USER'S GUIDE FOR
MULTI-TASKING OPERATING SYSTEM
MTOS-68

INDUSTRIAL PROGRAMMING, INC.

9 Northern Blvd.

Greenvale, N.Y. 11548

January 1976

User's Guide for MTOS

The information in this document is subject to change, update and revision without notice. Industrial Programming, Inc., assumes no responsibility for any errors that may appear in this manual or the associated software.

The software described in this document is furnished to the purchaser under a license which forbids reproduction in whole or part except as provided in writing by Industrial Programming, Inc.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF FIGURES

TABLE OF APPENDICES

USER'S GUIDE FOR MTOS

1. Introduction

An operating system is used to increase the effectiveness of a computer by allowing several activities to proceed simultaneously and to share common facilities without interference. Simultaneous operation increases throughput; facility sharing conserves memory.

A Multi-Tasking Operating System (MTOS), in particular, permits the user to divide his program into separate, individual segments called "tasks". A task is an independent program which can run either alone or simultaneously with other tasks. This independence, however, does not exclude coordination among tasks, or the sharing of resources such as a data base or set of utility subprograms.

The ability of a task to be run as a separate program does not imply that it is running at all times. Often it is not. Each task will be in one of four general states: dormant, waiting, ready, or active.

A dormant task is totally inactive; it is not running. Either it has never started, or it has run and terminated. A dormant task may be started by another task or by the operating system in response to some external event, e.g. recovery after a power failure.

A waiting task is currently running, but is temporarily blocked from continuing. There are several wait states depending upon the type of blockage. Generally, a task is either waiting to use some shared facility which is busy, waiting for a requested service to be completed, or waiting for some internal event such as the receipt

of a coordination (go ahead) signal from another task. While a task is in one of these wait states it does not compete for use of the computer's Central Processing Unit (CPU).

The active task is the one presently using the CPU. The ready tasks are those which could use the CPU if it were available.

MTOS supports the individual tasks by providing functions and services which are beyond the capability of any one task or which are most efficiently concentrated in a central area. For example, the allocation of the CPU must be an operating system function since dynamic, system-wide information is needed to resolve the conflicting needs of several tasks. In contrast, the input of discrete signals could in principle be programmed into each task which must react to such signals. In many cases, however, program efficiency is improved by centralizing the processing of discretes.

The broad categories of MTOS service are:

- o management of tasks
- o coordination of shared subprograms
- o management of time
- o input and output of Console message and discretes

Each of these categories is described in later sections of this document. The discussion will cover both those services which are provided automatically by MTOS (and thus which are not directly visible to the task programs), and those services which come into play only when directly requested by a task program.

In all cases, MTOS services are initiated by interrupts. Many of these interrupts are generated and serviced by MTOS with only

the indirect effects seen within the tasks. For example, the real-time clock periodically generates an interrupt. MTOS uses that interrupt in its timekeeping functions. However, unless a task has requested a pause for a given amount of time, or has requested some other time-dependent service, the clock interrupt has no direct consequence for that task.

2. Supervisor Calls

When a task needs service, it issues a supervisor call (SVC) using the following code:

```
SVCPGM EQU $E400
        .
        .
        .
       JSR SVCPGM
       FDB data block
```

The data block indicates the type of request and provides any required parameters.

The SVCs within MTOS are:

0. terminate the issuing task without automatic restart
1. terminate the issuing task with automatic restart after a given interval
2. pause for a given time interval
3. cancel the pause of a given task
4. start a given task if it is dormant
5. start a given task, queuing the request if the task is busy
6. enter a controlled shared subprogram
7. exit from a controlled shared subprogram
8. set/reset/test an external discrete
9. set/reset/test an internal event flag
10. wait until event flags are set
11. set an event flag after a given interval
12. read a message from the Console
13. write a message on the Console

All of the SVCs will be described in the sections to follow. Appendix B contains a summary of the required data blocks.

The supervisor call uses an internal MTOS subprogram (SVCPGM) rather than the software interrupt (SWI) instruction. This avoids two problems. First, the Motorola EXORciser uses the SWI for break-points. Our use of an internal subprogram eliminates this conflict and permits program development on the EXORciser. Second, if a non-maskable interrupt occurs during SWI processing, a bogus peripheral interrupt is generated by the hardware (see Motorola Applications Manual, Page A-10, Q9). MTOS is not subject to this anomaly.

The same data block may be referenced by several individual requests. For example, the data block to pause for 30 seconds is:

```
1000 02      PA30SC FCB   2        (=pause)
1001 02             FCB   2        (=seconds)
1002 1E             FCB   30       (=number of units)
```

Two separate requests for such a pause could be:

```
1500 BD E400        JSR   SVCPGM   PAUSE FOR 30 SECONDS
1503 1000           FDB   PA30SC
                     .
                     .
                     .
1641 BD E400        JSR   SVCPGM   PAUSE FOR ANOTHER 30 SECONDS
1643 1000           FDB   PA30SC
```

Furthermore, the data block may be in either read-only memory (ROM) or read-write memory (RAM). Thus, a pause for a variable number of seconds could be constructed as follows:

```
0120 003     PAUSE  RMB   3        (in RAM)

1710 CE 0202        LDX   #$0202   SET UP PAUSE WITH SECONDS
1713 FF 0120        STX   PAUSE    AS TIME UNIT
                     .
                     .    (calculate number of seconds)
                     .
174F B7 0122        STA A PAUSE+2  STORE CALCULATED NUMBER OF SEC(=A)
1752 BD E400        JSR   SVCPGM   PAUSE
1755 0120           FDB   PAUSE
```

Upon return from SVCPGM, the condition code is set to convey certain information about the SVC. The specific meaning of the individual bits varies with the particular type of request. However, the following general scheme has been adopted for all SVCs:

H (half carry):	always 0.
I (interrupt mask):	always 0.
N (negative):	always 0.
Z (zero):	=1 if the requested condition already exists so that the SVC has no effect. For example, if the SVC request was to start a given task if it is dormant, and that task was already started then Z is set to 1.
	=0 otherwise
V (overflow):	=1 if any of the parameters are beyond the proper range, such as an SVC type of 99.
	=0 if all parameters are within range
C (carry):	= original value of external discrete or internal event flag.

3. Task Management

Task management encompasses:

- o CPU management
- o task overhead
- o task startup
- o task termination
- o task coordination

3.1 CPU Management

While several tasks may be running simultaneously, only one task can use the computer's Central Processing Unit (CPU) at a time. This task is said to be "active". MTOS determines which one of the ready tasks will be active.

To facilitate CPU management, MTOS maintains a queue of all ready tasks. This Ready Tasks Queue (RTQ) is dynamic. For example, when a dormant task first starts it is placed on the RTQ. If at some point the task requests a pause for a given amount of time, it is temporarily removed from the RTQ; when the time elapses, the task is reinstated. When the task finally terminates, it is taken permanently from the RTQ. These processes are repeated for each subsequent restart of the task.

Ordering within the RTQ is determined by a current priority associated with each task. Priorities range from 0 (lowest) to 15 (highest). The priority is set when a task first starts and remains constant until the task terminates. However, each restart of the task may be made at a different priority. (Assignment of task

priority is discussed in Section 3.3.2.)

MTOS gives the CPU to that ready task having the highest priority. For tasks of the same priority: first-come, first-served. Since the RTQ is maintained in descending order of current priority, the active task is merely the head of the RTQ.

The active task retains control until an interrupt occurs. At the interrupt the active task is automatically suspended and the state of the machine (registers, condition code and program counter) is saved in the task's stack. (Each running task has its own stack.) MTOS then services the interrupt. The service may change task status so that a new task becomes the head of the RTQ. To resume task processing, MTOS loads the state of the machine from the stack of whatever task is at the head of the RTQ.

3.2 Task Overhead

CPU control is an internal function of the operating system and requires no effort on the part of the user. There is nevertheless certain storage overhead which must be provided for each task. These are described fully in Appendix A. In brief, the overheads are:

- A stack for use by MTOS to store the state of the microprocessor during interrupt processing, and by the task to store sub-program return addresses and temporary data. The user must keep 18 bytes within the stack always available to MTOS.
- A 7 byte entry within a table of Task Control Data (TCD). The TCD is fixed by the user and contains the program entry point, inherent priority, and similar static source data.

The information must be in ROM.

An 18 byte Task Control Block (TCB) which is created and used internally by MTOS. Since the TCB contains dynamic data, it must be in RAM.

3.3 Task Startup

3.3.1 Automatic Task Startup

When power is first applied to the processor (that is, during the servicing of a reset interrupt), MTOS sets all application tasks dormant except for a user-written "reset processing" task. Although there are no special restrictions on the reset task, it is expected that it will initialize certain data and then will start other tasks as required by the particular application.

Note, however, that recovery from a temporary power failure and initial start of the system are indistinguishable as far as the basic microprocessor is concerned; they both generate a "reset" interrupt. In fact, without some external hardware to make a distinction, turning on the processor is simply the recovery from an extended period of power outage.

Since the user may wish to proceed differently after temporary power failure than after system restart, MTOS provides an external initialize button which is used upon system restart. A depression of this button is treated exactly as a reset interrupt, except that a different user task--the initialize task--becomes active.

3.3.2 Unconditional Requests for Task Startup

The reset and initialize tasks generally start other tasks

which may, in turn, start still other tasks. The mechanism by which one task requests the start of another is through SVC 5.

All SVCs are coded as JSR SVCPGM followed by the address of a supplementary Supervisor Data Block (SDB). The SDB for requesting task startup is composed as:

```
FCB    5                                          (type)
FCB    parameter 1                                (options and control)
FDB    address of TCD for task to be started      (target TCD)
FDB    address of 7 byte RAM scratch area in      (RCB address)
         which to build queue block
```

Parameter 1 contains bit-by-bit control data:

- o If bit 7 is a 1 the value currently in the X register of the requesting task will be placed in the X register of the started task as a startup argument; otherwise the target task will start with the X register cleared. This mechanism provides for a single, 16-bit argument to be transmitted from the requesting to the target task. However, since the user can establish the convention that the argument always points to the start of an entire data array, there is no limit to the size of the effective argument list.
- o The target task can only be started if it is currently dormant. If it is running, the request is automatically queued and will eventually result in a restart of the target task. Bits 6 and 5 indicate the coordination between the requesting and target task:

  00 the requesting task continues without wait after the request is queued,

  01 the requesting task waits for the target task to start as a result of the current request,

10 the requesting task waits for the target task to start and terminate as a result of the current request,

11 not used.

- The requesting task controls the priority the target task will have when it starts via the current request. If bit 4 is 1 the target task will start with whatever priority is given in bits 3 to 0; if bit 4 is a 0 the priority will be computed by MTOS as the larger of the inherent priority of the target task and the current priority of the requesting task. (The inherent priority is a fixed property preserved within the TCD.) The priority also determines the order in which startup requests are queued to the target task. Requests are serviced in order of restart priority.

Note that the target task is indicated by giving the address of its TCD (as opposed to its TCB). This convention is carried throughout MTOS. Thus, if the TCD starts with a label, that label becomes effectively the "name" of the task.

The final element within the startup SDB is the address of a scratch area in which MTOS may form a startup Request Control Block (RCB). Any scratch area in RAM may be used. However, the area must not be modified by the user from the time the SVC is issued until the target task terminates from the request. Violation of this rule may cause errors which could be very difficult to trace. To help determine when a RCB is available for reuse, MTOS sets the leading two bytes to hexadecimal FFFF (all ones) when the block is no longer needed.

Some typical requests for task startup are shown in Figure 3-1. Although all examples use an SDB within ROM this is not an inherent restriction. An SDB can also be formed within RAM. In fact, this would be required if either the given priority or the target task were to be computed dynamically rather than being fixed at program assembly time.

All condition codes are reset to zero upon return from SVC 5.

For a task started by SVC 5, the stack pointer is set to the value given within the TCD and registers A and B have some arbitrary values.

3.3.3 Conditional Requests for Task Startup

The SVC 5 just described is an unconditional request for task startup. It always results in the eventual startup of the target task. If that task is busy, the request is simply queued.

In some cases, it is sufficient merely to have a target task running without reference to how the startup occurred. For these situations MTOS provides an SVC to startup the target task if it is dormant. If the task is already running, the SVC has no effect.

The format of the SDB is:

```
FCB     4                                   (type)
FDB     address of TCD for task to          (target.TCD)
          be started
```

When the requesting task continues after the SVC has been processed the Zero flag within its condition code will be 0 if the task was originally dormant or 1 if the task was already running (so that the SVC had no effect).

Case 1. Start task TS1 with priority 7, no argument, and wait for task to terminate:

```
            JSR   SVCPGM
            FDB   S5SDB1
             .
             .                          (in ROM)
             .
S5SDB1      FCB   5
            FCB   $57
            FDB   TS1TCD
            FDB   TEMP7B


TEMP7B      RMB   7                    (in RAM)
```

Case 2. Start task TS2 with computed priority, transmit 4 as an argument, and wait for task to start:

```
            LDX   #4
            JSR   SVCPGM
            FDB   S5SDB2
             .
             .                          (in ROM)
             .
S5SDB2      FCB   5
            FCB   $A0
            FDB   TS2TCD
            FDB   TEMP7B
```

Case 3. Start task TS3 with computed priority, transmit address of entire argument array (ARGARY), and continue without coordination:

```
            LDX   #ARGARY
            JSR   SVCPGM
            FDB   S5SDB3
             .
             .                          (in ROM)
             .
S5SDB3      FCB   5
            FCB   $80
            FDB   TS3TCD
            FDB   TEMP7B
```

Figure 3-1 Some Typical Requests for Task Startup

For a task started by an SVC 4, the stack pointer is set to the value given within the TCD and the registers have some arbitrary values.

3.3.4 Requests for Task Termination

When a task finishes its processing it issues a termination SVC. Two SVCs are available: one automatically reschedules a restart of the task after a given time interval; the other does not. In the former case, the task is considered busy and thus is not available for restart by another task. In the latter case, if a restart request had been queued it will now be honored; otherwise the task becomes dormant.

Every task would normally have at least one termination SVC; there may be several such calls within a given task. If there are several terminations some may request restart and others may not.

The SDB for termination without restart is a single zero byte. Thus, the entire SVC is a call of the SVC program followed by the address of a byte containing a zero:

```
        JSR     SVCPGM                  TERMINATE WITHOUT RESTART
        FDB     ZERO
         .
         .
         .
ZERO    FCB     0
```

The SDB for termination with automatic restart after a given time interval is composed as:

```
        FCB     1                       (type)
        FCB     parameter 1             (option and control)
        FCB     number of time units    (0 to 255)
```

Parameter 1 contains bit-by-bit information: bit 7 indicates if the restart is to be based on the last scheduled start time for the task (bit 7=0) or on the current (termination) time (bit 7=1). The former is normally used for cyclic tasks, i.e. tasks which become active on a regular period such as every 15 minutes. Bits 6 thru 3 are not used. Bits 2 thru 0 stipulate the time interval:

0 = ms
1 = 10 ms
2 = seconds
3 = minutes
4 = hours

Specifying a time interval code greater than 4 is invalid. In such cases, the task will start immediately with the Overflow bit of the condition code set to 1.

If a proper interval is given but the number of units is 0, the task will also restart immediately. This, however, is a valid restart and the Overflow bit will be reset.

Some time intervals can be composed in several ways. For example, 20 ms can be considered twenty intervals of 1 ms or two intervals of 10 ms. Both specifications are equivalent and neither offers any special advantage for interval processing.

Note that the last start time is the time the task first became ready, not the time it actually began processing. If there were higher priority ready tasks, the actual start may have been delayed. Furthermore, suppose a task is to be restarted every 5 minutes based on start time. Assume further, that the task started on

schedule at 11:00 but did not terminate until 11:07. Upon termination the task immediately restarts (since 11:07 is past the next scheduled restart time of 11:05). If the task terminates before 11:10 it will wait until then to restart; if it terminates after 11:10 it will again restart immediately trying to "catch-up".

While a task is waiting to restart at a future time it is considered running in a wait state, not dormant. Thus, it can not be restarted by another task (although other tasks may still queue future restarts). Furthermore, its stack (and TCB area) are reserved and should not be used by other tasks.

3.3.5 State of Task Upon Startup

Whenever a task starts (or restarts) processing begins at the entry point stipulated in the TCD, with the condition code set to zero. When a task resumes after termination with automatic restart, all registers (including the stack pointer) have the same value they had upon termination. For all other restarts, the stack pointer is reset to the initial value given in the TCD, the index register points to a restart argument or is 0, and the other registers have some arbitrary values.

3.3.6 Task Coordination Via Event Flags

Coordination between tasks may be accomplished in several ways. Section 3.3.2 described one method: a requesting task starts a target task. The requesting task has the option of proceeding immediately (no coordination), waiting until the target task starts from the request (concurrent running), or waiting until the target

task starts and terminates from the request (assured completion). A later section will outline coordination by pause cancellation.

MTOS also supports coordination via event flags. Within MTOS there is an array of 16 discrete, internal variables called "event flags". Each flag may be independently set to 1 or reset to 0. Upon system reset or initialization all flags are automatically cleared to zero.

The coordination is achieved by a pair of SVCs. One resets, sets or tests an individual flag; the other causes the issuing task to wait until 1 or more flags are set.

In a typical application, a target task might first reset one of the event flags and then issue a wait until that flag is set. Some time in the future a coordinating task can set the event flag to continue the waiting task. The assignment of individual event flags for the purposes of coordination is left completely to the user.

The SDB for the Reset, Set, or Test SVC is composed as:

```
FCB    9              (type)
FCB    parameter 1    (options and control)
```

Bits 7 and 6 of parameter 1 indicate the required function:

```
00 or 01 = test flag
10       = reset flag
11       = set flag
```

Bits 5 and 4 are not used. Bits 3 thru 0 give the flag number, 0 to 15.

For all functions, the value of the event flag at the time the SVC was issued is returned to the task in the Carry bit of the condition code. For the test function there is no other processing. For set or

reset the value is then forced to 1 or 0 respectively. In these cases, if the new value is the same as the old (so that the SVC causes no real change), the Zero bit of the task's condition code is also set to 1; otherwise it is reset. Some typical applications of SVC 9 by itself are shown in Figure 3-2.

The SDB for the Wait Until Event Flags Are Set SVC is 3 bytes long:

```
FCB     10      (type)
FDB     mask    (16 bits)
```

The 16 bit mask indicates which of the event flags are to end the wait, with the left-most bit corresponding to flag 0 and the right-most bit corresponding to flag 15. Any or all of the bits may be masked on. The task wait ends when any of the indicated flags are set. Figure 3-3 illustrates task coordination via an event flag.

Throughout MTOS all bit arrays (such as the event flags) are stored left-to-right in successive bits of successive bytes. Thus, flag 0 is stored in the left-most bit of byte 0, flag 1 is stored in the next bit of byte 0, . . ., flag 7 is stored in the right-most bit of byte 0, flag 8 is stored in the left-most bit of byte 1, . . ., flag 15 is stored in the right-most bit of byte 1. With this scheme flags 7 and 8 are stored in "adjacent" memory bits.

Note that Motorola numbers bits in the reverse order, i.e. right-to-left (even though successive bytes are numbered left-to-right). Thus, if we were to maintain consistency between bit designations and flag or discrete designations the sequence would have to be: 7, 6, 5,

4, 3, 2, 1, 0, 15, 14, 13, 12, 11, 10, 9, 8. To avoid the discontinuity between 0 and 15 we chose to be inconsistent with Motorola and consistent within ourselves.

Case 1. Set event flag 13:

```
        JSR     SVCPGM
        FDB     S9SDB1
         .
         .
         .
S9SDB1  FCB     9           (in ROM)
        FCB     $CD
```

Case 2. Reset event flag 15:

```
        JSR     SVCPGM
        FDB     S9SDB2
        BCS     WASONE      BRANCH IF FLAG WAS 1
         .
         .
         .
S9SDB2  FCB     9           (in ROM)
        FCB     $8F
```

Case 3. Test event flag whose number is given in A:

```
        STA A   S9SDB3+1
        LDA A   #9
        STA A   S9SDB3
        JSR     SVCPGM
        FDB     S9SDB3
        BCC     WASZRO      BRANCH IF FLAG WAS 0
         .
         .
         .
S9SDB3  RMB     2           (in RAM)
```

Figure 3-2 Typical Applications of SVC 9

Task A resets flags 1 and 2, and waits for either to be set:

```
        JSR   SVCPGM    RESET FLAG 1
        FDB   RSTEF1
        JSR   SVCPGM    RESET FLAG 2
        FDB   RSTEF2
        JSR   SVCPGM    WAIT FOR FLAG 1 OR 2 TO BE SET
        FDB   WAITEF
         .
         .
         .
RSTEF1  FCB   9,$81
RSTEF2  FCB   9,$82
WAITEF  FCB   10
        FDB   $6000
```

Task B sets flag 2 to continue task A:

```
        JSR   SVCPGM    SET FLAG 2
        FDB   SETEF2
         .
         .
         .
SETEF2  FCB   9,$C2
```

Figure 3-3 Task Coordination Via Event Flags

4. Coordination of Shared Subprograms

If a subprogram employs only registers or stack entries for all its arguments and temporary variables then several tasks may use the code simultaneously without interference. Tasks may invoke such reentrant subprograms freely without the aid of the operating system. However, it is not always convenient to make a subprogram reentrant.

For non-reentrant subprograms, MTOS contains a pair of SVCs which permit only one task at a time to use the code. All other tasks wishing entry are queued (on the basis of their current priority).

Subprograms used by only one task need not be controlled even if non-reentrant.

4.1 SVC for Entry To and Exit From a Controlled Subprogram

To enter a controlled subprogram a task issues an SVC with the following SDB:

```
FCB     6                           (type)
FDB     address of subprogram       (SCD address)
          control data
```

The subprogram control data (SCD) is part of the storage overhead which MTOS requires for its control function. An SCD is the direct analog of a TCD for tasks.

A controlled subprogram exits by issuing an SVC 7. The SDB has the form:

```
FCB     7                           (type)
FDB     address of subprogram       (SCD address)
          control data
```

A typical entry and exit sequence is shown in Figure 4-1.

Task RAM

```
0200 08      SUBSCB RMB   8          SUBPROGRAM CONTROL BLOCK (SCB)
```

Task ROM

```
12A0 06      USESUB FCB   6          SDB TO ENTER SUBPROGRAM
12A1 2152           FDB   SUBSCD     -POINTER TO SCD
                    .
                    .
                    .
     2152    SUBSCD EQU   *          SUBPROGRAM CONTROL DATA (SCD)
2152 C1B0           FDB   SUBENT     -ENTRY POINT
2154 0200           FDB   SUBSCB     -SCB AREA
```

Task Code

```
3204 BD E400        JSR   SVCPGM     ENTER SUBPROGRAM
3207 12A0           FDB   USESUB     -POINTER TO SDB
```

Subprogram Code/ROM

```
     C1B0   SUBENT EQU   *          SUBPROGRAM ENTRY POINT
                    .
                    .               (subprogram code)
                    .
C205 BD E400        JSR   SVCPGM     EXIT SUBPROGRAM
C208 C20A           FDB   XITSUB     -POINTER TO SDB

C20A 07      XITSUB FCB   7          SDB TO EXIT SUBPROGRAM
C20B 2152           FDB   SUBSCD     -POINTER TO SCD
```

Sequence of Program Control:

```
... 3204 C1B0 ... C205 3209 ...
    task  subprogram  task
```

Figure 4-1 Using a Controlled Subprogram

4.2 Controlled Subprogram Overhead

As with task management, control of shared subprograms requires storage overhead:

- Each controlled subprogram must have an area 8 bytes long within read/write memory in which MTOS may build a Subprogram Control Block (SCB). The SCB is maintained by MTOS as storage for current parameters such as a busy/available flag. It is not possible for several subprograms to share a common SCB.
- Each controlled subprogram must also have an entry within a table of basic Subprogram Control Data (SCD). Each SCD is exactly 4 bytes long and contains the following data:

BYTE	DATA
0,1	address of subprogram entry point
2,3	address (within read/write memory) at which an SCB may be formed

All entries must be in non-volatile memory to survive a power loss. Since SCDs contain fixed data, there is no possibility of sharing storage.

The relation between an SCD and SCB mirrors that between a TCD and TCB. The SCD is fundamental; it contains the permanent source parameters from which MTOS forms the SCB upon startup of the system. The SCB, on the other hand, contains the temporary, current information with which MTOS controls entry and exit to the subprogram.

4.3 Registers and Condition Code

When a controlled subprogram is entered, all registers (including

the stack pointer) have the values of the calling task at the time the SVC was issued. Similarly, there is continuity of register values upon returning to the calling task. The condition code, however, is cleared during both the entry and exit SVC processing.

5. Time Management

MTOS includes its own interrupt clock which periodically generates a peripheral interrupt. These interrupts drive an internal millisecond clock which is used to service time-dependent requests such as terminate with automatic future restart and pause for a given time interval.

The interrupt period is determined by a hardware clock included within the MTOS system. A typical value is 5 ms; the possible range is 1 to 255 ms. The user must select a value which is consistant with his application.

If a larger value is chosen (say 100 ms), then the internal clock would have a "granularity" of 100 ms. The granularity is the smallest interval which can be seen by the system. In other words, the internal clock would remain at the same value for 100 ms and then be incremented by 100 ms. As a result, if a pause of 1 to 99 ms were requested the actual pause would be 100 ms since it takes that long to see any change in time.

If a small value is chosen (say 1 ms) then the overhead in servicing the clock interrupts can become appreciable. This reduces the time available for task work.

The proper interrupt period is generally obvious from the time requirements of the application. Thus, if all pauses and other time-dependent servicing occur in multiples of 5 ms, the period would be set to this lowest common denominator.

The time kept by MTOS is purely internal and does not bear a fixed relation to the external, real-world time. Intervals are

correct: requesting a pause of 10 ms with respect to the internal clock causes a 10 ms pause with respect to a real-world clock. However, the base of the internal clock (the "time zero" point) shifts occasionally.

The necessity for base shifting arises from trying to maintain a perpetual clock in a finite (and preferably small) number of memory words. If the clock were simply a tally on the number of milliseconds since system startup (or some other fixed reference point) then eventually the tally would overflow. MTOS solves the problem by periodically decrementing both the internal clock and all internal references to that clock by a fixed amount.

A task may make any of the following time-related requests:

SVC 1 - Terminate the task with automatic restart after a given time interval

SVC 2 - Pause for given time interval

SVC 3 - Cancel pause of given task

SVC 14 - Pause for minimum time interval

SVC 11 - Set event flag after given interval

which is used in conjunction with:

SVC 10 - Wait until event flags are set

SVC 1 was described in Section 3.3.4; the others are described in the following subsections:

5.1 Pause for Given Time Interval

The SDB to specify a pause is:

```
FCB    2
FCB    time interval--
         0 = ms
         1 = 10 ms
         2 = seconds
         3 = minutes
```

4 = hours
FCB number of time units (0 to 255)

Certain intervals can be composed in several ways. For example, 2 seconds can also be expressed as 200 10-ms units. There is no difference in processing efficiency.

Specifying a time interval code greater than 4 is invalid. In such cases, the task will continue without pause and the Overflow bit of the condition code will be set. For a valid interval the Overflow bit is reset.

5.2 Cancel Pause for Given Task

Cancel pause can be used to coordinate task activity: One task issues a pause for an arbitrary, long time (say 24 hours); another task cancels that pause when it wants the waiting task to continue (see Figure 5-1).

Cancel pause can also be used to have a task wait for some event with timeout in case the event never occurs: One task issues a pause for the desired timeout interval; another task monitors the event and cancels the pause when the event is detected. If the event never occurs, the task automatically continues after the default period.

The SDB for the cancel pause SVC has the form:

```
FCB    3       (type)
FDB    task    (TCD address)
```

If the target task is not in the pause state when the SVC is issued, the Zero bit of the condition code is set to 1 to indicate that the SVC had no effect.

Task A pauses for an arbitrary long period (24 hours):

```
        JSR     SVCPGM          PAUSE FOR 24 HOURS
        FDB     PA24HR
          .
          .
          .
PA24HR  FCB     2,4,24
```

Task B cancels pause to continue A:

```
        JSR     SVCPGM          CONTINUE TASK A
        FDB     CPTSKA
          .
          .
          .
CPTSKA  FCB     3
        FDB     TKATCD          =TCD FOR TASK A
```

Figure 5-1 Task Coordination Via Pause Cancellation

5.3 Set Event Flag After Given Interval

SVC 11 - Set Event Flag After Given Interval permits a task to use an event flag as an internal alarm clock timer. At some point within a task, the timer may be set by issuing SVC 11 (see Figure 5-2). The task then continues with processing that may take a variable amount of time. When that processing is completed, the task issues an SVC 10 - Wait Unitl Event Flags Are Set referencing the same event flag as was specified in the SVC 11. MTOS then holds the task until the remainder of the original time interval runs out. This mechanism permits a task to initiate action after a predetermined time interval, and yet to use part of that wait interval for further processing.

The SDB for SVC 11 is composed as:

```
FCB    11                      (type)
FCB    time interval--
         0 = ms
         1 = 10 ms
         2 = seconds
         3 = minutes
         4 = hours
FCB    number of time          (0 to 255)
         units
FCB    event flag number       (0 to 7)
```

The event flag number is limited to the first eight (0 to 7). If either limit is exceeded, the Overflow bit within the task's condition code is set to 1 and the event flag is not altered.

The specified event flag is automatically reset by the SVC 11 and then set after the given interval. The flag may also be set early by an independent SVC 9. (This does not stop the SVC 11 "countdown".)

If a new (valid) SVC 11 is issued for the same event flag while

```
        JSR     SVCPGM      SET EVENT FLAG 2
        FDB     SF30MS        30 MS FROM NOW
          .
          .
          .
        any     code        CONTINUE CALCULATIONS
          .
          .
          .
        JSR     SVCPGM      WAIT UNTIL BALANCE OF
        FDB     WAITEF        30 MS IS UP
          .
          .
          .
SF30MS  FCB     11,0,30,2
WAITEF  FCB     10
        FDB     $2000
```

Figure 5-2 Setting an "Alarm Clock"

a previous one is in progress the original request is cancelled and the new one takes control. By convention if the number of time units is 0 (with a valid time interval specified) the flag is reset, but no set-flag countdown is started. Thus, a valid zero-interval SVC 11 can be used to cancel a previous SVC 11 for a given event flag.

5.4 Synchronization for Exact Time Intervals

It is sometimes necessary to have two events, such as the output of two signals ("A" and "B"), separated by a given interval. A straightforward method to achieve this might be: output "A", pause for required interval, output "B". However, because of the finite granularity of the clock, the pause interval is usually shorter than expected. (Since on the average half of the current clock period is already over when a pause is issued, the average pause is half a clock period shorter than requested.)

When precise intervals are required, it is best to first synchronize to the start of a clock period before the first event. Synchronization is achieved by issuing a pause for minimum time interval (see Section 5.5). The sequence for precise intervals would then be: pause for minimum interval, output "A", pause for required interval, output "B".

5.5 Pause for Minimum Time Interval

In a typical real-time system, there is at least one task which runs at the maximum rate. For example, a task which samples input data for changes is often activated each time the internal clock generates an interrupt.

A convenient method to structure such a task is to have an initialization section (which is entered just once) followed by a cyclic section. The cyclic section ends with a pause for a minimum interval and a branch back to itself:

```
ENTRY:
    initialization section code
LOOP:
    cyclic section code
    pause for miminum interval
    jump to LOOP
```

One method to request the required minimum pause is via SVC 2, with 1 ms. specified as the interval. Because of the finite granularity of the internal time processing a 1 ms. pause is automatically cancelled at the next clock interrupt (for any value of the interrupt interval).

System overhead can be reduced significantly, however, by using a special SVC which bypasses the normal pause processing but still achieves a pause until the next clock "tick". The SDB to specify a pause for minimum interval is:

```
FCB 14
```

Since there are no options, there are no parameters.

6. Input and Output of Discretes

Most systems are expected to have a variety of external, single-bit discrete signals. Some will be input, others will be output. Typical inputs reflect the state of a button or a switch; typical outputs control the state of an indicator, motor drive, or mechanical actuater.

The discretes are stored as individual bits, within one or more bytes. A given byte could contain all inputs, all outputs, or a mixture of inputs, outputs and unused bits. Furthermore, the bytes dedicated to discretes need not be consecutive.

The storage medium could be the data section of a PIA or could be a bipolar latch. If any of the discretes are implemented on PIAs, it is the user's responsibility to initialize the control sections upon system startup.

Because tasks run asynchronously with random interrupts, the output of discrete values must be controlled by MTOS. Consider, for example, the following situation: task "A" wishes to set the first discrete to 0. It attempts this by loading the first discretes byte, ANDing with hexadecimal 7F (which sets the first discrete to 0 and leaves the others unchanged), and then storing the modified value back in the first discretes byte. In most cases this would give the desired effect. However, suppose task "A" were interrupted, just before the store and a higher priority task, "B", were to become active. Suppose further that

discretes and then terminate. Task "A" then continues, completes its store and by this act cancels any changes made in the interim by task "B". To avoid such problems all output of discretes should be done via SVC 8.

SVC 8 can be used to reset, set or test the value of a given discrete. Reset or set is equivalent to output; testing is equivalent to input. While input can be accomplished directly (by a load and mask) without causing the problem described above, it is sometimes more convenient to use the SVC.

Note that the discretes of interest in this section are external signals. Thus, they are a completely different set of values from the internal event flags which were discussed in Section 3.3.6.

The SDB for the discretes SVC has the form:

```
FCB   8                          (type)
FCB   parameter 1                (options and control)
FDB   address of discrete byte
```

Parameter 1 contains the function:

```
bits 7-6:  00 or 01 = test flag
           10 = reset flag
           11 = set flag
     5-3:  not used
     2-0:  address of bit within byte (0=bit 7, ..., 7=bit 0)
```

For a valid index (i.e. an index value less than the total number of discretes) the Overflow bit of the condition code is reset; for any larger index the Overflow is set. Furthermore, for a valid index the value of the discrete at the time the SVC is issued is returned in the Carry bit. For the test function, no other actions are taken. For reset or set, the discrete is forced to 1 or 0 respectively; and if this represents no change in the value of the discrete, the Zero bit in the condition code is also turned on.

7. Input and Output of Console Messages

MTOS supports a system Console by supplying the machinery for reading and writing text messages to a teletype or teletype-compatible device. The device is interfaced through an ACIA. The functions provided are: requested output, requested input, and unrequested (unsolicited) input. Both inputs have certain line editing capabilities.

7.1 Requested Output

SVC 13 is used to output a message on the Console. The SDB is:

```
FCB    13                                  (type)
FCB    message buffer length (bytes)       (1 to 255)
         with 0 taken as 1
FDB    address of message buffer           (in RAM or ROM)
```

The message buffer is assumed to contain ASCII text and control data. The message must be fully formatted, i.e. any desired carriage returns and line feeds must exist within the text. MTOS does not supply any line control characters on its own. The buffer may be within read-write or read-only memory. Some examples of message output are shown in Figure 7-1.

7.2 Requested Input

To request input SVC 12 is issued. The SDB is similar to that for output:

```
FCB    12                                  (type)
FCB    input buffer length (bytes)         (1 to 255)
         with 0 taken as 1
FDB    address of input buffer             (in RAM)
```

To begin processing MTOS outputs a prompt string consisting of a question mark followed by a blank. This signals the operator that

Message to be output:

(CR)(LF)PROCESSED SECTION 4 SUCCESSFULLY

part 1 2 4

code:

```
                JSR     SVCPGM          OUTPUT HEADER
                FDB     PART1
                LDA  A  SECTNO          FORM AND OUTPUT
                ADD  A  #$30              SECTION NUMBER
                STA  A  TEMP
                JSR     SVCPGM
                FDB     PART2
                TST     FAILFG          OUTPUT 'NOT' IF
                BEQ     OUTPT4            UNSUCCESSFUL
                JSR     SVCPGM
                FDB     PART3
OUTPT4          JSR     SVCPGM          OUTPUT END OF MESSAGE
                FDB     PART4
```

data:

```
                .
                .
                .
PART1           FCB     13,20
                FDB     *+2
                FCB     $0D,$0A
                FCC     /PROCESSED SECTION /
PART2           FCB     13,1
                FDB     TEMP
PART3           FCB     13,4
                FDB     *+2
                FCC     / NOT/
PART4           FCB     13,13
                FDB     *+2
                FCC     / SUCCESSFULLY/
```

Figure 7-1 Console Output

input has been requested. MTOS will then fill the buffer area with edited input text up to and including the mandatory carriage return which marks the end of the message (see next section).

If more characters are entered from the Console than will fit in the buffer, the excess is discarded. In these cases, the Overflow bit of the condition code is set and there will always be a carriage return at the end of the buffer.

The operator must not wait more than 5 minutes before entering the first (or next) character. After 5 minutes without input the Console times out and a carriage return is automatically entered to close out the input message.

7.3 Input Text Editing

All input text lines, whether requested or not, can be edited as follows:

character	ASCII, hex.	function	output response
backarrow (shift O)	5F	delete last character, if any	none
rubout	7F	delete current line	CR/LF/?/blank
carriage return (CR)	OD	end of input	CR/LF
line feed (LF)	OA	end of line, but not end of input	CR/LF/blank

The first two permit character-by-character and entire line deletion. A succession of N backarrows deletes N characters (or fewer if N characters have not yet been input).

Input must be terminated by a carriage return. Once the CR is given, the line is no longer available for editing. The CR always appears in the input buffer.

The line feed may be used to return the carriage physically without ending the text. The LF is not stored in the input buffer.

7.4 Unrequested Text Messages

Unsolicited text may be entered at the Console. To initiate the process an "escape" character is entered. Assuming no solicited input or output activity is pending, MTOS responds by outputting the '?' prompt string. The operator may then input up to 72 edited characters including the final carriage return. When the CR is received a user-written Unsolicited Console Message Processor task will be activated and the address of the message will be passed as an argument. The analysis of the message and any subsequent actions are left completely to the user. However, until the Message Processor goes dormant another escape character will not be recognized.

The TCD for the Console Unsolicited Message Processor must be the third entry in the table of user TCDs (see Appendix A).

8. Peripheral Interrupts

Peripheral interrupts from any device other than the internal clock and initialize button must be serviced by the user. When any peripheral interrupt occurs, MTOS suspends the current task and then determines if the interrupt is internal or external to MTOS. If external, MTOS jumps to a user subprogram whose address is supplied within a common interface block (see Section 9). The user subprogram must service the interrupt and then return to MTOS via an RTS instruction.

The user peripheral processing must not re-enable interrupts.

9. Memory Allocation

9.1 MTOS Memory Map

The standard version of MTOS resides within hexadecimal addresses E000 to EFFF. Of these, the first 256 bytes are used as an internal scratchpad and the next 768 are not used (except for an interrupt latch at E2F0-E2F1). The program itself occupies the final 3K bytes.

9.2 MTOS/User Interface

MTOS requires certain information about the user's system to be supplied with a fixed interface area. The data must be both non-volatile and immediately available upon startup or initialization. The block is 32 bytes long and immediately precedes MTOS. The format is:

location	length	information
DFD0	2	address of user TCD table
DFD2	1	number of user tasks
DFD3	2	address of user SCD table
DFD5	1	number of user controlled subprograms
DFD6	1	real-time clock period (ms)
DFD7	2	address of Console ACIA data section (FCF5 for EXORciser)
DFD9	2	address of Console ACIA control section (FCF4 for EXORciser)
DFDB	1	Console ACIA initial control data (see Table 9-1)
DFDC	1	Console delay after carriage return (clock pulses, see Table 9-1)
DFDD	1	Console delay after other character (clock pulses, see Table 9-1)
DFDE	2	address of user peripheral interrupt processing subprogram
DFE0	16	reserved for future expansion (must be 0)

Typical values for the Console data are shown in Table 9-1.

In addition, the following properties of MTOS may be needed by the user:

```
SVCPGM   EQU   $E400   address of SVC subprogram
IRQINT   EQU   $E403   address of IRQ interrupt processing
RSTINT   EQU   $E406   address of reset (power on) interrupt
                         processing
```

The SWI and NMI interrupts are not used by MTOS (since these functions must be dedicated to the EXORciser).

The following procedure may be used to start MTOS-68 on an EXORciser:

1. Load application program.
2. Using MAID, change the top of memory address from 83FF to EFFF:

```
MAID
*FF00/83 EF (CR)
```

3. Depress "ABORT" button (to transfer interrupt vector).
4. Using MAID, start processing at E406:

```
MAID.
*E406;G
```

Console	Use	Initial Control Data	Delay After Carriage Return	Delay After Other Character
TI Silent 700	300 BAUD for printing, 1200 BAUD for tape	09	190 ms*	35 ms*

* To obtain number of clock pulses, divide by clock period. For example, for a 5 ms clock rate, the delay is 190/5 = 38 clock pulses.

Table 9-1 Typical Console Data

Appendix A: Task Storage Overhead

MTOS requires storage in both ROM and RAM in order to perform CPU management. This storage overhead is used for (1) task stacks, (2) task control data, and (3) task control blocks.

Each task must have a stack with 9 bytes always available for interrupt processing. When an interrupt is serviced the state of the machine is automatically pushed onto the stack; when the task resumes the state is automatically popped from the stack. Thus, the intermediate internal use of the stack will not be apparent to the task provided that the stack does not overflow.

Individual stacks may be placed anywhere in RAM; they need not be consecutive entries in a table. Furthermore, two (or more) tasks can share a common stack provided they can never be running at the same time. Running, here, means non-dormant.

Each task must also have an entry within a table of basic Task Control Data (TCD). Each TCD is exactly 7 bytes long and contains the following data:

BYTE	DATA
0	inherent (default) priority (0 to 15)
1,2	initial stack pointer address
3,4	address of program entry point
5,6	address (within RAM) at which a TCB may be formed

All entries must be in non-volatile memory so as to survive a power loss. Since TCDs contain fixed data, there is no possibility of sharing, even between mutually exclusive tasks.

All TCDs must be stored together within a user TCD table. Further-

more, the first three entries must be for the tasks that respond to the (1) power on (restart), (2) initialization button depressed, and (3) unsolicited Console input entered conditions respectively. Thereafter the TCDs may appear in any order.

Finally, each task must designate an area within RAM in which MTOS can build and maintain a Task Control Block (TCB). This block is 18 bytes long. In principle, TCB areas can be shared by mutually exclusive tasks. However, the conditions under which such sharing is permitted are both stringent and difficult to state. Thus, in most applications TCB areas should be dedicated to individual tasks. Furthermore, a TCB area should not be altered by any of the application programs even if the associated task is known to be dormant.

A memory allocation for a typical set of TCDs, TCBs and stacks is shown in Figure A-1. Except for the restrictions discussed above, the arrangement and labeling shown in the figure is arbitrary.

For a given task the TCD, TCB and stack are all closely related. The TCD is fundamental; it contains the permanent source parameters from which MTOS forms the TCB upon startup of the task. In contrast, the TCB contains the temporary, current information with which MTOS controls the task while it is running. The stack is used both by the task program (to store subprogram return addresses and temporary data) and by MTOS (to store the state of the machine during interrupt processing). The TCD supplies both the initial value of the stack pointer and the initial contents of the stacked program counter (program entry point).

```
* USER ROM

     3200  USTCDT EQU   *        USER TCD TABLE:

     3200  RSTTCD EQU   *        RESET (POWER ON) TASK
3200 0F            FCB   15       -PRIORITY
3201 0C66          FDB   RSTACK   -STACK
3203 325F          FDB   RSTENT   -ENTRY POINT
3205 0C00          FDB   RSTTCB   -TCB AREA

     3207  INITCD EQU   *        INITIALIZATION TASK
3207 0F            FCB   15       -PRIORITY
3208 0C66          FDB   RSTACK   -STACK
320A 3281          FDB   INIENT   -ENTRY POINT
320C 0C12          FDB   INITCB   -TCB AREA

     320E  CNUTCD EQU   *        CONSOLE UNSOLICITED INPUT TASK
320E 08            FCB   8        -PRIORITY
320F 0C73          FDB   CNUSTK   -STACK
3211 3305          FDB   CNUENT   -ENTRY POINT
3213 0C24          FDB   CNUTCB   -TCB AREA
                     .
                     .
                     .

* USER RAM

0C00 0012  RSTTCB RMB   18       TCB FOR RESET TASK
0C12 0012  INITCB RMB   18       TCB FOR INITIALIZATION TASK
0C24 0012  CNUTCB RMB   18       TCB FOR CNS UNSOLICITED INPUT TASK
                     .
                     .
                     .
0C5A 0C            RMB   24       STACK FOR RESET AND INIT TASKS
0C66 00    RSTACK  EQU   *-1

0C67 0C            RMB   24       STACK FOR CNS UNSOLICITED INPUT TASK
0C73 00    CNUSTK  EQU   *-1
                     .
                     .
                     .
```

Figure A-1 Typical TCD, TCB and Stack Allocation

Appendix B: Summary of SVC Data Blocks

The data block required for each of the 15 SVCs within MTOS is given on the following pages. The SVCs are:

0. terminate the issuing task without automatic restart
1. terminate the issuing task with automatic restart after a given time interval
2. pause for a given time interval
3. cancel the pause of a given task
4. start a given task if it is dormant
5. start a given task, queuing the request if the task is busy
6. enter a controlled shared subprogram
7. exit from a controlled shared subprogram
8. set/reset/test an external discrete
9. set/reset/test an internal event flag
10. wait until event flags are set
11. set an event flag after a given interval
12. read a message from the Console
13. write a message on the Console
14. pause for minimum time interval

SVC 0 - Terminate Issuing Task Without Automatic Restart

```
FCB     0       (type)
```

SVC 1 - Terminate Issuing Task With Automatic Restart After Given Time Interval

FCB	1	(type)
FCB	parameter 1	(option and control)
FCB	number of time units	(0 to 255)

Parameter 1

bit 7: 0 = base restart on last scheduled start time

1 = base restart on current (termination time)

6-3: unused

2-0: time interval--
0 = ms
1 = 10 ms
2 = seconds
3 = minutes
4 = hours

Condition Code

Overflow = 1 if time interval code is greater than 4

= 0 if above error condition does not exist

SVC 2 - Pause for Given Time Interval

FCB	2	(type)
FCB	time interval	(see below)
FCB	number of time units	(0 to 255)

Time Interval

bits 7-3: unused

2-0: time interval--
- 0 = ms
- 1 = 10 ms
- 2 = seconds
- 3 = minutes
- 4 = hours

Condition Code

Overflow = 1 if time interval code is greater than 4

= 0 if above error condition does not exist

SVC 3 - Cancel Pause of Given Task

```
FCB     3       (type)
FDB     task    (TCD address)
```

Condition Code

Zero: 0 = if task was originally in pause state

1 = if task was not originally in pause state

SVC 4 - Start Given Task if Dormant

FCB	4	(type)
FDB	address of TCD for task to be started	(TCD address)

Condition Code

Zero: 0 = if task was originally dormant

1 = if task was already running

SVC 5 - Queue Start of Given Task

FCB	5	(type)
FCB	parameter 1	(options and control)
FDB	address of TCD for task to be started	(target TCD)
FDB	address of 7 byte RAM scratch area in which to build queue block	(RCB address)

Parameter 1

bit 7: 0 = no argument to be transmitted (X register of target task will be set to 0 upon startup)

1 = transfer value currently in X register of requesting task to X register of target task as a startup argument

6-5: 00 = queue start request and continue without wait (no coordination)

01 = queue start request and wait until the task starts because of this request (concurrent running)

10 = queue start request and wait until the task terminates because of this request (assured completion)

11 = (illegal combination)

4: 0 = use larger of inherent priority of called task and current priority of calling task

1 = use priority given in bits 3-0

3-0: startup priority (value must be given only if bit 4 is 1)

SVC 6 - Enter Controlled Shared Subprogram

FCB	6	(type)
FDB	address of subprogram control data	(SCD address)

SVC 7 - Exit From Controlled Shared Subprogram

FCB	7	(type)
FDB	address of subprogram control data	(SCD address)

SVC 8 - Set/Reset/Test External Discrete (discretes input/output)

```
FCB     8                           (type)
FCB     parameter 1                 (options and control)
FDB     address of discrete byte
```

Parameter 1

bits 7-6:
- 00, 01 = test (input) discrete
- 10 = reset discrete (output 0)
- 11 = set discrete (output 1)

5-3: not used

2-0: address of bit within byte (0=bit 7, ..., 7=bit 0)

Condition Code

Carry = 1 if discrete was set at the time the SVC was issued

= 0 if discrete was reset at the time the SVC was issued

Zero = 1 if reset or set caused no change in discrete value

= 0 if reset or set caused a change in the discrete value, or if the function was test

SVC 9 - Set/Reset/Test Internal Event Flag

```
FCB     9               (type)
FCB     parameter 1     (options and control)
```

Parameter 1

bits 7-6: 00, 01 = test flag

10 = reset flag

11 = set flag

5-4: not used

3-0: event flag number (0 to 15)

Condition Code

Carry = 1 if flag was set at the time the SVC was issued

= 0 if flag was reset at the time the SVC was issued

Zero = 1 if reset or set caused no change in flag value

= 0 if reset or set caused a change in the flag value, or if the function was test

SVC 10 - Wait Until Event Flags Are Set

FBC	10	(type)
FDB	parameter 1	(mask)

Parameter 1

byte 1,	bit	7	continue	if	Flag	0	is	set, or
"	"	6	"	"	"	1	"	"
"	"	5	"	"	"	2	"	"
"	"	4	"	"	"	3	"	"
"	"	3	"	"	"	4	"	"
"	"	2	"	"	"	5	"	"
"	"	1	"	"	"	6	"	"
"	"	0	"	"	"	7	"	
byte 2,	bit	7	continue	if	Flag	8	is	set, or
"	"	6	"	"	"	9	"	"
"	"	5	"	"	"	10	"	"
"	"	4	"	"	"	11	"	"
"	"	3	"	"	"	12	"	"
"	"	2	"	"	"	13	"	"
"	"	1	"	"	"	14	"	"
"	"	0	"	"	"	15	"	

Note: Parameter 1 may be identically 0, but then the task will never continue.

SVC 11 - Set Event Flag After Given Interval

FCB	11	(type)
FCB	time interval	(see below)
FCB	number of time units	(0 to 255)
FCB	event flag number	(0 to 7*)

* Only the first 8 event flags may be used for this purpose.

Time Interval

0 = ms
1 = 10 ms
2 = seconds
3 = minutes
4 = hours

Condition Code

Overflow = 1 if event flag is greater than 7, or time interval code is greater than 4

= 0 if none of above error conditions exists

SVC 12 - Input a Message From the Console

```
FCB     12                                  (type)
FCB     input buffer length (bytes)         (1 to 255)
          with 0 taken as 1
FDB     address of input buffer
```

Condition Code

Overflow = 1 if excess input was discarded

= 0 if input fit within buffer

SVC 13 - Output a Message to the Console

FCB	13	(type)
FCB	message buffer length (bytes) with 0 taken as 1	(1 to 255)
FDB	address of message buffer	

SVC 14 - Pause for Minimum Time Interval

```
FCB     14      (type)
```