intel

PRELIMINARY

# 80130/80130-2 iAPX 86/30, 88/30, 186/30, 188/30 iRMX 86 OPERATING SYSTEM PROCESSORS

- **High-Performance 2-Chip Data Processors Containing Operating System Primitives**
- **Standard iAPX 86/10, 88/10 Instruction Set Plus Task Management, Interrupt Management, Message Passing, Synchronization and Memory Allocation Primitives**
- **Fully Extendable To and Compatible With iRMX® 86**
- **Supports Five Operating System Data Types: Jobs, Tasks, Segments, Mailboxes, Regions**
- **35 Operating System Primitives**
- **Built-In Operating System Timers and Interrupt Control Logic Expandable From 8 to 57 Interrupts**
- **8086/80150/80150-2/8088/80186/80188 Compatible At Up To 8 MHz Without Wait States**
- **MULTIBUS® System Compatible Interface**

The Intel iAPX 86/30 and iAPX 88/30 are two-chip microprocessors offering general-purpose CPU (8086) instructions combined with real-time operating system support. They provide a foundation for multiprogramming and multitasking applications. The iAPX 86/30 consists of an iAPX 86/10 (16-bit 8086 CPU) and an Operating System Firmware (OSF) component (80130). The 88/30 consists of the OSF and an iAPX 88/10 (8-bit 8088 CPU). (80186 or 80188 CPUs may be used in place of the 8086 or 8088.)

Both components of the 86/30 and 88/30 are implemented in N-channel, depletion-load, silicon-gate technology (HMOS) and are housed in 40-pin packages. The 86/30 and 88/30 provide all the functions of the iAPX 86/10, 88/10 processors plus 35 operating system primitives, hardware support for eight interrupts, a system timer, a delay timer and a baud rate generator.

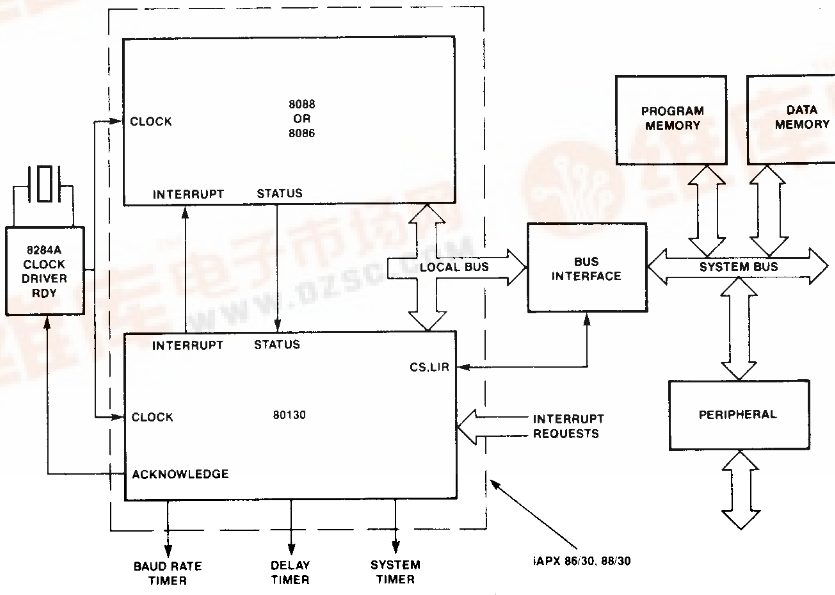


**Figure 1. iAPX 86/30, 88/30 Block Diagram**

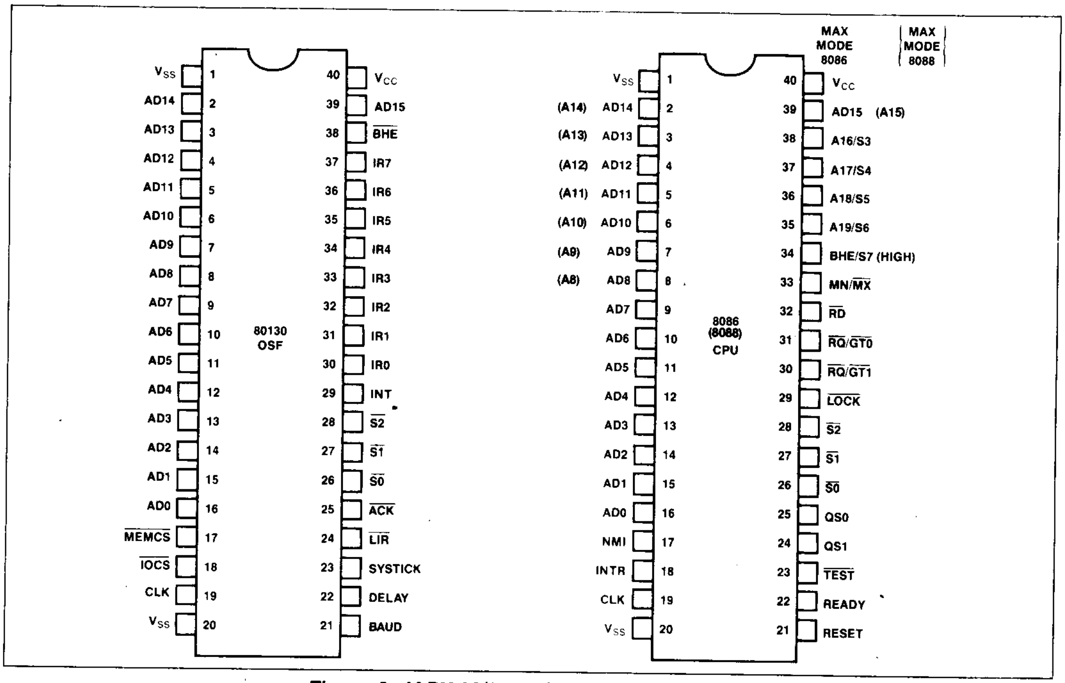


**Figure 2. iAPX 86/30, 88/30 Pin Configuration**

**Table 1. 80130 Pin Description**

| Symbol | Type | Name and Function |
|---|---|---|
| $AD_{15}$–$AD_0$ | I/O | **Address Data:** These pins constitute the time multiplexed memory address ($T_1$) and data ($T_2$, $T_3$, $T_W$, $T_4$) bus. These lines are active HIGH. The address presented during $T_1$ of a bus cycle will be latched internally and interpreted as an 80130 internal address if MEMCS or IOCS is active for the invoked primitives. The 80130 pins float whenever it is not chip selected, and drive these pins only during $T_2$–$T_4$ of a read cycle and $T_1$ of an INTA cycle. |
| $\overline{BHE}/S_7$ | | **Bus High Enable:** The 80130 uses the $\overline{BHE}$ signal from the processor to determine whether to respond with data on the upper or lower data pins, or both. The signal is active LOW. $\overline{BHE}$ is latched by the 80130 on the trailing edge of ALE. It controls the 80130 output data as shown.<br>$\overline{BHE}$ $A_0$<br>0 0 Word on $AD_{15}$–$AD_0$<br>0 1 Upper byte on $AD_{15}$–$AD_8$<br>1 0 Lower byte on $AD_7$–$AD_0$<br>1 1 Upper byte on $AD_7$–$AD_0$ |
| $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$ | I | **Status:** For the 80130, the status pins are used as inputs only. 80130 encoding follows:<br>$\overline{S_2}$ $\overline{S_1}$ $\overline{S_0}$<br>0 0 0 INTA<br>0 0 1 IORD<br>0 1 0 IOWR<br>0 1 1 Passive<br>1 0 0 Instruction fetch<br>1 0 1 MEMRD<br>1 1 X Passive |

Table 1. 80130 Pin Description (Continued)

| Symbol | Type | Name and Function |
|---|---|---|
| CLK | I | **Clock:** The system clock provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing. The 80130 uses the system clock as an input to the SYSTICK and BAUD timers and to synchronize operation with the host CPU. |
| INT | O | **Interrupt:** INT is HIGH whenever a valid interrupt request is asserted. It is normally used to interrupt the CPU by connecting it to INTR. |
| $IR_7$–$IR_0$ | I | **Interrupt Requests:** An interrupt request can be generated by raising an IR input (LOW to HIGH) and holding it HIGH until it is acknowledged (Edge-Triggered Mode), or just by a HIGH level on an IR input (Level-Triggered Mode). |
| $\overline{ACK}$ | O | **Acknowledge:** This line is LOW whenever an 80130 resource is being accessed. It is also LOW during the first INTA cycle and second INTA cycle if the 80130 is supplying the interrupt vector information. This signal can be used as a bus ready acknowledgement and/or bus transceiver control. |
| $\overline{MEMCS}$ | I | **Memory Chip Select:** This input must be driven LOW when a kernel primitive is being fetched by the CPU. $AD_{13}$–$AD_0$ are used to select the instruction. |
| $\overline{IOCS}$ | I | **Input/Output Chip Select:** When this input is low, during an IORD or IOWR cycle, the 80130's kernel primitives are accessing the appropriate peripheral function as specified by the following table: |

| $\overline{BHE}$ | $A_3$ | $A_2$ | $A_1$ | $A_0$ | |
|---|---|---|---|---|---|
| 0 | X | X | X | X | Passive |
| X | X | X | X | 1 | Passive |
| X | 0 | 1 | X | X | Passive |
| 1 | 0 | 0 | X | 0 | Interrupt Controller |
| 1 | 1 | 0 | 0 | 0 | Systick Timer |
| 1 | 1 | 0 | 1 | 0 | Delay Counter |
| 1 | 1 | 1 | 0 | 0 | Baud Rate Timer |
| 1 | 1 | 1 | 1 | 0 | Timer Control |

| Symbol | Type | Name and Function |
|---|---|---|
| $\overline{LIR}$ | O | **Local Bus Interrupt Request:** This signal is LOW when the interrupt request is for a non-slave input or slave input programmed as being a local slave. |
| $V_{CC}$ | | **Power:** $V_{CC}$ is the +5V supply pin. |
| $V_{SS}$ | | **Ground:** $V_{SS}$ is the ground pin. |
| SYSTICK | O | **System Clock Tick:** Timer 0 Output. Operating System Clock Reference. SYSTICK is normally wired to IR2 to implement operating system timing interrupt. |
| DELAY | O | **DELAY Timer:** Output of timer 1. Reserved by Intel Corporation for future use. |
| BAUD | O | **Baud Rate Generator:** 8254 Mode 3 compatible output. Output of 80130 Timer 2. |

## FUNCTIONAL DESCRIPTION

The increased performance and memory space of iAPX 86/10 and 88/10 microprocessors have proven sufficient to handle most of today's single-task or single-device control applications with performance to spare, and have led to the increased use of these microprocessors to control *multiple* tasks or devices in real-time. This trend has created a new challenge to designers—development of real-time, multitasking application systems and software. Examples of such systems include control systems that monitor and react to external events in real-time, multifunction desktop and personal computers, PABX equipment which constantly controls the telephone traffic in a multiphone office, file servers/disk subsystems controlling and coordinating multiple disks and multiple disk users, and transaction processing systems such as electronics funds transfer.

### The iAPX 86/30, 88/30 Operating System Processors

The Intel iAPX 86/30, 88/30 Operating System Processors (OSPs) were developed to help solve this

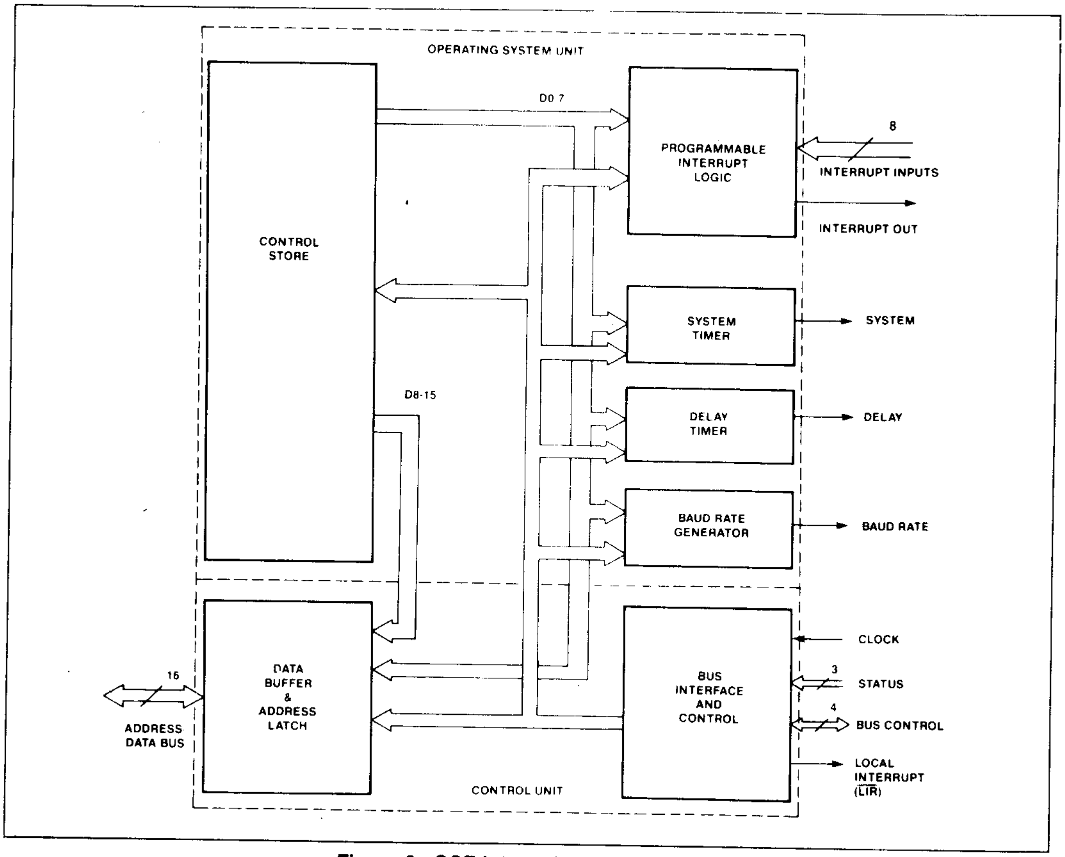


Figure 3. OSF Internal Block Diagram

problem. Their goal is to simplify the design of multitasking application systems by providing a well-defined, fully debugged set of operating system primitives implemented directly in the hardware, thereby removing the burden of designing multitasking operating system primitives from the application programmer.

Both the 86/30 and the 88/30 OSPs are two-chip sets consisting of a main processor, an 8086 or 8088 CPU, and the Intel 80130, Operating System Firmware component (OSF) (see Figure 1). The 80130 provides a set of multitasking kernel primitives, kernel control storage, and the additional support hardware, including system timers and interrupt control, required by these primitives. From the application programmer's viewpoint, the OSF extends the base iAPX 86, 88 architecture by providing 35 operating system primitive instructions, and supporting five new system data types, making the OSF a logical and easy-to-use architectural extension to iAPX 86, 88 system designs.

## The OSP Approach

The OSP system data types (SDTs) and primitive instructions allocate, manage and share low-level processor resources in an efficient manner. For example, the OSP implements task context management (managing a task state image consisting of both hardware register set and software control information) for either the basic 86/10 context or the extended 86/20 (8086+8087) numerics context. The OSP manages the entire task state image both while the task is actively executing and while it is inactive. Tasks can be created, put to sleep for specified periods, suspended, executed to perform their functions, and dynamically deleted when their functions are complete.

The Operating System Processors support event-oriented systems designs. Each event may be processed by an individual responding task or along with other closely related events in a common task. External events and interrupts are processed by the OSP interrupt handler primitives using its built-in interrupt controller subsystem as they occur in real-time. The multiple tasks and the multiple events are coordinated by the OSP integral scheduler whose preemptive, priority-based scheduling algorithm and system timers organize and monitor the processing of every task to guarantee that events are processed as they occur in order of relative importance. The 86/30 also provides primitives for intertask communication (by mailboxes) and for mutual exclusion (by regions), essential functions for multitasking applications.

## Programming Language Support

Programs for the OSP can be written in ASM 86/88 or PL/M 86/88, Intel's standard system languages for iAPX 86,88 systems.

The Operating System Processor Support Package (iOSP 86) provides an interface library for application programs written in any model of PL/M-86. This library also provides 80130 configuration and initialization support as well as complete user documentation.

## OSF PROGRAMMING INTERFACE

The OSF provides 35 operating system kernel primitives which implement multitasking, interrupt management, free memory management, intertask communication and synchronization. Table 4 shows each primitive, and Table 5 gives the execution performance of typical primitives.

OSP primitives are executed by a combination of CPU and OSF (80130) activity. When an OSP primitive is called by an application program task, the iAPX CPU registers and stacks are used to perform the appropriate functions and relay the results to the application programs.

## OSP Primitive Calling Sequences

A standard, stack-based, calling sequence is used to invoke the OSF primitives. Before a primitive is called, its operand parameters must be pushed on the task stack. The SI register is loaded with the offset of the last parameter on the stack. The entry code for the primitive is loaded into AX. The primitive invocation call is made with a CPU software interrupt (Table 4). A representative ASM86 sequence for calling a primitive is shown in Figure 4. In PL/M the OSP programmer uses a call to invoke the primitive.

```
                 SAMPLE ASSEMBLY LANGUAGE PRIMITIVE CALL

PUSH P1                              ;PUSH PARAMETER 1
PUSH P2                              ;PUSH PARAMETER 2
  .                                  ;  .  .  .
  .                                  ;  .  .  .
  .                                  ;  .  .  .
PUSH PN                              ;PUSH PARAMETER N
PUSH BP                              ;STACK CALLING CONVENTION
MOV BP,SP
LEA SI,SS:NUM__BYTES__PARAM   2[BP]
                                     ;SS:SI POINTS TO FIRST
                                     ;PARAMETER ON STACK
MOV AX, ENTRY CODE                   ;AX SETS PRIMITIVE ENTRY CODE
INT 184                              ;OSF INTERRUPT

                        OSP PRIMITIVE INVOKED

POP BP
RET NUM__BYTES__PARAM__              ;POP PARAMETERS
                                     ;CX CONTAINS EXCEPTION CODES
                                     ;DL CONTAINS PARAMETER NUMBER
                                     ;   THAT CAUSED EXCEPTION (IF
                                     ;   CX IS NON ZERO)
                                     ;AX CONTAINS WORD RETURN VALUE
                                     ;ES:BX CONTAINS POINTER
                                     ;   RETURN VALUE
```

**Figure 4. ASM/86 OSP Calling Convention**

## OSP Functional Description

Each major function of the OSP is described below. These are:

Job and Task Management
Interrupt Management
Free Memory Management
Intertask Communication
Intertask Synchronization
Environmental Control

The system data types (or SDTs) supported by the OSP are capitalized in the description. A short description of each SDT appears in Table 2.

## JOB and TASK Management

Each OSP JOB is a controlled environment in which the applications program executes and the OSF system data types reside. Each individual application program is normally a separate OSP JOB, whether it has one initial task (the minimum) or multiple tasks. JOBs partition the system memory into pools. Each memory pool provides the storage areas in which the OSP will allocate TASK state images and other system data types created by the executing TASKs, and free memory for TASK working space. The OSP supports multiple executing TASKs within a JOB by managing the resources used by each, including the CPU registers, NPX registers, stacks, the system data types, and the available free memory space pool.

When a TASK is created, the OSP allocates memory (from the free memory of its JOB environment) for the TASK's stack and data area and initializes the additional TASK attributes such as the TASK priority level and its error handler location. (As an option, the caller of CREATE TASK may assign previously defined stack and data areas to the TASK.) Task priorities are integers between 0 and 255 (the lower the priority number the higher the scheduling priority of the TASK). Generally, priorities up to 128 will be assigned to TASKs which are to process interrupts. Priorities above 128 do not cause interrupts to be disabled, these priorities (129 to 255) are appropriate for non-interrupt TASKs. If an 8087 Numerics Processor Extension is used, the error recovery interrupt level assigned to it will have a higher priority than a TASK executing on it, so that error handling is performed correctly.

## EXECUTION STATUS

A TASK has an execution status or execution state. The OSP provides five execution states: RUNNING, READY, ASLEEP, SUSPENDED, and ASLEEP-SUSPENDED.

— A TASK is RUNNING if it has control of the processor.

— A TASK is READY if it is not asleep, suspended, or asleep-suspended. For a TASK to become the running (executing) TASK, it must be the highest priority TASK in the ready state.

— A TASK is ASLEEP if it is waiting for a request to be granted or a timer event to occur. A TASK may put itself into the ASLEEP state.

— A TASK is SUSPENDED if it is placed there by another TASK or if it suspends itself. A TASK may have multiple suspensions, the count of suspensions is managed by the OSP as the TASK suspension depth.

— A TASK is ASLEEP-SUSPENDED if it is both waiting and suspended.

TASK attributes, the CPU register values, and the 8087 register values (if the 8087 is configured into the application) are maintained by the OSP in the TASK state image. Each TASK will have a unique TASK state image.

## SCHEDULING

The OSP schedules the processor time among the various TASKs on the basis of priority. A TASK has an execution priority relative to all other TASKs in the system, which the OSP maintains for each TASK in its TASK state image. When a TASK of higher priority than the executing TASK becomes ready to execute, the OSP switches the control of the processor to the higher priority TASK. First, the OSP saves the outgoing (lower priority) TASK's state including CPU register values in its TASK state image. Then, it restores the CPU registers from the TASK state image of the incoming (higher priority) TASK. Finally, it causes the CPU to start or resume executing the higher priority TASK.

TASK scheduling is performed by the OSP. The OSP's priority-oriented preemptive scheduler determines which TASK executes by comparing their relative priorities. The scheduler insures that the highest priority TASK with a status of READY will execute. A TASK will continue to execute until an interrupt with a higher priority occurs, or until it requests unavailable resources, for which it is willing to wait, or until it makes specific resources available to a higher priority TASK waiting for those resources.

TASKs can become READY by receiving a message, receiving control, receiving an interrupt, or by timing out. The OSP always monitors the status of all the TASKs (and interrupts) in the system. Preemptive scheduling allows the system to be responsive to the external environment while only devoting CPU resources to TASKs with work to be performed.

## TIMED WAIT

The OSP timer hardware facilities support timed waits and timeouts. Thus, in many primitives, a TASK can specify the length of time it is prepared to wait for an event to occur, for the desired resources to become available or for a message to be received at a MAILBOX. The timing interval (or System Tick) can be adjusted, with a lower limit of 1 millisecond.

## APPLICATION CONTROL OF TASK EXECUTION

Programs may alter TASK execution status and priority dynamically. One TASK may suspend its own execution or the execution of another TASK for a period of time, then resume its execution later. Multiple suspensions are provided. A suspended TASK may be suspended again.

The eight OSP Job and TASK management primitives are:

| | |
|---|---|
| CREATE JOB | Partitions system resources and creates a TASK execution environment. |
| CREATE TASK | Creates a TASK state image. Specifies the location of the TASK code instruction stream, its execution priority, and the other TASK attributes. |

| | |
|---|---|
| DELETE TASK | Deletes the TASK state image, removes the instruction stream from execution and deallocates stack resources. Does not delete INTERRUPT TASKS. |
| SUSPEND TASK | Suspends the specified TASK or, if already suspended, increments its suspension depth by one. Execute state is SUSPEND. |
| RESUME TASK | Decrements the TASK suspension depth by one. If the suspension depth is then zero, the primitive changes the task execution status to READY, or ASLEEP (if ASLEEP/SUSPENDED). |
| SLEEP | Places the requesting TASK in the ASLEEP state for a specified number of System Ticks. (The TICK interval can be configured down to 1 millisecond.) |
| SET PRIORITY | Alters the priority of a TASK. |

## Interrupt Management

The OSP supports up to 256 interrupt levels organized in an interrupt vector, and up to 57 external interrupt sources of which one is the NMI (Non-Maskable Interrupt). The OSP manages each interrupt level independently. The OSF INTERRUPT SUBSYSTEM provides two mechanisms for interrupt management: INTERRUPT HANDLERs and INTERRUPT TASKs. INTERRUPT HANDLERs disable all maskable interrupts and should be used only for servicing interrupts that require little processing time. Within an INTERRUPT HANDLER only certain OSF Interrupt Management primitives (DISABLE, ENTER INTERRUPT, EXIT INTERRUPT, GET LEVEL, SIGNAL INTERRUPT) and basic CPU instructions can be used, other OSP primitives cannot be. The INTERRUPT TASK approach permits all OSP primitives to be issued and masks only lower priority interrupts.

Work flow between an INTERRUPT HANDLER and an INTERRUPT TASK assigned to the same level is regulated with the SIGNAL INTERRUPT and WAIT INTERRUPT primitives. The flow is asynchronous. When an INTERRUPT HANDLER signals an INTERRUPT TASK, the INTERRUPT HANDLER becomes immediately available to process another interrupt. The number of interrupts (specified for the level) the INTERRUPT HANDLER can queue for the INTERRUPT TASK can be limited to the value specified in the SET INTERRUPT primitive. When the INTERRUPT TASK is finished processing, it issues a WAIT INTERRUPT primitive, and is immediately ready to process the queue of interrupts that the INTERRUPT HANDLER has built with repeated SIGNAL INTERRUPT primitives while the INTERRUPT TASK was processing. If there were no interrupts at the level, the queue is empty and the INTERRUPT TASK is SUSPENDED. See the Example (Figure 5) and Figures 6 and 7.

OSP external INTERRUPT LEVELs are directly related to internal TASK scheduling priorities. The OSP maintains a single list of priorities including both tasks and INTERRUPT LEVELs. The priority of the executing TASK automatically determines which interrupts are masked. Interrupts are managed by INTERRUPT LEVEL number. The OSP supports eight levels directly and may be extended by means of slave 8259As to a total of 57.

The nine Interrupt Management OSP primitives are:

| | |
|---|---|
| DISABLE | Disables an external INTERRUPT LEVEL. |
| ENABLE | Enables an external INTERRUPT LEVEL. |
| ENTER INTERRUPT | Gives an Interrupt Handler its own data segment, separate from the data segment of the interrupted task. |
| EXIT INTERRUPT | Performs an "END of INTERRUPT" operation. Used by an INTERRUPT HANDLER which does not invoke an INTERRUPT TASK. Reenables interrupts, when the INTERRUPT HANDLER gives up control. |
| GET LEVEL | Returns the interrupt level number of the executing INTERRUPT HANDLER. |
| RESET INTERRUPT | Cancels the previous assignment made to an interrupt level by SET INTERRUPT primitive request. If an INTERRUPT TASK has been assigned, it is also deleted. The interrupt level is disabled. |
| SET INTERRUPT | Assigns an INTERRUPT HANDLER to an interrupt level and, optionally, an INTERRUPT TASK. |

```
/*  CODE EXAMPLE A   INTERRUPT TASK TO KEEP TRACK OF TIME-OF-DAY
DECLARE SECOND$COUNT BYTE,
     MINUTE$COUNT BYTE,
     HOURS$COUNT BYTE;
TIME$TASK:  PROCEDURE;
     DECLARE TIME$EXCEPT$CODE WORD;
     AC$CYCLE$COUNT=0;
     CALL RQ$SET$INTERRUPT(AC$INTERRUPT$LEVEL, 01H),
          @AC$HANDLER,0,@TIME$EXCEPT$CODE);
     CALL RQ$RESUME$TASK(INIT$TASK$TOKEN,@TIME$EXCEPT$CODE);
     DO HOUR$COUNT=0 TO 23;
          DO MINUTE$COUNT=0 TO 59;
               DO SECOND$COUNT=0 TO 59;
                    CALL RQ$WAIT$INTERRUPT(AC$INTERRUPT$LEVEL,
                         @TIME$EXCEPT$CODE);
                    IF SECOND$COUNT MOD 5=0
                         THEN CALL PROTECTED$CRT$OUT(BEL);
                    END;    /*  SECOND LOOP  */
               END;    /*  MINUTE LOOP  */
          END;    /*  HOUR LOOP  */
     CALL RQ$RESET$INTERRUPT(AC$INTERRUPT$LEVEL, @TIME$EXCEPT$CODE);
     END TIME$TASK;
/*  CODE EXAMPLE B    INTERRUPT HANDLER TO SUBDIVIDE A.C. SIGNAL BY 60.  */
DECLARE AC$CYCLE$COUNT BYTE;
AC$HANDLER:  PROCEDURE INTERRUPT 59;
     DECLARE AC$EXCEPT$CODE WORD;
     AC$CYCLE$COUNT=AC$CYCLE$COUNT +1;
     IF AC$CYCLE$COUNT>=60 THEN DO;
          AC$CYCLE$COUNT=0;
          CALL RQ$SIGNAL$INTERRUPT(AC$INTERRUPT$LEVEL,@AC$EXCEPT$CODE);
          END;
     END  AC$HANDLER;
```

Figure 5. OSP Examples

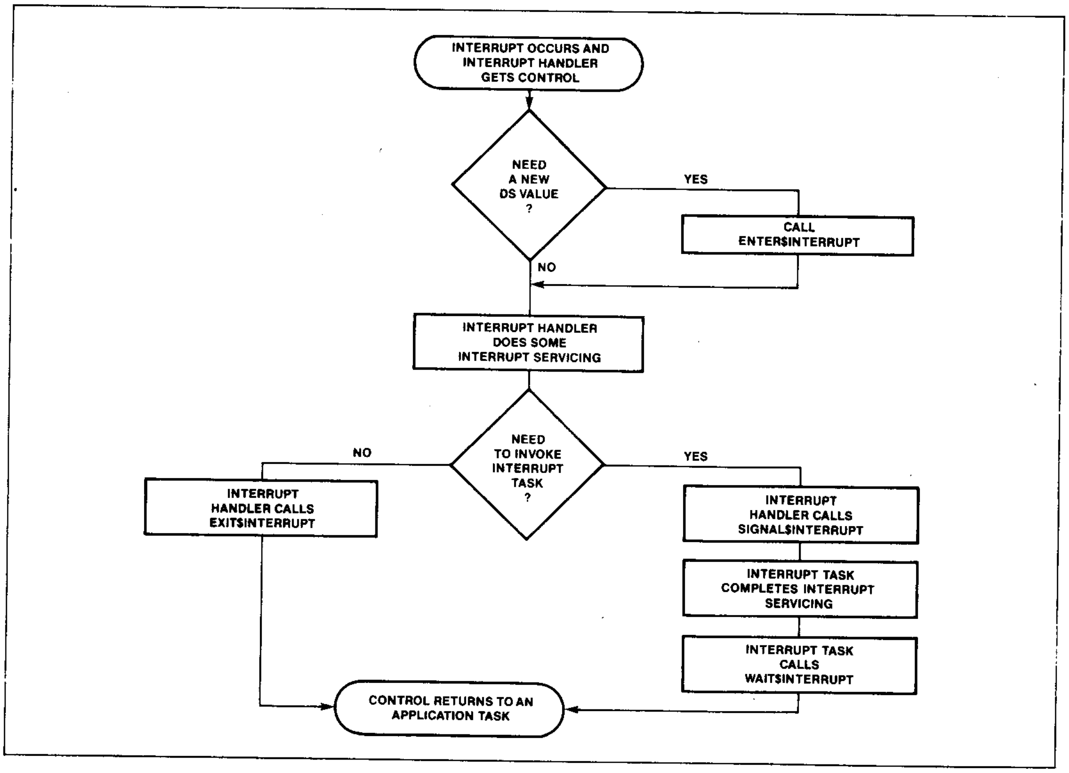

INTERRUPT OCCURS AND INTERRUPT HANDLER GETS CONTROL
NEED A NEW DS VALUE ?
YES
CALL ENTER$INTERRUPT
NO
INTERRUPT HANDLER DOES SOME INTERRUPT SERVICING
NEED TO INVOKE INTERRUPT TASK ?
NO
YES
INTERRUPT HANDLER CALLS EXIT$INTERRUPT
INTERRUPT HANDLER CALLS SIGNAL$INTERRUPT
INTERRUPT TASK COMPLETES INTERRUPT SERVICING
INTERRUPT TASK CALLS WAIT$INTERRUPT
CONTROL RETURNS TO AN APPLICATION TASK

Figure 6. Interrupt Handling Flowchart

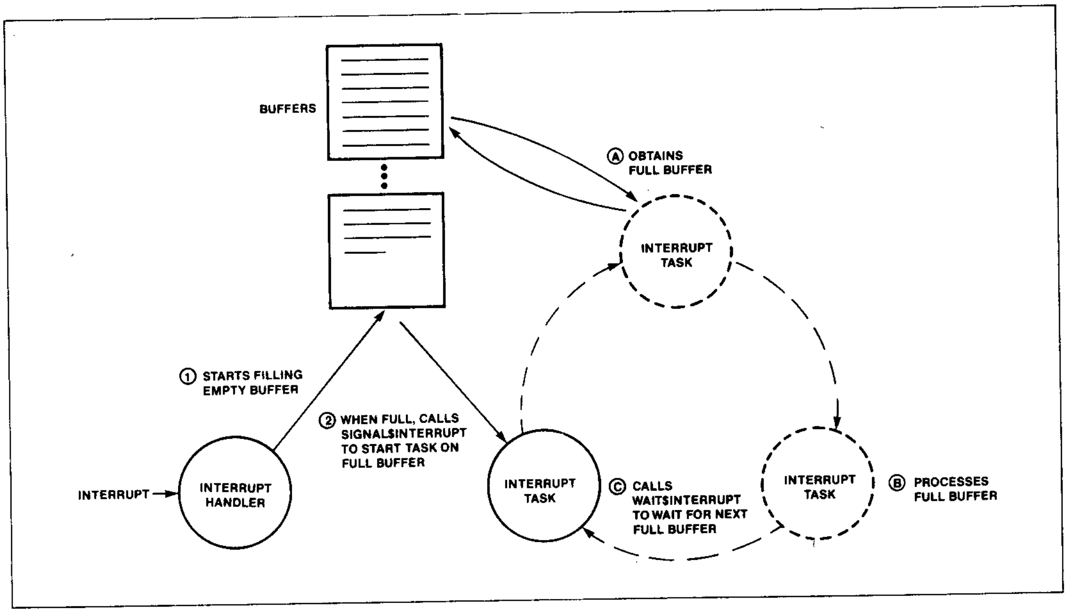

BUFFERS
Ⓐ OBTAINS FULL BUFFER
INTERRUPT TASK
① STARTS FILLING EMPTY BUFFER
② WHEN FULL, CALLS SIGNAL$INTERRUPT TO START TASK ON FULL BUFFER
INTERRUPT
INTERRUPT HANDLER
INTERRUPT TASK
Ⓒ CALLS WAIT$INTERRUPT TO WAIT FOR NEXT FULL BUFFER
INTERRUPT TASK
Ⓑ PROCESSES FULL BUFFER

**Figure 7. Multiple Buffer Example**

| | |
|---|---|
| SIGNAL INTERRUPT | Used by an INTERRUPT HANDLER to activate an Interrupt Task. |
| WAIT INTERRUPT | Suspends the calling Interrupt Task until the INTERRUPT HANDLER performs a SIGNAL INTERRUPT to invoke it. If a SIGNAL INTERRUPT for the task has occurred, it is processed. |

## FREE MEMORY MANAGEMENT

The OSP Free Memory Manager manages the memory pool which is allocated to each JOB for its execution needs. (The CREATE JOB primitive allocates the new JOB's memory pool from the memory pool of the parent JOB.) The memory pool is part of the JOB resources but is not yet allocated between the tasks of the JOB. When a TASK, MAILBOX, or REGION system data type structure is created within that JOB, the OSP implicitly allocates memory for it from the JOB's memory pool, so that a separate call to allocate memory is not required. OSP primitives that use free memory management implicitly include CREATE JOB, CREATE TASK, DELETE TASK, CREATE MAILBOX, DELETE MAILBOX, CREATE REGION, and DELETE REGION. The CREATE SEGMENT primitive explicitly allocates a memory area when one is needed by the TASK. For example, a TASK may explicitly allocate a SEGMENT for use as a memory buffer. The SEGMENT length can be any multiple of 16 bytes between 16 bytes and 64K bytes in length. The programmer may specify any number of bytes from 1 byte to 64 KB, the OSP will transparently round the value up to the appropriate segment size.

The two explicit memory allocation/deallocation primitives are:

| | |
|---|---|
| CREATE SEGMENT | Allocates a SEGMENT of specified length (in 16-byte-long paragraphs) from the JOB Memory Pool. |
| DELETE SEGMENT | Deallocates the SEGMENT's memory area, and returns it to the JOB memory pool. |

### Intertask Communication

The OSP has built-in intertask synchronization and communication, permitting TASKs to pass and share information with each other. OSP MAILBOXes contain controlled handshaking facilities which guarantee that a *complete* message will always be sent from a sending TASK to a receiving TASK. Each MAILBOX consists of two interlocked queues, one of TASKs and the other of Messages. Four OSP primitives for intertask synchronization and communication are provided:

| | |
|---|---|
| CREATE MAILBOX | Creates intertask message exchange. |
| DELETE MAILBOX | Deletes an intertask message exchange. |
| RECEIVE MESSAGE | Calling TASK receives a message from the MAILBOX. |
| SEND MESSAGE | Calling TASK sends a message to the MAILBOX. |

The CREATE MAILBOX primitive allocates a MAILBOX for use as an information exchange between TASKs. The OSP will post information at the MAILBOX in a FIFO (First-In First-Out) manner when a SEND MESSAGE instruction is issued. Similarily, a message is retrieved by the OSP if a TASK issues a RECEIVE MESSAGE primitive. The TASK which creates the MAILBOX may make it available to other TASKs to use.

If no message is available, the TASK attempting to receive a message may choose to wait for one or continue executing.

The queue management method for the task queue (FIFO or PRIORITY) determines which TASK in the MAILBOX TASK queue will receive a message from the MAILBOX. The method is specified in the CREATE MAILBOX primitive.

## Intertask Synchronization and Mutual Exclusion

Mutual exclusion is essential to multiprogramming and multiprocessing systems. The REGION system data type implements mutual exclusion. A REGION is represented by a queue of TASKS waiting to use a resource which must be accessed by only one TASK at a time. The OSP provides primitives to use REGIONs to manage mutually exclusive data and resources. Both critical code sections and shared data structures can be protected by these primitives from simultaneous use by more than one task. REGIONs support both FIFO (First-In First-Out) or Priority queueing disciplines for the TASKS seeking to enter the REGION. The REGION SDT can also be used to implement software locks.

Multiple REGIONs are allowed, and are automatically exited in the reverse order of entry. While in a REGION, a TASK cannot be suspended by itself or any other TASK, and thereby avoids deadlock.

There are five OSP primitives for mutual exclusion:

| | |
|---|---|
| CREATE REGION | Create a REGION (lock). |
| SEND CONTROL | Give up the REGION. |
| ACCEPT CONTROL | Request the REGION, but do not wait if it is not available. |
| RECEIVE CONTROL | Request a REGION, wait if not immediately available. |
| DELETE REGION | Delete a REGION. |

The OSP also provides dynamic priority adjustment for TASKs within priority REGIONs: If a higher-priority TASK issues a RECEIVE CONTROL primitive, while a (lower-priority) TASK has the use of the same REGION, the lower-priority TASK will be transparently, and temporarily, elevated to the waiting TASK's priority until it relinquishes the REGION via SEND CONTROL. At that point, since it is no longer using the critical resource, the TASK will have its normal priority restored.

## OSP Control Facilities

The OSP also includes system primitives that provide both control and customization capabilities to a multitasking system. These primitives are used to control the deletion of SDTs and the recovery of free memory in a system, to allow interrogation of operating system status, and to provide uniform means of adding user SDTs and type managers.

### DELETION CONTROL

Deletion of each OSP system data type is explicitly controlled by the applications programmer by setting a deletion attribute for that structure. For example, if a SEGMENT is to be kept in memory until DMA activity is completed, its deletion attribute should be disabled. Each TASK, MAILBOX, REGION, and SEGMENT SDT is created with its deletion attribute enabled (i.e., they may be deleted). Two OSP primitives control the deletion attribute: ENABLE DELETION and DISABLE DELETION.

### ENVIRONMENTAL CONTROL

The OSP provides inquiry and control operations which help the user interrogate the application environment and implement flexible exception handling. These features aid in run-time decision making and in application error processing and recovery. There are five OSP environmental control primitives.

### OS EXTENSIONS

The OSP architecture is defined to allow new user-defined System Data Types and the primitives to manipulate them to be added to OSP capabilities provided by the built-in System Data Types. The type managers created for the user-defined SDTs are called user OS extensions and are installed in the system by the SET OS EXTENSION primitive. Once installed, the functions of the type manager may be invoked with user primitives conforming to the OSP interface. For well-structured extended architectures, each OS extension should support a separate user-defined system data type, and every OS extension should provide the same calling sequence and program interface for the user as is provided for a built-in SDT. The type manager for the extension would be written to suit the needs of the application. OSP interrupt vector entries (224–255) are reserved for user OS extensions and are not used by the OSP. After assigning an interrupt number to the extension, the extension user may then call it with the standard OSP call sequence (Figure 4), and the unique software interrupt number assigned to the extension.

| | |
|---|---|
| ENABLE DELETION | Allows a specific SEGMENT, TASK, MAILBOX, or REGION SDT to be deleted. |
| DISABLE DELETION | Prevents a specific SEGMENT, TASK, MAILBOX, or REGION SDT from being deleted. |
| GET TYPE | Given a token for an instance of a system data type, returns the type code. |
| GET TASK TOKENS | Returns to the caller information about the current task environment. |
| GET EXCEPTION HANDLER | Returns information about the calling TASK's current information handler: its address, and when it is used. |
| SET EXCEPTION HANDLER | Provides the address and usage of an exception handler for a TASK. |
| SET OS EXTENSION | Modifies one of the interrupt vector entries reserved for OS extensions (224–255) to point to a user OS extension procedure. |
| SIGNAL EXCEPTION | For use in OS extension error processing. |

## EXCEPTION HANDLING

The OSP supports exception handlers. These are similar to CPU exception handlers such as OVERFLOW and ILLEGAL OPERATION. Their purpose is to allow the OSP primitives to report parameter errors in primitive calls, and errors in primitive usage. Exception handling procedures are flexible and can be individually programmed by the application. In general, an exception handler if called will perform one or more of the following functions:

—Log the Error.
—Delete/Suspend the Task that caused the exception.
—Ignore the error, presumably because it is not serious.

An EXCEPTION HANDLER is written as a procedure. If PLM/86 is used, the "compact," "medium" or "large" model of computation should be specified for the compilation of the program. The mode in which the EXCEPTION HANDLER operates may be specified in the SET EXCEPTION HANDLER primitive. The return information from a primitive call is shown in Figure 4. CX is used to return standard system error conditions. Table 7 shows a list of these conditions, using the *default* EXCEPTION HANDLER of the OSP.

## HARDWARE DESCRIPTION

The 80130 operates in a closely coupled mode with the iAPX 86/10 or 88/10 CPU. The 80130 resides on the CPU local multiplexed bus (Figure 8). The main processor is always configured for maximum mode operation. The 80130 automatically selects between its 88/30 and 86/30 operating modes.

The 80130 used in the 86/30 configuration, as shown in Figure 8 (or a similar 88/30 configuration), operates at both 5 and 8 MHz without requiring processor wait states. Wait state memories are fully supported, however. The 80130 may be configured with both an 8087 NPX and an 8089 IOP, and provides full context control over the 8087.

The 80130 (shown in Figure 3) is internally divided into a control unit (CU) and operating system unit (OSU). The OSU contains facilities for OSP kernel support including the system timers for scheduling and timing waits, and the interrupt controller for interrupt management support.

### iAPX 86/30, iAPX 88/30 System Configuration

The 80130 is both I/O and memory mapped to the local CPU bus. The CPU's status S0/-S2/ is decoded along with IOCS/ (with BHE and $AD_3$-$AD_0$) or MEMCS/ (with $AD_{13}$-$AD_0$). The pins are internally latched. See Table 1 for the decoding of these lines.

### Memory Mapping

Address lines $A_{19}$-$A_{14}$ can be used to form MEMCS/ since the 80130's memory-mapped portion is aligned along a 16K-byte boundry. The 80130 can reside on any 16K-byte boundry excluding the highest (FC000H-FFFFFH) and lowest (00000H-003FFH). The 80130 control store code is position-independent except as limited above, in order to make it compatible with many decoding logic designs. $AD_{13}$-$AD_0$ are decoded by the 80130's kernel control store.

### I/O Mapping

The I/O-mapped portion of the 80130 must be aligned along a 16-byte boundry. Address lines $A_{15}$-$A_4$ should be used to form IOCS/.

### System Performance

The approximate performance of representitive OSP primitives is given in Table 5. These times are shown for a typical iAPX 86/30 implementation with an 8 MHz clock. These execution times are very comparable to the execution times of similar functions in minicomputers (where available) and are an order of magnitude faster than previous generation microprocessors.

### Initialization

Both application system initialization and OSP-specific initialization/configuration are required to use the OSP. Configuration is based on a "database" provided by the user to the iOSP 86 support package. The OSP-specific initialization and configuration information area is assigned to a user memory address adjacent to the 80130's memory-mapped location. (See Application Note 130 for further details.) The configuration data defines whether 8087 support is configured in the system, specifies if slave 8259A interrupt controllers are used in addition to the 80130, and sets the operating system time base (Tick Interval). Also located in the configuration area are the exception handler control parameters, the address location of the (separate) application system configuration area and the OSP extensions in use. The OSP application system configuration area may be located anywhere in the user memory and must include the starting address of the application instruction code to be executed, plus the locations of the RAM memory blocks to be managed by the OSP free memory manager. Complete application system support and the required 80130 configuration support are provided by the iAPX 86/30 and iAPX 88/30 OPERATING SYSTEM PROCESSOR SUPPORT PACKAGE (iOSP 86).

### RAM Requirements

The OSP manages its own interrupt vector, which is assigned to low RAM memory. Working RAM storage is required as stack space and data area. The memory space must be allocated in user RAM.

OSP interrupt vector memory locations 0H-3FFH must be RAM based. The OSP requires 2 bytes of allocated RAM. The processor working storage is dynamically allocated from free memory. Approximately 300 bytes of stack should be allocated for each OSP task.

## TYPICAL SYSTEM CONFIGURATION

Figure 8 shows the processing cluster of a "typical" iAPX 86/30 or iAPX 88/30 OSP system. Not shown are subsystems likely to vary with the application. The configuration includes an 8086 (or 8088) operating in maximum mode, an 8284A clock generator and an 8288 system controller. Note that the 80130 is located on the CPU side of any latches or transceivers. See Intel Application Note 130 for further details on configuration.

### OSP Timers

The OSP Timers are connected to the lower half of the data bus and are addressed at even addresses. The timers are read as two successive bytes, always LSB followed by MSB. The MSB is always latched on a read operation and remains latched until read. Timers are not gatable.

### Baud Rate Generator

The baud rate generator is 8254 compatible (square wave mode 3). Its output, BAUD, is initially high and remains high until the Count Register is loaded. The first falling edge of the clock after the Count Register is loaded causes the transfer of the internal counter to the Count Register. The output stays high for N/2 [(N+1)/2 if N is odd] and then goes low for N/2 [(N−1)/2 if N is odd]. On the falling edge of the clock which signifies the final count for the output in low state, the output returns to high state and the Count Register is transferred to the internal counter. The whole process is then repeated. Baud Rates are shown in Table 6.

The baud rate generator is located at 0CH (12), relative to the 16-byte boundary in the I/O space in which the 80130 component is located ("OSF" in the following example), the timer control word is located at

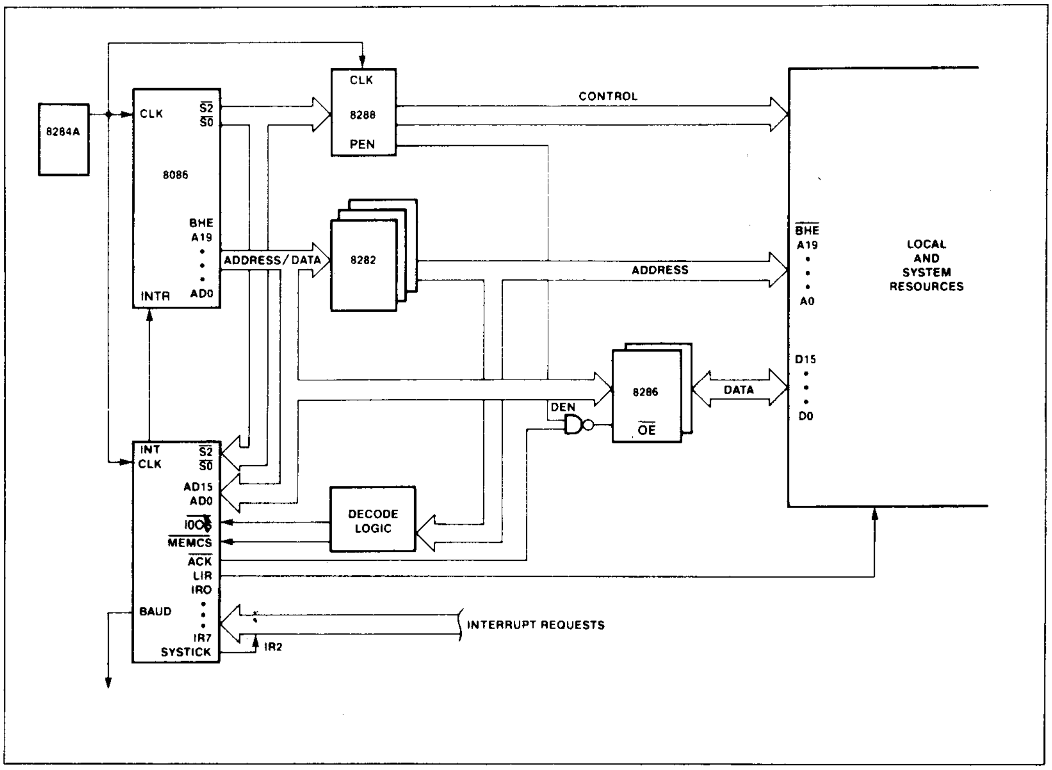


Figure 8. Typical OSP Configuration

relative address, 0EH(14). Timers are addressed with IOCS=0. Timers 0 and 1 are assigned to the use by the OSP, and should not be altered by the user.

For most baud-rate generator applications, the command byte

0B6H Read/Write Baud-Rate Delay Value

will be used. A typical sequence to set a baud rate of 9600 using a count value of 52 follows (see Table 6):

```
MOV  AX, 0B6H     ;Prepare to Write Delay to
                   Timer 3.
OUT  OSF+14,AX    ;Control Word.
MOV  AX, 52
OUT  OSF+12,AL    ;LSB written first
XCHG AL,AH
OUT  OSF+12,AL    ;MSB written after.
```

The 80130 timers are subset compatible with 8254 timers.

## Interrupt Controller

The Programmable Interrupt Controller (PIC), is also an integral unit of the 80130. Its eight input pins handle eight vectored priority interrupts. One of these pins must be used for the SYSTICK time function in timing waits, using an external connection as shown. During the 80130 initialization and configuration sequence, each 80130 interrupt pin is individually programmed as either level or edge sensitive. External slave 8259A interrupt controllers can be used to expand the total number of OSP external interrupts to 57.

In addition to standard PIC funtions, 80130 PIC unit has an $\overline{\text{LIR}}$ output signal, which when low indicates an interrupt acknowledge cycle. $\overline{\text{LIR}}$=0 is provided to control the 8289 Bus Arbiter SYSB/$\overline{\text{RESB}}$ pin. This will avoid the need of requesting the system bus to acknowledge local bus non-slave interrupts. The user defines the interrupt system as part of the configuration.

**INTERRUPT SEQUENCE**
The OSP interrupt sequence is as follows:

1. One or more of the interrupts is set by a low-to-high transition on edge-sensitive IR inputs or by a high input on level-sensitive IR inputs.
2. The 80130 evaluates these requests, and sends an INT to the CPU, if appropriate.
3. The CPU acknowledges the INT and responds with an interrupt acknowledge cycle which is encoded in $\overline{S_2}$-$\overline{S_0}$.
4. Upon receiving the first interrupt acknowledge from the CPU, the highest-priority interrupt is set by the 80130 and the corresponding edge detect latch is reset. The 80130 does not drive the address/data bus during this bus cycle but does acknowledge the cycle by making $\overline{ACK}$=0 and sending the $\overline{LIR}$ value for the IR input being acknowledged.
5. The CPU will then initiate a second interrupt acknowledge cycle. During this cycle, the 80130 will supply the cascade address of the interrupting input at $T_1$ on the bus and also release an 8-bit pointer onto the bus if appropriate, where it is read by the CPU. If the 80130 does supply the pointer, then $\overline{ACK}$ will be low for the cycle. This cycle also has the value $\overline{LIR}$ for the IR input being acknowledged.
6. This completes the interrupt cycle. The ISR bit remains set until an appropriate EXIT INTERRUPT primitive (EOI command) is called at the end of the Interrupt Handler.

## OSP APPLICATION EXAMPLE

Figure 5 shows an application of the OSP primitives to keep track of time of day in a simplified example. The system design uses a 60 Hz A.C. signal as a time base. The power supply provides a TTL-compatible signal which drives one of 80130 edge-triggered interrupt request pins once each A.C. cycle. The Interrupt Handler responds to the interrupts, keeping track of one second's A.C. cycles. The Interrupt Task counts the seconds and after a day deletes itself. In typical systems it might perform a data logging operation once each day. The Interrupt Handler and Interrupt Task are written as separate modular programs.

The Interrupt Handler will actually service interrupt 59 when it occurs. It simply counts each interrupt, and at a count of 60 performs a SIGNAL INTERRUPT to notify the Interrupt Task that a second has elapsed. The Interrupt Handler (ACS HANDLER) was assigned to this level by the SET INTERRUPT primitive. After doing this, the Interrupt Task performed the Primitive RESUME TASK to resume the application task (INITS TASKS TOKEN).

The main body of the task is the counting loop. The Interrupt Task is signaled by the SIGNAL INTERRUPT primitive in the Interrupt Handler (at interrupt level ACS INTERRUPTS LEVEL). When the task is signalled by the Interrupt Handler it will execute the loop exactly one time, increasing the time count variables. Then it will execute the WAIT INTERRUPT primitive, and wait until awakened by the Interrupt Handler. Normally, the task will now wait some period of time for the next signal. However, since the interface between the Handler and the Task is asynchronous, the handler may have already queued the interrupt for servicing, the writer of the task does not have to worry about this possibility.

At the end of the day, the task will exit the loop and execute RESET INTERRUPT, which disables the interrupt level, and deletes the interrupt task. The OSP now reclaims the memory used by the Task and schedules another task. If an exception occurs, the coded value for the exception is available in TIMES EXCEPTS CODE after the execution of the primitive.

A typical PL/M-86 calling sequence is illustrated by the call to RESET INTERRUPT shown in Figure 5.

**Table 2. OSP System Data Type Summary**

| | |
|---|---|
| Job | Jobs are the means of organizing the program environment and resources. An application consists of one or more jobs. Each iAPX 86/30 system data type is contained in some job. Jobs are independent of each other, but they may share access to resources. Each job has one or more tasks, one of which is an initial task. Jobs are given pools of memory, and they may create subordinate offspring jobs, which may borrow memory from their parents. |
| Task | Tasks are the means by which computations are accomplished. A task is an instruction stream with its own execution stack and private data. Each task is part of a job and is restricted to the resources provided by its job. Tasks may perform general interrupt handling as well as other computational functions. Each task has a set of attributes, which is maintained for it by the iAPX 86/30, which characterize its status. These attributes are:<br>its containing job<br>its register context<br>its priority (0–255)<br>its execution state (asleep, suspended, ready, running, asleep/suspended).<br>its suspension depth<br>its user-selected exception handler<br>its optional 8087 extended task state |
| Segment | Segments are the units of memory allocation. A segment is a physically contiguous sequence of 16-byte, 8086 paragraph-length, units. Segments are created dynamically from the free memory space of a Job as one of its Tasks requests memory for its use. A segment is deleted when it is no longer needed. The iAPX 86/30 maintains and manages free memory in an orderly fashion, it obtains memory space from the pool assigned to the containing job of the requesting task and returns the space to the job memory pool (or the parent job pool) when it is no longer needed. It does not allocate memory to create a segment if sufficient free memory is not available to it, in that case it returns an error exception code. |
| Mailbox | Mailboxes are the means of intertask communication. Mailboxes are used by tasks to send and receive message segments. The iAPX 86/30 creates and manages two queues for each mailbox. One of these queues contains message segments sent to the mailbox but not yet received by any task. The other mailbox queue consists of tasks that are waiting to receive messages. The iAPX 86/30 operation assures that waiting tasks receive messages as soon as messages are available. Thus at any moment one or possibly both of two mailbox queues will be empty. |
| Region | Regions are the means of serialization and mutual exclusion. Regions are familiar as "critical code regions." The iAPX 86/30 region data type consists of a queue of tasks. Each task waits to execute in mutually exclusive code or to access a shared data region, for example to update a file record. |
| Tokens | The OSP interface makes use of a 16-bit TOKEN data type to identify individual OSF data structures. Each of these (each instance) has its own unique TOKEN. When a primitive is called, it is passed the TOKENs of the data structures on which it will operate. |

Table 3. System Data Type Codes and Attributes

| S.D.T. | Code | Attributes |
|---|---|---|
| Jobs | 1 | Tasks<br>Memory Pool<br>S.D.T. Directory |
| Tasks | 2 | Priority<br>Stack<br>Code<br>State<br>Exception Handler |
| Mailboxes | 3 | Queue of S.D.T.s (generally segments)<br>Queue of Tasks waiting for S.D.T.s |
| Region | 5 | Queue of Tasks waiting for mutually exclusive code or data |
| Segments | 6 | Buffer<br>Length |

Table 4. OSP Primitives

| Class | OSP Primitive | Interrupt Number | Entry Code in AX | Parameters On Caller's Stack |
|---|---|---|---|---|
| JOB | CREATE JOB | 184 | 0100H | *See 80130 User Manual |
| TASK | CREATE TASK | 184 | 0200H | Priority, IP Ptr, Data Segment, Stack Seg, Stack Size Task Information, ExcptPtr |
| | DELETE TASK | 184 | 0201H | TASK, ExcptPtr |
| | SUSPEND TASK | 184 | 0202H | TASK, ExcptPtr |
| | RESUME TASK | 184 | 0203H | TASK, ExcptPtr |
| | SET PRIORITY | 184 | 0209H | TASK, Priority, ExcptPtr |
| | SLEEP | 184 | 0204H | Time Limit,ExcptPtr |
| INTERRUPT | DISABLE | 190 | 0705H | Level, ExcptPtr |
| | ENABLE | 184 | 0704H | Level #, ExcptPtr |
| | ENTER INTERRUPT | 184 | 0703H | Level #, ExcptPtr |
| | EXIT INTERRUPT | 186 | NONE | Level #,ExcptPtr |
| | GET LEVEL | 188 | 0702H | Level #, ExcptPtr |
| | RESET INTERRUPT | 184 | 0706H | Level #, ExcptPtr |
| | SET INTERRUPT | 184 | 0701H | Level, Interrupt Task Flag Interrupt Handler Ptr, Interrupt Handler DataSeg ExcptPtr |
| | SIGNAL INTERRUPT | 185 | NONE | Level, ExcptPtr |
| | WAIT INTERRUPT | 187 | NONE | Level, ExcptPtr |
| SEGMENT | CREATE SEGMENT | 184 | 0600H | Size, ExcptPtr |
| | DELETE SEGMENT | 184 | 0603H | SEGMENT, ExceptPtr |

Table 4. OSP Primitives (Continued)

| Class | OSP Primitive | Interrupt Number | Entry Code in AX | Parameters On Caller's Stack |
|---|---|---|---|---|
| MAILBOX | CREATE MAILBOX | 184 | 0300H | Mailbox flags, ExcptPtr |
| | DELETE MAILBOX | 184 | 0301H | MAILBOX, ExcptPtr |
| | RECEIVE MESSAGE | 184 | 0303H | MAILBOX, Time Limit ResponsePtr, ExcptPtr |
| | SEND MESSAGE | 184 | 0302H | MAILBOX,Message Response, ExcptPtr |
| REGION | ACCEPT CONTROL | 184 | 0504H | REGION, ExcptPtr |
| | CREATE REGION | 184 | 0500H | Region Flags, ExcptPtr |
| | DELETE REGION | 184 | 0501H | REGION, ExcptPtr |
| | RECEIVE CONTROL | 184 | 0503H | REGION, ExcptPtr |
| | SEND CONTROL | 184 | 0502H | ExcptPtr |
| ENVIRONMENTAL | DISABLE DELETION | 184 | 0001H | TOKEN,ExcptPtr |
| | ENABLE DELETION | 184 | 0002H | TOKEN,ExcptPtr |
| | GET EXCEPTION HANDLER | 184 | 0800H | Ptr,ExcptPtr |
| | GET TYPE | 184 | 0000H | TOKEN,ExcptPtr |
| | GET TASK TOKENS | 184 | 0206H | Request, ExcptPtr |
| | SET EXCEPTION HANDLER | 184 | 0801H | Ptr, ExcptPtr |
| | SET OS EXTENSION | 184 | 0700H | Code,InstPtr, ExcptPtr |
| | SIGNAL EXCEPTION | 184 | 0802H | Exception Code, Parameter Number, StackPtr,0,0,ExcptPtr |

**NOTES:**
All parameters are pushed onto the OSP stack. Each parameter is one word. See Figure 3 for Call Sequence.

**Explanation of the Symbols**

| | |
|---|---|
| JOB | OSP JOB SDT Token |
| TASK | OSP TASK SDT Token |
| REGION | OSP REGION SDT Token |
| MAILBOX | OSP MAILBOX SDT Token |
| SEGMENT | OSP SEGMENT SDT Token |
| TOKEN | Any SDT Token |
| Level | Interrupt Level Number |
| ExcptPtr | Pointer to Exception Code |
| Message | Message Token |
| Ptr | Pointer to Code,Stack etc. Address |
| Seg` | Value Loaded into appropriate Segment Register |
| ---- | Value Parameter. |

Table 5. OSP Primitive Performance Examples

| Datatype Class | Primitive Execution Speed* (microseconds) | |
|---|---|---|
| JOB | CREATE JOB | 2950 |
| TASK | CREATE TASK (no preemption) | 1360 |
| SEGMENT | CREATE SEGMENT | 700 |
| MAILBOX | SEND MESSAGE (with task switch) | 475 |
| | SEND MESSAGE (no task switch) | 265 |
| | RECEIVE MESSAGE (task waiting) | 540 |
| | RECEIVE MESSAGE (message waiting) | 260 |
| REGION | SEND CONTROL | 170 |
| | RECEIVE CONTROL | 205 |

*8 MHz iAPX 86/30 OSP Configuation.

Table 6. Baud Rate Count Values (16X)

| Baud Rate | 8 MHz Count Value | 5 MHz Count Value |
|---|---|---|
| 300 | 1667 | 1042 |
| 600 | 833 | 521 |
| 1200 | 417 | 260 |
| 2400 | 208 | 130 |
| 4800 | 104 | 65 |
| 9600 | 52 | 33 |

Table 7a. Mnemonic Codes for Unavoidable Exceptions

| | |
|---|---|
| E$OK | Exception Code Value = 0<br>the operation was successful |
| E$TIME | Exception Code Value = 1<br>the specified time limit expired before completion of the operations was possible |
| E$MEM | Exception Code Value = 2<br>insufficient nucleus memory is available to satisfy the request |
| E$BUSY | Exception Code Value = 3<br>specified region is currently busy |
| E$LIMIT | Exception Code Value = 4<br>attempted violation of a job, semaphore, or system limit |
| E$CONTEXT | Exception Code Value = 5<br>the primitive was called in an illegal context (e.g., call to enable for an already enabled interrupt) |
| E$EXIST | Exception Code Value = 6<br>a token argument does not currently refer to any object; note that the object could have been deleted at any time by its owner |
| E$STATE | Exception Code Value = 7<br>attempted illegal state transition by a task |
| E$NOT$CONFIGURED | Exception Code Value = 8<br>the primitive called is not configured in this system |
| E$INTERRUPT$SATURATION | Exception Code Value = 9<br>The interrupt task on the requested level has reached its user specified saturation point for interrupt service requests. No further interrupts will be allowed on the level until the interrupt task executes a WAIT$INTERRUPT. (This error is only returned, in line, to interrupt handlers.) |
| E$INTERRUPT$OVERFLOW | Exception Code Value = 10<br>The interrupt task on the requested level previously reached its saturation point and caused an E$INTERRUPT$SATURATION condition. It subsequently executed an ENABLE allowing further interrupts to come in and has received another SIGNAL$INTERRUPTcall, bringing it over its specified saturation point for interrupt service requests. (This error is only returned, in line, to interrupt handlers). |

Table 7b. Mnemonic Codes for Avoidable Exceptions

| | |
|---|---|
| E$ZERO$DIVIDE | Exception Code Value = 8000H<br>divide by zero interrupt occurred |
| E$OVERFLOW | Exception Code Value = 8001H<br>overflow interrupt occurred |
| E$TYPE | Exception Code Value = 8002H<br>a token argument referred to an object tha was not of required type |
| E$BOUNDS | Exception Code Value = 8003H<br>an offset argument is out of segment bounds |
| E$PARAM | Exception Code Value = 8004H<br>a (non-token,non-offset) argument has an illegal value |
| E$BAD$CALL | Exception Code Value = 8005H<br>an entry code for which there is no corresponding primitive was passed |
| E$ARRAY$BOUNDS = 8006H | Hardware or Language has detected an array overflow |
| E$NDP$ERROR | Exception Code Value = 8007H<br>an 8087 (Numeric data Processor) error has been detected; (the 8087 status information is contained in a parameter to the exception handler) |

## ABSOLUTE MAXIMUM RATINGS*

| | |
|---|---|
| Ambient Temperature Under Bins | 0°C to 70°C |
| Storage Temperature | −65°C to 150°C |
| Voltage on Any Pin With Respect to Ground | −1.0V to +7V |
| Power Dissipation | 1.0 Watts |

**NOTICE: Stresses above those listed under Absolute Maximum Ratings may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended period may affect device reliability.*

## D.C. CHARACTERISTICS ($T_A$ = 0°C to 70°C, $V_{CC}$ = 4.5 to 5.5V)

| Symbol | Parameter | Min. | Max. | Units | Test Conditions |
|---|---|---|---|---|---|
| $V_{IL}$ | Input Low Voltage | −0.5 | 0.8 | V | |
| $V_{IH}$ | Input High Voltage | 2.0 | $V_{CC}$ + .5 | V | |
| $V_{OL}$ | Output Low Voltage | | 0.45 | V | $I_{OL}$ = 2mA |
| $V_{OH}$ | Output High Voltage | 2.4 | | V | $I_{OH}$ = −400μA |
| $I_{CC}$ | Power Supply Current | | 200 | mA | $T_A$ = 25 C |
| $I_{LI}$ | Input Leakage Current | | 10 | μA | $0 < V_{IN} < V_{CC}$ |
| $I_{LR}$ | IR Input Load Current | | 10<br>−300 | μA<br>μA | $V_{IN} = V_{CC}$<br>$V_{IN} = 0$ |
| $I_{LO}$ | Output Leakage Current | | 10 | μA | $.45 = V_{IN} = V_{CC}$ |
| $V_{CLI}$ | Clock Input Low | | 0.6 | V | |
| $V_{CHI}$ | Clock Input High | 3.9 | | V | |
| $C_{IN}$ | Input Capacitance | | 10 | pF | |
| $C_{IO}$ | I/O Capacitance | | 15 | pF | |
| $I_{CLI}$ | Clock Input Leakage Current | | 10<br>150<br>10 | μA<br>μA<br>μA | $V_{IN} = V_{CC}$<br>$V_{IN}$ = 2.5V<br>$V_{IN}$ = 0V |

## A.C. CHARACTERISTICS ($T_A$ = 0-70°C, $V_{CC}$ = 4.5-5.5 Volt, $V_{SS}$ = Ground)

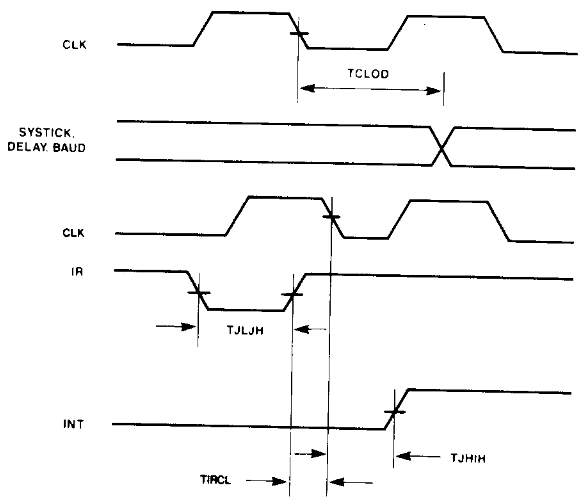
| | | 80130 | | 80130-2 | | | |
|---|---|---|---|---|---|---|---|
| Symbol | Parameter | Min. | Max. | Min. | Max. | Units | Test Conditions |
| $T_{CLCL}$ | CLK Cycle Period | 200 | – | 125 | – | ns | |
| $T_{CLCH}$ | CLK Low Time | 90 | – | 55 | – | ns | |
| $T_{CHCL}$ | CLK High Time | 69 | 2000 | 44 | 2000 | ns | |
| $T_{SVCH}$ | Status Active Setup Time | 80 | – | 65 | – | ns | |
| $T_{CHSV}$ | Status Inactive Hold Time | 10 | – | 10 | – | ns | |
| $T_{SHCL}$ | Status Inactive Setup Time | 55 | – | 55 | – | ns | |
| $T_{CLSH}$ | Status Active Hold Time | 10 | – | 10 | – | ns | |
| $T_{ASCH}$ | Address Valid Setup Time | 8 | – | 8 | – | ns | |
| $T_{CLAH}$ | Address Hold Time | 10 | – | 10 | – | ns | |
| $T_{CSCL}$ | Chip Select Setup Time | 20 | – | 20 | – | ns | |
| $T_{CHCS}$ | Chip Select Hold Time | 0 | – | 0 | – | ns | |
| $T_{DSCL}$ | Write Data Setup Time | 80 | – | 60 | – | ns | |
| $T_{CHDH}$ | Write Data Hold Time | 10 | – | 10 | – | ns | |
| $T_{JLJH}$ | IR Low Time | 100 | – | 100 | – | ns | |
| $T_{CLDV}$ | Read Data Valid Delay | – | 140 | – | 105 | ns | $C_L$ = 200 pE |
| $T_{CLDH}$ | Read Data Hold Time | 10 | – | 10 | – | ns | |
| $T_{CLDX}$ | Read Data to Floating | 10 | 100 | 10 | 100 | ns | |
| $T_{CLCA}$ | Cascade Address Delay Time | – | 85 | – | 65 | ns | |

## A.C. CHARACTERISTICS (Continued)

| Symbol | Parameter | 80130 Min. | 80130 Max. | 80130-2 Min. | 80130-2 Max. | Units | Notes |
|---|---|---|---|---|---|---|---|
| $T_{CLCF}$ | Cascade Addresse Hold Time | 10 | – | 10 | – | ns | |
| $T_{IAVE}$ | INTA Status t Acknowledge | – | 80 | – | 80 | ns | |
| $T_{CHEH}$ | Acknowledge Hold Time | 0 | – | 0 | – | ns | |
| $T_{CSAK}$ | Chip Select to ACK | – | 110 | – | 110 | ns | |
| $T_{SACK}$ | Status to ACK | – | 140 | – | 140 | ns | |
| $T_{AACK}$ | Address to ACK | – | 90 | – | 90 | ns | |
| $T_{CLOD}$ | Timer Output Delay Time | – | 200 | – | 200 | ns | $C_L$ = 100 pF |
| $T_{CLOD1}$ | Timer1 Output Delay Time | – | 200 | – | 200 | ns | $C_L$ = 100 pF |
| $T_{JHIH}$ | INT Output Delay | – | 200 | – | 200 | ns | |
| $T_{IRCL}$ | IR Input Set Up | 20 | | 20 | | ns | |

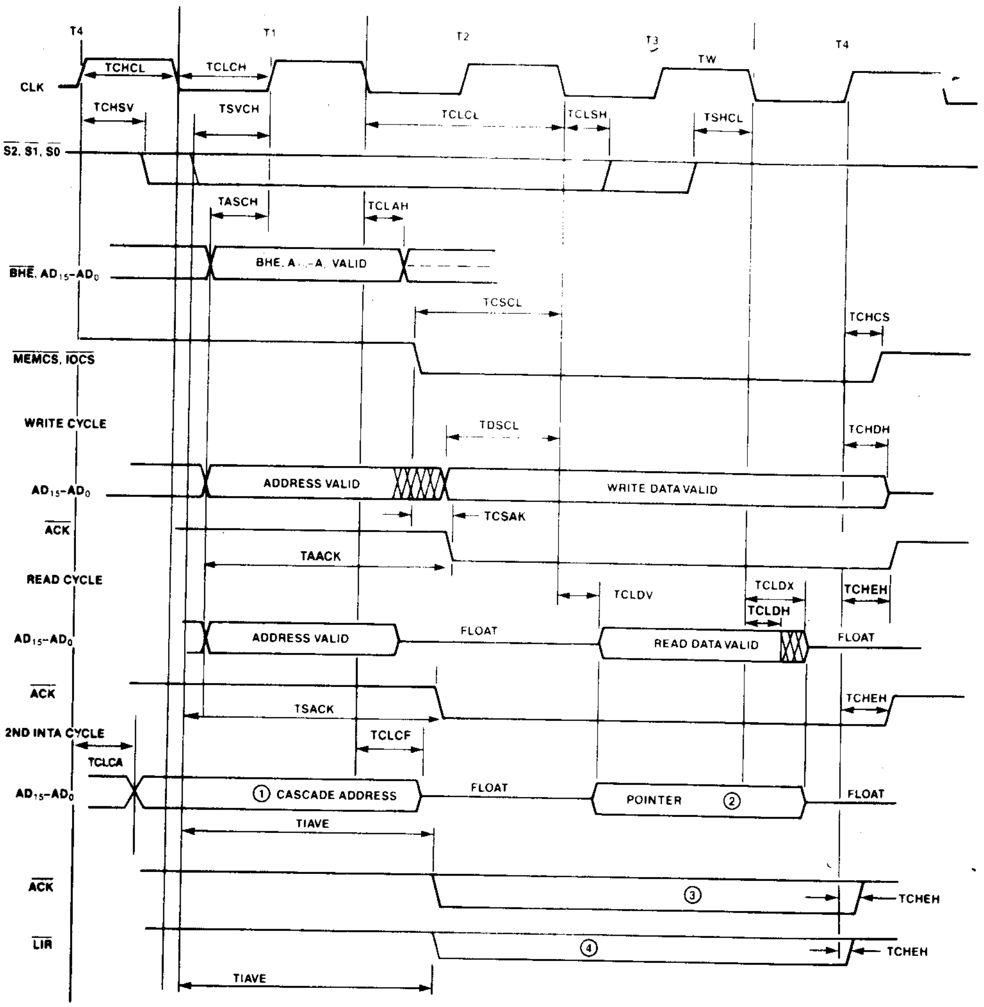
## WAVEFORMS

### A.C.



## WAVEFORMS

### A.C.



NOTES:
1. CASCADE ADDRESS PRESENTED ON AD8, AD9 AND AD10 CORRESPONDING TO CAS0, CAS1 AND CAS2 RESPECTIVELY. AD11-AD15 LINES ARE ACTIVE AND HAVE UNKNOWN VALUES. AD0-AD7 ARE TRISTATE.
2. POINTER VALUE IS ACTIVE ONLY IF POINTER IS GENERATED FROM THE 80150 AND NOT FROM EXTERNAL SLAVE UNIT.
3. ACTIVE LOW ONLY WHEN POINTER DATA IS BEING SUPPLIED BY THE 80150.
4. LOW ONLY FOR LOCAL INTERRUPT.