WESTERN DIGITAL
CORPORATION

Pascal MICROENGINE™ Product

80-013007-00A2
TECH NOTES

SUMMARY OF OSGO OPERATING SYSTEM

The following Tech Notes detail the fixes, changes and new features which have been incorporated in the GO release.
Before using the OSGO Operating system, the user must be aware of these modifications.

1. The cabling for serial port B (unit # 8) has been changed to allow handling of remote printers with baud rates greater than 4800. Remote cables must be modified for the GO release. The wire list in Appendix A indicates the change that must be made for either remote CRT's or printers.

2. The disk I/O drivers in the operating system have been changed to support double sided floppy disk drives. These new disk drivers now allocate double sided floppy disks in a cylindrical pattern. That is, each cylinder contains two tracks (one per each floppy side) and a disk step occurs after a total cylinder transfer. Users with double sided floppy drives should note that this new capability causes an incompatibility with double sided disks that were written with operating systems at level F1 or lower. The incompatability happens because the GO operating system uses the second side, whereas files written by pre-GO operating systems did not.

   In order to solve this incompatibility, the user should perform a two step process. First, the contents of disks with two-sided capability should be transferred to single sided disks using a pre-GO operating system. Next, the GO operating system should be booted and the single sided disk just made should be transferred to a double sided disk. This double sided disk is now GO compatible and may be used in all GO level systems. Of course, this GO disk cannot be read compatibly with pre-GO operating systems.

3. Segment procedure handling has been changed to handle problems in the original implementation. With these changes, a bug where programs with both UNIT's and SEGMENT PROCEDURE's could fail is now fixed. In addition, EXIT calls out of segment procedures also work. However, due to this change, only GO level software as provided on the GO release diskette will run correctly on the GO level operating system. Use of pre-GO software may cause improper program termination.

   This change in segment procedure handling also affects programs that were linked using the F1 level. It is recommended that programs that need linking be recompiled and relinked on GO. If this is not performed, running a program may cause improper program termination.

3128 REDHILL AVENUE, BOX 2180 NEWPORT BEACH, CA 92663 (714) 557-3550,TWX 910-595-1139

The GO release of the Operating System includes the following new features.

Debugger
See Appendix B

Physical Mode for Disk
See Appendix C

Double Sided Floppy Disk Support
See paragraph 1, item 2.

The following problems have been fixed.

Operating System

1. U(ser Restart now works
2. EXIT statement works for segments with more than 128 procedures.
3. UNITBUSY now returns correct status for units 1,2,8.
4. NEW intrinsic checks for stack overflow.
5. Programs with UNIT's and SEGMENT PROCEDURE's now execute properly.
6. The system now correctly reinitializes after an <etx> is typed at the outer level.
7. Serial port B now supports printers with baud rates greater than 4800.

Pascal Compiler

1. SYSTEM.SWAPDISK is now utilized correctly during compilation. This file holds symbol table information for the compiler during compilation of large programs having include files.

2. Case statements in large p edures now compile correctly.

3. Multiple USES statements are now allowed.

4. The compiler now emits a syntax error when a program allocates more than 32767 words at any nesting level. Without this syntax error, a run-time stack overflow would occur.

5. Parameter checking and passing for variables of type CHAR now works correctly.

6. Real constants without fractional parts now compile correctly.

7. Seven segments or units may be used in a program, up from six on the previous release. Note that the outer block of a program is a segment.

8. ARCTAN is recognized as a synonym for ATAN.

Editor

1. Padded blanks which are added at the end of lines for editor performance optimization will no longer accumulate.

2. The '?' prompt now displays the additional editor commands: move[<arrows>,<sp>,<ret>,=,P(age,direction[<,>],M(rgn,S(et,V(rfy.

3. When deleted text cannot be copied, but the deletion is accepted anyway, the copy buffer is marked invalid. It is also marked invalid when a buffer overflow occurs in insert mode, even if an <escape> is typed to escape the deletion.

4. During Delete in the forward direction, the screen display is correct.

5. If a file is too long for the editor's buffer, a message is displayed advising the user to use the L2 editor. This is done so that the last portions of a file are not lost.

6. The Adjust command with a down arrow will remove accumulated padded blanks at the end of each line.

7. In the Xchange command,the left and right arrow keys now move over text without changing it. This is convenient for multiple exchanges on a line.

8. Spaces inserted after a carriage return are no longer lost when only spaces are inserted on the new line.

Filer

1. Transfers from #8 to any other unit now work.

Other

1. SYSTEM.MISCINFO is changed so that the field 'Has Clock' is set false.

2. The FORMAT program now prompts to ask if a diskette is to be formatted single or double sided.

3. YALOE is fixed to prevent integer overflow for /D or /J commands.

4. CALC now displays its real number output in non-exponential form when possible.

5. PATCH is fixed to use SYSTEM.MISCINFO settings for moving the cursor.

6. L2 no longer fails when it encounters a line greater than 80 characters.

7. Format has an extra prompt to ascertain that both sides of a disk are to be formatted.

8. Setup no longer allows a PREFIXED KEY FOR BREAK.

9. Copydupdir and Markdupdir now recognize lower case input.

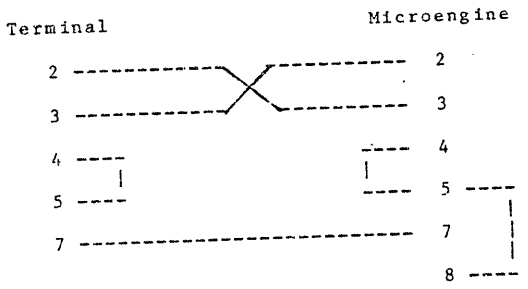
10. Librarian will now work correctly on terminals having a screen width less than 80 and screen height less than 24.

APPENDIX A

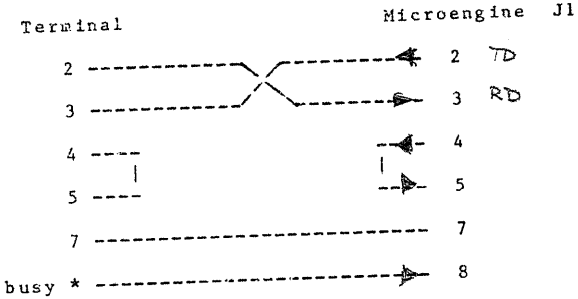
Wire Lists for Serial Port B (Unit #8)

The changes required in the wire list for serial port B depend on whether a CRT or a printer is attached to the port. The diagrams below show the new wire list.

CRT Wire List



Serial Printer Wire List



* Pin 19 is BUSY on NEC Spinwriter

Low (=0V) an pin 5 unterbricht die Ausgabe von pin 2 augenblicklich (d.h. auch im laufenden Byte.

Low (=0V) an pin 8 hält die Ausgabe an (vermutlich per SW), das gerade im Übertragung befindliche Zeichen wird jedoch noch korrekt übertragen.

APPENDIX B

T H E D E B U G G E R

The Debugger is provided as a tool to debug user programs running the operating system or to debug the operating system itself. The Debugger when invoked may insert or delete breakpoints in the work file or break at breakpoints in the work file. Use of the Debugger requires a familiarity with the UCSD operating system and compiler. In order to use the Debugger, it is often necessary to have a compiler generated listing of the program being worked on, and at times a disassembly listing may be needed. This is needed because breakpoints are inserted with reference to segment #, procedure #, and offset within a procedure. This information is listed on a compiler-produced listing.

This Debugger has a functionality that is similar to the Debugger distributed with the UCSD system level I.5, but now no longer distributed. The major difference is that the III.0 system allows interactive placement of breakpoints in a code file, whereas with the I.5 debugger, breakpoints were compiled in.

There are two portions of the Debugger: the Breakpoint Handler and the Debugger. The Debugger is invoked in two ways. The first is when a breakpoint is encountered in an executing work file. The second is when a run-time error occurs in any program. The Breakpoint Handler and both modes of Debugger invocation are described in the following sections.

1. The Breakpoint Handler:

Outer level command prompt:

Command: E(dit, R(un, F(ile, C(omp, L(ink, X(ecute, D(ebug ? (1)

Reply with D for D(ebug to invoke the Breakpoint handler;
If a code file does not currently exist, the system will compile the work file, just like R(un.

Breakpoint handler prompt:

Debug: R(esume, I(nsert, L(ist, C(lear breakpoints, Q(uit ? (2)

Reply:

R(esume: to continue running the user proram.

I(nsert: to insert one or more breakpoints { max number = 10 }
For each breakpoint, the Breakpoint handler will prompt:

Enter segment number: (enter number in decimal); (3)
Enter procedure number: -----
Enter procedure IPC: -----

Validity checking is done for each value.
If the insertion is successful, then info about the breakpoint will be displayed:

Index: i S# <seg> P# <proc> IPC <proc-ipc>(in hex) Op-code <op>(in hex)

then the Breakpoint handler will prompt:

Insert another breakpoint ? (Y or N)

Reply:
Y: to go back to (3);
N: to stop the insertion and go back to (2).

L(ist: to list all breakpoints or to display 'No breakpoints' then return to (2); { see breakpoint info. in I(nsert }

C(lear: to clear breakpoints. The Breakpoint handler will prompt

A(ll, S(ingle ?

Reply:

A(ll: to clear ALL breakpoints;
for each breakpoint displays info { see I(nsert } with 'removed'.
S(ingle: to clear a single breakpoint.
The Debugger will list all breakpoints { see L(ist } then prompt:

Clear breakpoint with index = (enter selected index number, as listed) (4)

if clearing is successful then the Breakpoint handler will prompt:

Continue clearing ? (Y or N)
Reply:
Y: to go back to (4);
N: to go back to (2).

Q(uit: to go back to (1), the outer command level.

Breakpoint information is kept in block zero of the code file.

Block zero layout:

```
zolayout= record
            otherdata: array [0..224] of integer;
            bkcntrl  : packed record
                         bkcnt: 0..maxbrk  {breakpoint count}
                       end;
            bkinfar  : array [0..maxindx]  of {bp info array}
                         packed record
                           relsblk,saveop,opseg,opproc: byte;
                           opsipc: integer;
                           oppipc: integer;
                         end;
          end;
```

where
maxbrk {max. number of breakpoint} = 10;
maxindx {max. index value} = maxbrk-1= 9.

2 . The Debugger: (Called when a breakpoint is executed)

When a breakpoint is executed the DEBUGGER is invoked and the message
Programmed break-point is output along with:

S# <seg.number>, P# <proc.number>, I# <proc-ipc>

Then, the debugger is invoked, Status is displayed (see S(tatus below)
and the debugger prompt is shown.

Prompt:

Debugger: R(esume, D(ump, B(reakpoint, X(amine, S(tatus, Q(uit ? (5)

Reply:

R(esume: to continue running the work file.

D(ump: to dump the whole memory into *SYSTEM.DEBINFO file.
The Debugger will prompt:

Input your Notice: (max size= 80 char.)

This notice (or <space>) will be saved in block 0 of SYSTEM.DEBINFO
along with:
- the contents of registers -3..13;
- the run time error code that caused debugger invocation;
- the segment#, proc# and the ipc of the corresponding opcode;
- and the date (as displayed at boot time).

The record describing this dumped information is:

```
dumplayout = record
               regs: array[0..16] of integer;
               errcode: integer;
               seg: integer;
               proc: integer;
               ipc: integer;
               date: daterec; { 1 word }
               filler: array[0..193] of integer;
               notice: packed array[0..79] of char;
             end;
```

B(reakpoint: to go to the Breakpoint handler
(Very much like in the breakpoint handling step in I.;
except:
- o The code file in memory will also be updated correspondingly for I(nsert and C(lear;
- o Q(uit will return to (5), instead of (1).)

S(tatus: to diplay the environment status:

User program BP = dec. value (hex value) [1,1]
Current MP = dec. value (hex. value) [seg,proc]

Q(uit: do return to (1), the outer command level.

User program MP is the pointer to the MSCW of the user program
at the time the breakpoint occurred

Current MP is the pointer to the current activation record as
performed by the C(hain command (see below).

NOTE: At the first call of the Debugger, user program MP = current MP.

X(amine: to go to the memory eXamine mode.
Prompt:

C(hain, O(ffset, re-D(isplay, A(lter, M(emory, S(tatus, R(adix, Q(uit ?

Reply:

C(hain: to move the current MP pointer along the dynamic or static links.
Prompt:

S(tatic, D(ynamic ? (7)

Reply:

D(ynamic: to follow the dynamic link chain field in the mark stack control word;
S(tatic: to follow the static link chain of the mark stack control word.

If D(ynamic then prompt: (8)

G(lobal, L(ocal ?

Reply:

L(ocal: to move toward more recently called procedures { Limit: procedure [4,1], PRINTERROR }.
G(lobal: to move toward previously called procedures { Limit: procedure [0,1] }.

Number of links: { enter n, a decimal number }

If n in [1.. maxint] then traverse n dynamic or static links.
If D(ynamic then the chaining will be stopped if one limits { see (8) } are reached; else S(tatic chaining will stop if [1,1] is reached.

NOTE: C(hain with S(tatic allows only G(lobal moves and limited to the user program domain.

After every C(hain the following will be displayed:

In [seg,proc]

O(ffset: to display the contents of memory at a word offset from the current MP (see C(hain).

Offset is convenient to access values of variables as all variables are allocated at offsets from a mark stack. The offset corresponds to variables offsets assigned by the compiler.

Prompt: (the Debugger will prompt if input data must be in Hex)
Offset= { enter the offset value }
Length= { enter number of WORDS to be displayed }

then, the requested words will be displayed.

re-D(isplay: to display whatever O(ffset or M(emory was just previously displayed in the "other" radix. This option does not change current radix. Re-display is not possible if the immediately previous command was not O(ffset or M(emory.

A(lter: to modify one word in memory.
The Debugger will display the current Radix and prompt:

Enter address: (in current radix)

then will display:

to Hexadecimal or vice versa. The debugger will prompt:

Radix switched from Decimal to Hex (or Hex to Decimal)

NOTE: This Radix option is always reset to Decimal when the (run time) debugger is first invoked.

Q(uit: to go back to (5).

NOTE:

Due to the mechanism used to return from a breakpoint the active breakpoint in memory will be replaced by the original P-code; and it will be restored only when another breakpoint is encountered. This means a single breakpoint will NOT be restored until another is encountered. However, the breakpoint is still preserved in the code file.

3. The Debugger: (called when a run-time error occurs).

Run-time errors other than Stack overflow will display the prompt:

D(ebug or Type <space> to continue

Reply:
<space>: to follow the usual path of execution for an error;
D(ebug: to go to the Debugger { prompt (5) }
If there is not enough room to load the Debugger, the system will prompt:
Not enough room for Debugger
For debugger invocation via a run time error, the invocation of the Breakpoint Handler will not be allowed, as an X(ecute { from the outer commands } of a program other than the workfile may have been requested.

APPENDIX C

Physical Sector Mode

To provide enhanced flexibility for systems programming, a mechanism is provided for directly accessing physical sectors of a disk. This mode may be enabled during the UNITREAD or UNITWRITE commands. The options for UNITREAD and UNITWRITE are as follows:

```
UNITREAD(unitnumber, array, length, [blocknumber], [flags]);
```

where flags is an integer that may specify physical mode. If bit 1 of flags is reset, logical sector mode, the normal mode on the Microengine is performed. If bit 1 is set, physical sector mode is enabled. This mode has the effect that block number is interpreted as the physical sector number. Conceptually in this mode the disk looks like an array of tracks where each track is an array of sectors. Physical sectors are numbered from 0 starting on track 0 of the diskette, continue ascending from 26 to 51 on track 1, etc. This mode is especially useful for accessing track 0 of a diskette, where the bootstrap resides. For example, the following code sequence reads all of track 0 into an array:

```
var TrackBuf : array[0..3327] of 0..255;

unitread(4,Trackbuf,3328,0{ sector 0 }, 2{ physical mode });
```

APPENDIX D

The following sections were originally a part of the O.F0 Tech note. The documentation is included here as information for users with earlier Operating Systems and for new users.

SETUP

The SETUP utility has been modified to add two fields:

VERTICAL DELAY CHARACTER: The pad character output after a slow terminal operation such as home or clearscreen. (The default vertical delay character is NUL=0.)

KEY TO BACKSPACE: Configures the backspace key for a terminal.

The SETUP program also has three fields, `DISK SEEK RATE`, `DISK READ RATE`, and `DISK WRITE RATE` that tailor disk accesses.

The operating system tailors disk I/O operations by means of these fields. This allows a user to configure the disk transfer delays and stepping rates of any type of floppy disk drive according to values set in SETUP. User tailoring of disk I/O commands is useful due to the wide variance of disk drives. By allowing user configuration of disk I/O commands, full advantage can be taken of each type of disk drive. For example, some floppy disk drives have a fast head stepping rate, so the system stepping rate would be modified using SETUP to specify fast step rates. The SYSTEM.MISCINFO that is shipped has fields that reflect the slowest step rates and disk transfer delays.

Values that can be inserted into `DISK SEEK RATE` are:

Hex	Decimal	Step Rate
1B	27	15 ms. (slowest)
1A	26	10 ms.
19	25	6 ms.
18	24	3 ms. (fastest)

Fast drives can have a value of 24 for this field due to their fast step capability. Slower drives may use a value of 27 or 26 as they have a slow step rate. Note that the SYSTEM.MISCINFO that is shipped has a value of 1F hex, 31 decimal, which is the slowest step rate and also requests the 1791 Controller to verify that the seek is on the destination track. The verify option may be removed to produce a command of hex 1B which is the slowest step rate.

The `DISK READ RATE` and `DISK WRITE RATE` fields specify if there is a delay before head load. The values for `DISK READ RATE` are:

Hex	Decimal	
90	144	no delay
94	148	delay

The values for `DISK WRITE RATE` are:

Hex	Decimal	
B0	176	no delay
B4	180	delay

These three fields correspond to Western Digital 1791 Floppy Disk Controller commands described in section 5.6.4 of the Pascal MICROENGINE Hardware Reference Manual.

UNITREAD AND UNITWRITE

UNITREAD and UNITWRITE now have an option not to convert DLE control characters into spaces, and an option not to append a line feed onto a carriage return.

The options for UNITREAD and UNITWRITE are as follows:
UNITREAD (unit number, array, length, [block number], [flags])
Where flags is an integer specifying the options:

BIT 0-1		Reserved
BIT 2	NOSPEC	SET implies no special character handling for DLE. RESET implies special character handling for DLE, i.e., expand to blanks.
BIT 3	NOCRLF	SET implies no LF's are appended to CR's. RESET implies LF's are appended to CR's.
BIT 4-15		Reserved

Example:

```
Var p: packed array [0..3] of char;
unitwrite (1,p,4,,4) {DLE not stripped and replaced by blanks}
```


SYSTEM.LIBRARY

In addition to the LONGINT UNIT, the SYSTEM.LIBRARY contains a UNIT SCREEN-CONTROL. This unit accesses fields in the record SYSCOM, which is set up by SYSTEM.MISCINFO. It contains several procedures that cause screen control action or return information about the user's terminal. In addition, there is a procedure that returns the date. The procedures the user may access in this UNIT are:

```
PROCEDURE HOME;                          {Homes the cursor}

PROCEDURE CLEAREOS;                      {Clears the screen starting at
                                          the current cursor position}

FUNCTION CLEAREOL;                       {Clears the line starting at the
                                          current cursor position}

FUNCTION SCREENWIDTH:  INTEGER;
        {Returns the width of the screen}

FUNCTION SCREENHEIGHT: INTEGER;
        {Returns the height of the screen}

PROCEDURE DATE(VAR M:MONTHS; VAR D:DAYS; VAR Y:YEARS);
        {Returns the current date as stored by the operating system.
        MONTHS, DAYS, and YEARS are types declared in the INTERFACE
        and are therefore available to the user.  The declarations
        are:

                TYPE MONTHS = 0..12;
                     DAYS   = 0..31;
                     YEARS  = 0..99;}
```

THE SYNTAX FOR UNIT DEFINITION

The following should replace Figure 3-4, Syntax for a Unit Definition, in the Pascal Operations Manual which is the second part of the WD/90 Pascal MICRO-ENGINE Reference Manual.

```
<Compilation unit>      ::= <Program heading>;{<Unit definition>;}
                            <Uses part> <Block> |
                            <Unit definition>{; <Unit definition>}.

<Unit definition>       ::= <Unit heading>;
                            <Interface part>
                            <Implementation part>
                            End

<Unit heading>          ::= Unit <Unit identifier>

<Unit identifier>       ::= Interface
                            <Uses part>
                            <Constant definition part>
                            <Type definition part>
                            <Variable declaration part>
                            <Procedure and function heading part>

<Procedure and function heading part>
                        ::= {<Procedure or function heading>}

<Procedure or function heading>
                        ::= <procedure heading>|<function heading>

<Implementation part>   ::= Implementation
                            <Label declaration part>
                            <Constant definition part>
                            <Type definition part>
                            <Variable declaration part>
                            <Procedure and Function declaration part>

<Uses part>             ::= Uses <Unit identifier>
                            {, <Unit Identifier>}; | <Empty>
```

SUMMARY OF OSHO OPERATING SYSTEM 80-013007-00A3
TECH NOTES

The H0 software release has improvements in software, hardware, and firmware. The most important aspect is that I/O interrupt capability is provided.

New Features

The interrupt capability is manifest to the user as four new features.

The typeahead queue allows typed characters to be stored until there is a programatic request. The typeahead queue is 80 characters in length.

The start/stop feature is the ability to suspend output to the terminal and then resume the output. The start/stop key is specified by the user in the SETUP program. The default setting is control- S.

The flush feature gives the ability to terminate output to the terminal. The flush key is specified in the SETUP program. The default setting is control-F.

The user break feature provides the capability to interrupt a program's execution at any time. The break key is specified in the SETUP program and its default setting is the break key or for terminals without break key it is control-@.

As a part of the H0 release, new microms are incorporated into the Microengine board. These microms correct several past microcode problems. The following changes will be noticed by the user.

1. Integer overflow reporting is inhibited. This was done to correspond to the UCSD standard.

2. A floating point underflow reporting is inhibited. This now corresponds to the UCSD standard for floating point. Now a floating point underflow causes the floating point result to be reported as 0.0.

3. The result of 0.0/0.0 now causes a floating point error.

4. The stack overflow run-time error is now reported correctly.

5. The MOD operator now generates a run-time error message (value range error) for I MOD J where J is less than or equal to zero. This corresponds to the proposed ANSI/IEEE Pascal standard.

6. A new operator, BNOT, has been added to the operator set. Its instruction code is 159. This operator replaces the operator , RBP, which was unused. The BNOT operator is now generated where the LNOT operator was generated previously. The LNOT operator complements all bits in a word, whereas the BNOT operator complements only the low order bit and zeroes the 15 high order bits. This fixes problems such as ORD(NOT FALSE) which formerly returned a negative value and ORD(I > 0) where I is an integer which returned a negative value when I was a negative number.

Note that the BNOT operator is always generated when a NOT is performed. Programs that need a whole word complemented must use the LNOT operator. This can be generated by use of the PMACHINE construct.

Concurrency

The START command is the system intrinsic that creates new tasks in the system. Refer to section 3.7.4 of the Microengine Pascal Operations Manual for a discussion of the START command. It may only be called from a main task, such as the outer block of a user program. If START is called from a sub-task, a run-time error is generated. As a part of the START calling sequence, the semaphore primitives SIGNAL and WAIT are executed. The purpose of this semaphore synchronization for START is to assure that parameters passed by a START call are received by the subtask before later execution may alter them.

A user should note that this type of task switch occurs as a part of task STARTing. Under the H0 operating system, it should be noted that calls to READ and WRITE execute the WAIT semaphore operator so a task switch may occur during I/O. Thus a user should realize that a task switch may occur at other times than he has explicitly programmed using SIGNAL and WAIT.

A correct concurrent program makes no assumptions about the order of operations during concurrent processing. The corollary is: a program must always be prepared for a task switch as interrupts may happen any time. In order to protect indivisible operations, a semaphore lock must be used. Note that a program that does not call START need not be concerned about concurrency and task switching as the operation for a single task will handle all intertask synchronization.

Under the H0 software release, I/O locks and associated critical regions are implemented at the unit level. These semaphore locks are used to assure that each I/O operation is not interrupted until it completes. This means that each UNITREAD, UNITWRITE, UNITCLEAR call is an indivisible operation for a specific unit and that no other task in the system may perform a unit operation on the same unit until the first operation completes.

3128 REDHILL AVENUE, BOX 2180 NEWPORT BEACH, CA 92663 (714) 557-3550,TWX 910-595-1139

Interrupts

Each time there is a hardware interrupt, a software semaphore is signalled. When a hardware interrupt occurs, all interrupts are disabled. Thus a typical I/O driver upon receipt of an interrupt must re-enable interrupts after checking status and capturing the I/O data.

Interrupts may be enabled programatically. In order to enable interrupts, the program must write to the interrupt enable register which is at address FC48 hex. For example, the Pascal procedure below enables interrupts.

```
procedure enableints;
var enabletrix: record
                  case boolean of
                    true:  (addr: integer);
                    false: (loc: ^integer);
                  end;
begin
  enabletrix.addr := -952;  { FC48 hex }
  enabletrix.loc^ := -1;
end;
```

Interrupts may be disabled by a program at the I/O device level. That is each peripheral device, such as the WD 1931 on the serial port or the AM8255 on the parallel port, has a specific bit or bits that disable interrupts for the device. Refer to sections 5.4, 5.5, and 5.6 of the Microengine Hardware Manual for a description of the specific control bits. For example, to disable interrupts on a serial port(bits 1 and 2 of control register 1), the request to send bit and the receiver enable bit must be reset.

When an interrupt driven I/O driver executes, a typical sequence is:

Set up device controller registers to perform I/O
Wait(device interrupt semaphore)
Capture data
Re-enable interrupts

In this sequence the microcode handles the conversion of a hardware interrupt signal to a software signal. For a discussion of semaphores, see section 3.7.4 of the Microengine Pascal Operations Manual. When an interrupt is generated by the hardware, interrupts are disabled for the entire system. The I/O driver sequence above re-enables interrupts as soon as possible after the interrupt signal is received. Due to the necessity of re-enabling interrupts after an I/O interrupt, it must be guaranteed that when a hardware interrupt attached semaphore is signalled, an I/O process is at a high enough priority that it can execute in order to re-enable interrupts.

When an I/O operation is requested by a program, the operating system temporarily raises the priority of the I/O calling task. The I/O interrupt causes an I/O task to run so that it can re-enable interrupts. This I/O priority switch changes the priority of the task requesting the I/O to run at a priority between 240 and 255. In order to keep the I/O tasks at highest priority, no other task in the system may run at a higher priority than this. To safeguard this, the START command will not let a task run at a priority higher than 240. If a task must be started at a higher priority, passing the stack space parameter as a negative number to the start command will override this restriction.

CONCURRENCY AND INTERRUPT INTRINSICS

Below is a description of the concurrency and interrupt intrinsics. See section 3.7.4 of the MICROENGINE Pascal Operations Manual for further details on the Concurrency Primitives and Interrupts.

PROCEDURE ATTACH(SEMAPHORE,INTEGER);

This procedure will attach the semaphore to the interrupt address specified by the integer, allowing a hardware interrupt to signal a semaphore. See section 5.1.3 Device Initiated Communication with the Processor: Interrupts in the MICROENGINE Computer User's Manual for the interrupt addresses.

PROCEDURE SEMINIT(SEMAPHORE,INTEGER);

This procedure initializes the semaphore. The integer value specifies the number of times the semaphore has been signalled. The following example initializes the semaphore SEM to not signalled.

```
SEMINIT(SEM,0);
```

PROCEDURE SIGNAL(SEMAPHORE);

The procedure increments the number of outstanding signals for the semaphore. If any tasks are waiting on the semaphore, the first task on the queue for the signal is decremented. The highest priority task not waiting on the semaphore will then execute.

PROCEDURE START(PROCESS(PARAMS),PROCESSID,INTEGER,INTEGER);

This procedure causes the process to be initiated asynchronously. The processid wil be assigned to point to the TIB initialized. The two integer parameters, STACKSPACE and PRIORITY respectively, specify the amount of stack space that the task will be allocated and the priority at which it will run. PRIORITY is of type 0..255. (See section 3.7.3 Resisters and Operating System Tables for a description of the TIB.)

PROCEDURE WAIT(SEMAPHORE);

This procedure will cause this task to wait until the semaphore has been signalled. If the semaphore has already been signalled, the task will be put on the ready queue and the number of outstanding

signals for the semaphore will be decremented. The highest priority task not waiting on a semaphore will then execute.

EXAMPLES OF CONCURRENCY AND INTERRUPT INTRINSICS

```
program ProcessExample;
  var pid1,
      pid2:processid;
      MessageLock,
      MessageReady,
      ReceivedMessage:semaphore;
      message:string;

process SendMessage(mess:string);
  {locals are allowed}
begin
  wait(MessageLock);
  message:=mess;
  signal(MessageReady);
  wait(RecievedMessage);
  signal(MessageLock);
end;   {SendMessage}

process PrintMessage;
begin
  wait(MessageReady);
  writeln(message);
  signal(ReceivedMessage);
end;  {PrintMessage}

begin
  seminit(MessageLock,1);
  seminit(MessageReady,0);
  seminit(ReceivedMessage,0);

  start(PrintMessage,pid1,85,200);
  start(SendMessage('The message'),pid2,85,200;
end.
```