tdelete addr	address of user supplied routine accessed when a task is deleted
sys tswap addr	address of user supplied routine accessed when a context switch occurs
(RESERVED)	address of Hunter & Ready future extensions to VRTX

embedded system, this new routine could be placed in ROM along with application software.

Now that programs in ROM have matured into silicon systems, the development of software for embedded systems may now follow a more hospitable development cycle. The particular method used to create embedded systems will, in general, fall into one of two paths represented by the two major camps.

On one hand, kernels in silicon from systems such as RMX-86 or the MSP from Hemenway Corp., Boston, Mass., for the 68000 or Z8000 are self-contained subsets of the full operating system. Consequently, software programmers may use the full development version of the same operating system as that in the eventual target to create the application package. On the other hand, development of application programs around the ZRTS system from Zilog Corp., Cupertino, Calif., or Hunter & Ready's VRTX for the Z8002, iAPX-86 family, or 68000 relies on the use of a separate development system to create software for the target microprocessor, since this software does not have development versions.

Two approaches

The significance of these two approaches as usual depends on the intended application. Hunter & Ready views VRTX as a set of processor-independent building blocks that programmers use to construct application packages for embedded systems. As such, the programmers employ the same development systems that they might use to build application code, but now with the benefit of a sophisticated set of ready-made system-software components.

In playing its part in Intel's systematic drive toward providing an integrated environment around the iAPX-86 family, the 80130 holds the anchor position in an interlocked set of components. Able to function independently of the upper layers of the operating system, it provides a hardware base for the rest of RMX-86. Serving as a viewport into this system-software base for the central processing unit, Intel's universal run-time and development interfaces offer the mechanism for software portability needed for the next stage in the company's plan to grow into higher-performance microprocessors, such as the 186, 286, and 386.

While interlocking with the software in this way, the 80130 also must play its role in the complementary relationships being established at the hardware level. As such, it includes on-chip hardware support for system-level functions, including timers, interrupt controller, bus control, and bus interface.

Meanwhile, Intel's plan for software-in-silicon becomes evident as it gathers the other pieces of the puzzle, such as the 82730 text-coprocessor chip, the 82586 local-network coprocessor, and the 82720 graphics processor chip. Similar to the 80130 software connection, the 82720 graphics part interlocks with the rest of the system at the software level through its support of another well-defined software interface—the virtual device interface. Yet to come are pieces for voice I/O support, as well as some level of hardware support for data-base access. □

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

intel

ARTICLE
REPRINT

AR-288

June 1983

Intel's Matchmaking Strategy: Marry iRMX™ Operating System with Hardware

Chappell Brown
Software Editor
Electronic Engineering Times

Copyright© 1983 CMP Publications, Inc , 111 E Shore, Rd . Manhasset, N Y 11030
Reprinted with permission from *Electronic Engineering Times*

210341-005

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Intel's Matchmaking Strategy: Marry iRMX™ Operating System With Hardware

Intel's major software product, the iRMX™-86 16-bit operating system, which is now in its fifth release, represented a three-year development investment which most independent software vendors would have found a daunting prospect in 1978 when the project was conceived.

The investment was essential. By the mid-1970s, feedback from OEMs working with Intel's hardware revealed problems with system integration—the marriage of software with hardware. It consequently slowed sales, with the prospect of even greater problems at higher levels of circuit integration. Intel management, looking for ways of coping with the ballooning software requirements of the rapidly accelerating hardware program, began stepping up software development programs in the mid-1970s.

"The RMX program illustrates a number of things one needs to keep in mind with developing a real-time operating system," explained Bill Lattin, Intel's OEM microcomputer systems manager. "Foundations must be well laid so the system can grow and evolve over time. And there is a need for the system to be open to modification by typical OEM-specific applications.

"Although the RMX program has been around since 1978, it has only recently hit its stride, as processor technology has advanced to use the full range of its features," Lattin said.

The fast-paced microcomputer market had created a new situation for systems designers in terms of a radical shift in the hardware/software cost ratio. Earlier hardware generations involved various expensive centralized facilities. Not only was software cheap in comparison, but the hardware environment changed slowly, so that it was also feasible to rewrite systems as needed.

But when the price of a computer drops to as low as $5, the hardware environment becomes volatile and software turns into a major investment. Intel was finding that customers might invest as much as two-thirds of their development costs in software, only to see it eclipsed by evolving VLSI technology.

It became evident that merely supplying components would become increasingly counterproductive. Thus, the Intel "total solution" emerged—a consistent systems approach to hardware sales, which naturally depends heavily on a viable software program.

Object-oriented programming is a method which has worked best in creating a software program blending with the component approach. By hiding data representation within an object with its own object manager, changes in the hardware environment that affect the data can be accommodated without having to change the rest of the software.



A price is paid in terms of program size with this approach, however. And it was difficult at the time to justify this kind of liability with the existing onboard memories of the 8-bit generation.

Bill Stevens, iRMX-86 program manager for release five, explained the difficult decisions that had to be made at the outset of the program. "Every engineering decision involves a trade-off. We wanted to optimize program productivity and we had to have modularity. The consequence of this was large size. It turned out that a minimum configuration was 12 kbytes wide and the full configuration was 128 kbytes. At the time we did not have 64k dynamic RAMs and 64k EPROMs, so we didn't have the technology to realize the systems of initial specifications times. Bruce Schafer has to take credit for making that decision to go ahead anyway, early on. . . it was a gutsy decision, and it turned out to be absolutely right!"

210341-005

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Had Intel known of the difficulty it was about to encounter in producing its 64-kbyte RAM, Schafer may have had second thoughts.

Schafer joined Intel in 1976 and began working on iRMX-80. "It was a nice little system," Schafer said. "A miniature dispatcher had evolved to handle multiple asynchronous events and became a primitive OEM operating system. It was tempting to do an enlarged version of it, mainly because I was already working on it for the 16-bit generation!"

Schafer soon found himself centrally involved in the task of heading off the 16-bit software crunch, laying groundwork for a system that could cover a wide range of applications, many of them unknown at the time, and a system which could also evolve with hardware advances.

"When you set out to design a system of that scope, you don't just sit down and start writing code. It's definitely a top-down process," explained Schafer. He discovered early in the project that the purely technical hurdles in writing software were minor compared to orchestrating a team of engineers on such a comprehensive project.

The iRMX-86 system is multi-layered, and the project had to be coordinated across these layers along with the sequence of planning, design and implementation. On top of that, a thorough testing program had to be coordinated with all phases.

"I had a difficult time convincing engineers on the project that documentation of their work was as important as the work itself. Specifications were absolutely crucial to the development phase!" said Schafer.

Schafer began with a customer survey to discover the kind of problems OEMs were experiencing with system design. He wrote a production implementation plan, which was critiqued by marketing and engineering personnel. This was approved in June 1978 and formed the basis for engineering specifications. A critiquing process evolved as the organizing principle behind initial product design; engineers on the project would exchange documentation and then meet to evaluate the progress of the system.

The sessions were lively and the problems of coordinating implementation, testing and design along with the pressure of deadlines for the whole program generated quite a bit of excitement.

Development testing turned out to be a particularly thorny problem—the asynchronous interrupts and multiple-processing aspects of real-time applications required a special test apparatus to simulate a real-world environment.

What they came up with is a nucleus executing directly on the 8086 and 8088 processors as the basic building block of the system. Together with the next layer—a basic I/O system—a minimal operating system can be configured, which has been found useful in many applications.

However, it was necessary to develop an application on the Series-III development system even though the target was going to be RMX. "We quickly realized that users want to be able to do development work on the machine they target on," said Schafer. "This is particularly important for field maintenance . . . you can't drag a Series-III out to an oil derrick!" To realize this goal, Intel built higher layers around iRMX so that program development could be done without a Series-III. Higher layers involve extended I/O and human interface facilities. After this, customer-written software can be added in high-level languages.

A major objective has been to provide a stable base for independent software vendors; with its latest release, Intel also announced an ISV program initially involving three major vendors; Microsoft, Digital Research and Mark Williams Inc.

The first release of iRMX-86 came out in April 1980. Since then, the system has been refined and released four more times, with release five appearing last December. An Interactive Configuration Utility appeared for the first time with release five, a further attempt to aid OEMs in putting their systems together. The system designer runs the ICU program on a terminal and is quizzed on his requirements, after which the program generates the unique iRMX software for his application.

"It has been a successful product in its own right, apart from its role in the hardware program, but I doubt that anyone would have wanted to invest in a three-year development process before there was a chance at some return," observed Stevens, who has been most excited by the diverse applications he has seen. "I've really enjoyed the iRMX symposiums. There is always some new system demonstrated. In Tokyo, I just saw an 8086-based scientific system with really first-class graphics put together by Seiko. Another time I saw a blood analyzer based on the system. There are even RMX-based personal computers."

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

ARTICLE
REPRINT

AR-337

APRIL, 1984

Industrial PC Systems Demands Real-Time Operating System

Kathryn S. Norris
Intel Corporation

Reprinted with permission from April, 1984 issue of Systems & Software, Hayden Publishing, 1984.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

"The tasks must be able to coordinate with one another."

PC ■ INTEGRATION SERIES

Industrial PC systems demands real-time operating system

Personal computers find a wide range of applications in an industrial process control environment. For example, the computers can be used for temperature monitoring and control, production line testing, wear analysis, frequency signature monitoring, analysis in noisy or hostile environments, and vibration analysis.

All of these jobs for the computer have certain characteristics in common. All require that the computer process asynchronously occurring events that are happening in real time. Moreover, handling multiple asynchronously occurring events, some of which can happen concurrently, demands that the computer process more than one operation or task at a time.

For example, in monitoring and controlling the temperature of a burn-in oven used to stress printed circuit boards, the computer must have a task that keeps track of time so that the temperature is read at prescribed intervals. It requires a second task to read the temperature sensor, and a third to control the application or removal of heat.

Furthermore, certain tasks are more time critical than others. For example, an error routine that detects a critical overtemperature condition in the burn-in oven must be given the highest priority and executed before any other routine vying for computer time.

Finally, the tasks must be able to coordinate with one another. They must be able to communicate so that the results of one needed by a second can be passed between the two. Tasks must also be able to exclude one another so that, for example, one can have use of commonly accessed data without interference from a second. Finally, tasks must be able to synchronize with one another to ensure, for example, control of a chemical reaction that requires an ordered sequence of elements be added to produce the desired result.

Kathryn S. Norris, Software Products Marketing Manager
Intel Corp
5200 N.E. Elam Young Pkwy
Hillsboro, Ore 97123

To meet the requirements of an industrial control environment, a personal computer must have an operating system like the iRMX-86 from Intel Corp. (Hillsboro, Ore.), which can meet the requirements just described. One recently announced industrial personal computer which offers iRMX-86 is the MSC 8807 Industrial PC from Monolithic Systems Corp. (Englewood, Colo.), see ("Industrial PC goes to work").

The operating system contains a nucleus which gives an applications program or task the means to monitor and control external events. Tasks running concurrently are called into execution by interrupts generated by the real-world process being controlled.

A scheduler inside the operating system decides whether an executing task should be interrupted to process data from a device generating an interrupt. In addition, three facilities inside the operating system provide for tasks to interact with one another. Each of the three is optimized for data transfer, synchronization or mutual exclusion.

Handling multiple tasks. The essence of real-time application systems is the ability to process numerous events occurring at random times. Any single program that attempts to process multiple, concurrent, asynchronous events is bound to be complex. It must process each event and remember which have already occurred and the order in which they happened. The complexity obviously grows greater as the system monitors more events.

Multitasking capability in an operating system unwinds this confusion. Rather than writing a single monolithic program to process N events, N programs are written, each of which processes a single event. Each of these N programs forms an iRMX 86 task. Multitasking eliminates the need to monitor the order in which events occur.

The operating system is an interrupt processor. When an interrupt occurs, it schedules a task to process the interrupt. This method of event detection improves the performance of an application system.

There are two ways that computer systems can schedule processing associated with detecting and

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

PC ■ INTEGRATION SERIES

"iRMX 86 uses preemptive, priority-based scheduling to decide which tasks runs at any instance."

controlling events in the real world: polling and interrupt processing. Polling has the major shortcoming of requiring a significant amount of the processor's time to test to see if events have occurred.

The second method of controlling processing is the interrupt. An event occurring generates an interrupt to the computer. Rather than executing the next sequential instruction, the processor begins to execute a task associated specifically with the detected event.

Interrupt processing allows a system to spend all of its time running the tasks that processes events, rather than executing a polling loop to see if events have occurred. Since there is a direct correlation between interrupts and tasks, a system can easily be modified to process different events. All that is needed is to write the tasks to process the new interrupts.

Because interrupt processing allows a system to respond to events by means of modularly coded tasks, system programs are more structured and easier to understand. Modular programs are less costly to develop and maintain, and modules can be developed more quickly than a monolithic program containing the equivalent of several independent modules.

Scheduling with priority. The iRMX 86 operating system uses preemptive, priority-based scheduling to decide which task runs at any instant. This technique ensures that if a more important task becomes ready while a less important task is running, the more important task begins execution immediately.

In multitasking systems, there are two common techniques for deciding which task is to be run at any given moment. One called time slicing, better known as the familiar round-robin approach, involves tasks running in rotation. Each task is allotted a fixed quantity of computer time in which to execute. If it does not complete in that time, it must relinquish the CPU and wait until its turn with the CPU comes up again. The technique is commonly employed in time-sharing systems.

The second technique, priority-based scheduling, uses assigned priorities to decide which task is to be run next. Within priority-based scheduling, there are two approaches. Non-preemptive scheduling allows a task to run until it relinquishes the processor. Even if while running it causes a higher priority task to become ready for execution, the original task continues to run until it explicitly surrenders the processor.

The second approach to priority-based scheduling is preemptive. Using preemptive scheduling, the system always executes the highest priority task that is ready to run. In other words, if the executing task or an interrupt causes a higher priority task to become ready, the operating system switches the processor to the higher priority task.

Preemptive, priority-based scheduling goes hand-in-hand with interrupt processing. The priorities of tasks can be tied to the relative importance of the events that they process. Thus, the processing of more-important events preempts the processing of less-important ones.

Allowing tasks to interact. The iRMX 86 operating system provides simple techniques for tasks to coordinate with one another. These techniques allow programs in a multitasking system to mutually exclude, synchronize, and communicate with each other. The processing of several events may be related. For instance, the task processing event A may need to know how many times event B has occurred since event A last occurred. This processing requires coordination between programs.

Tasks exchange information for two purposes. One is to pass data from one program to another. Suppose that one task accumulates keystrokes from a terminal until a carriage return is encountered. The keyboard program then passes the entire line of text to another task, which is responsible for decoding commands.

The second reason for passing data is to draw attention to a specific object, a mailbox for example, in the application system. In effect, one task says to another, "I am talking about that object."

The iRMX 86 system facilitates intertask communications by supplying objects called mailboxes along with system calls to manipulate them. The system calls associated with mailboxes are CREATE MAILBOX, DELETE MAILBOX, SEND MESSAGE, and RECEIVE MESSAGE. Tasks use the first two commands to build and eradicate a particular mailbox. They use the remainder to communicate with each other.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

"The priorities of tasks can be tied to the relative importance of the events that they process."

PC ■ INTEGRATION SERIES

If Task A wants task B to become aware of a particular object, it uses the SEND MESSAGE system call to send the object to the mailbox. Task B uses the RECEIVE MESSAGE system call to retrieve the object from the mailbox. Why don't tasks just send messages directly between each other rather than through mailboxes? Tasks are asynchronous; they execute in unpredictable order. Mailboxes allow tasks to communicate with each other even though tasks are asynchronous.

If the receiver uses the RECEIVE MESSAGE system call before the message has been sent, the receiver waits at the mailbox until a message arrives. Similarly, if the sender uses the SEND MESSAGE system call before the receiver is ready to receive, the message is held at the mailbox until a task requests a message from the mailbox.

Providing tasks exclusivity. Occasionally, when tasks are running concurrently, the following kind of situation arises. Task A is in the process of reading information from a memory segment. An interrupt occurs and task B, which has a higher priority preempts task A. Task B modifies the contents of the segment that task A was in the midst of reading.

Task B finishes processing its event and surrenders the processor and task A resumes reading the segment. However, task A might have information that is completely invalid. For instance, suppose the application is air traffic control. Task A is responsible for detecting potential collisions and task B is responsible for updating the plane location table with the new X- and Y-coordinates of each aircraft's location. Unless task A can obtain exclusive use of the plane location table, task B can make task A fail to spot a collision.

Here's how it could happen. Task A reads the X-coordinate of the plane's location and is preempted by task B. Task B updates the entry that task A was reading, changing both the X- and Y-coordinates of the plane's location. Task B finishes its function and surrenders the processor. Task A resumes execution and reads the new X- and Y-coordinate of the aircraft's location. As a direct result of task B changing the plane location table while task A was reading it, task A thinks the plane is at old X and new Y coordinates. This misinformation could easily lead to disaster.

This problem can be avoided by mutual exclusion. If task A can prevent task B from modifying the table until after A has finished using it, A can be assured of valid information. Somehow, task A must obtain exclusive use of the table. The iRMX 86 operating system provides a type of object that can be used to provide mutual exclusion in the form of the semaphore.

A semaphore is an integer counter that tasks can manipulate using four system calls: CREATE SEMAPHORE, DELETE SEMAPHORE, SEND UNITS and RECEIVE UNITS. The creation and deletion system calls are used to build and eradicate semaphores. The send and receive system calls can be used to achieve mutual exclusion.

Semaphores can only take on non-negative integer values. Tasks can modify a semaphore's value by using the SEND UNITS or RECEIVE UNITS system calls. When a task sends N units to a semaphore, the value of the counter is increased by N. When a task uses the RECEIVE UNITS system call to request M units from a semaphore, one of two things happens: If the semaphore's counter is greater than or equal to M, the operating system reduces the counter by M and continues to execute the task. Otherwise, the operating system begins running the task having the next highest priority, and the requesting task waits at the semaphore until the counter reaches M or greater.

To use a semaphore to achieve mutual exclusion, the task wanting exclusivity creates a semaphore with an initial value of one. Before any task uses the shared resource, it must receive one unit from the semaphore. Also, as soon as a task finishes using the resource, it must send one unit to the semaphore. This technique ensures that at any given moment, no more than one task can use the resource, and any other tasks that want to use it must await their turn at the semaphore.

Semaphores allow mutual exclusion; they don't enforce it. All tasks (there can be more than two) sharing the resource must receive one unit from the semaphore before using the resource. If one task fails to do this, mutual exclusion is not achieved. Also, each task must send a unit to the semaphore when the resource is no longer used. Failure to do this can permanently lock all tasks out of the resource.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

PC ■ INTEGRATION SERIES

"Occasionally a task must know that a certain event has occurred before it starts running."

Synchronizing tasks. Tasks are asynchronous. Nonetheless, occasionally a task must know that a certain event has occurred before it starts running. For instance, suppose that a particular application system requires that task A cannot run until after task B has been completed.

An application system can achieve synchronization also by using semaphores. Before executing either task A or task B, a semaphore is created with an initial value of zero. Task A issues RECEIVE UNITS requesting one unit from the semaphore. Task A is forced to wait at the semaphore until task B sends a unit. This achieves the desired synchronization.

Industrial PC goes to work

One portable computer designed exclusively for the multitasking industrial and scientific environment is the MSC Model 8807 Industrial PC from Monolithic Systems Corp. (Englewood, Colo.), the first of a planned family of system and support products that emphasizes software flexibility by offering two operating systems, four major languages and a variety of utility programs.

Based on the Intel Multibus architecture, the industrial PC integrates a 16-bit 80186 CPU, up to 512 kbytes of parity RAM, 9-in. CRT screen, dual 3½-in. floppy disk drives, parallel and serial I/O ports, and ACSI interface. The total weighs approximately 35 lb and includes a 100-W power supply, cooling fan and built-in carrying handle. One of two operating systems available for the PC is Intel's iRMX-86 operating system.

RMX-86 is a multi-user, multitasking, real-time operating system, which provides such advanced features as hierarchical file structure with variable file granularity. It schedules tasks with true real-time preemptive priorities. It enables dynamic memory allocation among concurrent applications, device independent I/O and intertask communication via mailboxes and semaphores.

The PC is built around the company's MSC 8186 single-board computer, which, in turn, is based on the Intel 80186 microprocessor. The processor board contains 128 kbytes of dual-port, dynamic parity RAM, a dynamic RAM controller, up to 64 kbytes of EPROM, and programmable parallel and serial I/O ports. Twenty-bit addressing, plus four bits for bank select, enables addressing up to 16 Mbytes of system memory. The serial I/O port is controlled by a programmable communications interface for operation in most synchronous or asynchronous data transmission formats. Parallel I/O is implemented with a basic 24 lines controlled by a programmable peripheral interface.

An on-board programmable interrupt controller allows the system to handle up to eight levels of interrupt priority under software control. The CPU operates at 8 MHz and has the enhanced 80186 instruction set. The processor board contains two iSBX bus connectors for piggyback expansion modules.

Product packaging also reflects the targeted industrial market. The enclosure is metal, rather than plastic, and the top of the unit is hinged for easy access to all internal parts. This permits a user to run an interface directly off the processor board in addition to external port connectors.

The chassis is designed with six board slots, of which three are intended for customer-specified modules. This is a tool for system integrators who specialize in factory automation, test systems, process control, R&D laboratories, and a multitude of other on-line applications where system portability is important.

The product can monitor units under test concurrent with statistical analysis or data processing applications. As a software development tool, the system can do large compiling jobs concurrent with code writing or editing.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Translators and Utilities for Program Development

3

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

intel

TRANSLATORS AND UTILITIES FOR PROGRAM DEVELOPMENT

Intel offers an extensive selection of program development tools for its microprocessor (8080, 8086, 8088, 80186, 80286) and microcontroller (8048, 8051, 8096 etc.) families. These tools include translators and programming utilities such as linkers, relocators, and library managers. These program development tools are high quality, time tested tools for the professional. Based on a set of well-defined standards, they provide an integrated development environment. The result is an extremely flexible and productive program development environment.

A LANGUAGE FOR EVERY NEED

The iAPX-86 family has the most comprehensive set of translators available for a microprocessor. These include a macro assembler and compilers for PL/M, Pascal, FORTRAN, and C (see Table 1). The macro assembler produces the most optimum code. PL/M is the most popular 8086 language for systems programming and provides the best of both optimal code and high level language capabilities.

The main advantage of 'C' is portability across different target machines. Pascal and FORTRAN are used extensively for applications programming. To allow applications to be portable, Pascal and FORTRAN conform to ISO and ANSI77 standards respectively, with many useful extensions for microprocessor applications.

Intel's microcontroller family (8048, 8051, 8096 etc.) is similarly the best supported in the industry. PL/M-51 was the first high level language ever to be introduced for a microcontroller. The 8096 is similarly supported with PL/M-96. Every microcontroller in the family is supported with an assembler and linkage utilities.

USE A MIXTURE OF LANGUAGES FOR MAXIMUM FLEXIBILITY

Programs are typically decomposed into modules to exploit the many benefits of modular programming. Intel's integrated programming technology allows different modules of the same program to be programmed in a variety of languages. For instance, the most performance-sensitive system modules may be coded in assembler or in PL/M. The application modules, on the other hand, can be written in Pascal to speed up programming. The system and application modules can then be linked into one program using the linker. Hence, the various modules of a program can each be coded in the most suitable programming language.

UTILITIES ENHANCE PROGRAMMING PRODUCTIVITY

A set of utilities is provided to support modular and position independent programming. The linkers combine the constituent modules of a program into one system. A locator is provided to position the code in memory. This allows code to be placed in appropriate ROM and RAM locations. Also, coding can be done in a position-independent way. The librarian provides a structured way of organizing frequently used routines. The routines needed by a particular program can be linked in by the linker. The linker automatically selects only those modules from the libaray that are needed by the program. For the protected, virtual-memory, and multi-tasking processor iAPX 286, a sophisticated operating system configuration utility BUILD-286, is provided.

FULL RANGE OF DEBUG SUPPORT

The programming tools are integrated with the debugging tools via the well-defined Intel object module format standard. iAPX-86 family programs may be debugged using any of the Intel 8086 debug tools. This includes PSCOPE which provides source level software debug, and the ICE products which provide in-target real-time debug. Microcontroller software is similarly supported by the various emulators and ICE units.

CHOOSE FROM A VARIETY OF HOST CONFIGURATIONS

The programming tools are provided on a variety of development host environments to meet the needs of different project sizes and development budgets (see Table 1). The environments span personal development systems (iPDS), stand alone development systems, network development systems (NDS-II) and even the *VAX/VMS microcomputer. The programming tools work identically, no matter which of the available host configurations is chosen. This allows the user to grow his development environment, as his needs grow, without impacting previous investment in software.

* VAX/VMS is a trademark of Digital Equipment Corporation.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Table 1. Intel Translator/Host Summary

Language	Component Family	Host Code
Macro Assembler + Utilities	2920	1,2
	MCS-85 Family	1
	MCS-48 Family	1
	MCS-51 Family	1
	iACX-96 Family	2
	iAPX-86 Family	1,2,3
	iAPX-286 (Protected Mode)	2,3
PL/M	MCS-85 Family	1
	MCS-51 Family	1
	iACX-96 Family	2
	iAPX-86 Family	1,2,3
	iAPX-286 (Protected Mode)	2,3
PASCAL	MCS-85 Family	1
	iAPX-86 Family	2,3
	iAPX-286 (Protected Mode)	2
FORTRAN	MCS-85 Family	1
	iAPX-86 Family	2
	iAPX-286 (Protected Mode)	2*
"C"	iAPX-86 Family	2,3*
	iAPX-286 (Protected Mode)	2*,3*
Ada	iAPX-86 Family	3*
	iAPX-286 (Protected Mode)	3*

NOTE: * = Planned

HOST CODES
1 = 8085 Based Development System
2 = iAPX-86 family Based Development System
3 = VAX/VMS Minicomputer

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

PL/M 80
HIGH LEVEL PROGRAMMING LANGUAGE

- **Provides Resident Operation on Intellec® Microcomputer Development System and Intellec® Series II Microcomputer Development Systems**
- **Produces Relocatable and Linkable Object Code**
- **Sophisticated Code Optimization Reduces Application Memory Requirements**
- **Speeds Project Completion with Increased Programmer Productivity**
- **Cuts Software Development and Maintenance Costs**
- **Improves Product Reliability with Simplified Language and Consequent Error Reduction**
- **Eases Enhancement as System Capabilities Expand**

The PL/M 80 High Level Programming Language Intellec Resident Compiler is an advanced, high level programming language for Intel 8080 and 8085 microprocessors, iSBC-80 OEM computer systems, and Intellec microcomputer development systems. PL/M has been substantially enhanced since its introduction in 1973 and has become one of the most effective and powerful microprocessor systems implementation tools available. It is easy to learn, facilitates rapid program development and debugging, and significantly reduces maintenance costs. PL/M is an algorithmic language in which program statements naturally express the algorithm to be programmed, thus freeing programmers to concentrate on system development rather than assembly language details (such as register allocation, meanings of assembler mnemonics, etc.). The PL/M compiler efficiently converts free-form PL/M programs into equivalent 8080/8085 instructions. Substantially fewer PL/M statements are necessary for a given application than would be using assembly language or machine code. Since PL/M programs are problem oriented and thus more compact, programming in PL/M results in a high degree of productivity during development efforts, resulting in significant cost reduction in software development and maintenance for the user.



©INTEL CORPORATION, 1983

MAY 1983
ORDER NUMBER:210327-002

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

FUNCTIONAL DESCRIPTION

The PL/M compiler is an efficient multiphase compiler that accepts source programs, translates them into object code, and produces requested listings. After compilation, the object program may be first linked to other modules, then located to a specific area of memory, and finally executed. The diagram shown in Figure 1 illustrates a program development cycle where the program consists of three modules: PL/M, FORTRAN, and assembly language. A typical PL/M compiler procedure is shown in Table 1.

Features

Major features of the Intel PL/M 80 compiler and programming language include:

Resident Operation — on Intellec microcomputer development systems eliminates the need for a large in-house computer or costly timesharing system.

Object Code Generation — of relocatable and linkable object codes permits PL/M program development and debugging in small modules, which may be easily linked with other modules and/or library routines to form a complete application.

Extensive Code Optimization — including compile time arithmetic, constant subscript resolution, and common subexpression elimination, results in generation of short, efficient CPU instruction sequences.

Symbolic Debugging — fully supported in the PL/M compiler and ICE-85 in-circuit emulators.

Compile Time Options — includes general listing format commands, symbol table listing, cross reference listing, and "innerlist" of generated assembly language instructions.

Block Structure — aids in utilization of structured programming techniques.

Access — provided by high level PL/M statements to hardware resources (interrupt systems, absolute addresses, CPU input/output ports).

Data Definition — enables complex data structures to be defined at a high level.

Re-entrant Procedures — may be specified as a user option.

Benefits

PL/M is designed to be an efficient, cost-effective solution to the special requirements of microcomputer software development as illustrated by the following benefits of PL/M use:

Low Learning Effort — even for the novice programmer, because PL/M is easy to learn.

Earlier Project Completion — on critical projects, because PL/M substantially increases programmmer productivity while reducing program development time.

Lower Development Cost — because increased programmer productivity requiring less programming resources for a given function translates into lower software development costs.

Increased Reliability — because of PL/M's use of simple statements in the program algorithm, which are easier to correct and thus substantially reduce the risk of costly errors in systems that have already reached full production status.

Easier Enhancement and Maintenance — because programs written in PL/M are easier to read and easier to understand than assembly language, and thus are easier to enhance and maintain as system capabilities expand and future products are developed.

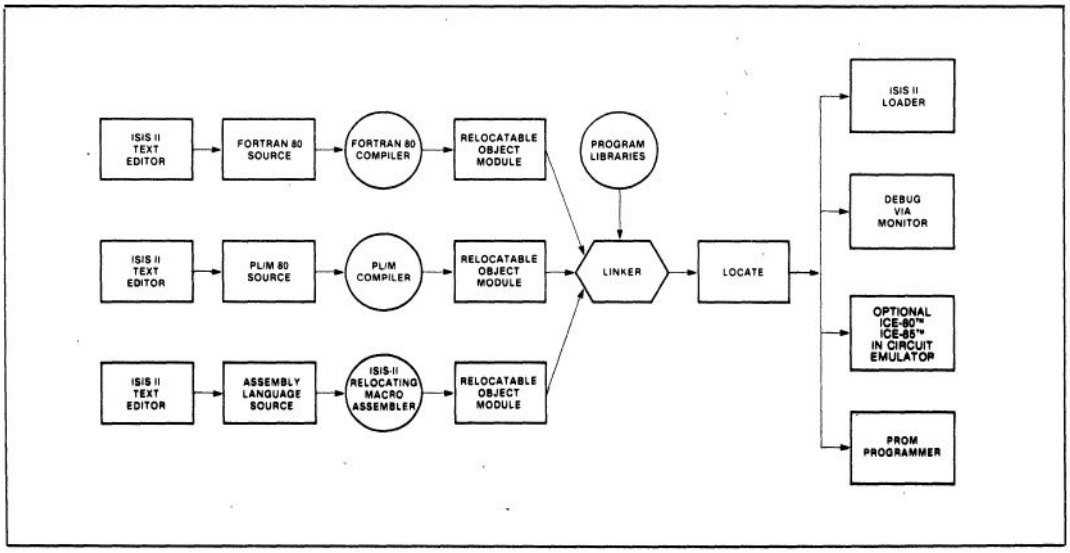


Figure 1. Program Development Cycle Block Diagram

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Simpler Project Development — because the Intellec microcomputer development system with resident PL/M 80 is all that is needed for developing and debugging software for 8080 and 8085 microcomputers, and the use of expensive (and remote) timesharing or large computers is consequently not required.

Table 1. PL/M-80 Compiler Sample Factorial Generator Procedure

```
                                    $OBJECT(:F1:FACT.OB2)
                                    $DEBUG
                                    $XREF
                                    $TITLE('FACTORIAL GENERATOR — PROCEDURE')
                                    $PAGEWIDTH(80)

 1                                  FACT:
                                    DO;

 2    1                             DECLARE NUMCH BYTE PUBLIC;

 3    1                             FACTORIAL: PROCEDURE (NUM,PTR) PUBLIC;
 4    2                                 DECLARE NUM BYTE, PTR ADDRESS;
 5    2                                 DECLARE DIGITS BASED PTR (161) BYTE;
 6    2                                 DECLARE (I,C,M) BYTE;

 7    2                                 NUMCH = 1; DIGITS(1) = 1;
 9    2                                 DO M = 1 TO NUM;
10    3                                     C = 0;
11    3                                     DO I = 1 TO NUMCH;
12    4                                         DIGITS(I) = DIGITS(I)*M + C;
13    4                                         C = DIGITS(I)/10;
14    4                                         DIGITS(I) = DIGITS(I) — 10*C;
15    4                                     END;

16    3                                     IF C<>0 THEN
17    3                                     DO;
18    4                                         NUMCH = NUMCH + 1; DIGITS(NUMCH) = C;
20    4                                         C = DIGITS(NUMCH)/10;
21    4                                         DIGITS(NUMCH) = DIGITS(NUMCH) — 10*C;
22    4                                     END
                                        END;

24    2                             END FACTORIAL;

25    1                             END;
```

SPECIFICATIONS

OPERATING ENVIRONMENT

Intel Microcomputer Development Systems
(Series II, Series III, Series IV)
Intel Personal Development System

DOCUMENTATION

PL/M 80 Programming Manual
ISIS-II PL/M 80 Compiler Operator's Manual

ORDERING INFORMATION

Product Code		Description
MDS	PLM	PL/M 80 High Level Language Compiler. Needs Software License

SUPPORT:

Hotline Telephone Support, Software Performance Report (SPR), Software Updates, Technical Reports, and Monthly Technical Newsletters are available

*MDS is an ordering code only and is not used as a product or trademark. 'MDS' is a registered trademark of Mohawk Data Sciences Corporation

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

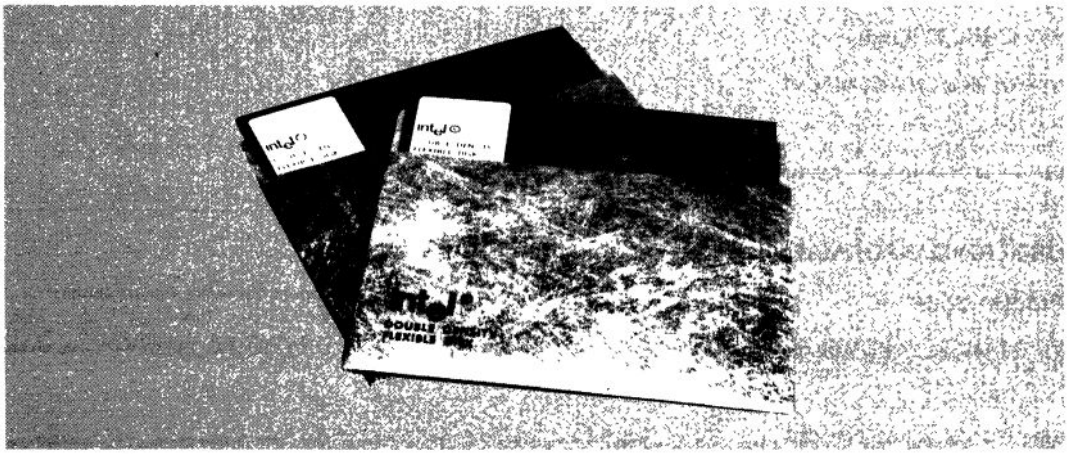
FORTRAN 80
8080/8085 ANS FORTRAN 77
INTELLEC® RESIDENT COMPILER

- **Meets ANS FORTRAN 77 Subset Language Specification plus adds Intel microprocessor extensions**
- **Supports Intel Floating Point Standard with the FORTRAN 80 software routines, the iSBC-310™ High Speed Mathematics Board, or the iSBC-332™ math multimodule**
- **Executes on Intellec Microcomputer Development System, Intellec Series II Microcomputer Development System, and Personal Development System**
- **Supports full symbolic debugging with ICE-80™ and ICE-85™**
- **Produces relocatable and linkable object code compatible with resident PL/M 80 and 8080/8085 Macro Assembler**
- **Provides optional run-time library to execute in RMX-80™ environment**
- **Has well defined I/O interface for configuration with user-supplied drivers**

FORTRAN 80 is a computer industry-standard, high-level programming language and compiler that translates FORTRAN statements into relocatable object modules. When the object modules are linked together and located into absolute program modules, they are suitable for execution on Intel 8080/8085 Microprocessors, iSBC-80 OEM Computer Systems, Intellec Microcomputer Development Systems and Personal Development Systems. FORTRAN 80 meets the ANS FORTRAN 77 Language Subset Specification[1]. In addition, extensions designed specifically for microprocessor applications are included. The compiler operates on the Intellec Microcomputer Development System and Personal Development System under the ISIS-II Disk Operating Systems and produces efficient relocatable object modules that are compatible for linkage with PL/M 80 and 8080/8085 Macro Assembler modules.

The ANS FORTRAN 77 language specification offers many powerful extensions to the FORTRAN language that are especially well suited to Intel 8080/8085 Microprocessor software development. Because FORTRAN 80 conforms to the ANS FORTRAN 77 standard, the user is assured of compatibility with existing FORTRAN software that meets the standard as well as a guarantee of upward compatibility to other computer systems supporting an ANS FORTRAN 77 Compiler.

[1] ANSI X3J3/90



©INTEL CORPORATION, 1983

MAY 1983
ORDER NUMBER:400610-001

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

FORTRAN 80 LANGUAGE FEATURES

Major ANS FORTRAN 77 features supported by the Intel FORTRAN 80 Programming Language include:

- Structured Programming is supported with the IF ... THEN ... ELSE IF ... ELSE ... END IF constructs.
- CHARACTER data type permits alphanumeric data to be handled as strings rather than characters stored in array elements.
- Full I/O capabilities include:
  - — Sequential and Direct Access files
  - — Error handling facilities
  - — Formatted, Free-formatted, and Unformatted data representation
  - — Internal (in-memory) file units provide capability to format and reformat data in internal memory buffers
  - — List Directed Formatting
- Supports arrays of up to seven dimensions.
- Supports logical operators
  - .EQV. — Logical equivalence
  - .NEQV. — Logical nonequivalence

Major extensions to FORTRAN 77 in Intel FORTRAN-80 include:

- Direct 8080/8085 port I/O supported by intrinsic subroutines.
- Binary and Hexadecimal integer constants.
- Well defined interface to FORTRAN-80 I/O statements (READ, OPEN, etc.), allowing easy use of user-supplied I/O drivers.
- User-defined INTEGER storage lengths of 1, 2 or 4 bytes.
- User-defined LOGICAL storage lengths of 1, 2 or 4 bytes.
- REAL STORAGE lengths of 4 bytes.
- Bitwise Boolean operations using logical operators on integer values.
- Hollerith data constants.
- Implicit extension of the length of an integer or logical expression to the length of the left-hand side in an assignment statement.
- A format descriptor to suppress carriage return on a terminal output device at the end of the record.

FORTRAN 80 COMPILER FEATURES

- Supports multiple compilation units in single source file.
- Optional Assembly Language code listing.
- Comprehensive cross-reference, symbol attribute and error listing.
- Compiler controls and directives are compatible with other Intel language translators.
- Optional Reentrancy.
- User-defined default storage lengths.
- Optional FORTRAN 66 Do Loop semantics.
- Source files may be prepared in free format.
- The INCLUDE control permits specified source files to be combined into a compilation unit at compile time.
- Transparent interface for software and hardware floating point support, allowing either to be chosen at time of linking.

FORTRAN 80 BENEFITS

FORTRAN 80 provides a means of developing application software for Intel MCS-80/85 products in a familiar, widely accepted, and computer industry-standardized programming language. FORTRAN 80 will greatly enhance the user's ability to provide cost-effective solutions to software development for Intel microprocessors as illustrated by the following:

- *Completely Complementary to Existing Intel Software Design Tools* — Object modules are linkable with new or existing Assembly Language and PL/M Modules.
- *Incremental Runtime Library Support* — Runtime overhead is limited only to facilities required by the program.
- *Low Learning Effort* — FORTRAN 80, like PL/M, is easy to learn and use. Existing FORTRAN software can be ported to FORTRAN 80, and programs developed in FORTRAN 80 can be run on any other computer with ANS FORTRAN 77.
- *Earlier Project Completion* — Critical projects are completed earlier than otherwise possible because FORTRAN 80 will substantially increase programmer productivity, and is complementary to PL/M Modules by providing comprehensive arithmetic, I/O formatting, and data management support in the language.
- *Lower Development Cost* — Increases in programmer productivity translates into lower software development costs because less programming resources are required for a given function.
- *Increased Reliability* — The nature of high-level languages, including FORTAN 80, is that they lend themselves to simple statements of the program algorithm. This substantially reduces the risk of costly errors in systems that have already reached production status.
- *Easier Enhancements and Maintenance* — Like PL/M, program modules written in FORTRAN 80 are easier to read and understand than assembly language. This means it is easier to enhance and maintain FORTRAN 80 programs as system capabilities expand and future products are developed.
- *Comprehensive, Yet Simple Project Development* — The Intellec Microcomputer Development System and Personal Development System, with the 8080/8085 Macro Assembler, PL/M 80 and FORTRAN 80 are the most comprehensive software design facilities available for the Intel MCS-80/85 Microprocessor family. This reduces development time and cost because expensive (and remote) timesharing or large computers are not required.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

SAMPLE FORTRAN-80 SOURCE PROGRAM LISTING

```
*    **  THIS PROGRAM IS AN EXAMPLE OF ISIS-II FORTRAN-80 THAT
*    **  CONVERTS TEMPERATURE BETWEEN CELSIUS AND FARENHEIT

      PROGRAM CONVRT

      CHARACTER*1 CHOICE, SCALE

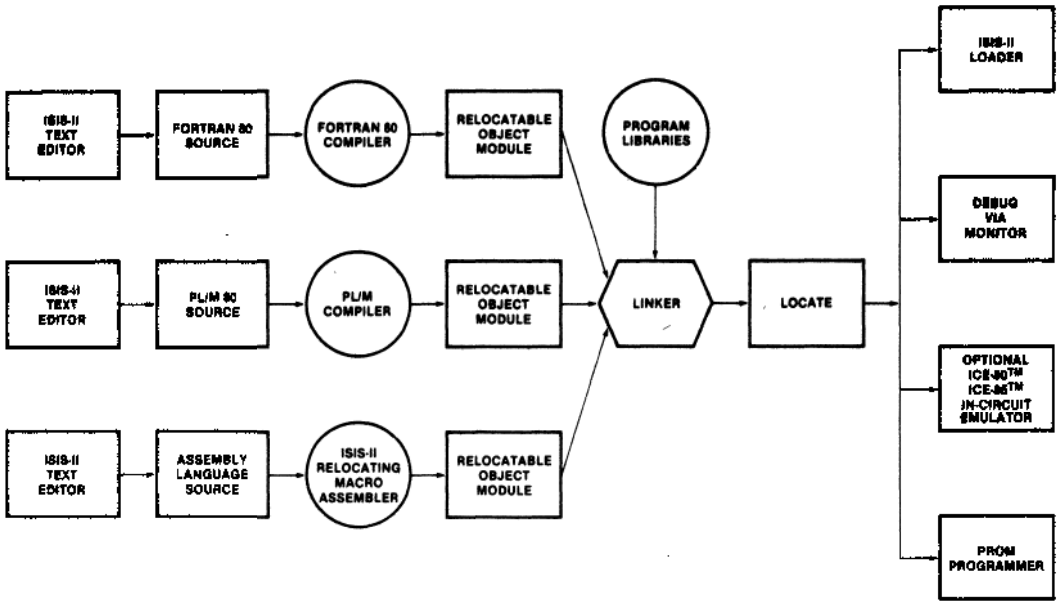
      PRINT 100
*     **  ENTER CONVERSION SCALE (C OR F)
10    PRINT 200
      READ (5,300) SCALE

        IF (SCALE .EQ. 'C')
     +     THEN
              PRINT 400
*             **  ENTER THE NUMBER OF DEGREES FARENHEIT
              READ (5,*) DEGF
              DEGC = 5./9.*(DEGF-32)
*             **  PRINT THE ANSWER
              WRITE (6,500) DEGF,DEGC
*             **  RUN AGAIN?
20            PRINT 600
              READ (5,300) CHOICE
                 IF (CHOICE .EQ. 'Y')
     +              THEN
                       GOTO 10
                 ELSE IF (CHOICE .EQ. 'N')
     +              THEN
                       CALL EXIT
                 ELSE
                    GOTO 20
                 END IF
        ELSE IF (SCALE .EQ. 'F')
     +     THEN
*             **  CONVERT FROM FARENHEIT TO CELSIUS
              PRINT 700
              READ (5,*) DEGC
              DEGF = 9./5.*DEGC+32.
*             **  PRINT THE ANSWER
              WRITE (6,800) DEGC,DEGF
              GOTO 20
        ELSE
*          **  NOT A VALID ENTRY FOR THE SCALE
           WRITE (6,900) SCALE
           GOTO 10
        END IF
100   FORMAT(' TEMPERATURE CONVERSION PROGRAM',//,
     +' TYPE C FOR FARENHEIT TO CELSIUS OR',/,
     +' TYPE F FOR CELSIUS TO FARENHEIT',//)
200   FORMAT(/,' CONVERSION? ',$)
300   FORMAT(A1)
400   FORMAT(/,'ENTER DEGREES FARENHEIT: ',$)
500   FORMAT(/,F7.2,' DEGREES FARENHEIT = ',F7.2,' DEGREES CELSIUS')
600   FORMAT(/,' AGAIN (Y OR N)? ',$)
700   FORMAT(/,' ENTER DEGREES CELSIUS: ',$)
800   FORMAT(/,F7.2,' DEGREES CELSIUS = ',F7.2,' DEGREES FARENHEIT',/)
900   FORMAT(/,1H ,A1,' NOT A VALID CHOICE - TRY AGAIN!',/)
      END
```


Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

FORTRAN 80

The FORTRAN 80 Compiler is an efficient, multiphase compiler that accepts source programs, translates them into relocatable object code, and produces requested listings. After compilation, the object program may be linked to other modules, located to a specific area of memory, then executed. The diagram shown below illustrates a program development cycle where the program consists of modules created by FORTRAN 80, PL/M 80 and the 8080/8085 Macro Assembler.



SPECIFICATIONS

OPERATING ENVIRONMENT

Required Hardware:

1. Intel Microcomputer Development Systems
   —MDS-800 and Series II

   or

2. Personal Development System

DOCUMENTATION PACKAGE

FORTRAN-80 Programming Manual

ISIS-II FORTRAN-80 Compiler Operator's Manual

FORTRAN-80 Programming Reference Card

ORDERING INFORMATION

PART NO.	DESCRIPTION
Model MDS-301	FORTRAN 80 Compiler for Intellec Microcomputer Development Systems

Requires Software License.

SUPPORT

Intel offers several levels of support for this product which are explained in detail in the price list. Please consult the price list for a description of the support options available.

*MDS is an ordering code only and is not used as a product name or trademark. MDS is a registered trademark of Mohawk Data Sciences Corporation.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

MICROSOFT*, INC. BASIC-80 INTERPRETER SOFTWARE PACKAGE

- **Compatible with other Microsoft BASIC compilers and interpreters**
- **Sophisticated string handling and structured programming features for applications development**
- **Direct transfer of BASIC programs to the 8085, 8086 and 8088**
- **Random and sequential file manipulation where random file record length is user-definable**
- **Read or write memory location capabilities**
- **Meets the requirements for the ANSI subset standard for BASIC, and supports many enhancements**
- **Extensive text editing features built-in**
- **Automatic line number generation and renumbering**
- **Supports assembly language subroutine calls**
- **Trace facilities for easier debugging**

BASIC Release 5.0 from Microsoft is an extensive implementation of BASIC. Microsoft BASIC gives users what they want from a BASIC—ease of use plus the features that are comparable to a minicomputer or large mainframe.

BASIC-80 meets the requirements for the ANSI subset standard for BASIC, as set forth in document BSRX3.60-1978. It supports many unique features rarely found in other BASICs.

FEATURES

—Four variable types: Integer (−32768, +32767), String (up to 255 characters), Single-Precision Floating Point (7 digits), Double-Precision Floating Point (16 digits).

—Trace facilities (TRON/TROFF) for easier debugging.

—Error trapping using the ON ERROR GOTO statement.

—PEEK and POKE statements to read or write any memory location.

—Automatic line number generation and renumbering, including reference line numbers.

—Matrices with up to 255 dimensions.

—Boolean operators OR, AND, NOT, XOR, EQV, IMP.

—Formatted output using the PRINT USING facility, including asterisk fill, floating dollar sign, scientific notation, trailing sign, and comma insertion.

—Direct access to I/O ports with the INP and OUT functions.

—Extensive program editing facilities via EDIT command and EDIT mode subcommands.

—Assembly language subroutine calls (up to 10 per program) are supported.

—IF/THEN/ELSE and nested IF/THEN/ELSE constructs.

—Supports variable-length random and sequential disk files with a complete set of file manipulation statements: OPEN, CLOSE, GET, PUT, KILL, NAME, MERGE.

©INTEL CORPORATION, 1983

MAY 1983
AFN-02086C

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BASIC-80 Commands, Statements, Functions

AUTO	RENUM	NAME
LIST	WIDTH	SAVE
NULL	CONT	EDIT
TROFF	MERGE	NEW
CLEAR	RUN	TRON
LOAD	DELETE	

Program Statements

CALL	RANDOMIZE	RETURN
GOSUB	COMMON	WAIT
END	DEF FN	ON GOSUB
GOTO	ERROR	DIM
STOP	POKE	FOR/NEXT/
WHILE/	RESUME	STEP
WEND	SWAP	IF/THEN/
CHAIN	DEFDBL	ELSE
DEF USR	DEFSTR	ON ERROR
LET	DEFSNG	GOTO
REM	DEFINT	OPTION BASE

Input/Output Statements and Functions

CLOSE	GET	NAME
KILL	POS	PUT
OUT	FIELD	EOF
RESTORE	LSET/RSET	SPC
READ	PRINT	INKEY$
TAB	USING	INPUT
DATA	LOC	OPEN
LINE	MKI$	CVD
INPUT	MKS$	CVI
PRINT	MKD$	CVS
WRITE	LLIST	
LPRINT	LPOS	

Arithmetic Functions

ABS	SIN	LOG
INT	CDBL	FIX
SGN	CSNG	COS
ATN	CINT	RND
EXP	SQR	TAN

String Functions

ASC	STR$	INSTR
LEN	HEX$	RIGHT$
STRING$	OCT$	MID$
CHR$	VAL	SPACE$
LEFT$		

Operators

÷	*	XOR
^	<=	NOT
<	+	EQV
>	< >	MOD
-	\	IMP
/	>=	OR
		AND

Special Functions

ERL	ERR	VARPTR
USR	FRE	PEEK

SPECIFICATIONS

Operating Environment

The standard disk version of Microsoft BASIC-80 occupies 24K bytes of memory. Microsoft BASIC-80 Interpreter is compatible with Intel's ISIS operating system or CP/M* operating system.

Required Hardware

Intellec Microcomputer Development System

—iPDS (Personal Development System)
—minimum of 1 diskette drive

Required Software

ISIS Operating System or CP/M Operating System.

Documentation Package

One copy of each manual is supplied with the software package.

Description
BASIC-80 Reference Manual
BASIC Reference Book

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

ORDERING INFORMATION

Order Code	Description
SD102CPM80F	Microsoft BASIC-80 Interpreter Software Package, CP/M version (Double-Sided, Double Density 5¼" Floppy) IPDS format
SD102ISS80F	Microsoft BASIC-80 Interpreter Software Package, ISIS version (Double-Sided, Double Density 5¼" Floppy) IPDS format

SUPPORT

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.

An Intel Software License required.
*Microsoft is a trademark of Microsoft, Inc.
*CP/M is a registered trademark of Digital Research, Inc.
*MP/M-II is a trademark of Digital Research, Inc.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

MICROSOFT*, INC. BASIC-80 COMPILER SOFTWARE PACKAGE

- **Produces highly optimized, true machine code**
- **Compiled programs are fast and compact because of extensive optimizations performed during compilation**
- **Supports all the commercial language features of the Microsoft BASIC interpreter (except direct mode commands)**
- **Supports double-precision transcendental functions**
- **Machine code for application programs may be placed on diskette, ROM, or other Media**
- **Provides source program security because only compiled code need be distributed to end-users**
- **Loader format identical to Microsoft's MACRO-80 assembler, COBOL-80 compiler, and FORTRAN-80 compiler: Compiled BASIC programs can be loaded and linked with any of these languages**

Microsoft's BASIC-80 compiler is a powerful tool for programming BASIC applications or microprocessor system software. The single-pass compiler produces extremely efficient, optimized 8080 machine code that is in Microsoft-standard, relocatable binary format. Execution speed is typically 3–10 times faster than Microsoft's BASIC-80 interpreter.

FEATURES

Optimized, Compatible Object Code

The BASIC compiler produces object code that is highly optimized for speed and space, relocatable, and compatible with other Microsoft software products. The loader format is identical to that of the MACRO-80 assembler, COBOL-80 compiler and FORTRAN-80 compiler, so programs written in any one of these four languages can be loaded and linked together. The compiler can also provide a formatted listing of the machine code that is generated.

Compiled programs are fast and compact due to extensive optimizations performed during compilation:

—Expressions are reordered to minimize temporary storage and (wherever possible) to transform floating point division into multiplication.

—Constant multiplications are distributed to allow more complete constant folding.

—Constants are folded wherever possible. The expression reordering finds "hidden" constant operations.

—Peephole optimizations are performed, including strength reduction.

—The code generator is template-driven, allowing optimal sequences to be generated for the most commonly used operations.

—String operations and garbage collection are extremely fast.

Compiled BASIC-80 programs are the ideal end product for BASIC applications' programmers. The machine code for any application program may be placed on a diskette, ROM, or other media. The program not only runs faster than with the interpreter, but the BASIC source program need not be distributed. Thus the original application program is protected from unauthorized alteration.

Language Features

The Microsoft BASIC-80 Compiler supports all the commercial language features of Microsoft BASIC-80, except those commands that are not usable in the compiler environment (i.e., direct mode commands such as LOAD, AUTO, SAVE, EDIT, etc.). That means you get the BASIC language compatible with other Microsoft BASIC packages.

© INTEL CORPORATION 1983

MAY 1983
ORDER NUMBER:210247-002

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

In addition, the compiler supports double-precision transcendental functions (SIN, COS, TAN, ATN, LOG, EXP, SQR), %INCLUDE, CHAIN and COMMON. The %INCLUDE compiler directive brings another source file into the compilation without retyping the main source file.

BRUN Runtime Module

The BRUN runtime module contains the most common runtime routines needed for most programs. Using the BRUN module provides faster link loading of program modules and allows the user to link much larger programs because the runtime routine library does not reside in memory during linking. The executable files saved on disk are also much smaller since the BRUN module exists separately.

Utility Software Package

The BASIC-80 package includes the Microsoft Utility Software Package. The Utility Software Package includes the MACRO-80 macro assembler, the LINK-80 linking loader and the CREF-80 Cross-Reference Facility. Refer to the description of the Microsoft Utility Software Package for full details.

SPECIFICATIONS

Operating Environment

The BASIC Compiler requires a minimum of 34K bytes of memory (exclusive of the operating system). Microsoft recommends that 48K bytes be available for compiling medium to large programs. The compiler itself occupies about 28K bytes. At runtime, the BRUN module occupies approximately 15.5K bytes. If, as an option, the BRUN module is not used, the runtime library occupies 8K–18K bytes.

Required Hardware

Intellec Microcomputer Development System
—iPDS (Personal Development System)
—minimum of 1 diskette drive

Required Software

CP/M* Operating System

Documentation Package

One copy of each manual is supplied with the software package.

Description
BASIC Compiler User's Manual
BASIC-80 Reference Manual
BASIC Reference Book
Microsoft Utility Software Manual

(Specify by Alpha Character when ordering.)

ORDERING INFORMATION

Order Code	Description
SD124CPM80F	Microsoft BASIC-80 Compiler Software Package, CP/M version (iPDS Format)

SUPPORT:

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.

An Intel Software License required
*Microsoft is a trademark of Microsoft, Inc
*CP/M is a registered trademark of Digital Research, Inc
*MP/M-II is a trademark of Digital Research, Inc

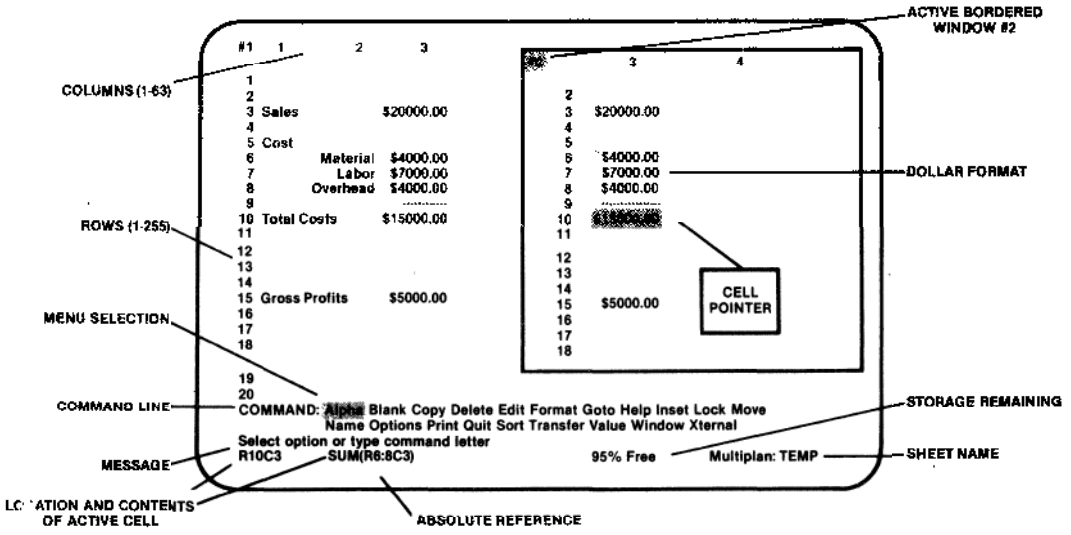
Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

MICROSOFT*
MULTIPLAN* SPREADSHEET

- **Simplifies the design and use of very large spreadsheets, and multiple interrelated spreadsheets**
- **Automatically updates subtotals, totals, percentages, growth curves, etc**
- **Can perform multiple iterations to solve closed-loop problems**
- **Formulas automatically revised when reordering rows and columns in displays**
- **Can be used in time, monetary, and inventory budgeting**
- **Wide array of sophisticated functions to simplify formulas**
- **Cells and areas can be named for clarity**
- **Can reference and update several interrelated spreadsheets at once**
- **Simple to use, intuitive commands. Single keystroke command entry**
- **"Windows" allow several portions of large sheets to be viewed at once**
- **Contains the features of the most popular spreadsheet programs, as well as its contribution of new features**

Multiplan is a productivity tool designed to help the user to analyze data in spreadsheet format. As an aid to both business and personal needs, Multiplan is an extremely powerful modeling and planning tool.

Multiplan is easy to learn and use, yet its versatility is enhanced by the skill of the user. Multiplan allows the user to operate in as intuitive a way as possible, and its widespread capabilities allow accomplishment of a variety of tasks. Advanced users are unencumbered by simplifying features, and have enough power to satisfy their needs.



Typical Multiplan Screen Display

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products: BXP, CREDIT, i, iCE, iCS, Im, Insite, Intel, INTEL, Intelevision, Intellink, Intellec, iMMX, iOSP, iPDS, iRMX, iSBC, iSBX, Library Manager, MCS, MULTIMODULE, Megachassis, Micromainframe, MULTIBUS, Multichannel, Plug-A-Bubble, PROMPT, Promware, RUPI, RMX/80, System 2000, UPI, and the combination of iCS, iRMX, iSSC, iSBX, ICE, I²ICE, MCS, or UPI and numerical suffix. Intel Corporation Assumes No Responsibility for the use of Any Circuitry Other Than Circuitry Embodied in an Intel product. No Other Patent Licenses are implied. © INTEL CORPORATION, ©INTEL CORPORATION, 1983
*Microsoft & Multiplan are trademarks of Microsoft Corp

MAY 1983
ORDER NUMBER:210767-002

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

FEATURES

— Names can be used to express "cells" (worksheet elements), or groups of cells. These names, in turn, can be used as parts of formulas and commands. Named areas can be combined in various ways for ease of use.

— A wide range of functions unique to Multiplan is available in addition to the functions typical to the most popular spreadsheet programs. These functions allow the user to select windows, sort data, draw from other worksheets, and a number of other important operations.

— Expressions can be clarified by the use of names as in "PROFIT = SALES – COSTS" rather than "R12C1 = R1C3 – R5C12".

— Active sheets can draw data automatically from inactive "supporting" sheets through the use of named cells and areas. This unique feature allows the user to streamline the processing of data, and to generate an entire pad full of interrelated spreadsheets.

— Multiplan offers a worksheet size of up to 255 rows by 63 columns, a broad worksheet simulator in which words, numbers, and formulas may be entered into information cells. Added to the access of data in inactive sheets, this large sheet size allows the user to perform very rigorous analyses in a minimum amount of time.

— With Multiplan the user gains the capability to plan against several different situations to allow comparison of one set of circumstances against another. A good example of this would be the generation of several sheets, one based on steady growth versus others based on several potential problems. This way, contingency planning will become less tedious and more effective.

— By altering a single critical number, the impact on other dependent numbers will be automatically updated to help the user observe sensitivities and interdependencies. This helps the user to plan resources efficiently, and schedule more effectively.

— Multiplan overcomes the limitations of paper worksheets by allowing the user to instantly move, insert, or delete entire rows or columns of data. The remaining rows, columns, or free space will expand or contract automatically as necessary, thereby eliminating the costly and tiresome work of typing or hand-printing the worksheet over and over.

— All commands can be invoked by a single keystroke and selections are menu driven. Multiplan even offers proposed responses to commands, to encourage its use by even the most unskilled user. Multiplan's commands, prompts, and messages, as well as the screen and keyboard, communicate with each other and the user directly and naturally to allow the untrained user to accomplish objectives easily.

— A special edit area helps the user to make additions and deletions quickly and easily.

— Up to eight windows are available to allow users to view different parts of a very large worksheet simultaneously. The windows can be aligned, scrolled together, opened, or closed at will.

— An iteration option allows the simulation of closed-loop problems involving mutually interdependant formulas. The number of iterations can be chosen, or iterations can continue until a given constraint is met.

— Formulas can be moved from one worksheet location to another without having to be rewritten by the user.

— Reference to a particular cell need not be in absolute terms, but can be expressed as a location relative to other cells. A formula containing this sort of relative reference may be copied into other cells and will be automatically changed to reflect its new position.

— The sheet display may be redesigned or formatted in various ways without affecting the data stored in Multiplan. Thus, the same data can be presented in different order in different reports with a minimum of effort.

AFN-00649A

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Commands

The following is a brief list of commands available under Multiplan. All of these commands are invoked by the single keystroke of their first letter (i.e. "C" for Copy or "F" for Format) with the exception of eXternal, which is invoked by typing an "X."

Several of the commands offer a number of selections of operational modes, which are displayed when the command is invoked. In order to choose a mode, either press the TAB key until the cursor rests over the selected mode, then hit RETURN, or type the first letter of the selected mode, then hit RETURN.

For more detailed descriptions of the commands, please see the Multiplan User's Manual.

ALPHA
Replaces the contents of the active cell with a character string. If the active cell already contains a string, that string is the proposed response of the command, so that it can be edited.

BLANK
Deletes contents of all specified cells. Names are not affected; if a cell was referred to by a name before use of this command, that name will still apply.

COPY
Presents a choice of three ways of copying the contents of some cells into other cells. To duplicate one cell across several to its right, choose Right. To duplicate one cell across several below it, choose Down. To copy any cell or cells to any others, choose From.

DELETE
Presents a two-way choice to delete cells. To delete a row or rows, choose R. To delete a column or columns, choose C. To blank out the cells, use the Blank command.

EDIT
Makes contents of the active cell available for editing. Place the cell pointer on the cell to be edited and press E. The cell's contents are then placed on the command line for modification. The edit cursor is placed at the end of the current contents rather than highlighting the whole command, as is done for other defaults. If the cell contains a string, it is presented in double quotes. After having edited the cell's contents, press RETURN to put the changed contents back in the cell (or press ABORT to cancel any changes).

FORMAT
Presents a choice of three kinds of format adjustment. To set a specific format for a cell or group of cells, choose Cells. To set the width of a column or columns, choose Width. To set the default format—the format that applies wherever a specific format hasn't been set—choose Default.

GOTO
Presents a choice of ways to move the cell pointer over the sheet. To display a specific row and column, choose Row-col. To display a named area, choose Name.

HELP
Provides helpful information about Multiplan. When help is requested, the spreadsheet is replaced by text from the HELP file and the HELP command menu appears on the screen. Help is available in the areas of Applications, Commands, Editing, Formulas, and the Keyboard. The spreadsheet display is reinstated when the RESUME subcommand is entered.

INSERT
Presents a choice of ways to insert new cells into the sheet. To insert new rows choose Row. To insert new columns choose Column.

LOCK
Provides two ways to lock cells in protection against accidental change. Either individual cells or all cells containing formulas can be moved, deleted, formatted or sorted after having been locked, but their contents cannot be changed.

MOVE
Presents a choice of ways to move cells around the sheet. To move whole rows, choose Row. To move whole columns, choose Column.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Table 1. Multiplan Commands

ALPHA	— Replaces cell contents with a character string.
BLANK	— Clears cell contents.
COPY DOWN	— Used to fill a column with identical values.
COPY FROM	— Duplicates one or a number of cells to another location.
COPY RIGHT	— Used to make a row of identical values.
DELETE COLUMN	— Removes columns from the spreadsheet.
DELETE ROW	— Removes rows from the spreadsheet.
EDIT	— Allows editing of the contents of a single cell.
FORMAT CELLS	— Used to help align cells in a column.
FORMAT WIDTH	— Limits the width of all cells in a given column.
FORMAT DEFAULT CELLS	— Sets formats for all previously unformatted cells.
FORMAT DEFAULT WIDTH	— Sets formats for all previously unformatted columns.
FORMAT OPTIONS COMMAS	— Displays numbers with commas separating every third digit.
FORMAT OPTIONS FORMULAS	— Displays formulas instead of their values.
GOTO ROW-COL	— Moves the cell pointer to the specified row and column.
GOTO NAME	— Moves the cell pointer to the named area.
GOTO WINDOW	— Places the specified cell within the given window.
HELP APPLICATIONS	— Illustrates solutions to a number of common problems.
HELP COMMANDS	— Lists and describes all commands.
HELP EDITING	— Describes Editing functions.
HELP FORMULAS	— Gives Formula construction rules.
HELP KEYBOARD	— Explains special functions of the keyboard.
HELP NEXT	— Gives the next screenful of HELP text.
HELP PREVIOUS	— Gives the previous screenful from HELP call.
HELP RESUME	— Returns to the spreadsheet from HELP call.
HELP START	— Begins the HELP tutorial.
INSERT COLUMN	— Used to add a column to an existing spreadsheet.
INSERT ROW	— Used to add a row to an existing spreadsheet.
LOCK CELLS	— Protects the indicated cell from alteration.
LOCK FORMULAS	— Locks out alteration of all cells containing formulas or text.
MOVE COLUMN	— Changes the order of the columns on the sheet.
MOVE ROW	— Changes the order of the rows on the sheet.
NAME	— Assigns a name to a cell or number of cells.
OPTIONS	— Allows the user to disallow recalculation upon every change of a cell value, to mute the audible alarm, or to enable the iteration option.
PRINT FILE	— Outputs the spreadsheet to a diskette file.
PRINT MARGINS	— Sets up the margins on the printed output.
PRINT OPTIONS	— Allows optional printing modes to be used.
PRINT PRINTER	— Prints the spreadsheet on the system's printer.
QUIT	— Ends the Multiplan session without saving the active sheet.
SORT	— Sorts a range of rows to put values in a specified column into ascending or descending numerical order.
TRANSFER CLEAR	— Clears the active sheet.
TRANSFER DELETE	— Deletes the specified file.
TRANSFER LOAD	— Loads a sheet from the disk file.
TRANSFER OPTIONS	— Modifies the context of the following transfer operation.
TRANSFER RENAME	— Renames the active sheet.
TRANSFER SAVE	— Saves the active sheet on diskette.
VALUE	— Enters a value or formula into the active cell.
WINDOW BORDER	— Changes the border of the specified window.
WINDOW CLOSE	— Removes a window from the screen.
WINDOW LINK	— Sets or breaks link for synchronized scrolling between windows.
WINDOW SPLIT HORIZONTAL	— Horizontally divides a window into two windows.
WINDOW SPLIT VERTICAL	— Vertically divides one window into two windows.
WINDOW SPLIT TITLES	— Divides one window into two or four which scroll together.
XTERNAL COPY	— Copies data from an inactive sheet to the active sheet.
XTERNAL LIST	— Displays the relationships between the active sheet and the other sheets.
XTERNAL USE	— Sets a substitute name for a supporting sheet.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Commands (Continued)

NAME
Assigns a name to a cell or area of cells. The name defined may then be used wherever a reference to that cell or area is needed in a command or formula.

OPTIONS
The Options command can be used to set and reset various options provided with Multiplan.

The Recalc option controls how often Multiplan performs formula calculations. If the option is on, Multiplan recalculates all formulas whenever a cell is changed. If the option is off, recalculation is done only when the Recalc control key is pressed or during Transfer Save.

The Recalc option has an effect on how quickly Multiplan finishes entering a new value in a cell. The length of time Multiplan takes to recalculate the sheet depends on how many cells are in use, and on the complexity of the formulas in them. When you want to make a number of entries on a busy sheet, turn the Recalc option off to get the quickest response. Turn it on again when you are interested in seeing the effect of each change.

The Mute option silences Multiplan's audible alarm.

The Iterate option gives the user a means of solving problems which involve circular or "closed loop" references. Whereas formulas which count on each other's results (i.e., A = B + C, B = A + C) are disallowed in other spreadsheet programs, Multiplan allows spreadsheets with such references to be reiterated upon in an orderly manner either until a maximum number of iterations has been reached, or until a cell has reached a predetermined value.

PRINT
Presents a choice of four actions related to printing the active sheet. To begin printing, choose Go. To put printable output in a disk file, choose File. To set the margins that will be used on the printed output, choose Margins. To fix the part of the worksheet to be printed, or to insert a control line at the top of the output, choose Options.

QUIT
Ends the Multiplan session without saving the active sheet. Multiplan requests confirmation; if it is given, Multiplan terminates, returning control to the computer operating system. The active sheet is lost unless it has previously been saved.

SORT
Reorders the rows on the spreadsheet so that the data in a specified column appears in ascending or descending numerical order. The column to be sorted may contain numbers, text, or other values, and if such values are mixed, they are presented in ascending order numerically, alphabetically and by error value, after which any blank cells follow.

TRANSFER
Offers a choice of five commands, which affect an entire sheet.

To load a saved sheet, replacing the active sheet, choose Load.

To save the active sheet in a disk file, choose Save.

To give the active sheet a new name, choose Rename.

To clear the active sheet, deleting all its contents, and restoring all its default settings, choose Clear.

To delete the disk copy of the active sheet, choose Delete.

VALUE
This command is used to enter a formula or number into the active cell. VALUE may either be selected from the command menu or by typing a numerical value, a mathematical symbol, or a left parentheses.

WINDOW
Presents a choice of four things that can be done with windows.

To open a new window by splitting the active window horizontally or vertically, or to open a window used strictly for titles, choose Split.

To close a window by removing it from the screen, choose Close.

To synchronize scrolling of windows, choose Link.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

To move a window to a particular part of the sheet, choose Home.

To add or remove a decorative border around a window, choose Border.

XTERNAL
Presents a choice of actions relating to the use of data from other sheets in the formulas of the active sheet.

To copy data, or blocks of data from an inactive spreadsheet to the active sheet, choose Copy.

To display the relationships between the active sheet and other sheets, showing which sheets support (provide values for) the active one and which sheets depend on (use values from) the active sheet, choose List.

To assign a substitute name for an inactive sheet, specify Use.

Table 2. Multiplan Functions

ABS	— Calculates the absolute value of an argument.
AND	— True if, and only if, all values are true; otherwise returns false.
ATAN	— Gives the arctangent of an argument.
AVERAGE	— Returns the average of all cells referenced by up to 5 arguments.
COLUMN	— Gives the current column number.
COS	— Calculates an argument's cosine.
COUNT	— Finds the number of cells fitting the referenced criteria.
DOLLAR	— Formats numbers as dollar amounts.
EXP	— This is the inverse natural logarithm of the argument.
FALSE	— Returns the logical false value.
FIXED	— Rounds the first argument to the precision specified by the second.
IF	— Returns value specified after "THEN" if argument is true, or the "ELSE" specified value if false.
INDEX	— Returns the value of the cell in a named area offset by an index value.
INT	— Truncates the argument's fractional part.
ISERROR	— Returns true if, and only if, the argument is an error value.
ISNA	— Returns true if, and only if, the argument is an #N/A value.
LEN	— Gives the number of characters in the argument's string.
LN	— Calculates the natural logarithm of its argument.
LOG10	— Returns the common logarithm of its argument.
LOOKUP	— Used to search for dependent variables in a lookup table.
MAX	— Finds the largest numeric value in an area of cells.
MID	— Produces the middle characters of a string.
MIN	— Finds the smallest numeric value in an area of cells.
MOD	— Gives the remainder of the integer division of the two arguments.
NA	— Returns the #N/A value.
NOT	— Gives the logical inverse of the argument.
NPV	— Calculates the net present value of a constant annuity.
OR	— True if, and only if, any of the arguments are true; otherwise returns a false.
PI	— Returns PI (3.14159...).
REPT	— Forms a string consisting of a repeated substring.
ROUND	— Rounds the first argument to the precision specified by the second.
ROW	— Gives the current row number.
SIGN	— Performs the Signum function on the argument.
SIN	— Returns the sine of the argument.
SQRT	— Calculates the square root of the argument.
STDEV	— Calculates the standard deviation of the arguments.
SUM	— Adds the sum of all cells in a specified area.
TAN	— Calculates the tangent of the argument.
TRUE	— Returns the logical true value.
VALUE	— Used to extract numbers from strings.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

BENEFITS

Unlike other spreadsheet programs, Multiplan allows the user to create and view as many as eight different windows within the screen display area. Complete control is allowed over each window, allowing windows without borders, and the freezing and scrolling of title columns and rows.

Multiplan allows formulas to describe the contents of any cell. Formulas are written in a method similar to standard programming languages, and are evaluated according to priority of functions, a unique feature among spreadsheet programs. Parentheses are allowed to clarify the order of calculation. Formulas can use a string of characters as a variable name, and variables may be either numerical data, or strings of characters which may be manipulated to concatenate words and phrases. These are all unusually powerful and intuitively easy-to-learn features many of which are unique to Multiplan.

Multiplan gives the user an unusual amount of flexibility in rearranging the format or layout of a spreadsheet with its three forms of addressing: absolute, relative, or symbolic (by name). Any of the three can be combined in any order to produce the exact results needed in any case.

One of the features that sets Multiplan apart from other spreadsheet programs is the ability to name all cells. The NAME command allows the naming of single cells, an area of cells of any shape, or even a list of unconnected areas of cells. That name can then be used in functions, or even as a response in a command. NAME also allows the user to review all cell names in their proper position on the screen in order to reduce confusion.

Multiplan commands can be entered by single letters on the command lines, after which the program will fill in the rest of the command. This speeds the user through complex operations without leaving any doubt about their functions. Versatile commands handle not only single data cells, rows, or columns as do other spreadsheet programs, but these commands allow Multiplan to move multiple rows or columns, or insert, delete, or handle any rectangular area. All relative references are automatically adjusted to account for these changes.

Multiplan automatically updates all entries affected by a change in a single cell, without requiring the user to command it to do so. This feature allows the user to fiddle with numbers and test for sensitivities and trouble spots.

Another unique benefit of Multiplan is its ability to employ values from one sheet in the formulas of another. This "sheet linkage" can be used to construct a hierarchy of worksheets, with detailed worksheets feeding their totals to a summary worksheet. When a detail sheet is updated and saved on diskette, the dependent summary sheet will be automatically updated the next time it is loaded.

SPECIFICATIONS

Operating Environment

REQUIRED HARDWARE:

Multiplan requires a minimum system which contains at least:

- 64K bytes of RAM
- 8080/8085 CPU
- Console with absolute cursor positioning
- One Diskette drive

OPTIONAL HARDWARE:

- Line printer

REQUIRED SOFTWARE:

- CP/M* Operating System

Documentation Package

Multiplan users manual

Shipping Media

(Specify by Alphabetical Character when ordering)

A - Single density IBM 3740/1 compatible 8" diskette
B - Double density IBM 3740/1 compatible 8" diskette
F - IPDS™ compatible 5-1/4" diskette

*CP/M is a registered trademark of Digital Research, Inc.

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

ORDERING INFORMATION

Order Code	Shipping Media	Product Description
SD109CPM80A	A—Single-density 8" diskette	Multiplan spreadsheet program for use under CP/M* on 8080/8085 based small computers.
SD109CPM80B	B—Double-density 8" diskette	
SD109CPM80F	F—iPDS Format 5¼" diskette	

SUPPORT

Intel offers several levels of support for this product, depending on the system configuration in which it is used. Please consult the price list for a detailed description of the support options available.

AFN-00649B

Artisan Technology Group - Quality Instrumentation ... Guaranteed | (888) 88-SOURCE | www.artisantg.com

Artisan Technology Group is an independent supplier of quality pre-owned equipment

Gold-standard solutions

Extend the life of your critical industrial, commercial, and military systems with our superior service and support.

We buy equipment

Planning to upgrade your current equipment? Have surplus equipment taking up shelf space? We'll give it a new home.

Learn more!

Visit us at **artisantg.com** for more info on price quotes, drivers, technical specifications, manuals, and documentation.

Artisan Scientific Corporation dba Artisan Technology Group is not an affiliate, representative, or authorized distributor for any manufacturer listed herein.

We're here to make your life easier. How can we help you today?

(217) 352-9330 | sales@artisantg.com | artisantg.com

ARTISAN® TECHNOLOGY GROUP