



CHAPTER SEVEN

# Multi-Tasking

## IN THIS CHAPTER

- |   |     |
|---|-----|
| <b>IN THIS CHAPTER</b>                        |     |
| • Introduction to Multi-Tasking .....         | 204 |
| • Using 6K Resources While Multi-Tasking..... | 213 |
| • Multi-Tasking Performance Issues .....      | 217 |
| • Multi-Tasking Application Examples .....    | 219 |

# Introduction to Multi-Tasking

**What is Multi-Tasking?** Multi-tasking is the ability of the 6K to run more than one program at the same time. This allows users to manage independent sets of axes for multiple machines or unrelated parts of a single machine.

## Why use Multi-Tasking?

- Multiple independent *tasks* can act on the same process or same set of axes.
- A supervisory program can control and coordinate multiple processes or axes sets.

**What is a Task?** A system task is a program execution environment, not a program. Each task runs independently. Each task can be used to run independent programs. Any task can run any program. Multiple tasks could even be running the same program simultaneously. Each task can either be:

- Running a program
- Monitoring ON, ERROR, or INSELP conditions
- Inactive (**not** running a program or monitoring conditions)

## Using Multi-Tasking to Run Programs

### Use the Wizard

Motion Planner provides a Multi-Tasking wizard, accessed from the Editor. The wizard leads you, step by step, through the process of setting up for multi-tasking.

Multi-tasking is initiated by the *Task Supervisor*. The supervisor is the 6K controller's main program execution environment — it runs the STARTP program and contains the command buffer and parser. If the multi-tasking feature is not being used, the supervisor is the lone program execution environment of the 6K controller, as shown in Figure 1. When multi-tasking is in operation, the supervisor is referred to as "Task 0".

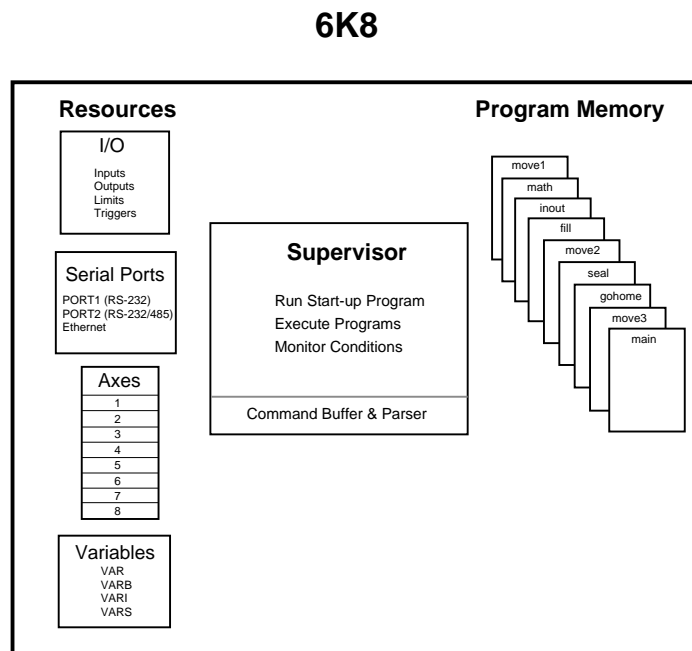


Figure 1: 6K Supervisor, with no Multi-Tasking

The 6K can have a maximum of 10 tasks, in addition to the Task Supervisor (Task 0). Each task acts as an individual program execution environment. The boxes shown under Resources in Fig. 1, and the following Figures, represent the resources that the supervisor and tasks

monitor and manipulate (see list below). These resources can be shared with other tasks.

- I/O..... Inputs and Outputs (onboard I/O and expansion I/O bricks)
- Communication ports ... “RS-232”, “RS-232/485”, and “ETHERNET”
- Axes..... Axes available with the 6K
- Variables ..... Real (VAR), binary (VARB), integer (VARI), and string (VARS)

“Program Memory” is the 6K’s non-volatile memory where programs are stored. Any task can run any program. Multiple tasks could even be running the same program simultaneously. Programs in multi-tasking are defined as they typically are with the 6K

### 6K8

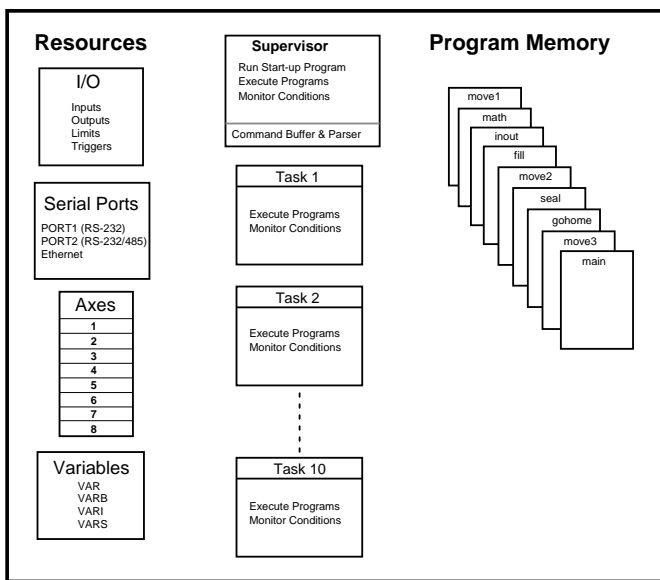


Figure 2: 6K8 - Multi-Tasking with 10 tasks.

The Task Identifier (%) prefix is used to specify that the associated command will **affect** the indicated task number. For most simple multi-tasking applications, the % prefix will only be used to start a program running in a specific task. Multi-tasking programs can be started through the communication ports (see Figs. 3a-3c), or from a program (see Figs. 4a-4d). Note that the programs are run in the tasks specified with the % prefix.

#### Starting Tasks from a Communications Port

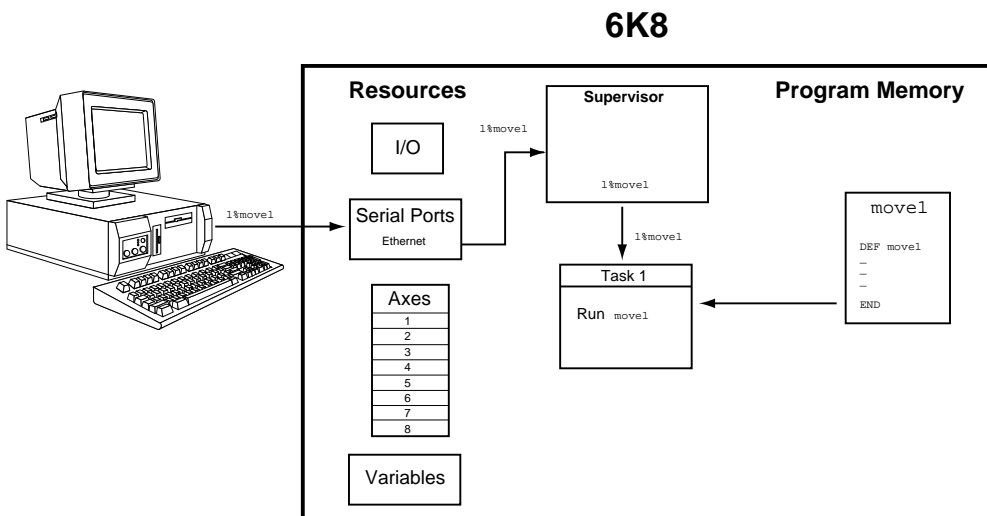


Figure 3a: Multi-Tasking initiated from a computer terminal.

## 6K8

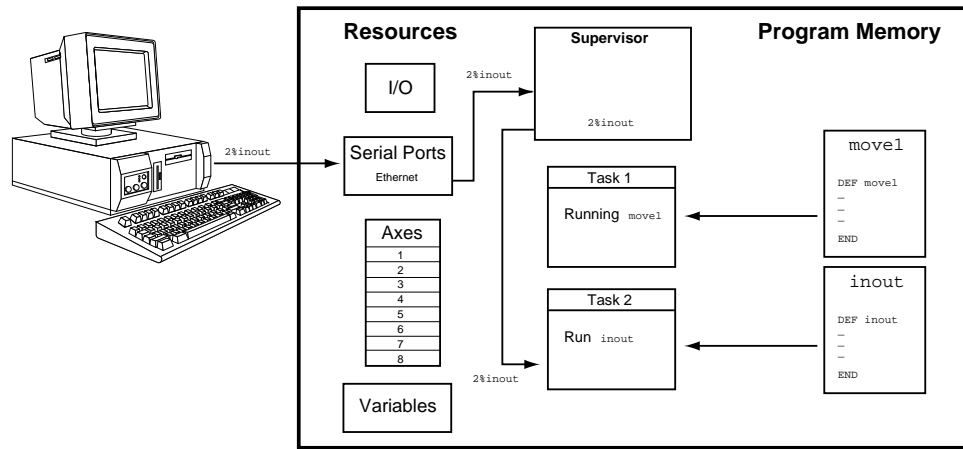


Figure 3b: Multi-Tasking initiated from a computer terminal (cont'd).

## 6K8

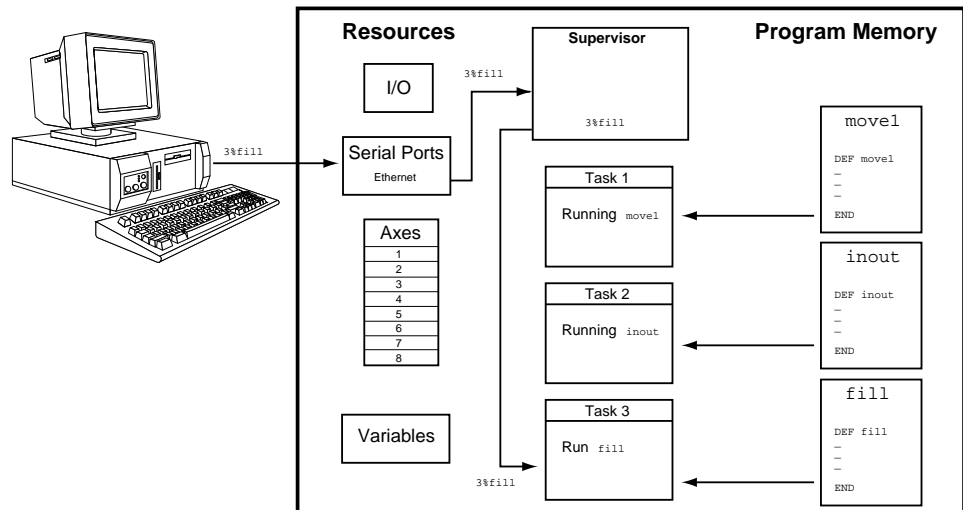


Figure 3c: Multi-Tasking initiated from a computer terminal (cont'd).

### Starting Tasks from a Program

## 6K8

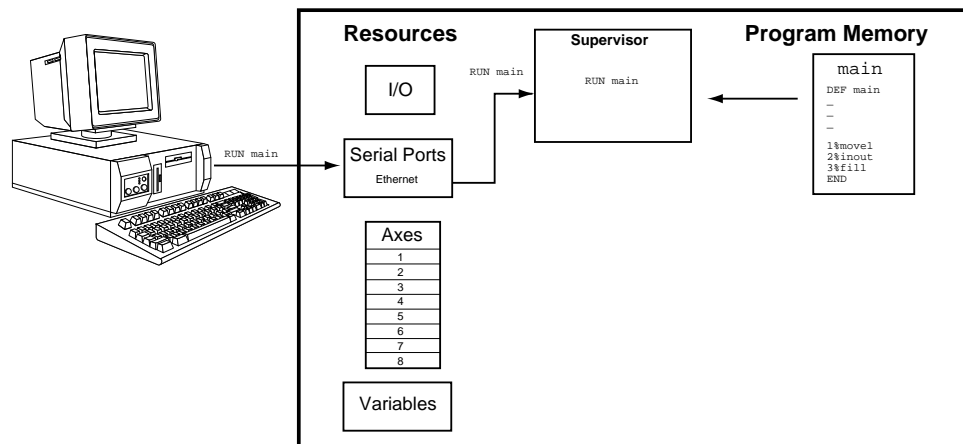


Figure 4a: Multi-Tasking initiated from a program.

## 6K8

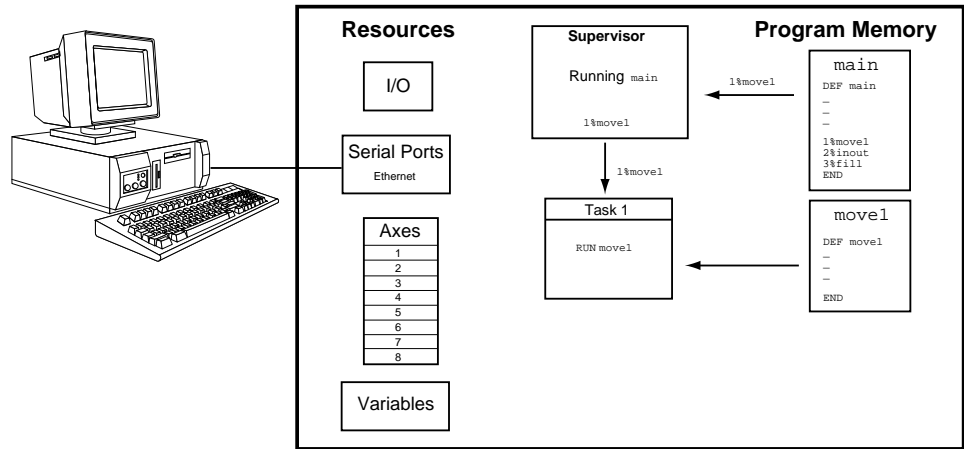


Figure 4b: Multi-Tasking initiated from a program (cont'd).

## 6K8

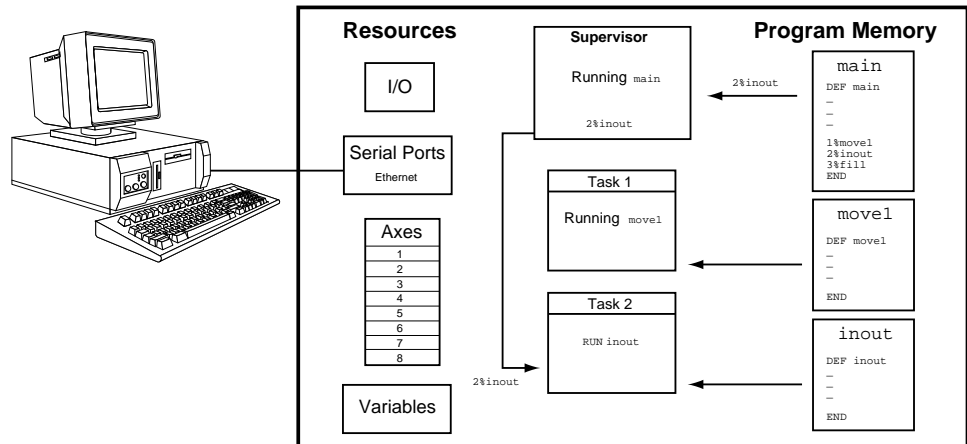


Figure 4c: Multi-Tasking initiated from a program (cont'd).

## 6K8

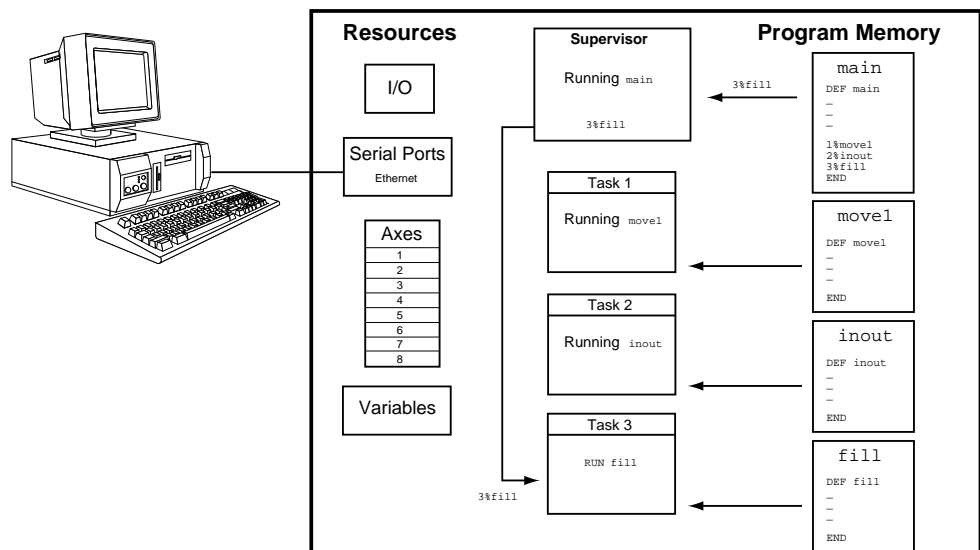


Figure 4d: Multi-Tasking initiated from a program (cont'd).

## Interaction Between Tasks

A new task is initiated by identifying the task number with the Task Identifier (%) prefix, followed by the name of the program to be run in the specified task. Because the % prefix specifies the task number that the associated command will **affect**, new tasks can be started from within other tasks. Fig. 5b shows the `move1` program in task 1 being started by the main program in the supervisor. Fig. 5c shows the `fill` program in task 3 being started by the `move1` program in task 1 with the `3%fill` command. Fig. 5d shows the `inout` program in task 2 being started by the `fill` program in task 3 with the `2%inout` command.

Program execution in a task is controlled by commands executed from the program the task is running. In the example shown in Figs 5a-5d, program execution in Task 1 is controlled by commands executed from the `move1` program, program execution in Task 2 is controlled by commands executed from the `inout` program, and program execution in Task 3 is controlled by commands executed from the `fill` program.

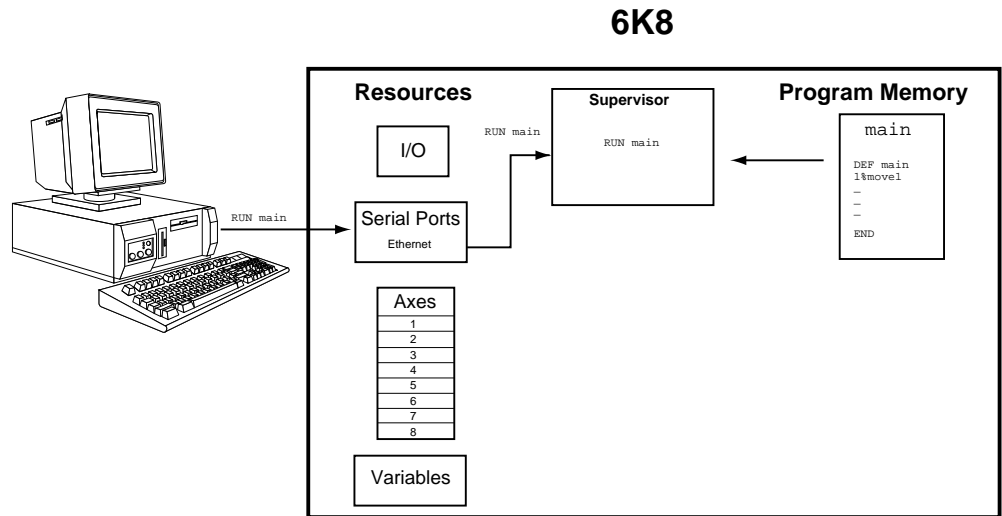


Figure 5a: Initiating multi-tasking.

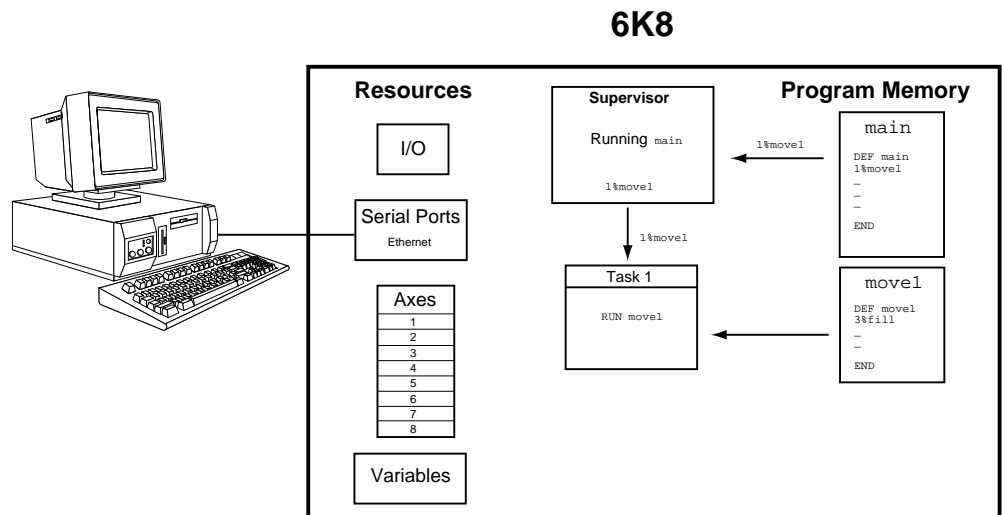


Figure 5b: Task initiated from another task (cont'd).

## 6K8

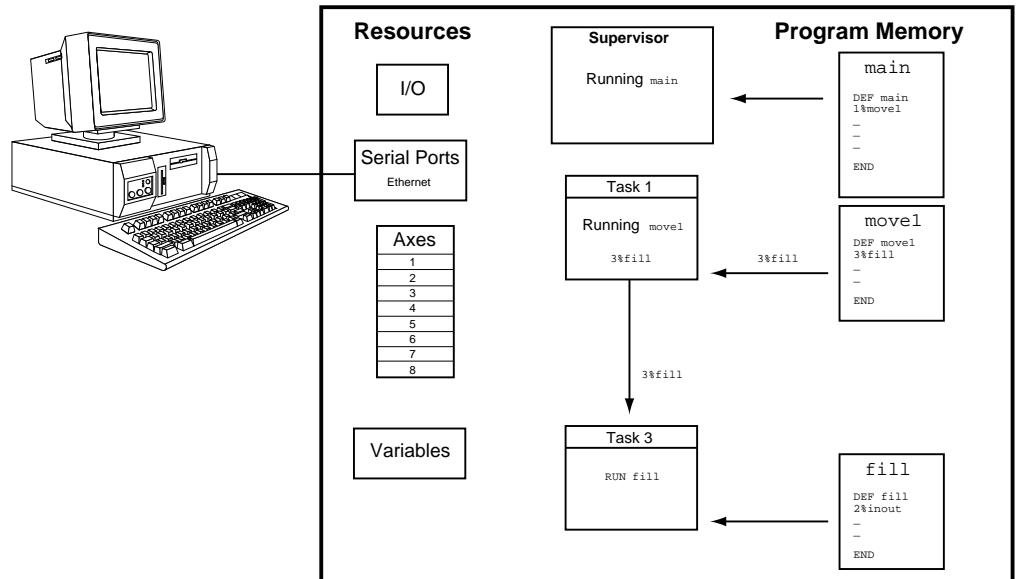


Figure 5c: Task initiated from another task (cont'd).

## 6K8

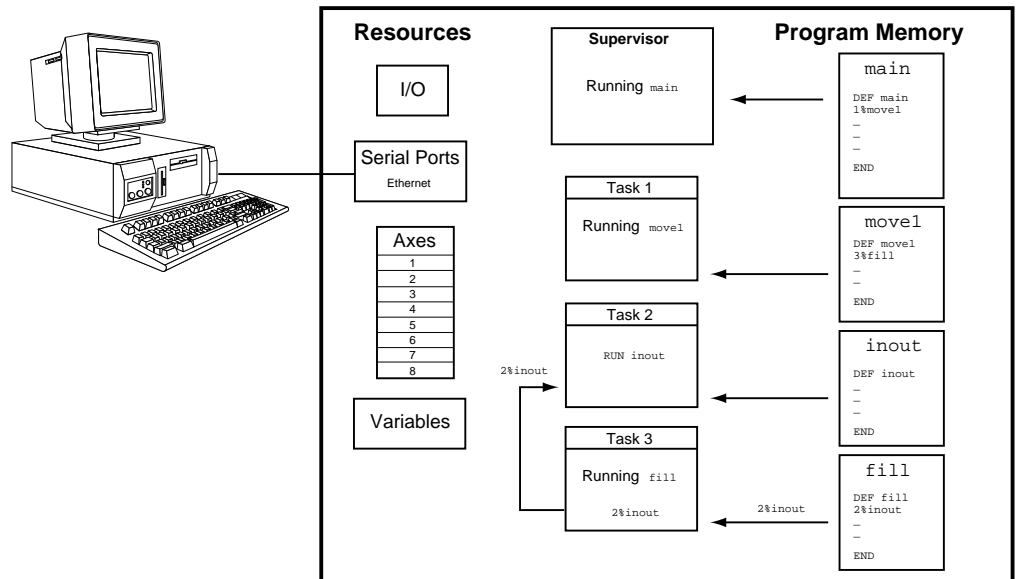


Figure 5d: Task initiated from another task (cont'd).

## Axes & Tasks

Refer to page 213 for more details about associating axes with tasks.

The default condition in multi-tasking is that each task is associated with all controller axes, as shown in Fig. 6a. This means that when an axis being used in a task hits an end-of-travel limit, program execution will be killed within that task, and in all other tasks, because they all share that axis. It can therefore be necessary to assign a set of axes to a given task to allow a multi-axis controller to be used as more than one independent program execution environment.

The TSKAX command allows you to assign axes to specific tasks, thus constraining task response and control to a smaller set of axes. A task is allowed to control only its associated axes. This axis association covers all interaction between axes commands, conditions or inputs and task program flow. For example, if a 6K controller is controlling two independent machines that do not share common axes, with control of each machine as a separate task, a limit hit by an axis in one machine can kill the task running that machine, but will not kill the task running the other machine.

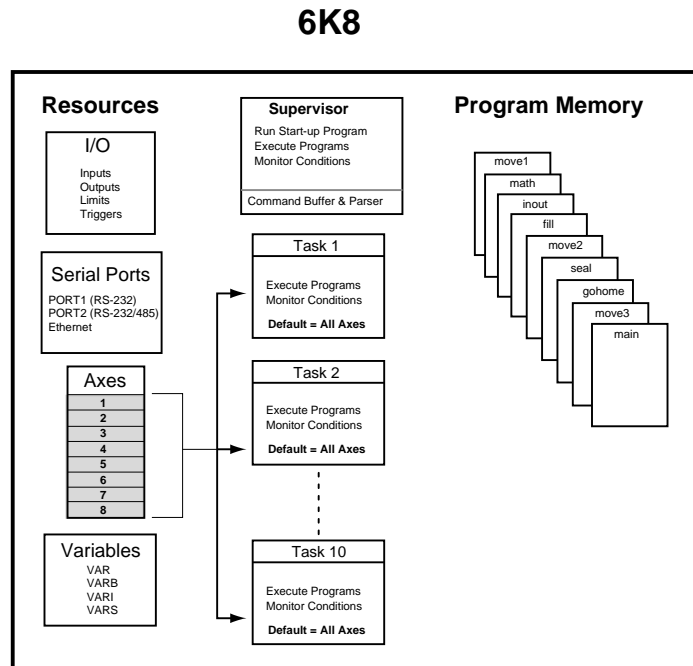


Figure 6a: Each task is associated with all controller axes in default condition.

The TSKAX command syntax identifies the first and last axis numbers (range) associated with the task. Thus, the axes associated with a task will always be consecutive. For example, the following commands will associate axes 1-3 with Task 1, axes 4-6 with Task 2, and axes 7 and 8 with Task 3. These associations are illustrated in Figures 6b-6d below. If axis 3 in task 1 hits a limit, program execution in task 1 will be killed, but task 2 and task 3 can continue to run because they are independent and do not share axis 3.

```
DEF main          ; Begin definition of program called "main"
1%TSKAX1,3       ; Associate axes 1-3 with Task 1 (see Fig 6b)
2%TSKAX4,6       ; Associate axes 4-6 with Task 2 (see Fig 6c)
3%TSKAX7,8       ; Associate axes 7 & 8 with Task 3 (see Fig 6d)
1%move1          ; Execute the "move1" program in Task 1 (affects axes 1-3)
2%inout          ; Execute the "inout" program in Task 2 (affects axes 4-6)
3%fill          ; Execute the "fill" program in Task 3 (affects axes 7 & 8)
END              ; End definition of program called "main"
```

## 6K8

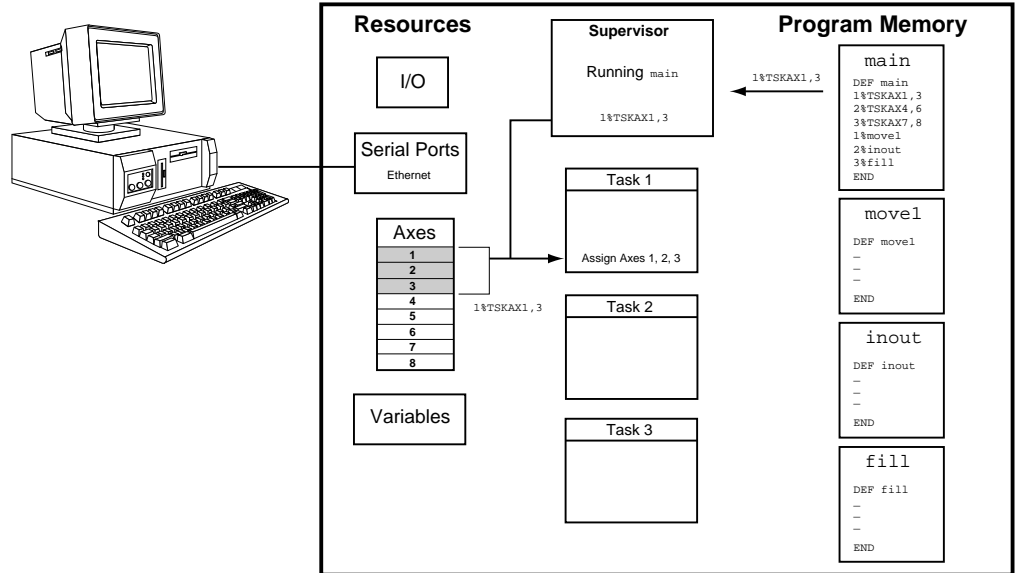


Figure 6b: Executed from the “main” program, the `1%TSKAX1,3` command assigns axes 1-3 to Task 1.

## 6K8

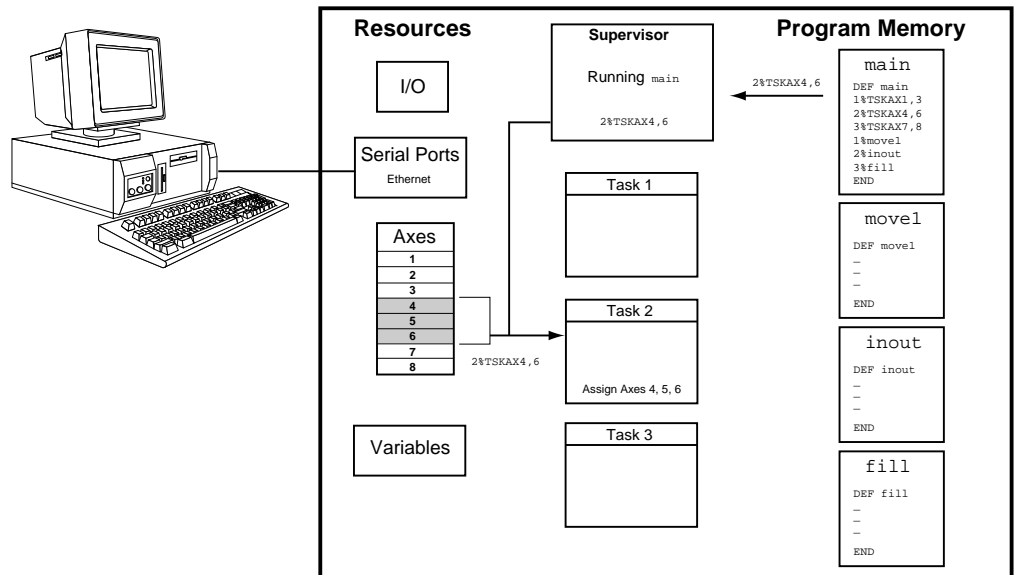


Figure 6c: Executed from the “main” program, the `2%TSKAX4,6` command assigns axes 4-6 to Task 2.

## 6K8

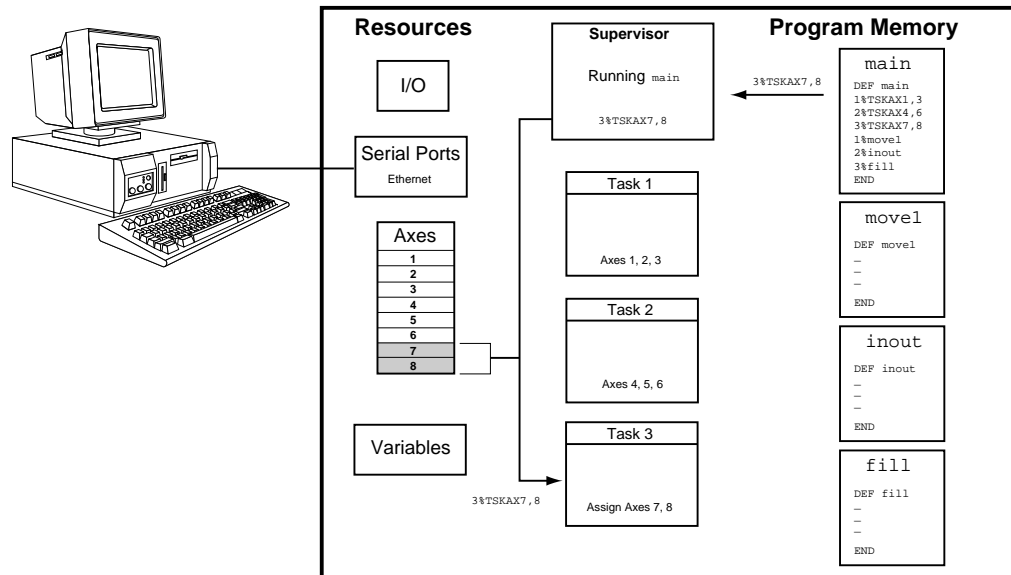


Figure 6d: Executed from the “main” program, the 3%TSKAX7,8 command assigns axes 7 and 8 to Task 3.

## How a “Kill” Works While Multi-Tasking

The general rule of command execution within a task is that the command affects only the task in which it was executed or to which it was directed via the % prefix. This includes almost all motion commands when tasks have associated (TSKAX) axes. The exception is the Kill command. A “K” or “!K” without prefix or parameters will kill all controller axes, and programs running in all tasks. This exception is made to allow one short, familiar command to effectively stop all controller processes. As usual, a “K” with parameters (e.g., Kx11) will kill motion only on the specified axes, and will not kill the program. A “K” prefixed with a task specifier but without parameters (e.g., 3%K) will kill the program and motion of all axes associated with that task.

Certain inputs can be used to kill programs and motion. These inputs and their results under multi-tasking are:

- **Drive Fault input.** Kills program execution and associated axes for all tasks that are associated with the faulted axis.
- **Kill input** (INFNCi-C or LIMFNCi-C). If prefixed with the task identifier (e.g., 2%INFNC3-C), it will kill the designated task, its program and associated axes. If not prefixed with a task identifier, it will kill all controller axes, and programs running in all tasks. It will also cause a branch to the task’s error program if the task’s error-checking bit 6 is enabled (n%ERROR.6-1).
- **User fault input** (INFNCi-F or LIMFNCi-F). This input functions the same as the kill input noted above, with the exception that it will also cause a branch to the task’s error program if the task’s error-checking bit 7 is enabled (n%ERROR.7-1).

## Associating Axes with Tasks

The default condition in multi-tasking is that each task is associated with all controller axes. The `TSKAX` command allows you to assign axes to specific tasks, thus constraining task response and control to a smaller set of axes. The `TSKAX` command allows you to specify the first and last axis numbers associated with the task; thus, the axes associated with a task will always be consecutive. This association covers all interaction between axes commands, conditions or inputs and task program flow, including the following:

- **Axis data commands** (e.g., A, V, D). These commands will affect only the associated axes. If data is supplied in the fields of non-associated axes, it will be ignored. To alter the data of an axis associated with another task, that task number must prefix the command (e.g., if axis 3 is associated with task 2 but not task 1, then to change axis 3 velocity to 5 units/sec from task 1, the command `2%3V5` can be issued).
- **Stop, pause, and continue commands and inputs**. These commands will affect motion only on the associated axes of the task receiving the command: `S` (stop) command — without parameters, `PS` (pause) command, and `C` (continue) command. Inputs programmed with these functions will affect motion only on the associated axes of the task specified in the input function assignment (with the `n%` prefix): Kill input (`n%INFNCi-C` or `n%LIMFNCi-C`), stop input (`n%INFNCi-D` or `n%LIMFNCi-D`), pause/continue input (`n%INFNCi-E` or `n%LIMFNCi-E`), and user fault input (`n%INFNCi-F` or `n%LIMFNCi-F`).
- **Drive fault and limit inputs**. If a drive fault or end-of-travel limit occurs on an axis, only tasks that include that axis as an associated axis will be killed.
- **ON conditions**. If user status information includes axis status bits, and `ONUS` is enabled via `ONCOND`, then only bits of an associated axis will be monitored by that task.
- **ERROR conditions**. Each task has its own error status register (`ER`, `TER`, `TERF`), error-checking bits (`ERROR`), and error program (`ERRORP`). If an axis-related error occurs, only the task that is associated (`TSKAX`) with the axis is affected.
- **Command buffer control**:
  - `COMEXC`. Pauses command processing of that task until no associated axes are moving.
  - `COMEXL`. Allows specified associated axes to not kill that task's program. (`COMEXL` applies to axes, and any task that includes an axis will be affected by that axis' `COMEXL` status. An axis can't be enabled for one task and disabled for another.)
  - `COMEXR`. Motion on only the associated axes will also pause if a task-specific pause input becomes active.
- **OUTFNC bits**. Output bits defined without an axis specifier will become active if the condition becomes true on any associated axis.

## Sharing Common Resources Between Multiple Tasks

Controller resources other than processing time must also be shared between multiple tasks. All physical inputs and outputs are shared (i.e., each task can read all inputs and write to all outputs). Some of the functions associated with that I/O (defined with LIMFNC, INFNC and OUTFNC commands) are unique for each task. All variables (VAR, VARI, VARB, VARS), DATA programs, and the DATPTR are shared. If one task modifies a variable, a DATA program, or DATPTR, the new value is the same for all tasks. There are no “private” variables. The serial ports are shared (i.e., any task can read from, or write to, any serial port). When a task writes to a serial port, the characters are all sent to the output buffer at once, but execution proceeds to the next command without waiting for those characters to be transmitted.

The data assignment functions READ, DREAD, or DREADF, and TW are also shared between tasks, but in these cases, the task is not allowed to execute the next command until the data has been received and assigned. This requires waiting for human or host input, or thumbwheel strobe time. During this time, that task is “blocked”, and that data assignment function (READ, DREAD, or DREADF, or TW) is owned by the blocked task. If another task is active, it will continue to execute commands. If this second task attempts to access a data assignment function that is still owned by the first task, the second task becomes blocked also. This situation persists until the first task receives and assigns its data. At that point, ownership of the data assignment function passes to the second task, until it too receives and assigns its data. All this blocking and ownership is automatic, not requiring coordination by the user.

## Locking Resources to a Specific Task

### Examples:

```
LOCK1,1 locks COM1
LOCK1,0 unlocks COM1
LOCK2,1 locks COM2
LOCK2,0 unlocks COM2
LOCK3,1 locks swapping
LOCK3,0 unlocks swapping
```

The LOCK command can be used by a system task to gain or release exclusive ownership of a resource. This will allow that task to complete a sequence of commands with that resource while preventing another task from using the resource in between commands. Any other task attempting to access or LOCK that resource will become blocked (i.e., become temporarily totally inactive) until the resource is released by the task that owns it. Resources that can be LOCKed are:

- COM1 — the “RS-232” connector or the “ETHERNET” connector
- COM2 — the “RS-232/485” connector
- Task Swapping — When task swapping is locked to a specific task, command statements in all other tasks will not be executed until the task swapping is again unlocked. For more information on task swapping, see page 218.

**NOTE:** A resource can be locked by a task only while that task is executing a program. If program execution is terminated for any reason (e.g., stop, kill, limit, fault, or just reaching the END of a program), all resources locked by that task will become unlocked.

For example, to report the current position of a robotic arm, one might program:

```
VAR10 = 2PM ; position of axis two
LOCK1,1 ; lock COM1 ("RS-232" port) resource
WRITE "Arm Position is" ; write string
WVAR10 ; write position
WRITE "inches\13" ; write string and carriage return
LOCK1,0 ; release COM1 to other tasks
; Between the LOCK commands, other tasks will run unless they
; attempt to access COM1, in which case they become
; temporarily blocked. The same concept applies to COM2 and task
; swapping. In the extreme, the LOCK command may be used to prevent
; other tasks from doing anything, allowing one task all command
; processing resources. This might be done to maximize the speed
; of a group of commands, or to protect a sequence which is not
; otherwise covered with the LOCK options.
```

## How Multi-tasking and the % Prefix Affect Commands and Responses

Multi-tasking affects many, but not all, of the 6K commands and features. The table below lists the commands that are affected by multi-tasking and the task-identifier (%) prefix. Any command that is not listed can be considered unaffected by multi-tasking. Any command that has axis data (e.g., A, V, D) is only affected if the axes have been associated with a task via the TSKAX command. Any command listed below as affected by the % prefix can also accept the @ character as a task number; in this case, all tasks will be affected by the command.

The response (if any) sent to the terminal by the command will be prefixed with the task number and % sign if the command had been executed by any task other than Task 0, the Supervisor Task. For commands executed in Task 1, if the command itself had the 1% prefix, then the response will also have the prefix. Otherwise, the response will not be prefixed. Therefore, if no multi-tasking is used, no responses will have task prefixes. If multi-tasking is used, however, you can determine which task originated the response. A response could be the result of a transfer command (e.g., TER), a command with no parameters (e.g. MC), trace output, or an error message.

Command(s)	Effect of % Prefix	Effect of Multi-tasking
GOTO, IF, ELSE, NIF, WHILE, NWHILE, REPEAT, UNTIL, L, LN, LX	None (ignored)	These commands direct program flow only on the task from which they are executed.
GOSUB, RUN, <programe>, JUMP, HALT, BP, PS, T, WAIT, C	Effectively inserts command into numbered task's program	These commands initiate, interrupt or continue program flow only on implicitly or explicitly specified task.
COMEXC, COMEXR, COMEXS	Affects mode on numbered task	These commands affect program flow only on implicitly or explicitly specified task.
S, K	Specifies affected task and axes, or axes only	These commands affect program flow and/or associated axes (depending on parameters given) only on implicitly or explicitly specified task. A "K" with no prefix or parameters kills all tasks.
ERROR, ERRORP, TER, TERF, ER	Affect/report on numbered task	These commands affect/report error conditions and programs only on implicitly or explicitly specified task (each task has its own error status register and error program).
TCMDER, TSS, TSSF, SS, TSTAT	Report on numbered task	These commands report command errors and status only on implicitly or explicitly specified task.
TTIM, TIMINT, TIMST, TIMSTP, TIM	Command refers to timer of numbered task	Each task has an independent timer. Also, TIMST0,# resets the timer to # msec., TIMST1,# restarts the timer with the TIM value of task #.
TRACE, TRACEP, TEX, STEP, #	Command refers to numbered task	Each task has its own trace and step mode. This allows tracing on a single task, or combined tasks commands.
ONCOND, ONIN, ONP, ONUS, ONVARA, ONVARB	Command refers to numbered task	Each task has its own set of ON conditions and its own ONP program.
LIMFNCi-C, LIMFNCi-D,	Command refers to	These functions are task specific.

LIMFNCi-E, LIMFNCi-F, LIMFNCi-P, INFNCi-C, INFNCi-D, INFNCi-E, INFNCi-F, INFNCi-P, INSELP, OUTFNCi-C	numbered task	All others affect the entire controller and cannot be affected by the % prefix.
--	---------------	---

---

## Input and Output Functions and Multi-tasking

The 6K has inputs and outputs (onboard and on external optional I/O bricks) that can be assigned various I/O functions with these function assignment commands:

- `limfnc` ..... Assigns input functions to the dedicated limit inputs on the “LIMITS/HOME” connectors (factory default functions are the respective R, S, and T limit input functions).
- `INFNC`..... Assigns input functions to the onboard digital inputs (trigger input on the “TRIGGERS/OUTPUTS” connectors) and to the digital inputs on external I/O bricks.
- `OUTFNC` ..... Assigns output functions to the onboard digital outputs (outputs on the “TRIGGERS/OUTPUTS” connectors) and to the digital outputs on external I/O bricks.

The 6K’s I/O and events involving programmed I/O are related to tasks only if:

- The input or output function assignment is associated with a specific task. For example, because of the 1% prefix, the 1%`INFNC3-F` command assigns the “user fault” function to onboard input 3 and directs its function to be specific to Task 1.
- An input or output is assigned an axis-specific function and the axis is associated with a specific task. For example, if axes 2 and 3 are associated with Task 1 (1%`TSKAX2, 3`) and onboard input 4 is assigned as a “stop” input for axis 2 (`INFNC4-2D`), then when onboard input 4 goes active axis 2 is stopped; thus Task 1 is affected by input 4.

Each input or output group (`LIMFNC`, `INFNC`, and `OUTFNC`) is limited to a maximum of 32 function assignments at one time. Exceptions are: function A, `LIMFNC` functions R, S, and T, and `INFNC` function H. A given input or output can be assigned only one function, although for task-specific functions, this I/O point can perform the same function for multiple tasks (by using the % prefix). This only counts as one function against the maximum total of 32, even though it is shared by multiple tasks. A given I/O point cannot perform different functions for different tasks. Re-assigning a I/O point’s function in any (same or different) task will result in removing the assignment of the previous function in the previous task(s). For example:

```
1%2INFNC4-C ; module 2's 4th input will kill task 1
2%2INFNC4-C ; module 2's 4th input now kills tasks 1 & 2
3%2INFNC4-D ; module 2's 4th input will stop all axes in task 3,
              ; it no longer has a kill function for tasks 1 & 2
```

## Axis-Specific I/O Functions

The table on page 215 shows the LIMFNC, INFNC and OUTFNC functions that can be shared by multiple tasks. A special case is the “stop” input function (INFNCi-aD or LIMFNCi-aD). Here, “a” is the optional axis number. If it is not included, the function stops the program and all associated (via TSKAX) axes of a task. In this case, the function is task specific, and can be shared by multiple tasks. If an axis number is included, it simply stops motion on the specified axis, and does not affect program flow. In this case, the function is not task specific; it associates an input with only one physical axis. The same is true for any INFNC or OUTFNC function that can accept (or require) an axis identifier. An I/O function can be associated with only one physical axis. The actual physical axis associated with the input or output is determined at the time the LIMFNC, INFNC or OUTFNC command is issued, in the context of current value of TSKAX for the current or % designated task. That association does not change if TSKAX changes later. As with all commands, if the specified axis is not part of the associated axes group (via TSKAX) of the task, the command will be ignored. Refer to the example below.

*Example*    TSKAX1,4            ; task 1 associated with physical axes 1-4  
              2%TSKAX5,8       ; task 2 associated with physical axes 5-8

## Multi-Tasking Performance Issues

---

### When is a Task Active?

Tasks are always ready to become active if commanded to do so. No special command is required to allow a task to do something. A task is active if it is:

- Executing a program (SS.3=B1)
- Executing a T or WAIT command (even if not running a program)
- Monitoring drive faults conditions (DRFEN1)
- Monitoring ONCOND conditions (SS.15=B1)
- Waiting in Program Select Mode (INSELP1 and SS.18=B1)
- Monitoring ERROR conditions (ERROR has any non-zero bit)

A task becomes active when it receives a command to perform something from the list above. A task that is not active does not create any overhead (delays) in command processing for tasks that are active. Once a task becomes active, however, it takes its turn in the sequence of swapping (see below), and checks for all of the conditions listed above. This adds a processing burden that will slow the command execution rate of programs running in other tasks.

A task becomes inactive if nothing from the above list is occurring. There is no special command “kill task” to disable all these modes/conditions/activities at once. The standards methods for terminating these (S, INSELP0, ERROR0, etc.) are needed to make a task completely inactive, and therefore not creating command processing overhead. Task 0, the Task Supervisor, will always be active, even if nothing from the above list is occurring, because Task 0 always checks the input command buffer for commands to execute.

The TSWAP command reports a binary bit pattern indicating the tasks that are currently active. Bit 1 represents task 1, bit 2 represents task 2, etc. A “1” indicates that the task is active, and a “0” indicates that the task is inactive. The SWAP assignment operator allows the same information to be assigned to a binary variable, or evaluated in a conditional statement such as IF or WAIT. This is useful for determining which tasks have any activity, whereas the system status (SS) and error status (ER) states reveal exactly what activity a given task has at that time.

## Task Swapping

The 6Kn controller has only one processor responsible for executing programs, therefore, multiple tasks are not actually executing simultaneously. Tasks take turns executing when they are active. A task's "turn" consists of executing one command and checking its input, output, ON, ERROR and INSELP conditions before relinquishing control to the next task. The process of changing from one task to the next is called **task swapping**. The 6K controller determines when to swap tasks. The user is not required (or allowed) to determine how long a task should run, or when to relinquish control to another task.

Although the user does not determine how long a task should run, or when to relinquish control to another task, the number of "turns" a task gets before swapping can be set with the TSKTRN command. The default value is one for all active tasks. Thus, each task has equal weight, swapping after every command. If a task needs a larger share of processing time however, the TSKTRN value for that task can be increased. For example, if task 2 issued a TSKTRN6 command, while the others stayed at 1, programs running in task 2 would execute six commands before relinquishing. The TSKTRN value for a task can be changed at any time, allowing a task to increase its weight for an isolated section of program statements.

The TTASK command reports to the display the task number of the task that executed the command. This could be used for diagnostic purposes, as a way to indicate which task is executing a given section of program. The corresponding TASK assignment allows the program itself to determine which task is executing it. The current task number TASK can be assigned to a variable or evaluated in a conditional statement such as IF. This allows a single program to be used as a subroutine called from programs running in all tasks, yet this routine could contain sections of statements that are executed by some tasks and not others.

## Task Execution Speed

Two terms describe the execution speed of tasks in multi-tasking environments.

### Performance

Performance is a measure of how fast a task executes commands, and could be described in terms of commands per second. Because tasks must take turns executing, performance will decrease when tasks are added. For example, the performance of a task sharing the controller with two other tasks will be just one third of its performance if it were running by itself. Using the TSKTRN command described above, the relative performances of tasks can be altered.

### Determinism

Determinism is a measure of how independent the performance of one task is from the commands and conditions involved in another task. If task swapping were done via a periodic interrupt (time slicing), task performance would be completely determinate, i.e., each task would run exactly for its time slice, not more or less. In the 6K products, task swapping is not time sliced, rather each task is allowed to execute one command and check its input, output, ON, ERROR and INSELP conditions before relinquishing control to the next task. For this reason, the performance of each task is somewhat determined by the commands and conditions underway in the other tasks. Most commands execute in approximately the same time. The others (some math commands, and line and arc commands) are broken into sections with durations that approximate those of the average command (approximately 2 ms). Swapping also takes place between these sections, allowing all task performance to be reasonably determinate regardless of which commands are being executed by other tasks.

Although time slicing would give completely determined task performance, the task swapping would need to include additional task save and restore functions to completely protect the task from the interrupt. This additional overhead would effectively rob processor time from the time slice, resulting in lower task performance. Task swapping in the 6K products does not add this overhead, and is therefore extremely efficient, taking less than 0.5% of task execution time.

# Multi-Tasking Application Example

---

## One machine multi-tasking application

A machine must perform three processes to a product. They are:

1. Fill the bottle with product. (axes 1,2)
2. Put a cap on the bottle. (axes 3,4)
3. Put the bottle in a box. (axes 5,6)

Each process must operate independently, because they wait on inputs that are not synchronized with each other. Even so, if one process slows too much, the others must also slow, to accommodate overall product flow. If a drive fault or limit occurs on any axis of any process, the entire machine must stop.

### Solution

Create programs CAP, FILL, BOX, SETUP, and MAIN. The first three will control the respective processes, and will each be run from their own task. The STARTP program will be SETUP. SETUP will initialize all I/O and motion parameters, launch the tasks that run CAP, FILL and BOX, then jump to MAIN, which monitors the overall product flow. The TSKAX command is not used, with the result that all tasks are affected by limits or faults on any axis.

```
DEF SETUP
INFNC1-A           ; set up I/O (assign onboard inputs 1-7 as
INFNC2-A           ; "general-purpose" inputs)
INFNC3-A           ;
INFNC4-A           ;
INFNC5-A           ;
INFNC6-A           ;
INFNC7-A           ;
OUT011001         ; initialize outputs
@A100             ; set up all accelerations
V4,5,7,6,32,12    ; set up all velocities
VARB1=B1          ; process 1 (filler) flag
VARB2=B1          ; process 2 (capper) flag
VARB3=B1          ; process 3 (boxer) flag
2%FILLER          ; start filler in task 2
3%CAPPER          ; start capper in task 3
4%BOXER           ; start boxer in task 4
JUMP MAIN
END

DEF MAIN
WHILE(IN.1=B1)     ; while process is running
  IF (IN.5=B1)     ; If filling too fast (high-flow input on)
    VARB1=B0
  ELSE
    VARB1=B1
  NIF
  IF (IN.6=B1)     ; If capping too fast (high-flow input on)
    VARB2=B0
  ELSE
    VARB2=B1
  NIF
  IF (IN.7=B1)     ; If boxing too fast (high-flow input on)
    VARB3=B0
  ELSE
    VARB3=B1
  NIF
NWHILE
END
```

*(continued on next page)*

*(continued from previous page)*

```
DEF FILLER
WHILE(IN.1=B1)           ; while process is running
  IF(VARB1=B1)           ; if allowed by MAIN
    WAIT(IN.2=B1)       ; fill input
    D4000,6000
    GO11
    WAIT(IN.2=B0)
    D-4000,-6000
    GO11
  NIF
NWHILE
END

DEF CAPPER
WHILE(IN.1=B1)           ;while process is running
  IF(VARB2=B1)           ;if allowed by MAIN
    WAIT(IN.3=B1)       ;cap input
    D,,7000,5000         ;could write "3D7000,5000"
    GOXX11              ;could write "3GO11"
    WAIT(IN.3=B0)
    D-,,7000,-5000      ;could write "3D7000,-5000"
    GOXX11              ;could write "3GO11"
  NIF
NWHILE
END

DEF BOXER
WHILE(IN.1=B1)           ;while process is running
  IF(VARB3=B1)           ;if allowed by MAIN
    WAIT(IN.4=B1)       ;cap input
    D,,,7000,5000       ;could write "5D7000,5000"
    GOXXX11            ;could write "5GO11"
    WAIT(IN.4=B0)
    D-,,,7000,-5000    ;could write "5D7000,-5000"
    GOXXX11            ;could write "5GO11"
  NIF
NWHILE
END

STARTP SETUP
```