Control Technology Corporation

Quickstep™ Language and Programming Guide

Doc. No. MAN-1010A

Contents

Appendix A - Sample Programs

Appendix B - Default Symbolic Names

Glossary

Notes To Readers

The *Quickstep™ Language and Programming Guide* provides the following information:

- A description of the Quickstep programming language
- How to write multi-tasking programs
- A language reference describing Quickstep instructions
- How to write Quickstep programs that effectively use controller resources and specialized I/O and motion control
- Sample Quickstep programs for a variety of applications
- The list of default symbol names available with the Quickstep editor

Related Documents

The following documents contain additional information

- For information on Quickstep, refer to the *Quickstep™ User Guide*.
- For information on general purpose and special registers, refer to the *Register Reference Guide*.
- For a tutorial using Quickstep, refer to the *Quickstep™ for Windows™ Tutorial*.
- For information on your controller and its modules, refer to the appropriate Installation and Applications Guide.
- For information on Microsoft Windows or your PC, refer to the manuals provided by the vendor.

Book conventions

The following conventions are used in this book:

ALL CAPS BOLDFACE	Identifies file names and fields on the Quickstep Editor user interface. It also identifies DOS, Windows, installation program file names.
Boldface	Indicates information you must enter, an action you must perform, or a selection you can make on a dialog box or menu.
Italics	Indicates a word requiring an appropriate substitution. For example, replace *filename* with an actual file name. It can also indicate a manual, book, or chapter title.
Text_Connected_With_Underlines	Indicates symbol names used in Quickstep programs. Step names are ALL_CAPITALS. Other symbol names can be Initial_Capitals or lower_case.
SMALL CAPS	Identifies the names of Quickstep instructions in text.
`Courier font`	Identifies step names, comment, output changes, and Quickstep instructions appearing in the Quickstep editor window or program steps
ArtCode – DN-24	Identifies the file name of a particular graphic image.

How to Contact Control Technology Corporation

Control Technology Corporation is located in Massachusetts, and wer are open from 8:30 a.m. to 5:00 p.m. eastern time. Contact us at 508 435-9595 and 800 282-5008 or Fax 508 435-2373

See us on the web at www.ctc-control.com.

Your Comments

We welcome your suggestions and comments about this or any other Control Tech document. Comment forms are in the file called BUGRPT.WRI, which was installed in your QSWIN directory during your Quickstep installation. you can also email comments to techpubs@ctc-control.com.

Introduction

Contents

Introduction

Multi-tasking
Programs written in Quickstep can also match the modular nature of many machines, since it gives you the ability to write multi-tasking programs. Multi-tasking executes multiple program modules simultaneously; each module can control a separate sequence of events. For more information on multi-tasking, see the section on *Multi-tasking*.

Quickstep™, Control Technology Corporation's (CTC) programming language, is used to program our automation controllers. The overall format of a Quickstep program consists of a sequence of events, described as a series of program steps. Each step is a collection of instructions that determines the state of an automated machine or system for an interval of time. A step can contain both instructions which initiate motions and instructions that monitor various sensors. Steps can also contain specific high-level instructions for functions such as stepping motor and servo control.

The sequential format of Quickstep is very similar to the thought processes that originally enter into the design of a new machine. New machines are usually designed to perform a specific task or series of tasks. The task can be an assembly operation, a series of machining operations, or a chemical process consisting of a sequence of etching, depositions, and/or reactions. Machine designers take this task and break it down into a number of discrete operations. These discrete operations possess a natural sequence or order and must occur in that sequence.

Machine designers often plan the sequence of tasks in advance and record them on paper in one of the following formats:

- Written narrative description
- Flow chart
- Timing diagram

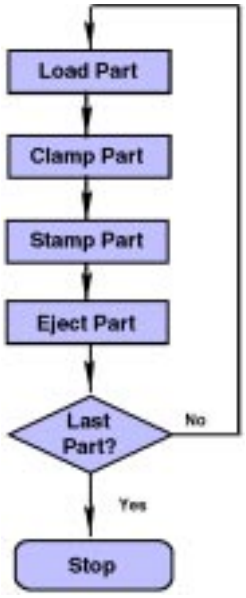
Each of these approaches describes the machine's operation in a similar manner—as a series of states through which the machine proceeds as it accomplishes its task. Quickstep mirrors the states that a machine proceeds through, while also allowing you to program parallel, asynchronous functions.

By describing a machine's operation as a series of states, a Quickstep program maintains a one-to-one correspondence with the functions of the machine you are programming.

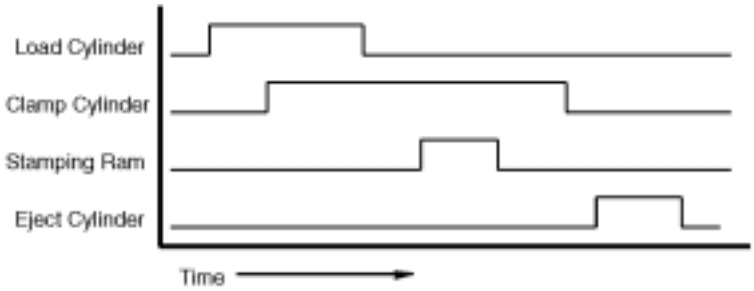
Sequence of Events Expressed as a Narrative Description

1. Extend load cylinder. Wait for limit switch to be hit before proceeding.
2. Extend Clamps. Allow 0.3 second for clamps to be fully extended.
3. Once clamps have extended, retract load cylinder. Allow 0.5 second for load cylinder to retract.
4. Actuate stamping ram. Wait for a limit switch.
5. Retract stamping ram. Allow 0.5 second for ram to retract.
6. Actuate ejection cylinder. Allow 0.3 second for part to be ejected.
7. Retract ejection cylinder.
8. Increment parts counter.

Sequence of Events Shown in a Flow Chart



Sequence of Events Shown as a Timing Diagram



What is a Step?

A Quickstep program uses steps to define each new state of a machine. A complete program is made up of a series of steps executed in a defined pattern. Steps are made up of the following two elements:

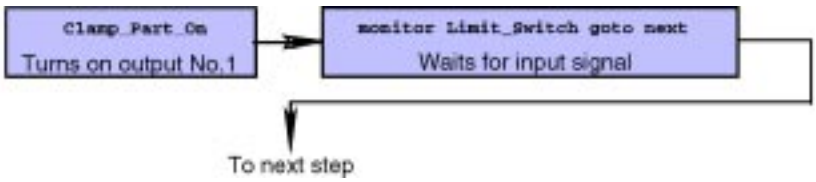
- One or more instructions changing the controller's outputs by turning them on or off or an instruction maintaining their current state. Turning one of the outputs on or off usually creates one or more motions on a machine and can establish the machine's new state.
- One or more instructions for leaving the step. These instructions establish the duration of that state.

Since you can program more than one instruction in a step, there can be several possible paths for leaving a step. The path taken by the controller would depend on the conditions sensed when the step is executed. The controller can monitor one or more conditions on the machine (limit switches, sensors) and only proceed to the next step once a condition is met. Another option is to program a time delay and have the controller move to a new step after a specified amount of time.

You can also program specialized motion control instructions, such as specifying the velocity of a stepping or servo motor, or program instructions for various internal controller functions, like counters, flags, and math instructions.

How a Simple Step Works

The following example shows a simple step. This step extends a pneumatic cylinder and waits for it to strike a limit switch at the end of its travel before proceeding to the next step.



Symbolic Names
In many of the examples shown in this manual the inputs, outputs, stepping motors, etc. have symbolic names. Starting with Quickstep 2.0 you can give resources like these symbolic names that can identify their function,. For example, an output that turns on a pneumatic cylinder for a stamping press can be called Stamp_Press_On. For more information on symbolic names, see Using *Symbolic Names in Quickstep Instructions* in Chapter 2.

There must be two devices on the machine connected to the controller to accomplish this task.

1. A solenoid valve which directs compressed air to the appropriate cylinder port. The solenoid is controlled by one of the controller's outputs, labeled Clamp_Part_On in the illustration.
2. A limit switch positioned so that the pneumatic cylinder strikes the limit switch at the end of its travel. The limit switch is connected to one of the controller's inputs (labeled Limit_Switch in the illustration). The limit switch sends a signal to the controller's input.

When the controller executes the step, it first executes the output instruction. In this case, it turns on Clamp_Part_On. Next, it executes any remaining instructions in order. This step only has one additional instruction, `monitor Limit_Switch goto next`. When the controller receives the input signal, it proceeds to the next step.

Multiple Instructions in a Step

Steps can contain multiple instructions. This example adds a fault-monitoring function to the simple step discussed previously. The following example makes use of two of the rules of Quickstep program execution:

1. You can program more than one instruction in a step.
2. Instructions with a destination are executed repeatedly, in the order programmed, for the duration of the step.

Using these two features we can expand the simple step example. In it, the limit switch is used to detect the completion of a pneumatic cylinder's stroke. If the cylinder jammed in mid-stroke, the controller would never receive the signal from the limit switch. It would remain in its current step forever, waiting for the limit switch signal. By estimating the longest possible normal stroke time for the cylinder, you can have the controller automatically sense a jam condition and have it take appropriate action. One method of accomplishing this is to program a DELAY instruction (time delay) in the same step of the program.

When the controller enters the step, it instantly turns on output 1, telling the cylinder to start extending. Next, it begins executing both the MONITOR and DELAY instructions. The instruction that is satisfied first takes the controller out of the step. If the cylinder reaches the limit switch before the time delay elapses, the controller continues to the next step, which is the normal operating sequence. The time delay instruction effectively disappears at this time, because the controller is no longer executing this step.

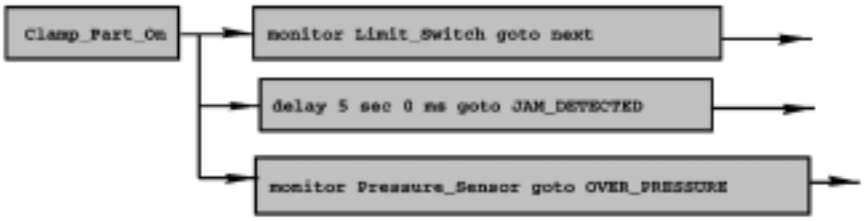
If the controller does not receive the signal from the limit switch on time and the DELAY instruction times out, the controller proceeds to step JAM_DETECTED. At step JAM_DETECTED you can program an instruction to shut down the machine, ring an alarm, or begin an automatic unjam sequence for the machine.

Monitoring a Fault Sensor

Using the same example, we can also program additional fault sensing by including more MONITOR instructions. To detect an over pressure condition you can:

- Connect a pressure switch to another of the controller's inputs.
- Add an instruction to monitor this input.

You can program the second MONITOR instruction to send the controller to a different step. The controller monitors inputs Limit_Switch and Pressure_Sensor at the same time it is timing the DELAY instruction. Quickstep executes all three instructions until one of them is satisfied. This means, when the machine reaches an over pressure condition, the controller does not wait for the other instructions before leaving the step. The illustration on the next page shows this step.



In some cases, you may need to have certain universal monitoring instructions that are always executed by the controller, for example, an instruction to monitor an over-temperature switch that must be monitored during an entire sequence of events. Although you can place a MONITOR instruction in each step of your program, an easier method of accomplishing this is by multi-tasking. For additional information on multi-tasking, see the section on *Multi-tasking*.

Using Control Instructions

Controller Resources
CTC controllers provide the following internal controller resources that you can use when writing your Quickstep program:
- Counters, for functions such as batch counting or production counting.
- Flags, which are memory locations used to store yes/no information.
- General-purpose numeric registers used to store numbers.
- Special purpose registers, which may store numbers, but are also reserved for special purpose functions.
- A Data Table, which stores an array of numbers to use within a Quickstep program.

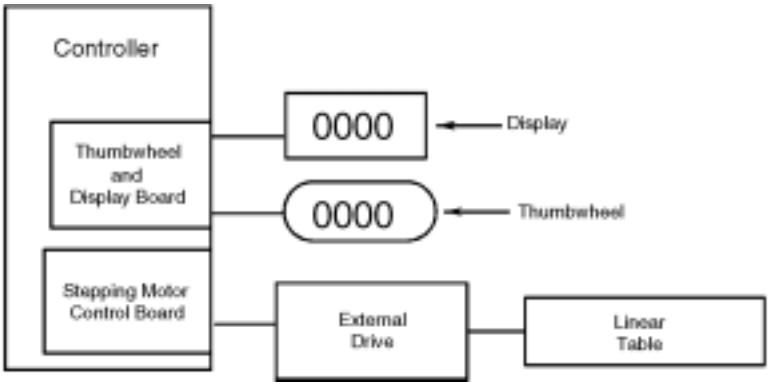
Controller resources are described in Chapter 3.

The previous examples explained the general format of a sequential program, showed several simple forms of steps, and illustrated the use of MONITOR and DELAY instructions to control the flow of a program from one step to another. Both of these instructions force the controller to a new step when a given event occurs.

Steps can also contain two other types of instructions:

- High-level instructions to initiate certain events which can span several steps. For example, instructions to initiate motion on a stepping motor or servo.
- Internal control instructions that affect the status of internal controller resources, including math instructions.

In this example, an external thumbwheel preset, which allows an operator to dial in a desired coordinate for a linear motion table, is connected to the controller. The table is driven by a stepping motor. Although the controller moves the table in units of steps, we want the operator to be able to dial in the coordinate expressed in thousandths of an inch of travel. The following illustration shows the setup of the controller, thumbwheel, and linear table.



Let us assume each step of the stepping motor corresponds to .0005 inch of table travel. To obtain the appropriate number of steps to move the motor, we know that we must multiply the coordinate position by a factor of 2. The first instruction in the program is a math instruction:

```
store Distance * 2 to Travel_Distance
```

Where:

Distance is the symbolic name for thumbwheel 1
Travel_Distance is the symbolic name for register 10

This instruction takes the current setting of thumbwheel 1 (Distance), multiplies it by two, and stores the result in one of the controller's internal numeric registers (Travel_Distance).

A second instruction takes the value stored in Travel_Distance and instructs the stepping motor to move to that coordinate:

```
turn Motor_1 to Travel_Distance
```

Since we want the controller to wait for the motor to reach its new position before the controller proceeds with its program, a third instruction monitors the motor:

```
monitor Motor_1:stopped goto next
```

The controller waits until the motor has stopped before going to the next step.

The Importance of Order

Specialized Motion Control and I/O Devices
Quickstep combined with CTC's controller hardware allows you to program:
- Specialized motion control devices, such as servo and stepping motors.
- Data acquisition and processing for analog I/O devices
- Specialized I/O such as numeric displays and thumbwheels.

Specialized motion control and I/O devices are described in Chapter 3.

In the previous example the value used for the TURN MOTOR instruction was calculated in the same step as the TURN MOTOR instruction. The instruction that performed the calculation was programmed before the TURN MOTOR instruction. This works because the controller executes the instructions within a step in the order in which you program them. The specific rules governing the execution of Quickstep instructions are as follows:

- When first entering a step, the controller turns on or off any digital outputs specified.
- The controller then executes any additional instructions in the step. It does this in the order that they are programmed.
- Instructions without a destination, that is, instructions that do not specify how to leave the step, are executed only once. These instructions include mathematical calculations, storing a value to a register, turning a motor or servo on, and motor or servo PROFILE instructions.
- Instructions with a destination are executed repetitively, in the order programmed, for the duration of the step.
- Time DELAY instructions are set up initially and checked repetitively for a time-out as long as the controller is in the step.
- If an instruction with a destination is already satisfied when a controller first begins to execute a step, the controller leaves the step without executing any subsequent instructions in that step.

In the following example the value for a pressure transducer is scaled and offset in the same step. The table below shows that the pressure transducer provides a signal voltage ranging from 0.5 to 5.5 volts with an applied pressure ranging

from 0 to 500 PSIG. The analog I/O board in the controller uses a conversion factor of 1000 counts per volt and reads this signal as a number from 500 to 5500.

Specification	Value
Applied pressure	0 - 500 PSIG
Voltage output	0.5 - 5.5 volts
Analog value	500 - 5500

The following example shows how you can program a series of instructions to display a correct pressure reading directly in PSIG.

Controller Resource	Symbolic Name
Analog input 1	Pressure_Trans
Register 10	Pressure_Value
Display 1	Pressure

```
[31]  CALCULATE_PRESSURE
      ____________________________________
         <NO CHANGE IN DIGITAL OUTPUTS>
      ____________________________________
     store Pressure_Trans - 500 to Pressure_Value
     store Pressure_Value/10 to Pressure_Value
     store Pressure_Value to Pressure
     if Pressure_Value >= 150 goto OVER_PRESSURE
```

In the step shown above, we first remove the offset to derive a numeric value that corresponds directly to PSIG. The following math instruction takes the value from the analog input (Pressure_Trans), subtracts 500, and stores the result in register 10 (Pressure_Value).

```
store Pressure_Trans - 500 to Pressure_Value
```

The subtraction results in a pressure reading between 0 and 5000. To complete the ranging of the analog signal, we must divide the number in the Pressure_Value by 10 as follows:

```
store Pressure_Value / 10 to Pressure_Value
```

The result of the division is stored back in register 10 and takes the place of the previous value.

The following instruction displays the pressure reading on the numeric display.

```
store Pressure_Value to Pressure
```

It is possible to take the value calculated by Pressure_Value / 10 and store it directly into the numeric display. However, by first storing the pressure value in register 10, we can refer to it later. An example of this is the last instruction in the step. It tests the value for an over pressure condition and jumps to step OVER_PRESSURE when such a condition is found.

```
if Pressure_Value >= 150 goto OVER_PRESSURE
```

NOTE: The controller processes the math calculation and the STORE instruction only once when executing the step. Any subsequent change in pressure will not be registered until the controller re-executes this step.

You can program the step so that the pressure reading will be continuously updated by adding the following instruction to the end of the step:

```
[31]  CALCULATE_PRESSURE
      ____________________________________
          <NO CHANGE IN DIGITAL OUTPUTS>
      ____________________________________
      store Pressure_Trans - 500 to Pressure_Value
      store Pressure_Value/10 to Pressure_Value
      store Pressure_Value to Pressure
      if Pressure_Value >= 150 goto OVER_PRESSURE
      goto CALCULATE_PRESSURE
```

where CALCULATE_PRESSURE is the name of the current step. The controller exits the step once it has been executed and jumps to the same step and re-executes it again repetitively. The controller continues the operation until the IF instruction causes the controller to jump to step OVER_PRESSURE.

Selective Execution of Instructions

When a controller first executes a step that contains several instructions, it executes the instructions in the order in which they appear in the step. What happens when one of the instructions wants to immediately send the controller to a new step? The following step is an example of this situation.

```
[1]  MONITOR_AND_TALLY
     ____________________________________
         <NO CHANGE IN DIGITAL OUTPUTS>
     ____________________________________
     monitor In_1_On goto next
     count up Parts_Produced
     goto next
```

The first instruction monitors input 1 to determine if a sensor switch has closed the input. This could indicate that the machine has sensed a bad part. If the machine senses a bad part, the MONITOR instruction sends the controller to the next step before the remaining two instructions are executed.

NOTE: The instruction goto next refers to the next step in the sequence, not the next instruction.

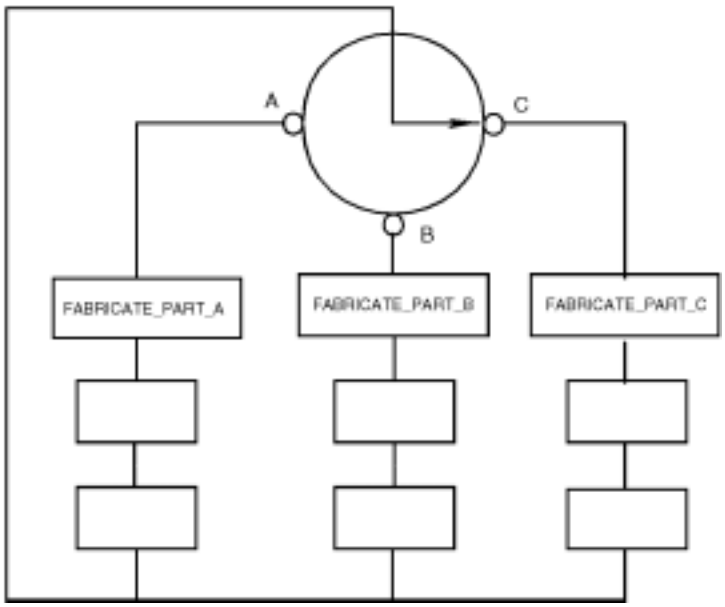
The second instruction in this step adds one to the count stored in counter 1 (Parts_Produced). The controller executes this instruction only if the previous MONITOR instruction did not send it to the next step. If the counter is tallying the number of good parts produced, only good parts are counted.

The last instruction in the step sends the controller to the next step unconditionally. In either case the controller jumps to the next step, remaining in the MONITOR_AND_TALLY step only for the time required to execute the appropriate instructions.

Storing Multiple Programs

You may need to program a machine to accomplish one of several possible tasks. For example, a machine may be required to fabricate one of three parts, A, B, or C. Each of these parts requires a different sequence of instructions. You can program all three of these sequences into the controller and use a manual selector switch to select the sequence you want. You can program one sequence in a series of steps, beginning with a step called FABRICATE_PART_A. The second sequence can be programmed in a series of steps beginning with a step called FABRICATE_PART_B and the third sequence could begin with a step called FABRICATE_PART_C.

To have the controller execute the correct sequence, you can connect three of the controller's inputs to the various points on the selector switch. Then program three MONITOR instructions in the beginning of the program. These instructions monitor the three inputs on the controller (See table) and jump to the correct step.



Controller Resource	Symbolic Name
Input 11 on	Fab_Part_A
Input 12 on	Fab_Part_B
Input 13 on	Fab_Part_C

```
[2]  MONITOR_FABRICATE_PART
     ------------------------------------------
         <NO CHANGE IN DIGITAL OUTPUTS>
     ------------------------------------------
     monitor Fab_Part_A goto FABRICATE_PART_A
     monitor Fab_Part_B goto FABRICATE_PART_B
     monitor Fab_Part_C goto FABRICATE_PART_C
```

Multi-Tasking

All of the previous examples have one common factor—they each consist of a single, linear sequence of events. These examples may have shown decision making or allowed the controller to follow one of several optional paths in the program. For many machines this type of program is appropriate. Most machines possess a natural sequence of events that must be executed to properly control them.

However, as machines grow in complexity, there are often multiple mechanisms on a machine that must be sequenced asynchronously. For example, a machine may contain the following:

- A loading mechanism to feed in a new workpiece
- An assembly station to perform an assembly operation
- An off-loading mechanism to remove the finished piece

To achieve optimum speed of operation, you may want to overlap a portion of, or all of, the sequences for the three mechanisms.

A similar situation exists with an index-table based machine. An index table has several workstations arranged around a rotary table, each performing specific operations on a workpiece as it goes around the table. Each time the table is indexed, all of the workstations must be sequenced simultaneously through their own independent sequence of events.

Machines like these have more of the characteristics of being several independent machines rather than one device. It is difficult to describe their combined operations as a single linear sequence. Any attempt to do so would require a careful analysis of the relative speeds of operation of each actuator on each mechanism. Such a program would also create inevitable inefficiencies as one mechanism is required to wait for another to actuate prior to proceeding with its own sequence.

Using Multi-Tasking to Execute Several Tasks

Using Quickstep, you can program the controller to execute several tasks at once. This is called multi-tasking. In programming a complex machine, multi-tasking allows you to think of each mechanism as a separate machine. You can write an independent task to control each mechanism using the standard Quickstep format. You then write a master program that calls up several of these independent tasks to operate simultaneously. It is as though the controller had split into several controllers running at the same time, each controlling one of the mechanisms.

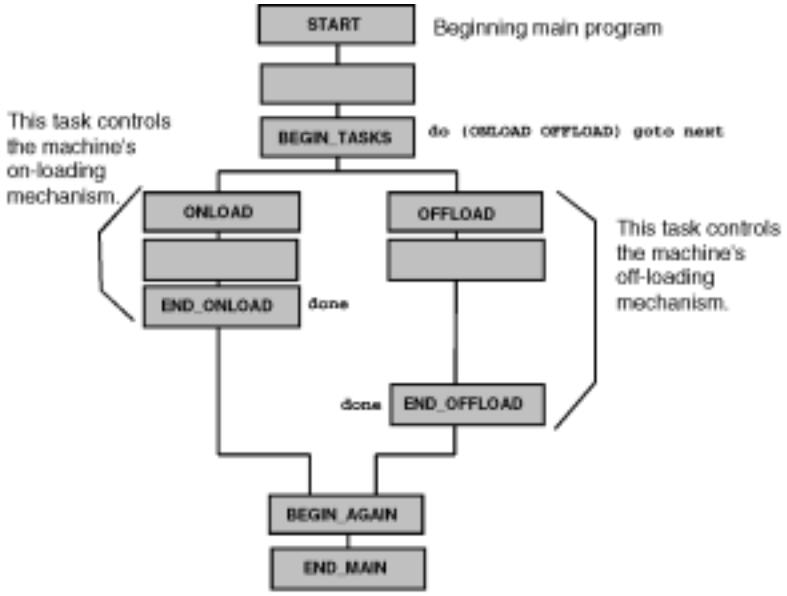
Depending on the controller model, it can have up to 28 separate tasks running at the same time. Refer to the installation guide for your controller for the number of tasks it can run.

Just as you might take a machine which is mechanically complex and break it down into modules for the sake of simplification, multi-tasking allows you to break your program into individual tasks–each task controlling one portion of the overall machine. Breaking down the program into tasks has the following advantages:

- Simplifies and shortens your programming efforts
- Increases the program's reliability (i.e., reduces the possibility of bugs)
- Makes each program easier to modify at a later date

Format of a Multi-tasking Program

The following diagram illustrates the format of a multi-tasking program. It shows each step of the program as a simplified block. In this instance the machine being controlled has a loading mechanism to load a new workpiece into position and an unloading mechanism to remove the workpiece completed on the previous cycle. The loading and unloading mechanisms are run simultaneously, because we assume that no mechanical conflict exists between them.



Nesting
Any task in a multi-tasking program can contain other tasks inside of it. Tasks contained within a task are called nested tasks. Nested tasks must start and end during the execution of its parent task and follow the rules for multiple tasks.

You can now write a separate program for each mechanism or task.

First write the program (task) to control the machine's loading mechanism. In this example, the task for the loading mechanism is only three steps long. When writing the loading task, you only need to be concerned with the operation of the loading mechanism, unless the possibility of mechanical interference with another part of the machine exists. At the end of the task, a DONE instruction indicates that it is completed.

Next, write the task to control the machine's off-loading mechanism, placing another DONE instruction at the end of this task.

NOTE: All controller resources and specialized motion control and I/O devices are globally accessible.

Starting the Tasks

The main program for the machine begins like a normal Quickstep program, as a sequence of events. In the third step of the main program, we program the instruction, `do (ONLOAD OFFLOAD) goto next`. This instruction causes the main program to suspend operation at its current step and starts two separate tasks running; one starting at step ONLOAD and one at step OFFLOAD.

The two tasks run independently as two separate programs. The effect is similar to having two separate controllers running the two portions of the machine. The

tasks operate asynchronously, one task may run through its steps quickly (based on the instructions contained in the steps) and the other task may wait for a long DELAY instruction to time out.

Individual tasks do not necessarily have to follow a sequential series of steps, although doing so improves program clarity. A task may jump around through any combination of steps in the program. A task may even contain subtasks (nested tasks) by incorporating DO instructions within the task itself.

Ending the Tasks

Each of the tasks continues until it reaches a DONE instruction. When one of the tasks executes a DONE instruction, it ceases operation and signals the original DO instruction in the main program that the task is complete. Once all of the tasks originally started by the DO instruction are complete, the main program continues. Since the DO instruction reads `do (ONLOAD OFFLOAD) goto next`, the main program will continue with its next step.

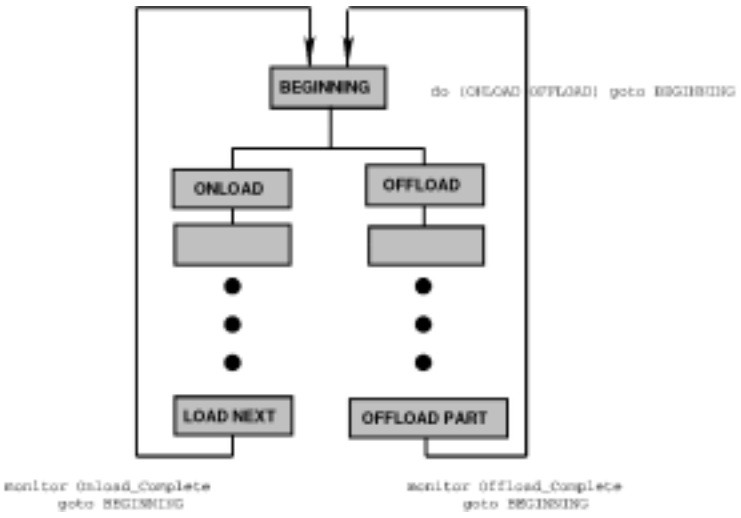
Counting the Number of Tasks

The task limit that a controller is capable of executing is an important limit. Violating the task limit for a controller results in a software fault, and the controller ceases operation. In the ONLOAD/OFFLOAD program example the controller was executing three tasks. The first task started at step 1, START. The DO instruction at step three started two more tasks. This is a total of three tasks. If the ONLOAD task contained additional DO instructions, those tasks would be added to the total number of tasks the controller was running.

The Problem of Recursion

Normally, programs are written to avoid violating the controller task limit. It is possible to write a program improperly and violate the task limit unintentionally.

The following diagram shows such a situation. At first glance, it may seem as though only two tasks are being started by the controller's main program. A careful analysis of the program indicates a structural error that would ultimately result in an infinite number of tasks running.



In the first step of the program the instruction `do (ONLOAD OFFLOAD) goto next` starts two separate tasks as shown. However, there is no DONE instruction at the end of each task. Although this may be allowable under some circumstances, in this example there is an instruction at the end of each task that returns the controller to the first step:

```
monitor Onload_Complete goto BEGINNING
monitor Offload_Complete goto BEGINNING
```

Onload_Complete and Offload_Complete are symbolic names for inputs.

Since tasks are allowed to jump around within a program, the controller views these instructions as continuations of their respective tasks. Each monitor instruction independently causes its task to jump back to the step called BEGINNING. Then, each task independently encounters the DO instruction in the first step again, and each task sets up two additional tasks—one commencing at step ONLOAD and one commencing at step OFFLOAD. Each of these two tasks ultimately returns to the first step and in turn sets up an additional two tasks each, and so on.

This problem is well known in computer programming and is called recursion. Recursion results whenever a subroutine (equivalent to a task) includes an instruction to restart the same subroutine. Eventually, the computer has to keep track of so many subroutines that it crashes, sometimes with disastrous consequences.

In CTC's controllers this results in a software fault, leaving all outputs, and other controller resources in the state they were in at the time of the error. Refer to the *Register Reference Guide* for a description of special purpose register 13009 and its use in software fault detection.

Avoiding Recursion

With Quickstep, recursion occurs when a task or one of its subtasks encounters the Do instructions which started it and re-executes it. Recursion can be avoided by clearly planning what you want to accomplish.

Do you want each program to loop back on itself continuously? Then change the MONITOR instruction to send the controller back to the beginning of that task. For example:

```
monitor Onload_Complete goto ONLOAD
monitor Offload_Complete goto OFFLOAD
```

This avoids re-execution of the original DO instruction.

Do you want the programs to execute in parallel and, when both tasks are completed, begin again simultaneously? Then add a step at the end of each program with a DONE instruction to signal the end of each task. The DONE instructions close out the tasks, signaling the original DO instruction to return to the first step. The controller returns to the first step and re-executes the DO instruction.

Using Modular Programs

Often a program can be written as a collection of tasks. If the construction of a machine is highly modular, you may be able to write a separate program module (task) for each mechanical module on the machine. The main program for the controller becomes a series of DO instructions that start the various tasks in the appropriate order.

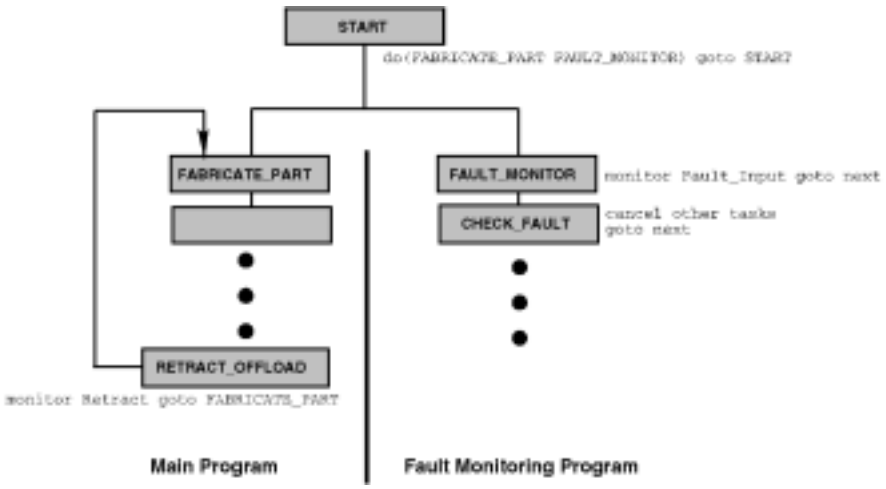
Creating segmented or modular programs is a powerful technique for simplifying the programming of a complex machine. When initially debugging the machine, you can quickly identify and rectify a programming problem in any given area. The location in the machine where the problem is occurring points to the appropriate program module to change. Writing modular programs also makes it very unlikely that modifying one task has any unintended side effect on another portion of the machine.

Fault Monitoring

Multi-tasking also allows you to perform continuous monitoring of a sensor during a machine's operation. By using multi-tasking it is possible to create a separate fault-monitoring program that does nothing but watch one or more fault sensors during the machine's operation. Only when the controller detects a fault condition does this program take action.

CAUTION: It is the machine designer's responsibility to assess the possibility of human injury and economic risk inherent in a machine's design and function. The machine designer must take adequate steps to protect against those risks. Under no condition should any one system or element on a machine represent the sole protection against injury or risk. Good design practice requires independent back up systems in such instances, preferably incorporating differing technologies in their design. The appropriateness of such measures must be assessed by the machine designer according to generally accepted safety practices in his or her industry.

The following illustration is an example of a fault monitoring program.



Multi-tasking starts in the first step of the program:

```
do (FABRICATE_PART FAULT_MONITOR) goto START
```

NOTE: The destination for the original Do instruction, goto start, is programmed to satisfy the format of the Do instruction and, in this instance, has no significance in the program.

The first of the two tasks, starting at step FABRICATE_PART, is the main task for running the machine. This task can take advantage of multi-tasking by

nesting additional subtasks within it. At the end of the task, step RETRACT_OFFLOAD loops back to the beginning of the task without executing a DONE instruction.

At the same time the fault-monitoring task remains at step FAULT_MONITOR, where an instruction continuously monitors a fault sensor on the machine. Only if the fault sensor becomes active does this program jump to step CHECK_FAULT. This step cancels all other tasks.

At step CHECK_FAULT you can program a number of different reactions to the fault condition. In the illustration the instruction, `cancel other tasks`, causes any other tasks that may be operating to cease operation. Cancel leaves all controller resources in the states they were in just prior to the CANCEL instruction. From this time on, the controller executes only one program, the fault monitoring program. This program can then proceed to take control over the machine, taking it through an orderly shutdown.

Sometimes less drastic action than shutting down all tasks is desired in response to a fault condition. For example, by programming the instruction, `stop goto FAULT_MONITOR`, at step CHECK_FAULT of the program, the entire machine would stop sequencing when a fault condition is sensed (including FABRICATE_PART and any subtasks). When the machine's operator corrects the fault and starts the controller again, all of the active tasks would continue execution from where they left off and the fault monitoring program could return to step FAULT_MONITOR.

CAUTION: The CANCEL and STOP instructions do not stop any of the motor motions in progress; nor do they turn off outputs that may be causing continuous motion, e.g., motor starters.

Rules for Writing Multi-tasking Programs

1. DO instructions can start up to eight tasks.
2. DO and DONE instructions must be in steps by themselves.
3. Obey the tasks limit for your controller model.
4. More than the maximum number of tasks can be included in a program, as long as no more than the maximum number are active at the same time.
5. A task may have other tasks nested inside of it.
6. Recursive programs should be avoided.
7. A DONE instruction signals the end of the task.
8. A CANCEL OTHER TASKS instruction or a reset input signal can be used to force an end to multi-tasking.

Sensing Multiple Faults

If more than one fault condition is possible on a machine, the fault monitoring program may include several instructions in its first step. Each instruction would monitor a separate condition. If different responses are required to the various faults, each instruction may cause the controller to jump to a different step. One factor that must be considered in setting up this structure is the

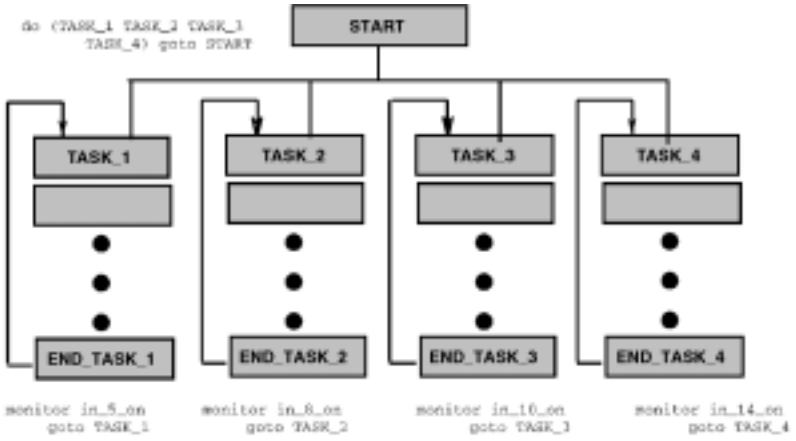
possibility of simultaneous faults and the desired priority and reaction to such a condition.

Running Several Machines with One Controller

It is sometimes possible to use multi-tasking to control several different machines from the same controller. Although there are a number of disadvantages to this approach, one obvious advantage is the ability to apportion the costs associated with the controller's capabilities over a number of machines.

You can use multi-tasking's ability to run several programs simultaneously to accomplish this. You must use a slightly different approach to do this. The most important difference is that each individual program is made to loop back on itself. It ends with some instruction, such as a monitor or time delay, that sends it back to its beginning.

The illustration below shows several machines run from the same controller using multi-tasking.



In some of the previous examples, the DO instruction waited for all tasks to end before continuing with the program. When running multiple machines, you would not want to have one of the machines, after completing its task, to wait for all the other machines to finish before starting a new cycle. By eliminating the DONE instruction, each program becomes one long task that never ends and loops forever in a circle. This allows each program to proceed at its own rate of speed without having to wait for another program to finish.

NOTE: The destination for the original DO instruction, goto start, is programmed to satisfy the format of the DO instruction and, in this instance, has no significance in the program.

One of the major disadvantages of this approach is the dependence of a number of machines on a single control system. This usually means that if one machine is stopped, all the machines being run by the same controller must also be stopped. The controller's dedicated stop, start, reset, and single step functions act on all of these machines simultaneously. When the economic advantages outweigh these concerns, multi-tasking represents a viable approach to the control of multiple machines.

Quickstep Instructions

Contents

Introduction

This chapter presents an overview of the Quickstep instructions. The following is a list of Quickstep instructions.

- Cancel
- Clear Flag
- Count Up
- Count Down
- Delay
- Disable Counter
- Do
- Done
- Enable Counter
- Goto
- If
- Monitor Boolean
- Monitor Flag
- Monitor Input
- Monitor Motor
- Monitor Servo
- Profile Motor
- Profile Servo
- Reset Counter
- Rotate Flag
- Search and Zero Motor
- Search and Zero Servo
- Set Flag
- Shift Flag
- Start Counter
- Stop (Controller)
- Stop Motor
- Stop Servo
- Store
- Test And Set Flag
- Turn Motor
- Turn Servo
- Zero Motor
- Zero Servo

Using Symbolic Names in Quickstep Instructions

In many of the examples shown in this manual the inputs, outputs, registers, etc. have symbolic names. Starting with Quickstep 2.0 you can give resources, such as registers, inputs, or stepping motors, symbolic names that can identify their function. For example, an output that turns on a pneumatic cylinder for a stamping press, can be called Stamp_Press_On. Or you can give several servo motors different symbolic names, e.g., Transverse, Rotate, Spindle, rather than calling them Servo_1, Servo_2, Servo_3.

You can define symbolic names for the following items:

- Steps
- Controller resources - counters, flags, numeric registers, Data Table columns.
- Specialized motion control devices - servo and stepping motors.
- Specialized I/O devices - analog inputs and outputs, thumbwheels, and numeric displays.
- Digital inputs and outputs.
- Numeric constants used in a Quickstep program, e.g., maximum speed of a stepping motor or a temperature value.

Symbolic names are created using the Symbol Browser. For a description of the Quickstep editor and the Symbol Browser, refer to the *Quickstep™ User Guide*.

Previous versions of Quickstep did not allow you to use symbolic names. When you use the new Quickstep editor to update a version 1.6 or 1.7 Quickstep program, it changes the names of all the controller resources to the default symbolic names. The examples in this chapter use the default symbolic names. The table on the following page lists the default symbolic names used in this chapter. The Quickstep editor includes a file, **DEFAUTS.SYM**, which contains default symbolic names for additional controller resources. You can use them when writing a Quickstep program. Appendix B contains a list of the default symbolic names.

We recommend that you write all new Quickstep programs using symbolic names that identify their function.

Symbolic Name	Controller Resource
ain_1	analog input - ain#1
aout_1	analog output - aout#1
col_1	data table column - col#1
ctr_5	counter - ctr#5
disp_5	display - disp#5
flag_3	flag - flag#3
in_3A	input normally open - in#3a
in_3B	input normally closed - in#3b
OUT_1_ON	output on - 1+
OUT_1_OFF	output off - 1-
motor_1	stepping motor - motor#1
reg_50	register - reg#50
servo_1	servo motor - servo#1
twhl_1	thumbwheel - twhl#1

Delay Instructions

The DELAY instruction causes the controller to proceed to a new step after a specified amount of time has passed. A time delay can be a specific amount of time, or it can be derived from another source, such as a thumbwheel or register.

You can use any of the following devices to specify the amount of time for the delay:

- An analog input specifying minutes, seconds, or hundredths of seconds.
- A value from registers 1 through 128 specifying minutes, seconds, or hundredths of seconds.
- An integer specifying hours, minutes, seconds, or milliseconds.
- The value from the Data Table specifying minutes, seconds, or hundredths of seconds.
- A thumbwheel specifying: hours and minutes, minutes and seconds, or seconds and hundredths of seconds

To set a time delay for a specific time, the format of the DELAY instruction is as follows:

```
delay 2 sec 300 ms goto next
```

This instruction sets the delay for 2.3 seconds.

When using an integer to specify a millisecond time delay, the least significant digit should always be zero. This is because the time resolution of the controller is in ten milliseconds.

To set a time delay using the value in a register or from an analog input, the format of the DELAY instruction is as follows:

```
delay reg_15 sec goto next
delay ain_6 min goto next
```

To set a time delay using a thumbwheel, the format of the DELAY instruction is as follows:

```
delay twhl_2 ssff goto next
```

The value ssff represents the four digits of the thumbwheel, ssff = seconds and hundredths of seconds, mmss = minutes and seconds, hhmm = hours and minutes.

Monitor Instructions

The Monitor instructions allow the controller to monitor inputs, flags, and stepping and servo motors. Monitor instructions can check any of the following states:

- Check if an input is open or closed
- Check if a flag is set or clear
- Check if a stepping or servo motor is running or stopped
- Use a Boolean expression to check or monitor the state of any combination of inputs, flags, stepping motors, and servo motors

Monitoring Inputs

MONITOR INPUT checks the status of one or more of the controller's digital inputs. The controller goes to a new step if a specified condition is met. For example:

```
monitor in_12A goto next
```

monitors the state of digital input 12. The switch is monitored as a normally open switch, and the controller proceeds to the next step only when the switch closes. You can also monitor inputs as normally closed.

```
monitor in_12B goto next
```

NOTE: To check the status of an analog input, use an IF instruction.

Monitoring Flags

MONITOR FLAG checks the state of a flag. You can detect whether the flag is set or clear. For example,

```
monitor flag_12:set goto next
```

monitors the state of flag 12. The controller proceeds to the next step only when the flag is set.

TEST AND SET FLAG is a special flag monitoring instruction. It is often used in multi-tasking, when two mechanisms are contending for the same resource, for example, two separate robotic arms that are invading the same workspace. In this case, TEST AND SET FLAG tests a flag to see if it is clear, representing the workspace being empty. If the flag is set, the controller does not take any action. If the flag is clear, the controller immediately sets the flag and proceeds to the next step.

```
testandset flag_12 goto next
```

The testing of the flag is essentially simultaneous with the setting of the flag. This allows you to use the flag to arbitrate the use of the workspace without any risk of collision.

Monitoring Stepping Motors and Servos

MONITOR MOTOR and MONITOR SERVO check the status of a motor control module. The stepping or servo motor can be monitored to see if it is running or stopped. The controller does not go to a new step until the control module indicates that the motor is in the desired state.

```
monitor motor_2:running goto next
```

```
monitor servo_1:stopped goto next
```

NOTE: The definition of stopped for the purposes of the monitor instruction is that the control module has completed its execution of the last motion. For example, the controller has no knowledge of any continued motor motion caused by an external force on a motor.

Monitoring Boolean Statements

MONITOR BOOLEAN allows you to perform the monitoring of complex combinations of inputs, flags, stepping motor, and servo motor states. Multi-level nesting of Boolean functions is supported. You can use the following Boolean algebra functions in a monitor statement.

- AND — Requires all listed states to be true.
- OR — Requires any one or more of the listed states to be true.
- XOR — (Exclusive or) requires exactly one of the listed states to be true.
- NAND — Requires any one or more of the listed states to be false.
- NOR — Requires all listed states to be false.
- NXOR — (Not exclusive or) requires either more than one or none of the listed states to be true.

The following examples show how to use Boolean statements:

```
monitor (and in_16A in_19B in_24A) goto next
```

requires that inputs 16 and 24 be closed and input 19 be open for the statement to be true.

```
monitor (or in_5A (and servo_1:stopped servo_2:stopped))
goto next
```

requires either input 5 to be closed or both servos to be stopped.

Store Instructions

A simple STORE instruction copies numeric data from one location in the controller to another location. For example,

```
store twhl_1 to reg_307
```

You can use a numeric value from the following sources as input for a STORE instruction:

- An analog input
- A column in a data table
- A counter (registers 1-8)
- A numeric register, including any special purpose registers having read access
- The servo error of a servo motor
- The position of a servo motor
- A thumbwheel preset
- A fixed number from -2,147,483,648 to +2,147,483,647

You can store numeric data to the following destinations:

- Analog output
- Counter
- Numeric display
- Numeric register, including any special purpose registers having write access
- Data Table row and column

In addition, a STORE instruction can perform arithmetic, Boolean, modulo, and rotate instructions as follows:

- Perform mathematical operations between any two sources of numeric data and store the result in any destination for numeric data.

```
store reg_307 + twhl_1 to reg_934
```

  Math operations performed are addition (+), subtraction (-), multiplication (∗), and division (/)

- Perform a modulo operation. A modulo operation stores the remainder of an integer division in any destination that accepts a numeric value.

  ```
  Store 10 mod 6 reg_15
  ```

  The value stored in register 15 is 4.

- Perform bit-wise Boolean algebra on numbers using Boolean STORE instructions. You can use the Boolean STORE instruction to mask data from 8-, 16-, and 32-bit data sources.

  The following instruction reads the binary value represented by the state of the controller's first set of 32 inputs and performs a boolean AND operation

with the number 4095. As a result of this instruction, the binary representation of the first 12 inputs only is stored in register 10.

```
store reg_10001 and 4095 to reg_10
```

The value in reg_10001 is: 0100 0110 1111 0100 0000 1111 1001 1101

4095 in binary is: 0000 0000 0000 0000 0000 1111 1111 1111

The resulting number stored in register 10 is: 0000 0000 0000 0000 0000 1111 1001 1101

The following instruction performs a bit-wise Exclusive OR operation between the binary value of the first 32 inputs and the binary value of the second 32 inputs, storing the result in register 10. This instruction compares the status of these two groups of inputs. Only when the two groups are identical does register 10 contain a zero. The value in register 10 can be tested with an IF instruction.

```
store reg_11001 xor reg_11002 to reg_10
```

The following boolean instructions can be performed: AND, OR, XOR (Exclusive or), NAND, NOR, NXOR (Not exclusive or), and NOTAND. For additional information on using Boolean STORE instructions see, Chapter 3, *Performing Boolean Operations on Binary Numbers*. For additional examples, refer to the Application Note, *Bit Level Operators and CTC Controllers*.

- Rotate the bit pattern in a register right or left one or more bits. The rotate instruction replaces the value in a bit (either 1 or 0) with the value in the bit preceding or following it. The first bit in the series inherits the value of the last bit in the series. For example:

  The number 33 is stored in register 75 and forms the following bit pattern:

  ```
  0000 0000 0000 0000 0000 0000 0010 0001
  ```

  After the controller executes the instruction `store reg_75 rol 1 to reg_75`, the bit pattern in the register is as follows:

  ```
  0000 0000 0000 0000 0000 0000 0100 0010
  ```

  You can rotate the bit pattern several places, for example,

```
store reg_75 rol 4 to reg_75
```

Storing data to the Data Table

You can store data to a specific row and column in a Data Table using the following set of instructions:

`store 5 to reg_131` (selects row 5 of the Data Table)

`store 8 to reg_132` (selects column 8 of the Data Table)

`store 368 to reg_9000` (stores 368 in column 8 row 5 of the Data Table)

For 2600 series and higher controller models, you can also store data to a specific row and column in a Data Table using the following instructions:

`store 5 to reg_126` (selects row 5 of the Data Table)

`store 368 to col_8` (stores 368 in column 8 row 5 of the Data Table)

For more information, see *The Data Table* in Chapter 3.

Flag Instructions

Flags are memory elements within a controller that can be either set or clear and are used to store yes/no types of information. You can also use them to store information from one part of a machine's cycle to another or during multi-tasking to communicate from one task to another. Flags can also be used as elements in a shift register. There are 32 flags available in a controller.

- The SET FLAG instruction sets a flag. The flag remains set until it is cleared or until the controller is reset or the power to it is turned off.

  ```
  set flag_8
  ```

- The CLEAR FLAG instruction clears a flag that was previously set.

  ```
  clear flag_8
  ```

- The SHIFT FLAG and ROTATE FLAG instructions treat a series of flags as a shift register. They automatically move information from a flag to the next flag within a specified range of flags. Any sequential series of flags can be shifted, and you can establish several shift registers.

  — The SHIFT FLAG instruction replaces the status of a flag (either set or clear) with the status of the flag preceding or following it. The first flag in the series is automatically cleared.

  — The ROTATE FLAG instruction replaces the status of a flag (either set or clear) with the status of the flag preceding or following it. The first flag in the series inherits the status of the last flag in the series.

  To shift or rotate a series of flags up

  ```
  shift flag_5 >> flag_10
  rotate flag_5 >> flag_10
  ```

  To shift or rotate a series of flags down

  ```
  shift flag_11 << flag_20
  rotate flag_5 << flag_10
  ```

  To shift or rotate a series of flags multiple shifts

  ```
  shift flag_11 << flag_20 2 times
  rotate flag_5 >> flag_19 3 times
  ```

For information on monitoring flags (MONITOR FLAG and TEST AND SET FLAG), refer to *Monitor Instructions* in this chapter.

If Instructions

The IF instructions allow the controller to perform a comparison between any two numeric quantities within the controller. If the comparison is true, the controller goes on to a new step. For example:

```
if reg_15 >= twhl_2 goto next
```

compares the value stored in register 15 to thumbwheel 2. If the value in register 15 is greater than or equal to the value entered on external thumbwheel 2, the controller goes to the next step

IF instructions can perform any of the following comparisons:

- Greater than, >
- Less than, <
- Greater than or equal to, >=
- Less than or equal to, <=
- Equal to, =
- Not equal to, <>

An IF instruction can draw its comparison values from the following numeric resources:

- Analog input
- Column of a data table
- Integer from -2,147,483,648 to +2,147,483,647
- Register
- Counter
- Servo motor position or error
- Thumbwheel

For example,

```
if ain_2 > reg_25 goto next

if ctr_2 >= 7500 goto next

if servo_1:error >= reg_38 goto next

if servo_2:position <= col_3 goto next
```

Goto Instructions

The GOTO instruction tells the controller which step to execute next. You can tell the controller to proceed to any step including the current step.

The following examples are three different ways to use GOTO instructions:

- This instruction tells the controller to jump to the step named SHUT_DOWN, where ever the step is in your program.

  ```
  goto SHUT_DOWN
  ```

- A special form of the GOTO instruction tells the controller to proceed to the next step in numeric sequence.

  ```
  goto next
  ```

- It is possible to have a GOTO instruction which jumps back to the same step. This results in the controller re-executing any instructions which normally executed only once upon entering the step, for example STORE instructions.

  ```
  goto TEST_AGAIN
  ```

  This tells the controller to re-execute the step TEST_AGAIN.

You can use goto instructions to move around within a program. However, you must take care when using GOTO instructions in programs or you can write a recursive program. Recursion results whenever a task in a program encounters an instruction to restart the same task.

For additional information on recursion, see the section on *Multi-Tasking* in Chapter 1.

Multi-tasking Instructions

You can control multi-tasking functions using the DO, DONE and CANCEL instructions.

The DO instruction can initiate from one to eight tasks. Each task is an independent program beginning at the step specified in the DO instruction. If multiple tasks are started, they run independently and asynchronously of each other. The format of a DO instruction is:

```
do (FABRICATE_PART FAULT_MONITOR OVER_PRESSURE) goto START
```

A task continues until it encounters a DONE instruction. DONE signals the completion of a task. After it encounters a DONE instruction for each task started by the DO instruction, the controller proceeds to the destination specified by the DO instruction.

A DO instruction can also loop through a task or tasks up to 99 times. For example:

```
do (ONLOAD OFFLOAD FAULT_MONITOR) 50 times goto next
```

The controller executes all the tasks until each encounters a DONE instruction and then begins all the tasks over again until it reaches the number of iterations specified.

CANCEL shuts down all other tasks.

```
cancel other tasks
```

For additional information on multi-tasking, refer to the section on *Multi-Tasking* in Chapter 1.

Counter Control Instructions

The counter instructions control the eight internal counters in a controller. These instructions can start a counter and increase or decrease the value in one of the controller's counters. They also reset, enable, and disable any of the controller's counters.

- START COUNTER initializes a counter. The counters overlay the first eight registers (i.e., counter No. 1 = register No. 1). When starting the counter, you can assign three of the controller's inputs to perform the count-up, count-down, and reset functions:

  ```
  start ctr_1 up (in_5A) down (in_6B) reset (in_7A)
  ```

  This initializes the counter and assigns functions to three of the controller's inputs. These inputs continue sending input signals to the counter until the counter is re-initialized or disabled.

  — Up (in_5A) specifies that input No. 5 is being used for the count up function and increments the counter for each switch closure. The A specifies that the input is a normally-open input. A normally-open input means that the count occurs when the switch closes.

  — Down (in_6B) specifies that input No. 6 is being used for a count down function and decreases the value in the counter by one for each time the switch opens. The B specifies that the input is a normally closed input. A normally-closed input means that the count occurs when the switch opens.

  — Reset (in_7A) resets the value in the counter to zero when the switch closes.

  ```
  start ctr_1 up (in_5A:30) down (in_6B:30) reset (in_7A:100)
  ```

  The debounce time (shown as `:30`) allows you to use mechanical switches for counting, even though they normally bounce a number of times when they actuate. Although there is no increase in reaction time, the maximum count frequency is reduced.

NOTE: Debounce is not available in the model 2600 and 2700 series controllers. The controller ignores the debounce parameters.

- COUNT UP adds one to the current value in the counter.

  ```
  count up ctr_3
  ```

- COUNT DOWN subtracts one from the current value in the counter.

  ```
  count down ctr_3
  ```

- RESET returns the value in the counter to zero.

  ```
  reset ctr_3
  ```

- DISABLE temporarily disables a counter so that it does not accept any count up, count down, or reset pulses until it is enabled.

  ```
  disable ctr_3
  ```

- ENABLE reactivates a counter that was temporarily disabled.

  ```
  enable ctr_3
  ```

Stop Instructions

The STOP instruction tells the controller to stop running the program. This usually causes the mechanism being run by the controller to stop.

All controller resources remain in the state they were in before the controller executed the STOP instruction. To restart the controller and maintain the current state of the controller resources, use the start dedicated input function. For a description of the dedicated input functions, see *Using Dedicated Inputs* in Chapter 3.

```
stop goto stepname
```

When the controller is restarted following the execution of a STOP instruction, the program will continue with the step specified in the STOP instruction. If the program was multi-tasking at the time the STOP instruction was executed, the other tasks also continue from where they have stopped.

WARNING!

Stepping and servo motor motions which were in progress at the time the controller executed the Stop instruction will continue to completion. In the event of a velocity mode instruction, a motor will continue turning indefinitely. Also, digital or analog outputs retain their current state, which may result in a continuous on state for a mechanism. All such conditions should be evaluated for possible dangers when using the STOP instruction.

Stepping Motor Instructions

Using the stepping motor instructions, you can set the motion parameters for a stepping motor, start it turning, and specify when and how it will stop. You can also establish a zero or home reference position for a stepping motor. The stepping motor instructions are:

- Profile Motor
- Turn Motor
- Zero Motor
- Search and Zero Motor
- Stop Motor
- Monitor Motor (described in the section *Monitor Instructions*)

You can define motion parameters for up to 16 stepping motors.

NOTE: Currently, the stepping motor instructions are used only with the model 2205 Stepping Motor Control Module. Other stepping motor control modules use the more flexible servo motor instructions. Refer to the *Installation and Applications Guide* for your stepping motor for more information

For information about stepping motor applications and special functions, see *Using Stepping Motors* in Chapter 3. For programming examples, see Appendix A.

Profile Motor Instruction

The PROFILE MOTOR instruction sets the motion parameters for a stepping motor as follows:

- Stepping Mode — Specifies either half-step mode or full-step mode with one or two coils on. You can only use this parameter to specify the stepping mode using the on-board drivers of the model 2205. We recommend using half-step mode when using an on-board driver since it is smoother and has less resonance.

  When using an external drive, the external drive determines the stepping mode.

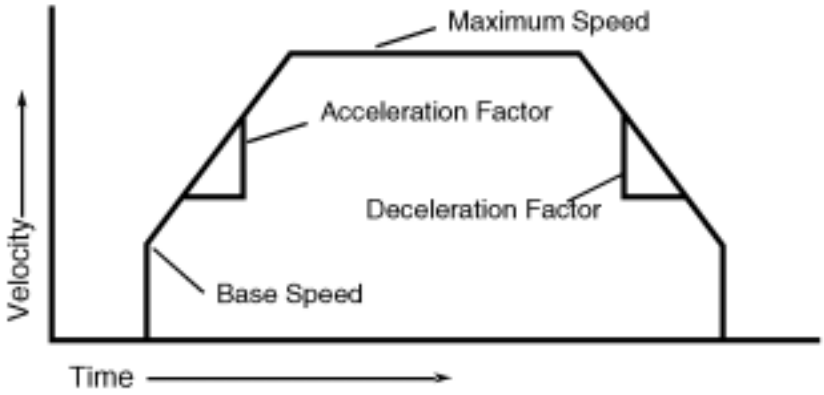
- Base Speed (basespeed)– defines the speed, in steps per second, at which the motor motion begins. The motor must be capable of virtually instantaneous acceleration to this speed.

- Maximum Speed (maxspeed) – defines the speed, in steps per second, at which the motor ceases accelerating during a motion. The maximum speed is maintained until the stepping motor control module calculates that the motor must begin to decelerate.

- Acceleration Factor (accel) – determines the rate at which the motor accelerates to maximum speed. It is expressed as a relative factor from 1 to 500. 500 represents instantaneous acceleration. Typical values range from 200 to 475.

- Deceleration Factor (deceleration) – determines the rate at which the motor decelerates to a stop. It is expressed as a relative factor from 1 to 500. 500 represents instantaneous deceleration. Typical values range from 200 to 475.

The following illustration shows the velocity profile for a stepping motor.



Any of the numeric parameters for a stepping motor can be drawn from any of the controller's numeric resources, instead of being expressed as a fixed number. For additional information, refer to the section on the STORE instruction.

The following example is for a PROFILE MOTOR instruction for a motor using half step mode:

```
profile motor_2 (half) basespeed=reg_200 maxspeed=reg_201
  accel=reg_202 decel=reg_203
```

The following example is for a PROFILE MOTOR instruction for a motor using full step mode with two coils on:

```
profile motor_1 (2 coils) basespeed=reg_10 maxspeed=reg_11
  accel=reg_12 decel=reg_13
```

An alternative form of a PROFILE MOTOR instruction allows you to establish the zero position of the motor as its present position.

```
profile and zero motor_1 (2 coils) basespeed=reg_10
  maxspeed=reg_11 accel=reg_12 decel=reg_13
```

The PROFILE MOTOR instruction does not start the motor motion. To initiate motion use the TURN MOTOR instruction. You may respecify the motor profile parameters any number of times in the same program, but only when the motor is stopped. If you do not wait until it stops, a software fault results.

Zero Motor and Search and Zero Instructions

The ZERO MOTOR and SEARCH AND ZERO MOTOR instructions set a zero or home reference position for a stepping motor as follows:

- The ZERO MOTOR instruction sets the current position of the stepping motor as its zero or home reference position. The stepping motor must be stopped when the controller executes this instruction.

  ```
  zero motor_1
  ```

- The SEARCH AND ZERO MOTOR instruction starts the motor turning in either a clockwise or counter clockwise direction at the base speed until the controller's motor control module senses a contact closure on the home limit

switch input. At that time, the motor stops and the current position is set as the zero point. You can specify whether the motor turns in a clockwise or counter clockwise direction.

```
search cw and zero motor_1
```

Turn Motor Instruction

TURN MOTOR instructions initiate motor motion and define either the distance to be traveled or the new desired coordinate position. The controller must have executed a profile instruction at some time before the first turn instruction for any given axis. If it hasn't, the controller reports a software fault stating "motor not ready." The motor uses the parameters from the last-executed PROFILE MOTOR instruction for that axis.

The TURN MOTOR instruction defines the distance the motor travels using one of the two following methods:

- Relative — Turns the motor either clockwise or counter clockwise a specified number of steps from the current motor position.

```
turn motor_5 ccw 750 steps
```

  This instruction has the motor move 750 steps in the counter clockwise direction. The current position of the motor is irrelevant to this instruction, since the distance of the motor motion is defined relative to the motor's current position. However, even in relative moves the controller keeps track of the absolute position of the motor.

- Absolute — Turns the motor a calculated number of steps based on the distance from a predetermined zero or home position. This type of turn instruction is more powerful, especially for applications involving the positioning of an actuator. Using an absolute turn instruction the stepping motor control module maintains the motor's current absolute position in memory. For example,

```
turn motor_1 to 1500
```

  This instruction has the motor turn to a position 1500 steps clockwise from a preestablished reference (zero or home) position. Since the new position is expressed in terms of an absolute coordinate, this form of instruction relieves the designer of the task of maintaining knowledge of the motor's current position and calculating the required motion to reach a new position.

NOTE: Do not issue PROFILE or TURN MOTOR, ZERO, or SEARCH AND ZERO instruction when a stepping motor is in motion. Program a `monitor motor:stopped` instruction before any new instruction for the motor. Failure to do so may result in a CPU software fault, halting execution of the program.

Stop Motor Instruction

The STOP MOTOR instruction stops the generation of pulses to a stepping motor. Use this instruction with caution, since the inertial load often carries the motor forward and consequently the absolute position may be lost.

```
stop motor_2
```

WARNING: Do not use a STOP MOTOR instruction to implement an emergency stop function where danger to human safety or substantial economic damage exists. Such functions should be implemented with independent, external systems.

Servo Motor Instructions

Using the servo motor instructions, you can set the motion parameters for a servo motor, start it turning, and specify when and how it will stop. You can also establish a zero or home reference position for a servo motor. The servo motor instructions are:

- Profile Servo
- Turn Servo
- Zero Servo
- Search and Zero Servo
- Stop Servo
- Monitor Servo (described in the section *Monitor Instructions*)

You can define motion parameters for up to 16 servo motors.

The Model 2206 Stepping Motor Control Module also uses servo motor instructions. For information about servo motor applications and special functions, see *Using Servo Motors* in Chapter 3. For programming examples, see Appendix A.

Profile Servo Instruction

The Profile Servo instruction sets the motion parameters for a servo as follows:

- Maximum Speed (max) — Establishes the maximum speed of the servo. It is defined in encoder pulse edges (steps) per second (fully decoded).

NOTE: The highest permissible maximum speed varies according to the module used. Refer to the installation guide for your module.

- Acceleration Rate (accel)— Specifies the acceleration rate of the servo. Defined in encoder pulse edges (steps) per second per second (steps/sec^2). This parameter also sets the deceleration rate.

  If you want the acceleration and deceleration values to be different, store the deceleration value to a special purpose register. The following example sets the deceleration rate:

  ```
  profile servo_1 max=50000 accel=100000
  store 20000 to reg_15006
  ```
  (axis No. 1 deceleration register)

- P (Proportional) Filter - Specifies the factor applied to the sensed position error to create a correction signal. It is expressed as a multiplication factor from 0 to 255.
- I (Integral) and D (Derivative) Filters - Determine the characteristics of the built-in digital compensation filter.

NOTE: Check the Applications Guide for your servo control module. Some servo modules give you additional compensation filter choices.

- Holding Mode — Specifies the status of the servo when stopped, using one of the following parameters:

  — Servo at position - Once the servo reaches the desired position, the actuator will continuously seek this position. If the actuator is forced from its position, the servo control module sends a correction signal to attempt to correct the perceived error.

— Deadband of __ at position - The servo control module senses position errors but does not correct them unless the error is out of the range of the Deadband.

— Off at position - Once the servo reaches position no further corrective action occurs. This allows manual adjustment or another external force to change the position of the servo.

— None - Indicates that the controller should use the holding mode specified in a previous PROFILE SERVO instruction.

The PROFILE SERVO instruction does not start the servo motion. To initiate motion use the TURN SERVO instruction. You may respecify the servo profile parameters any number of times in the same program. Any of the numeric parameters for a servo motor can be drawn from any of the controller's numeric resources, instead of being expressed as a fixed number. For additional information, refer to the section on the STORE instruction.

Unlike a stepping motor, you can execute a new PROFILE SERVO instruction while the servo is still in motion. You can change any parameter except the acceleration rate.

```
profile servo_3 servo at position maxspeed=15000 accel=35000
P=10 I=95 D=50
```

Zero Servo

The ZERO SERVO instruction sets the current position of the servo as its zero or home reference position.

```
zero servo_1
```

Search and Zero Servo

The SEARCH AND ZERO SERVO instruction sets a zero or home reference position for a servo. The SEARCH AND ZERO SERVO instruction starts the servo turning at the rate specified in the PROFILE SERVO instruction until the servo control module senses a contact transition on its home limit switch input (dedicated input). Depending on the model of the servo control module you have, the instruction functions differently. For examples showing how to establish and find a home position, see *Using Servo Motors* in Chapter 3.

```
search and zero servo_1
```

Turn Servo Instruction

The TURN SERVO instructions initiate a new servo motion. The controller must have executed a PROFILE SERVO instruction to define the motion parameters. TURN SERVO defines the distance the servo travels using one of the following methods:

- Absolute — Turns the servo a calculated number of steps based on the distance from a predetermined zero position. For example,

  ```
  turn servo_1 to 1500
  turn servo_2 to 1500 on_start_switch
  ```

  The second absolute distance instruction also requires a contact closure on the servo module's dedicated start input before the servo motion begins.

NOTE: step = encoder signal transition

- Relative — Turns the servo clockwise or counter clockwise a specified number of steps from the current motor position. For example,

```
turn servo_5 cw 70000 steps
```

- Velocity — Begins continuous clockwise or counter clockwise motion. The servo remains in motion until the controller executes a STOP SERVO instruction or the servo control module senses a stop input signal. For example,

```
turn servo_5 ccw
```

NOTE: Do not issue another TURN SERVO instruction for a servo while the servo is still in motion. If the servo is still turning, the controller reports a software fault (servo not ready) and halts execution of the program. Before issuing another turn instruction, you should program a `monitor servo:stopped` instruction prior to any subsequent turn instruction.

Stop Servo Instruction

The STOP SERVO instruction brings the servo to a halt. You can choose one of the following methods to stop the servo:

- Soft Stop — Causes the servo to stop at the deceleration rate specified in the last profile instruction.

```
stop (soft) servo_2
```

- Hard Stop — Causes the controller to attempt to stop the servo instantly. However, because of momentum (caused by the inertial load), the servo does not stop instantly and consequently the absolute position may be lost and instability may result.

```
stop (hard) servo_1
```

In either case, you should use a monitor instruction before issuing another turn instruction.

WARNING: Do not use a STOP SERVO instruction to implement an emergency stop function where danger to human safety or substantial economic damage exists. Such functions should be implemented with independent, external systems.

Using the Quickstep Programming Language

Contents

Introduction

CTC's automation controllers support high-level instructions for a number of internal resources, specialized I/O, and motion control devices to make writing an automation program with Quickstep easier. These resources include:

- Counters
- Flags
- Numeric registers
- Data Table
- Special purpose registers
- Stepping motor control modules
- Servo motor control modules
- Analog input/output modules
- Specialized input/output features
- Thumbwheel array and numeric display interface modules
- Dedicated inputs on the controller
- High-speed counting modules

Counters

CTC controllers can be programmed to automatically count pulses from the controller's inputs. You can use the first eight numeric registers as counters, and each counter can be programmed to accept input signals from three of the controller's inputs.

Counters work in the background. Once started, a counter operates much like an external, independent device within the controller. You can assign counting inputs. The counter continuously monitors the inputs for counts, while the controller proceeds on through subsequent steps of its program. You can assign any of the controller's inputs to a counter.

Possible uses for counters include:

- In packaging applications, counting a product passing by a sensor.
- Using a counter wheel to count the linear footage of a material being metered out.
- Detecting the motion of a workpiece over a specific distance.
- Tracking the quantity of a product in a holding area.

Programming Counters

Counter control instructions can be programmed to perform the following functions:

- Start counter — initializes a counter, resets it to zero, and, optionally, assigns inputs to the counter.
- Count up — adds one to the accumulated count.
- Count down — subtracts one from the accumulated count.
- Reset — resets the accumulated count to zero.
- Disable — disables a counter temporarily so that it does not accept any count up or count down pulses.
- Enable — reactivates a counter that was temporarily disabled.

The maximum range of a counter is -2,147,483,648 to +2,147,483,647.

Each counter input can be selected to take effect on a normally-open or normally-closed basis. A normally-open operation means that the count occurs when the switch closes, and a normally-closed operation means that the count occurs when the switch opens.

Since counters are treated similarly to numeric registers (they overlap with the first eight numeric registers), you can use any instruction that references a numeric register to reference a counter. For example, you can display the value of a counter on an external numeric display:

```
store ctr_3 to dis_1
```

Once a counter is started with a START COUNTER instruction, it continues to run, and its total accumulated count can be tested at any time with an IF instruction.

```
if ctr_1 >= 1500 goto next
```

Debouncing

You can program an independent debounce interval for each counter input. Debouncing allows the counter to register the multiple pulses typically received from a mechanical switch contact as a single count.

NOTE: Debounce is available in 2200XM and 2800 series controllers only.

When a mechanical switch closes, the switch contacts will bounce for a period of time (typically from one to 15 milliseconds). The bouncing is due to the imperfect electromechanical coupling that occurs when two pieces of metal come together under spring tension.

Debouncing allows a programmable interval from one to 250 milliseconds for a switch closure to settle. No degradation of reaction time occurs when using debouncing; a contact closure is instantly recognized and registered by the counter. However, from that time forward no further counts are allowed until the debounce time has expired, which allows the switch to settle.

Counting Speeds

One advantage of the internal counters is that they can attain greater counting rates than those attained from numeric registers. This faster counting rate is obtained because the controller handles the counters as an independent activity, unrelated to the execution of your Quickstep program.

The counting speed capabilities are related to the number of inputs assigned to all counters at any given time. This includes the count up, count down, and reset inputs of active counters. The following table lists the maximum counting rates:

1 to 3 inputs:	500 Hz (assumes a 50% duty cycle)
4 to 6 inputs:	250 Hz
7 to 8 inputs:	166 Hz

If your application requires a higher counting speed, you can obtain a separate high speed counter module.

Example

The following example uses a counter in a simplified application involving the metering of a fixed amount of material. In this application output 1 is used to control the motor that feeds the material.

```
[1] SET_INITIAL_STATUS
    ;;; This step contains instructions that set the
    ;;; initial conditions for each iteration of our
    ;;; counter.  The controller turns off all the
    ;;; digital outputs. It then starts the counter so
    ;;; it can begin measuring the material once the
    ;;; motor starts in the next step. The last
    ;;; instruction is a 2.1 second time delay. When
    ;;; the delay is up, the controller moves to the
    ;;; next step.
    ;;;
    ;;;     Travel = counter 1
    ;;;     Count_up = input 30
    ;;;     Count_Dn = input 31
    ;;;     Reset_Inp = input 32
    ;;;     Feed_Motor_On = output 1 on
    ;;;     Feed_Motor_Off = output 1 off
```

```
    ;;;     Cut_On = output 2 on
    ;;;
    ____________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________
    start Travel up(Count_Up) down(Count_Dn) reset
     (Reset_Inp)
    delay 2 sec 100 ms goto next

[2] START_MOTOR
    ;;; In this step we turn on the output that
    ;;; controls the feed motor for the material. The
    ;;; If instructions continuously monitors the
    ;;; counter, and when it reaches or exceeds 1500,
    ;;; the controller proceeds to the next step.
    ;;;
    ____________________________________
    Feed_Motor_On
    ____________________________________
    if Travel >= 1500 goto next

[3] DELAY
    ;;; In this step the controller turns the output
    ;;; that controls the feed motor off, and, after a
    ;;; 1 second delay, it proceeds to the next step.
    ;;;
    ____________________________________
    Feed_Motor_Off
    ____________________________________
    delay 1 sec goto next

[4] CUT
    ;;; In this step the controller turns on output 2
    ;;; to cut the material and goes to the first step
    ;;; in the program.  At the first step, the
    ;;; controller turns off all the outputs, restarts
    ;;; the counter, and waits for the time delay.
    ;;;
    ____________________________________
    Cut_On
    ____________________________________
    monitor Limit_Contact goto SET_INITIAL_STATUS
```

This example illustrates two important points:

1. The START COUNTER instruction automatically zeros the counter, and no additional instruction is necessary to reset the counter before each new cycle.

   This also means, if your application requires a continuous count throughout multiple cycles, the START COUNTER instruction must be outside the main program cycle to avoid resetting the counter.

2. When testing a counter with an IF instruction, you should use a greater than or equal (`if ctr_1 >= 1500`) comparison. Sometimes due to a rapid counting rate, more than one count is registered between two successive tests of the counter by the controller, and the target count is exceeded. The >= comparison still allows the test to be met.

Flags and Shift Registers

Flags are memory locations within the controller that can store yes/no types of information. CTC controllers have 32 flags.

At any given time flags are either set or clear. Using Quickstep instructions you can specify a flag's state as either set or clear and, in a subsequent instruction, change its state. Other instructions can monitor the state of a flag and send the controller to a new step if the flag is in the specified state. Quickstep also provides instructions to shift the data within a group of flags, allowing you to program the flags as a shift register.

Possible uses for flags are:

- Storing status information from one portion of a machine's cycle to another.
- Communicating information from one task to another when multi-tasking.
- Implementing a shift register for progressive assembly machines, e.g., index tables.
- Setting up handshaking when using computer data communications.

Flags are very straightforward in function. Once the state of a flag is changed with either a SET FLAG or CLEAR FLAG instruction, the flag remains in that state until a subsequent instruction changes its state or power is removed from the controller. When a controller is powered up, it automatically clears all flags. Resetting the controller or downloading a new program also resets all the flags.

You can use the information stored in a flag during one portion of a program's operation to affect the course of a later portion of the program. For example, you can set a flag in one step, program an instruction in a later step to monitor the flag, and proceed to a new step depending on the state of the flag.

Monitoring Flags

You can check a flag and determine whether the flag is in a specified state, either set or clear. If the test is met, the instruction sends the controller's program to a new step. For example,

```
monitor flag_18:set goto next
```

Using Monitor Boolean Instructions

The MONITOR BOOLEAN instruction allows you to monitor combinations of flags or combinations of flags, inputs, and motor states (running or stopped). MONITOR BOOLEAN supports nested combinations using the Boolean algebra functions. The following is a sample instruction:

```
monitor (or (and flag_5:set Motor_1:stopped)
 (and flag_12:set Motor_2:stopped)) goto next
```

This instruction requires one of two conditions to be true. Either flag 5 must be set and motor 1 be stopped, or flag 12 must be set and motor 2 be stopped.

Using Shift Registers

Shift registers consist of a sequential series of flags, (e.g., flags 5 through 10) and can be used to track the status of a series of different workpieces. After executing a shift instruction, the first flag in the range specified will be clear and the data in the last flag in the range will be lost. The format of a SHIFT FLAG instruction is

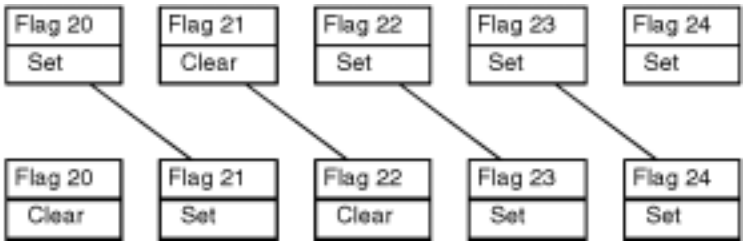
```
shift flag_20 >> flag_24
```

You can program multiple shifts with a single instruction:

```
shift flag_20 >> flag_24 3 times
```

The arrows within the instruction indicate the direction of data travel. It is possible to program a shift instruction that shifts the data in descending order:

```
shift flag_15 << flag_20
```



Shift registers are often used with machines that perform progressive operations on workpieces simultaneously. An index table, for example, can have a number of workstations designed to perform a series of progressive operations on a workpiece as it travels around a rotary table. Typically, at any given time there is a workpiece at each station. After all workstations have completed their cycles, the table rotates, carrying each workpiece to the next station in sequence.

In progressive operation machines there is often a requirement to remember some status information for each part in process, e.g., good part/bad part or type A part/type B part. Remembering this information is made more complex because the workpieces are continuously moving from station to station, new parts are being fed onto the table, and completed parts are being removed from the table.

Shift registers provide a mechanism for tracking the information along with the workpieces in process. To track the information you must first assign a flag, in sequence, for each successive workstation that contains a workpiece. The status of the flag, either set or clear, indicates the status of the workpiece residing in the associated station.

At the point in the program where the workpieces are advanced to the next workstation, you program an instruction such as shift flags 1>>8. This instruction causes the information within flags 1 through 8 to shift one position. In this manner, the controller transfers the status of flag 1 into flag 2, the status of flag 2 is transferred into flag 3, and so on.

If the status of a part is initially determined and loaded into the first flag being used, the information will follow the part throughout all subsequent operations. You can test the information at any workstation by using a MONITOR FLAG instruction to monitor the flag associated with that workstation.

Using Multiple Shift Registers

Since the action of a SHIFT FLAG instruction is limited to the range of flags specified in the instruction, you can create more than one shift register within the controller. For example, if flags 1 through 8 are used to store one piece of information for parts at eight workstations, you can use flags 9 though 16 to store a second piece of information for those same parts.

HINT: Using flags 11 through 18 may be easier to remember.

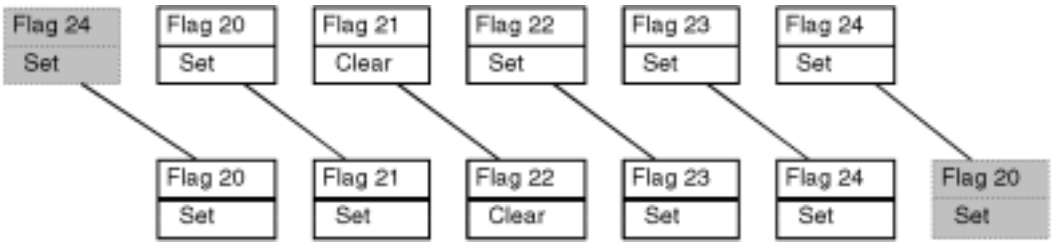
Each time the parts are indexed within the machine two shift instructions are then executed:

```
shift flags 1>>8
shift flags 9>>16
```

Rotating Flags in a Shift Register

ROTATE FLAG, like SHIFT FLAG, sets up a series of flags as a shift register. However, when the program executes a ROTATE FLAG instruction, the data from the last flag in the range specified is returned to the first flag in the range. The data travels in a circle and no data is lost as a result of the instruction. The format of a ROTATE FLAG instruction is:

```
rotate flag_20 >> flag_24
```



As with SHIFT FLAG you can also program multiple shifts and indicate the direction of data travel:

```
rotate flag_20 >> flag_24 2 times
rotate flag_15 << flag_20
```

Avoiding Mechanical Contention

Flags can also be used to avoid contention in instances where two or more mechanisms might, at some time, conflict with each other mechanically. An example of this is in a multi-tasking program that controls two pick-and-place mechanisms moving within the same work space in a machine. If the mechanisms are triggered asynchronously, a situation may arise where they would collide.

To avoid this, you can use a flag to indicate when the workplace is occupied. Each program controlling one of the pick-and-place mechanisms would be required to monitor this flag prior to entering the work space. The program would then set the flag, indicating that it will be occupying the work space. Once the mechanism is finished, the program would clear the flag.

Quickstep provides a special instruction that eliminates the very slight possibility of the following scenario: Task A monitors a flag, finds it clear and before setting the flag, a second task, Task B, also monitors the flag and finds it clear.

To eliminate this possibility, use the TEST AND SET Flag instruction. In this case, Task A monitors a flag and, if the flag is clear, automatically sets it. Task B cannot access the flag between the testing of the flag by Task A and its subsequent setting. The program can proceed without any chance of mechanical contention.

Example

The following example uses a flag to communicate information between two tasks in a multi-tasking program. The flag insures that one of the tasks does not proceed before the other task reaches a given step.

```
[1] INITIALIZE
    ;;; This step sets up the initial conditions for a
    ;;; program that cuts and places a piece of
    ;;; material on a stamping press. The cutting and
    ;;; placing operations take longer that the
    ;;; stamping press. The flag signals the stamping
    ;;; press that the material is in place. The first
    ;;; step turns off all digital outputs, monitors
    ;;; two inputs to make sure everything is ready,
    ;;; and proceeds to the next step.
    ;;;
    ;;;     Part_In_Place = flag 1
    ;;;     Stamp_Arm_Up = input 5 open
    ;;;     Feed_Part = motor 3
    ;;;     Stamp_Press_On = output 5 on
    ;;;     Stamp_Press_Off = output 5 off
    ;;;
    ____________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________
    monitor (and Stamp_Arm_Up Feed_Part:stopped) goto
   next

[2] START_TASKS
    ;;; This step starts the multi-tasking sequence.
    ;;;
    ____________________________________

    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    do (CUT_AND_PLACE STAMP_PART) goto START

[10] CUT_AND_PLACE
    ;;; This is the first step of a multi-step task.
    ;;; For the purposes of our example we are not
    ;;; interested in what happens here until step 20
    ;;; where the flag is set.
    ____________________________________
    Out_1_On
    ____________________________________
    monitor In_1A goto next
```

Steps 11 - 19 not shown here

```
[20] SET_FLAG
    ;;; This step sets flag Part_In_Place. This
    ;;; notifies task STAMP_PART to proceed with its
    ;;; next step which is stamping the part.
    ;;;
    ________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ________________________________
    set Part_In_Place
    goto CUT_AND_PLACE

[50] STAMP_PART
    ;;; This step starts the task that controls the
    ;;; stamp press. It begins with a monitor flag
    ;;; instruction. Once the flag is set, the
    ;;; controller proceeds to the next step in the
    ;;; task.
    ;;;
    ________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ________________________________
    monitor Part_In_Place:set goto next

[51] ACTIVATE_STAMPING_MECHANISM
    ;;; This step turns on the output that operates the
    ;;; stamp press. For the purposes of the example we
    ;;; will not show any more steps in this task.
    ;;;
    ________________________________
    Stamp_Press_On
    ________________________________
    delay 2 sec 500 ms goto next
```

Numeric Registers

Numeric registers are storage locations within the controller for numbers. They are capable of storing numbers from -2,147,483,648 to +2,147,483,647. The controller can use the numbers stored in numeric registers in arithmetic operations. There are two types of registers: general purpose registers and special purpose registers. Special purpose registers can store numbers, but are reserved for special functions.

CTC controllers have two types of general purpose registers: volatile and nonvolatile. Volatile registers lose their data when power is removed from the controller. Nonvolatile registers retain their data when power is removed from the controller.

The number of volatile registers differs depending on the model of your controller. For information on this, refer to your installation guide. For a list of special purpose registers, refer to the *Register Reference Guide*.

Possible uses for numeric registers are

- Storing intermediate results from mathematical equations.
- Maintaining counts of program cycles, parts processed, etc.
- Storing learned (experienced) parameters within intelligent adaptive programs, e.g., motor positions.
- Maintaining machine history information (e.g., total cycles) in nonvolatile registers.

Using Numeric Registers in Quickstep Programs

A broad array of instructions are available for storing numbers into registers and retrieving them. These instructions range from the STORE instruction (`store thwl_3 to reg_10`) to motion control instructions (`profile servo_1 maxspeed=reg_15`).

You can program any instruction that incorporates numeric data to derive data from a numeric register. This allows you to store calculated data in a register (`store ain_5 * 100 to reg_10`) and then use it in a subsequent instruction (`turn Motor_1 ccw reg_10 steps`).

Quickstep allows you to use the same register as both an operand in a mathematical calculation and as the destination for the result (`store reg_20 + 1 to reg_20`).

When operating parameters (e.g., motor speed and positions, time delays) for a machine are transferred from an external computer, the numeric registers are a convenient location for transfer of the information. The computer can load the data into a specific register and an instruction in the controller's program can reference that register (`delay reg_15 sec goto next`).

Nonvolatile Registers

Data stored in nonvolatile registers is maintained even during power-down periods. This allows you to store long-term data, such as weekly, monthly, or lifetime cycle counts.

CAUTION: The provision for maintaining data in nonvolatile registers varies by controller model. Some controllers use lithium cell batteries. Refer to the specifications for your controller for battery life information.

A potential use of nonvolatile registers is to write a program that is capable of fine-tuning certain parameters (time delays, motor speeds) and sensing the effect on the process being controlled. You can automatically store optimum, experimentally-determined values and continuously update them in nonvolatile registers. These parameters could potentially automatically adjust the machine to accommodate variations in tooling, lubrication, etc. over a long period of time.

You can also use the nonvolatile registers to store parameters from an external device, such as a computer, which are changed infrequently.

Using Registers

The following paragraphs are examples showing different types of instructions using registers.

- Transferring numeric information from one location to another within the controller.

```
store reg_15 to disp_1
store ain_3 to reg_10
```

- Performing a mathematical operation and storing the result in a register.

```
store twhl_1 * 25 to reg_10
store reg_45 + 500 to reg_45
```

- Performing a relational test (>, <, =, >=, <=, <>) using the value in a register.

```
if reg_36 >= 1500 goto next
if reg_51 < reg_50 goto next
```

- Specifying a time delay.

```
delay reg_14 sec goto next
```

  Only registers 1 through 128 can be used for time delays.

- Storing the operating parameters for a stepping motor or servo motor.

```
profile Motor_1 (half) basespeed=reg_10 maxspeed=
 reg_11 accel=reg_12 decel=reg_13
```

  These parameters can be calculated in a computer and transferred to the controller.

- Specifying the number of steps for TURN MOTOR or TURN SERVO instruction, if desired.

  For a relative turn:

```
turn Motor_1 ccw reg_38 steps
```

  For an absolute turn:

```
turn Servo_2 to reg_18
```

Example

The following program is designed to interact with an external computer, using one of the controller's registers as a means of communicating new positions for a servo motor.

```
[1] INITIALIZE
    ;;; In this program the computer can communicate a
    ;;; new coordinate position for the servo using
    ;;; either the serial protocols or an Ethernet
    ;;; connection. The new coordinate is stored in a
```

```
    ;;; register. After it has sent the coordinate, the
    ;;; computer sets a flag, indicating that the data
    ;;; is ready.
    ;;;
    ;;; The first step sets up the profile for the
    ;;; servo motion, using data stored in several
    ;;; nonvolatile registers. This data can be stored
    ;;; by the designer of the program and tuned as
    ;;; required via serial or network communication
    ;;; ports.
    ;;;
    ;;;     Servo_1 = servo 1
    ;;;     New_Pos = flag 1
    ;;;     Servo_Pos = register 25
    ;;;
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    profile Servo_1 servo at position maxspeed=Reg_501
     accel=Reg_502 P=Reg_503 I=Reg_504 D=Reg_505
    search and zero Servo_1
    goto next

[2] WAIT_FOR HANDSHAKE
    ;;; At this step, the controller is waiting for the
    ;;; external computer to set the flag, indicating
    ;;; that there is a new servo position waiting in
    ;;; register Servo_Pos.
    ;;;
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    monitor New_Pos:set goto next

[3] MOVE_SERVO
    ;;; Here, the controller knows that the new
    ;;; position information is available, it initiates
    ;;; the servo motion.
    ;;; First, we must range the value in register
    ;;; Servo_Pos. The data from the computer is
    ;;; expressed in thousandths of an inch of linear
    ;;; travel, but we've previously calculated that
    ;;; the servo must turn 4 steps to move its
    ;;; actuator 0.001 inch. So first, we multiply the
    ;;; number in register Servo_Pos by 4.
    ;;;
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    store Servo_Pos * 4 to Servo_Pos
    turn Servo_1 to Servo_Pos
    clear New_Pos
    monitor Servo_1:stopped goto WAIT_FOR_HANDSHAKE
```

The Pointer and Phantom Registers

The pointer and phantom registers are a pair of special-purpose registers that you can use to access most of the controller's resources. The pointer and phantom registers are registers 127 and 128 and are used in conjunction with each other. Register 128 is the phantom register, and register 127 acts as its pointer. Register 128 is called the phantom register because it does not actually exist as a physical register; it acts as a window through which other resources are accessed.

Any instruction referring to register 128 actually accesses another of the controller's resources, for example, an input or a flag. By storing the number of a general or special purpose register in the pointer register, the phantom register can then access the controller resource being pointed to. The specific resource being accessed is determined by the number of the register stored in the pointer register. Using this approach you can use this feature to access any of the controller's inputs, outputs, numeric displays, the position of a stepping motor, or other resources.

One of the major uses of the pointer and phantom registers is to allow iterative (repetitive loop) programs to access a sequential array of inputs, outputs, registers, etc. When requirements exist for performing the same operations on a number of I/O points, this technique allows you to dramatically shorten programs. Possible applications include automatic testing of switches and other electrical devices or monitoring of a number of identical workstations.

The following table is a partial list of special purpose registers you can use to reference controller resources. The first column in the table is the number stored in register 127, and the second is the controller resource accessed. Refer to the *Register Reference Guide* for a more complete listing of special purpose registers.

Store in Register 127	Access in Register 128
1 - 1000	Numeric registers 1 - 1000
1001 - 1999	Outputs 1 - 999
2001 - 3024	Inputs 1 - 1024
7001 - 7016	The position of stepping motors 1 -16
8001 - 8256	Analog output 1 - 256 settings
10001 - 10032	Groups of 32 outputs as a binary number
11001 - 11032	Groups of 32 inputs as a binary number
13201 - 13232	Flags 1 - 32

Special purpose registers 1001 through 1999 reference the controller's outputs from output 1 to 999, respectively. Storing 1018 in register 127 specifies output 18 and storing the number 1 in the phantom register (register 128) turns on the output. Storing 0 in register 128, turns the output off. The following instructions turn output 18 on.

```
store 1018 to Pointer
store 1 to Phantom
```

In the instructions shown in this section, register 127 has the symbolic name Pointer, and register 128 is named Phantom.

This technique is known as indirect addressing, because the location being addressed is not explicitly stated in the instruction being executed. Since the pointer register (register 127) can be incremented as part of a programmed loop, you can write a program to go through a series of sequential I/O points, performing the same operation on each set of I/O.

You can use the phantom register to access any numeric register. For example, storing 115 in register 127, tells the phantom register to access register 115. You can use any of the instructions that access the controller's general purpose registers to access the phantom register. To the controller, the phantom register looks like any other register. The difference is that the phantom register is automatically steered to the resource indicated in the numeric code previously stored in register 127. The following sample instructions show how to use the phantom register.

```
store Reg_10 to Phantom
store Phantom to Aout_3
if Phantom <> 0 goto next
if Ain_1 >= Phantom goto next
turn Motor_1 to Phantom
delay Phantom sec goto next
```

Accessing Digital Inputs and Outputs

Even though inputs and outputs are bi-stable (either on or off) their states are expressed as a number when accessed via the phantom register. The number 1 indicates that the input or output is on and the number 0 indicates that it is off. Inputs can only be read via the phantom register, not changed. Outputs can be either read or changed via the phantom register.

Example

The following example uses the phantom register for an output scanning operation.

```
[1] INITIALIZE
    ;;; This program example uses the phantom register
    ;;; in a program loop. The program sequentially
    ;;; turns on, then off, each of the controller's
    ;;; first 32 outputs.
    ;;; This step stores the number of the special
    ;;; purpose register which corresponds to output 1
    ;;; (1001) to the pointer register. It then
    ;;; proceeds into the program loop.
    ;;;
    ;;; Pointer = Register 127
    ;;; Phantom = Register 128
    ;;;
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    store 1001 to Pointer
    goto next

[2] OUTPUT_ON
    ;;; The program loop starts here. The first time
    ;;; this step is executed, the pointer register
    ;;; contains the value 1001, meaning that it is
    ;;; pointing to digital output 1.
    ;;; By storing the value 1 to the phantom register
    ;;; we are, in effect, turning output 1 ON.
    ;;;
    ;;; The second time through this loop, the pointer
    ;;; register will point to output 2, so that is
    ;;; the output which is turned on. Each successive
```

```
    ;;; iteration of the program loop turns on the next
    ;;; output in the sequence.
    ;;;
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    store 1 to Phantom
    delay 1 sec goto next

[3] OUTPUT_OFF
    ;;; This step first turns off the output we turned
    ;;; on in the previous step. Since the phantom
    ;;; register is still pointing to the same output
    ;;; we turned on in the previous step, we can do
    ;;; this by storing the value 0 into the phantom
    ;;; register.
    ;;;
    ;;; Then, before moving to the next output to be
    ;;; cycled, we have the program test to see if
    ;;; we've reached the last output in our sequence.
    ;;; The If instruction checks to see if we've just
    ;;; cycled output 32 on and off and, if so, takes
    ;;; us out of the loop. If not, the store
    ;;; instruction increments the pointer register
    ;;; and returns to step OUTPUT_ON.
    ;;;
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    store 0 to Phantom
    if Pointer >= 1032 goto DO_SOMETHING_ELSE
    store Pointer + 1 to Pointer
    goto OUTPUT_ON
```

Programming Hints

When a program loop is written, there is usually some condition programmed for leaving the loop. The IF instruction in step OUTPUT_OFF tests the pointer register to determine if it is pointing to output 32.

Tracking Multiple Resources

In instances where the I/O points are being accessed in a program loop, you may need to maintain more than one pointer for the phantom register. For example, each pass through a program loop could require the actuation of three outputs and the monitoring of three inputs, all of which require indexing for each new cycle.

You can accomplish this by using some of the general purpose numeric registers for storing a pointer for each I/O point to be accessed. Just prior to the required access, the program stores the number from that register into register 127. At the end of each cycle, all of the general-purpose registers being used to store pointers can be incremented individually.

Using Stepping Motors

Quickstep provides direct support for the control of stepping motors through a series of stepping motor modules that plug directly into your controller and are programmed using Quickstep instructions. The stepping motor instructions are high-level instructions and allow you to:

- Set or change stepping motor motion parameters
- Establish a home or zero position for the motor
- Start the motor in motion
- Stop the motor motion

Quickstep supports relative and absolute turn instructions.

The stepping motor instructions can derive some or all of their parameters from any of the controller's numeric resources. For a description of the stepping motor instructions, see Chapter 2, *Stepping Motor Instructions*. Some stepping motor modules must be programmed using servo motor instructions. Refer to the Installation and Applications Guide for your stepping motor module.

Since the stepping motor control modules contain independent processors, the controller can continue executing a program during a turn instruction, even though the motor may still be in motion. This gives you more flexibility for multi-axis applications or for the preparation of tooling when a workpiece is in motion. The modules also keep track of the motor's position at all times, allowing you to program absolute (coordinate-based) positioning instructions.

When using on-board motor drivers, the modules automatically generate the sequence of pulses necessary to turn a stepping motor in either full-step or half-step mode.

HINT: We recommend using half step mode for most applications, since it results in smoother operation of the motor.

Programming Stepping Motors

A typical sequence for controlling a stepping motor is as follows:

- Initialize the motion parameters using a PROFILE MOTOR instruction.
- If the absolute position of the motor is important, home the motor using either the ZERO MOTOR or SEARCH AND ZERO MOTOR instruction.
- Initiate motor motion using a TURN MOTOR instruction.
- Wait for the motor motion to stop.

Example

The following sample program shows the typical sequence for stepping motor motion and control described previously. For an additional example using stepping motors, see the sample programs in Appendix A.

```
[1] INITIALIZE
    ;;; The profile instruction provides initial operating
    ;;; parameters for the motor. We'll then search for the
    ;;; motor's home position. Once this is found and the
    ;;; motor stops, we'll proceed to the next step.
    ;;;
    ------------------------------------------
    <TURN OFF ALL DIGITAL OUTPUTS>
    ------------------------------------------
```

```
    profile Motor_1 (half) basespeed=200 maxspeed=500
     accel=400 decel=400
    search ccw and zero Motor_1
    monitor Motor_1:stopped goto Next

[2] TURN_MOTOR
    ;;; Here, we'll start the motor turning using an
    ;;; absolute turn instruction.  The motor will move to
    ;;; a position 1000 steps clockwise from the home
    ;;; position sensed in the previous step.
    ;;;
    ------------------------------------------
    <NO CHANGE IN DIGITAL OUTPUTS>
    ------------------------------------------
    turn Motor_1 to 1000
    monitor Motor_1:stopped goto REST_OF_SEQUENCE
```

Subsequent TURN MOTOR instructions will use the motion parameters established in the previous profile instruction for that axis. You can program new profile instructions to create different speeds or acceleration/deceleration rates for different moves as many times as necessary during a program. One method of changing parameters in a profile is to place the profile instruction in a program loop that references a register or Data Table for various motion parameters. This is a convenient technique for programming different parameters for a long series of motions.

When a motor is in motion, the controller should not execute another turn or profile instruction for that motor or a software fault will result. After programming a TURN MOTOR instruction, you should have the controller execute a MONITOR MOTOR instruction to see if the motor is stopped prior to any subsequent turn or profile instructions.

Reading Stepping Motor Position

CTC controllers provide a series of special-purpose registers that display the position of a stepping motor. The register numbers are 7001 through 7016 for motor axes 1 to 16. Any instruction that can access a numeric register (e.g., STORE) can access these registers.

When using a Model 2205 Stepping Motor Control Module, access these registers when the associated motor is stopped. If the motor is still running, the controller returns a value of -1 for the position.

Establishing a Home Position

The SEARCH AND ZERO instruction establishes a home position for a stepping motor. Using the SEARCH AND ZERO instruction you can start the motor turning in either a clockwise or counterclockwise direction. The stepping motor continues turning in that direction until a switch closure is sensed on the motor module's home input. To get a motor in the home position consistently, perform a multi-step homing sequence. Search and zero the motor once; then search and zero the motor again at a slower speed. Re-homing the motor a second time at the slower speed provides a more consistent home position. The following example shows how to search and zero a motor and find the home position consistently:

```
[55] BEGIN_HOME_SEQUENCE
    ;;; Begin initial home sequence for motor 1
    ------------------------------------------------
        <NO CHANGE IN DIGITAL OUTPUTS>
    ------------------------------------------------
```

```
    profile motor_1 (half) basespeed=800 maxspeed= 2000
     accel=400 decel=400
    search ccw and zero motor_1
    monitor motor_1 stopped goto next

[56] MOVE_OFF_HOME
    ;;; Move off the home switch in order to
    ;;; re-home the motor
    ------------------------------------------------------
        <NO CHANGE IN DIGITAL OUTPUTS>
    ------------------------------------------------------
    turn motor_1 to 50
    monitor motor_1 stopped goto next

[57] FIND_HOME
    ;;; Set the profile for slower speed then
    ;;; re-home motor
    ------------------------------------------------------
        <NO CHANGE IN DIGITAL OUTPUTS>
    ------------------------------------------------------
    profile motor_1 (half) basespeed=100 maxspeed=200
      accel=400 decel=400
    search ccw and zero motor_1
    monitor motor_1 stopped goto next
```

Programming Concepts

The following list contains hints and notes that may be helpful in understanding some of the concepts involved in writing stepping motor control programs using Quickstep.

- The PROFILE MOTOR instruction can also derive parameter values from a changeable source, such as an operator interface or a thumbwheel. The actual motion parameters in use are only updated when the profile instruction is re-executed. If you wanted to update parameters each time through the loop, you would need to include the PROFILE MOTOR instruction within the program loop.
- Although, it is common practice to include a MONITOR MOTOR instruction in each step with a motor motion instruction, this is not a general requirement. MONITOR MOTOR instructions can serve two purposes:
    1. To avoid sending the motor a new motion instruction while the previous motion is still under way (This can cause a CPU software fault).
    2. To avoid mechanical conflict (e.g., insuring that the motor has stopped prior to sending the stamping press down).
- In programs involving batch production of many workpieces, CTC recommends that you SEARCH AND ZERO the motor each cycle to insure that mechanical interference hasn't caused the motor to lose position. If the additional time required to rehome the motor every cycle is a problem, add a cycle counter to the program to jump to a SEARCH AND ZERO instruction every nth cycle, minimizing the number of potentially defective workpieces.
- When using the controller's dedicated stop function, any stepping motor motions that have already been initiated will continue to their completion. If this is undesirable, you can program a separate task to create a customized stop function that invokes a STOP MOTOR instruction.

Using Servo Motors

Quickstep provides direct support for the control of servo motors through a series of servo control modules that plug directly into your controller and are programmed using Quickstep instructions. The servo instructions are high-level instructions and can derive some or all of their parameters from any of the controller's numeric resources. They allow you to:

- Set or change servo motion parameters
- Establish a home or zero position for the servo
- Program relative, absolute, and continuous velocity turn instructions
- Start the servo in motion
- Stop the servo motion

The PROFILE SERVO instruction gives you control of acceleration, deceleration, velocity, and servo closed loop gain characteristics. CTC's servo control modules contain a programmable compensation filter which provides program control over the stability characteristics of the servo and allows you to write programs to automatically compensate for changes in the dynamics of the servo system.

Since our servo control modules contain independent microprocessors, the controller's program can continue execution after a turn instruction, even though the servo may still be in motion. This provides freedom for multi-axis applications or for the preparation of tooling while a workpiece is still in motion. In addition, the controller can read the servo error and instantaneous servo position even while the servo is in motion, providing a useful tool for sensing performance.

Some stepping motor modules must be programmed using servo motor instructions. Refer to the Installation and Applications Guide for your stepping motor module.

Programming and Initiating Servo Motions

The two major instructions for setting up and initiating servo motions are PROFILE SERVO and TURN SERVO. The PROFILE SERVO instruction establishes the following motion parameters: holding mode, maximum speed, acceleration rate, and proportional, integral, and differential compensation filters. All the numeric parameters in a PROFILE SERVO instruction can be derived from any of the controller's numeric resources, which facilitates several important capabilities:

- Tuning a servo using data in numeric registers, from an operator interface, from a Data Table, etc.
- Modifying the servo parameters by observing the servo performance and automatically modifying certain parameters.
- Modifying servo parameters with an external PC using CTCMON.

The TURN SERVO instruction initiates servo motion. There are three different modes of servo motion:

- Relative – the servo turns a specified number of steps clockwise or counterclockwise from the servo's current position. For example,

  ```
  turn servo_1 ccw 500 steps
  ```

NOTE: A step refers to one edge transition on either encoder input (A or B) for that axis. For example, a 500 line encoder generates 2000 steps.

- Absolute – the servo turns to a new absolute, or coordinate, position based on some predetermined zero reference point. For example,

  ```
  turn servo_1 to 500
  ```

  Once a zero or home reference location is set, all subsequent absolute instructions specify a coordinate position based on the zero position. The servo control module automatically determines the direction and distance to turn the actuator to the new position.

- Velocity – Begins continuous clockwise or counterclockwise motion. The servo remains in motion until the controller executes a Stop Servo instruction or the servo control module senses a Stop input signal. For example,

  ```
  turn servo_5 cw
  ```

Example

The following sample program shows the typical sequence for servo motor control. For additional examples, see the sample programs in Appendix A.

```
[1] INITIALIZE
    ;;; The Profile command establishes the initial
    ;;; motion parameters for the servo.  We then use
    ;;; the Search and Zero command to find the servo's
    ;;; home reference position.  The Monitor command
    ;;; determines when the servo has sensed home and
    ;;; come to a stop.
    ;;;
    ____________________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________________
    profile servo_1 servo at position maxspeed=2000
     accel=500 P=8 I=250 D=219
    search and zero servo_1
    monitor servo_1:stopped goto Next

[2] MOVE_THE_SERVO
    ;;; The Turn instruction below is an "absolute"
    ;;; turn; in other words, the servo will turn to a
    ;;; position 5100 steps forward from the predeter-
    ;;; mined home position. Once again, we'll wait for
    ;;; the motion to be complete before continuing.
    ;;; Alternatively, you may continue with the
    ;;; program without waiting by substituting a
    ;;; goto next instruction.
    ;;;
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    turn servo_1 to 5100
    monitor servo_1:stopped goto Next
```

Programming Notes

Once the controller has executed a turn instruction for a specific axis, no subsequent turn instruction can be executed for that axis until the first motion has ended. Executing a second turn instruction results in a software fault halting the execution of the program.

Unlike the profile instruction for stepping motors, the controller can execute a new PROFILE SERVO instruction for a specific axis when the servo is still in motion. The new PROFILE SERVO instruction can change any parameter except the acceleration rate.

You can use this programming technique when operating a servo in velocity mode to make periodic changes in the velocity of the servo actuator. A PROFILE SERVO instruction specifying a new maximum speed causes the servo to accelerate or decelerate to the new speed. For example, if the current maximum speed is stored in a numeric register, incremental changes in speed could be programmed with instructions such as:

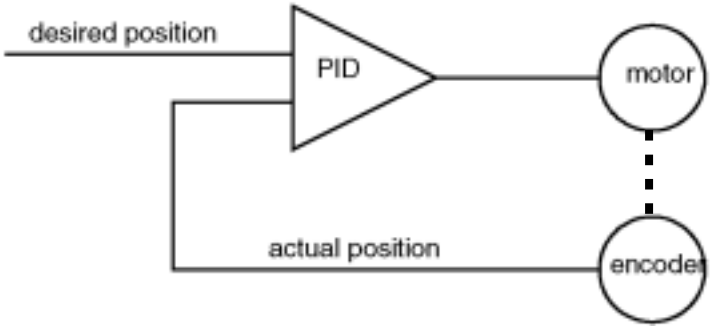
```
store reg_10 + 400 to reg_10
profile servo_1 maxspeed=reg_10
```

This type of change could be made in response to any condition the controller is capable of sensing or deriving.

Tuning a Servo

All servos have some form of sensor, typically an encoder, that provides position and velocity feedback to the controller. The servo controller compares the servo's current position to a calculated ideal instantaneous position and, based on the difference, sends a corrective signal to the actuator. CTC's servo control modules use the on-board microprocessor to perform this comparison.

The ideal position mentioned previously is not the ultimate target position for the complete servo motion being executed. Instead, it is the position where the servo motor should be at that instant to successfully execute the desired motion profile. This motion profile takes into account the desired acceleration and deceleration characteristics of the actuator in the real, mechanical world.



Once the difference between the ideal and actual positions (instantaneous error) is known, a correction signal is sent to the actuator. The strength of this correction signal, for a given amount of error, is a critical parameter in any servo system. The correction signal is determined, in part, by a multiplication factor applied to the error. This factor, called the system gain, is specified in the Servo Profile instruction as the P parameter.

Using the P Parameter

If the P parameter or gain is set too high, the correction signal sent to the actuator causes it to overshoot the ideal position. This results in a sensed error in the opposite direction that, in turn, is disproportionately corrected. The result of this instability may be continuous oscillation around the ideal position.

If the P parameter is set too low, the resulting correction signal is insufficient to fully correct the position error. The result is a slow, inaccurate servo system. Because of this, there is a trade-off between accuracy and stability.

By applying a filter to the sensed error signal, you can maintain stability at a higher gain factor. The filter can dampen the tendency to oscillate while allowing a high DC gain and results in high accuracy. The microprocessor in the servo control module implements a filter for this purpose.

Using the I Parameter

The additional parameters used in programming this filter are the I (integral) and D (differential) parameters. The I parameter is useful to obtain increased accuracy at low frequencies, or when stopped. The I factor integrates, or builds up, a corrective signal in response to steady-state error. A greater I factor will cause the filter to build up a corrective signal for even small amounts of error, greatly increasing the terminal accuracy of each move.

Using the D Parameter

The D parameter, on the other hand, senses and responds to rapidly changing rates of error. It is therefore most useful in increasing system response to varying loads and frictions at high speeds.

Using the P, I, and D parameters to tune a servo system requires knowledge of servo system basics. CTC has application notes available to assist in the process, as well as tutorials on servo theory to provide some necessary background information. We encourage you to take advantage of this information.

Some models of CTC servo control modules have advanced filters offering a choice of compensation techniques, including feed forward compensation. For more information about these functions, refer to the installation guide for your control module.

When using the Servo Profile command in conjunction with a model 2206 Stepping Motor Control Module, the P, I, and D parameters do not apply and may be ignored.

Using the Servo Position and Error Parameters

The instantaneous position and error of a servo can be easily accessed by your Quickstep program. This information provides an important set of tools for tuning, control, and diagnostics.

A few of the ways in which this information can be used include:

- The servo position may be used to trigger events. For example, you can commence a servo motion in velocity mode, then trigger a change in velocity or perhaps a motion on another axis when the servo reaches a specific position. This may be done with an instruction such as:

  ```
  if servo_1:position >= 15000 goto next
  ```

  A number of such instructions may be used throughout a servo move to trigger a variety of events.

- The servo error may be used as a tuning aid. By observing the servo error throughout a motion (e.g., `store servo_1:error to disp_2`) you can detect portions of a move where excessive error is building. For example, if error is high during acceleration, you may be attempting to accelerate at a rate faster than your motor can accommodate. If, on the other hand, the error starts building as you reach maximum speed, and then continues to build, you may be attempting to run faster than your motor allows. Such problems often masquerade as instability; observing the servo error can help you determine what is really happening.

- Often, excessive servo error can be used to indicate a jam condition. If a physical obstacle prevents a servo from keeping pace with your motion command, error will build quickly. This may be detected with a fault monitoring program containing an instruction such as:

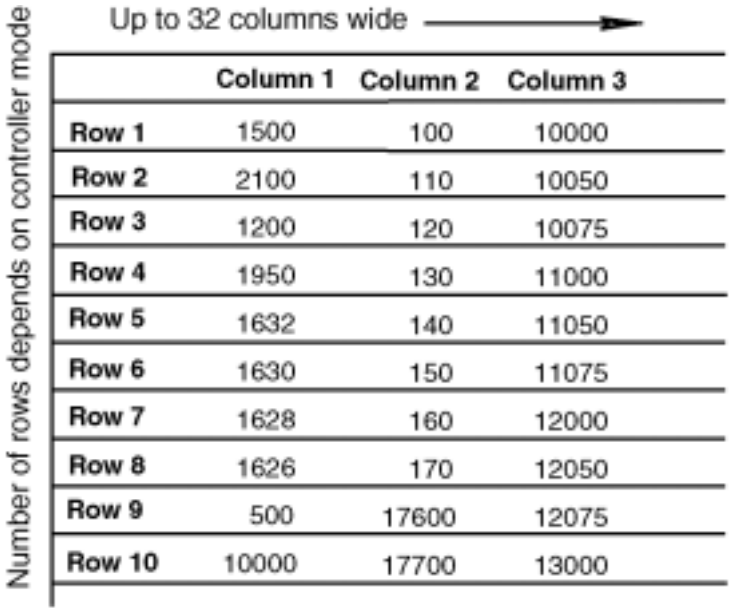
```
if servo_1:error >= 100 goto JAM_DETECTED
```

Establishing a Home Position

The SEARCH AND ZERO instruction establishes a home position for a servo. Each servo axis on a servo control module has a dedicated home input that is used in conjunction with the SEARCH AND ZERO instruction to set a home position for that axis. The precise function of this instruction depends on the servo control module being used. For information on using the SEARCH AND ZERO SERVO instruction, refer to the instalation and applications guide for your servo module.

The Data Table

The Data Table is a two-dimensional array of numbers that can be stored in memory along with your Quickstep program. The numbers in a Data Table can range from 0 to 65,535. For the maximum allowable size of a Data Table, refer to the specifications for the controller model you are using. The following illustration shows an example Data Table.



	Column 1	Column 2	Column 3
Row 1	1500	100	10000
Row 2	2100	110	10050
Row 3	1200	120	10075
Row 4	1950	130	11000
Row 5	1632	140	11050
Row 6	1630	150	11075
Row 7	1628	160	12000
Row 8	1626	170	12050
Row 9	500	17600	12075
Row 10	10000	17700	13000

Storing information in the Data Table can make your program easier to maintain by keeping related numeric parameters together in an organized format. You can change the data in a Data Table by using the Data Table editor (part of the Quickstep Editor), by instructions in a program, or by remote loading of data from an external computer as supported by the CTC serial protocols or by using CTCMON.

There are two major uses for the Data Table:

- In iterative (repetitive loop) programs, you can store parameters which must change for each iteration in rows of the Data Table. Each time through the loop, your program would “point” to the next row to obtain its data.
- Flexible assembly machines must often be reconfigured to manufacture a variety of products. The Data Table allows you to store the parameters for multiple products in the rows of the table. To make a specific product, you point to the appropriate row.

An additional use of the Data Table is to store results information for each workpiece being produced as your program runs. This information may then be transferred to a remote computer on a batch basis.

How you use a Data Table is determined by the references to it within your Quickstep program. The data itself may be used as motor motion parameters and coordinates, analog set points, time delay durations, and other manufacturing parameters. There is also a special function in most CTC controllers for sending a row of the Data Table out through the controller’s serial port as an ASCII message.

Accessing the Data Table Using the Row Pointer

In many applications, you will want to use information from the Data Table on a row-by-row basis. The easiest way to do this is by using the row pointer, register 126. The number stored in this register determines the row from which you'll obtain data when you access the Data Table using col_1, col_2, etc. For instance, the following sequence of instructions results in servo_1 turning to the position stored in Data Table row 15, column 2:

```
store 15 to reg_126
turn servo_1 to col_2
```

The benefit of using this approach is that you can select the desired row of the Data Table at one point in your program. You can then program a number of subsequent instructions to use data from the various columns of that row.

Example

One use for this technique is for programming an X-Y table, for positioning a workpiece using two servo motors controlling the X and Y axes of motion. The sets of coordinates for these axes can be stored in successive rows of the Data Table, using column 1 for the X coordinate and column 2 for the Y coordinate.

After initially pointing to row 1, a short program loop could be written that would:

1. Send servo_1 to the position in col_1 and send servo_2 to the position in col_2.
2. Perform the inspection, drilling or some other function on the workpiece at the new location.
3. Move the row pointer to the next row by adding 1 to the value in register 126.
4. Return to the beginning of the loop to process the next row.

Using the Data Table, what could have been a lengthy program with position information scattered throughout, has become a very compact loop. The program stores all the position information separately where it can be easily maintained.

Storing Data Using the Row Pointer

For 2600 series and higher controllers, you can store data to a specific row of a data table using the row pointer. The following instructions store a value (368) in row 5, column 8 of the Data Table:

```
store 5 to reg_126
store 368 to col_8
```

Using Row and Column Pointers

In the example using the row pointer, it varied for each new cycle and the column designation was fixed. You can also reference the Data Table so that both the row and column numbers are expressed as variables. Register 131 points to the active row and register 132 points to the active column. You then use a special purpose register (register 9000) to access the target Data Table

location. The following instructions store the position of a servo motor in the fourth row, third column of a Data Table:

```
store 4 to reg_131
store 3 to reg_132
store servo_1:pos to reg_9000
```

Example - Using the Data Table in Quickstep Programs

A broad array of instructions can obtain and use the numeric data stored in the Data Table, for example:

```
store col_1 to dis_1
profile servo_1 maxspeed=col_8
```

The following example uses numeric values from the Data Table in math calculation:

```
store col_1 + 500 to reg_10
turn Motor_1 ccw reg_10 steps
```

STORE instructions can perform a math function and store the result in any numeric destination for use in a subsequent instruction. This allows you to store data in the Data Table in engineering units (e.g., inches of travel, degrees C). Later, your program can convert it into units required by the controller for motor motions, analog values, etc. This can increase the maintainability of your program.

A program can also use an IF instruction to test a value in the Data Table against another source of numeric data. If the test is satisfied, the instruction takes the controller to a new step.

```
if ain_1 >= col_1 goto next
if servo_1:pos < col_3 goto next
```

You can use a value in the Data Table to generate a time delay, with the number in table representing time in units of minutes, seconds, or hundredths of seconds.

```
delay col_4 sec goto next
```

You can derive any or all of the operating parameters for a stepping or servo motor.

```
profile Motor_2 (half) basespeed=col_1 maxspeed=col_2
  accel=col_3 decel=col_4
```

In a relative turn instruction:

```
turn servo_2 ccw col_1 steps
```

In a absolute turn instruction:

```
turn Motor_1 to col_1
```

For programming examples using the Data Table, see Appendix A.

For information on using the Data Table for message transmission, refer to the Application Note, *ASCII Message Transmitting with CTC Controllers*.

Using Analog Inputs and Outputs

Analog signals are signals in which information is carried in the form of a voltage value (or, sometimes, a current value). While digital signals are either on or off at any given time, analog signals can vary continuously within a range.

Quickstep allows analog data to be accessed through a variety of analog input and output modules. Modules are available with resolution ranging from 10-bit (1 part in 1024) to 15-bit (1 part in 32,768), and with a number of different voltage range options. Possible uses for analog input and output points include:

- Monitoring and/or set-point control of temperatures, pressures, levels, etc.
- Acquiring data for statistical process control.
- Controlling analog actuators, e.g., proportionate solenoid valves, motor speed controls, etc.
- Programmable control of mass flow controllers.

Representing Analog Signals in Quickstep

Microprocessors are digital devices and can only process analog information when it is expressed in the form of numeric data. An analog input module converts an input voltage into a numeric representation using a process called analog-to-digital (A/D) conversion.

For example, a module with 12-bit resolution is capable of resolving 1 part in 4096. If the allowable input voltage range is 0 to +10 volts, the module can sense a change as small as 2.44 mv (.00244 volt). Although the numeric value created directly by the conversion is a number from 0 to 4095, the controller automatically ranges this value into a number from 0 to 10,000 to make programming more convenient. A voltage level of 2.5 volts is represented by the value 2,500.

NOTE: Some high-resolution modules will represent 2.5 volts as 2,500,000. Refer to the installation and applications guide for your analog I/O module.

Using Analog Input Data

Any instruction that uses numeric data can use analog input data. For example, the value of an analog input can be frozen by storing it to a numeric register using a STORE instruction, `store ain_1 to reg_10`. You can use a math instruction to scale the analog input value and store the result to a numeric display, `store ain_1 * 25 to dis_1`.

NOTE: Such a transfer only occurs once each time this instruction is executed. No permanent relationship is established between the analog input and the display. To continuously refresh the displayed data, the instruction would have to be continuously executed in a program loop.

You can also test an analog value by comparing it to another source of numeric data using an IF instruction, `if ain_1 >= reg_10 goto next`.

Motion control instructions such as turn or profile can also derive numeric data from analog input values. Usually a math instruction is necessary to scale the analog data into an appropriate range of values.

Using Analog Outputs

Analog outputs create a voltage level which performs some controlling function in the process or machine being controlled. For example, you can use an analog output to control the intensity of a heater (using a proportionate controller) or the feed rate of a vibratory feeder. Analog outputs can also be used to control the speed of a motor drive system or provide control signals for mass flow controllers.

As with analog inputs, analog output values are represented numerically with values that translate easily to voltage levels. For example, a value of 5000 sent to an analog output will result in a voltage level of 5 volts appearing on that output. This may be done with a STORE instruction, for example:

```
store 5000 to aout_1
```

You can use any source of numeric data to derive the number sent to an analog output. For example, the instruction, `store reg_10 to aout_1`, reads the value from register 10 and creates a proportional voltage level at analog output 1.

NOTE: As with analog inputs, such an instruction only acts once and does not create any ongoing relationship between the register and the analog output. To continuously update the analog output, the instruction must be placed in a program loop that continuously re-executes it.

You can use a math instruction to calculate a numeric value and send the value to an analog output, for example:

```
store thwl_1 * 25 to aout_1
```

You can use one or more such math instructions to convert a value expressed in engineering units to the desired voltage level that must appear at an analog output.

Using various combinations of time delay, input monitoring and math instructions, you can establish complex interrelationships between analog or digital inputs and analog outputs. You can even program analog ramping via the creation of a program loop that, based on a timed interval, gradually increases an analog output value.

For a programming example, see Appendix A.

Programming Hints

Using If Instructions

The If instruction allows any source of numeric data, including an analog input, to be tested with any of the six possible relational tests, greater than (>), less than (<), equal to (=), greater than or equal (>=), less than or equal (<=), not equal to (<>). If a test is satisfied, the instruction takes the controller to a new step of its program.

```
if ain_5 >= 1500 goto next
if ain_1 < ain_2 goto next
```

The value from an analog input can be tested against any other numeric value accessible to the controller.

Using a Delay instruction

You can use the value from an analog input to generate a time delay with the numeric value representing time in units of hours, minutes, seconds, or hundredths of a second:

```
delay ain_1 sec goto next
```

For typical applications, math commands would first be used to scale the value appropriately, storing the intermediate result in a register:

```
store ain_1 - 500 to reg_10
delay reg_10 sec goto next
```

Using Special-Purpose Registers

The controller can also access analog outputs via a set of special-purpose registers. This allows you to use the Phantom register to indirectly address the analog outputs in a repetitive loop. References to registers 8001 to 8256 actually access analog outputs 1 through 256, respectively. References to registers 8501 to 8756 give you read only access analog inputs 1 through 256, respectively.

Using Thumbwheel Arrays and Numeric Displays

Quickstep supports high-level access to thumbwheel arrays and numeric display modules. These devices are connected to your controller via a thumbwheel and display interface module. The interface modules allow you to avoid significant programming effort and the use of large quantities of I/O. With this approach, thumbwheels and displays may be used with any Quickstep instruction that requires or provides numeric data.

Possible uses of thumbwheels and displays include:

- Entering and reading numeric parameters.
- Entering and displaying batch counts or footage counts.
- Entering and displaying analog information, such as pressure, weight, and temperature.
- Entering motor motion parameters such as velocity or distance, for runtime tuning of operation.
- Modifying motion parameters for initial set-up of a servo or stepping motor system.

Examples of instructions referencing thumbwheels and displays are shown below. The controller automatically handles all of the scanning of digits and decoding of data, simplifying the programming effort required.

```
store thwh_1 + 500 to reg_15
turn Motor_1 ccw thwl_1 steps
store ctr_1 to dis_1
```

Prescaling Values Automatically

Frequently the numeric information used in a control program is not expressed in units that are convenient for the operator of an automated machine. For example, the motor motions are expressed in steps within the controller's program, while the machine's operator may think in terms of thousandths of an inch.

You can use math instructions to prescale values derived from thumbwheel arrays or rescale values sent to numeric displays. For example, in a case where a stepping motor must be driven 2 steps for each thousandth of an inch of resulting linear travel on a machine, the following sequence of instructions would allow the operator to enter the motion as thousandths of an inch:

```
store twhl_1 *2 to reg_10
turn Motor_1 cw reg_10 steps
```

Register 10 stores the result of the multiplication until it is used by the TURN MOTOR instruction. You can program both instructions in the same step, as long as the math instruction is first.

Sometimes a program requires more than one math instruction to properly scale a value. For example, a pressure transducer provides a 0.5 to 5.5 Volt signal which represents a pressure range from 0 to 500 PSIG. This signal is read by the analog input of the controller as a number from 500 (0.5 Volt) to 5500 (5.5 Volts). For information on analog inputs, see *Using Analog Inputs and Outputs*.

To display this value in units of PSIG, we need to offset and scale the value to properly range it. This could be accomplished using the following instructions:

```
store ain_1 - 500 to reg_20
store reg_20 / 10 to disp_1
```

The first instruction reads the signal from analog input 1 and then removes the offset of 500 prior to storing it in register 20. The second instruction ranges the value in register 20 to a number from 0 to 500 by dividing the number by 10. The result of this math operation is then sent to the display.

For programming examples showing thumbwheel arrays and displays, see Appendix A.

Accessing Four-digit Displays Using Special Purpose Registers

You can also access a numeric display by storing a number to special purpose registers 3001 to 3016 (representing displays 1 through 16). This allows the displays to be accessed using the Phantom register or from a remote computer using CTC's serial communications protocols or CTCMON.

For example, `store 3820 to reg_3002`, displays the number 3820 in numeric display 2.

Using Eight-digit Thumbwheels

The controller views eight-digit thumbwheels and numeric displays as a pair of four-digit devices. Because of this you must read the number from an eight-digit thumbwheel as two separate four-digit numbers.

In the following example, the most significant four digits of the thumbwheel array are connected in position twhl_1, and the least significant four digits are connected in position twhl_2.

```
store twhl_1 * 1000 to reg_10
store reg_10 + twhl_2 to reg_10
```

Register 10 then contains the eight-digit value read from the thumbwheel array.

Using Eight-digit Displays

A series of special purpose registers (registers 4001 to 4016) provide access to eight-digit numeric displays. Any two successive four-digit display connections may be used to connect an eight-digit numeric display.

Register 4001 automatically accesses an eight-digit display connected in positions disp_1 and disp_2. To access an eight-digit display connected in positions disp_2 and disp_3, register 4002 would be used, and so on.

The following example sends the current count stored in counter 1 to an eight-digit display connected in positions disp_3 and disp_4:

```
store ctr_1 to reg_4003
```

Registers 4001 to 4016 are write-only registers. The information in them cannot be read by subsequent instructions.

Setting a Decimal Point

A fixed decimal point may be displayed on any four- or eight-digit numeric display. This is accomplished by using special purpose registers 6001 to 6016, representing four-digit displays 1 through 16.

For example, the instruction, `store 3 to reg_6002`, displays a decimal point on display 2 in the third position. Therefore, the number 9999 would appear on the display as 9.999.

When setting a decimal point on an eight-digit display, use the register corresponding to the least significant four digits. For example, to display a decimal point in the fifth position of an eight-digit display connected in positions disp_1 and disp_2, use the instruction:

```
store 5 to reg_6001
```

In this case, the number 1234567 would appear as 12.34567.

When using a decimal point, the display automatically shows zeros for unused digits to the right of the decimal point, e.g., .00012.

Using Dedicated Inputs

You can assign the first four inputs connected to your controller as dedicated functions. They are as follows:

- **Start** – Starts or continues execution of a program
- **Stop** – Stops the execution of a program in place
- **Reset** – Re-initializes the controller and starts executing the program at the first step
- **Step** – Advances the controller one step in the program

These functions are enabled as part of the programming process, using the Parameter editor. Once programmed, they are triggered automatically when an external switch connected to the appropriate input closes. The inputs are active at every step of your Quickstep program, subject to the priorities and rules listed for each input. (See the following paragraphs.)

Start Input Functions

The Start input causes the controller to begin or continue the execution of its program from its current step onward. If multiple tasks are active, all tasks start executing again. The Start input only functions when the controller is stopped:

- At power-up
- In response to a stop instruction
- In response to stop signal from the dedicated Stop input

If, during the execution of a program, the controller is stopped and then restarted using the dedicated Start input, each task continues based on the following priorities:

- If there is a MONITOR instruction whose conditions are already satisfied when the controller is restarted, the controller jumps to the step destination specified by that instruction. If there is more than one such instruction, the first one occurring in the step takes precedence.
- If there are no MONITOR instructions satisfied in the step, any DELAY instruction in the step is instantly timed out, and the controller jumps to the step specified by the DELAY instruction

The controller ignores the dedicated Start input when:

- It is already running
- The Stop input has a continuous switch closure applied
- The dedicated Reset input has a continuous switch closure applied

If you do not select **Start on Input #1** from the Parameter editor, the controller automatically wakes-up running at step one of its program when power is applied.

Stop Input Functions

The Stop input causes the controller to stop executing its program. Any tasks currently running are suspended at their current step, and no further progress in any of these tasks takes place.

When the Stop input halts the controller, the following rules apply:

- Any stepping motor or servo motions that have begun will continue to completion. A velocity mode instruction will continue indefinitely.

- The controller will begin executing its program, in the step that was active, when you restart the computer after a Stop input. If you also triggered the Reset input, the controller begins executing its program at the first step when it is restarted.

When the Stop input is active, the Start and Step inputs are inactive, but the Reset input is active.

WARNING! In any circumstances where human injury or substantial economic damage is possible, you must use a separate, overriding method to implement emergency stop functions. Never use an automation controller to implement back-up safety circuits. Refer to the Application Note, *Thinking about Safety*, for further information.

Reset Input Functions

The Reset input re-initializes the controller. The Reset input is active whether the controller is stopped or running. Triggering the Reset input causes the following actions:

- Cancels all tasks, and the controller returns to first step of the program.
- Turns off all outputs, and then turns on only those specified in first step of the program.
- Resets all volatile general-purpose registers and counters to zero.
- Terminates all counters previously established with Start Counter instructions.
- Blanks all numeric displays.
- Clears all flags.
- Stops all stepping motors.
- Resets all high-speed counting modules
- Resets, i.e., de-profiles, all servos.
- Turns off the outputs of any customization boards in the system.

The controller remains in the state described above as long as the reset input is held active. The Reset input functions as follows:

- If the controller is running at the time of the reset, it continues to run from the first step in the program when the reset signal is removed.
- If the controller is stopped at the time of the reset, it remains stopped, but at the first step in its program.
- If the controller receives a Stop input signal when the Reset input is active, the controller stops.
- When the Reset input is active, the controller ignores the Start and Step inputs.

Step Input Functions

The Step input performs a jog function, allowing the controller to advance one step in its program. If multiple tasks are running, each task is allowed to advance one step. The Step input is active only when the controller is stopped, and the controller automatically stops again once the single step has been executed.

The Step input does not function when the controller is running or when the either the Reset or Stop inputs are held active.

After the Step input is toggled, the controller makes one pass through the current step of each active task. The Step input functions as follows:

- If the current step contains one or more MONITOR instructions that are satisfied, the first such MONITOR instruction causes the controller to proceed to a new step.
- If no MONITOR instructions are satisfied and a DELAY instruction exists within the step, the time delay is instantly timed out and the controller jumps to the specified step.
- If any tasks are allowed to proceed to a new step as the result of the Step input, the controller executes the output instructions and other instructions in the new step, but does not execute any instructions capable of taking the controller on to another step.

If any task does not contain instructions that, according to the above rules, allow it to proceed at the instant the Step input signal is applied, the opportunity to proceed to the next step is lost for that task until the Step input is released and pressed once again. The Step input does not arm the controller to proceed at some future time. It only allows it to proceed, if permitted by specific instructions within a step, during the instant the Step input is first activated.

WARNING!

If your program relies upon the sensing of an input condition to stop the progress of an actuator, using the Step function could cause a dangerous situation. An example is a step which triggers a hydraulic valve causing a cylinder to begin extending, and which also contains a MONITOR instruction sensing a limit switch to stop the motion by proceeding to a new step. If the Step input is used to advance into such a step, the valve would be actuated, but the MONITOR instruction would not be activated until the Step input is released and actuated again. This may cause damage or a safety hazard.

Using High Speed Counting Modules

High Speed Counting Modules provide hardware support for count rates faster than that possible using the software counters described earlier. These modules also support quadrature counting, used to automatically track the position of encoders and similar devices.

Counting Functions Supported

The high speed counting modules support two counting functions: continuous totalizing or frequency counting (which can also be used for speed sensing). These functions are accessible from your Quickstep program using special purpose registers.

Up to eight counting channels can be present in a given controller. The current value of a high speed counter can be accessed using registers 5001 through 5008 for counters 1 through 8 respectively. You can also preset the counters to a specific value by storing a number to these registers, or reset the counter by storing zero to the associated register:

```
store 1234567 to reg_5001

store 0 to reg_5005
```

These registers can be accessed with any instruction which can be used with numeric registers.

Frequency Counting

The first high speed counter (only) may be configured as a frequency counter. This will cause the counter to be sampled over a fixed number of milliseconds, from 1 to 60,000 (one minute).

To enable this mode of operation, store the sample period to register 5501. Any subsequent reading of register 5501 will return the number of pulses received during the last sample period. For example, the following instruction sets the sample period to 10 milliseconds:

```
store 10 to reg_5501
```

The counter may be returned to normal operation by storing 0 to register 5501.

NOTE: Frequency counting is not available in 2200 and 2200XM controllers.

8-, 16-, or 32-bit Access to Input/Output Points

Normally, digital inputs and outputs are accessed individually using the specific instructions provided for that purpose. Sometimes, however, you may wish to access a group of consecutive I/O points, treating the bits represented by these I/O points as a binary value. The two major applications for this capability are:

- Communicating with an external intelligent system (for example, a digital thermometer) that provides or requires data in the form of a parallel word up to 32 bits wide.
- Allowing an external computer to communicate output information to the controller in the form of 8-, 16-, or 32-bit numbers. This can result in shorter data transmission time.

Quickstep provides this type of binary access to groups of 8, 16, or 32 I/O points via a group of special-purpose registers.

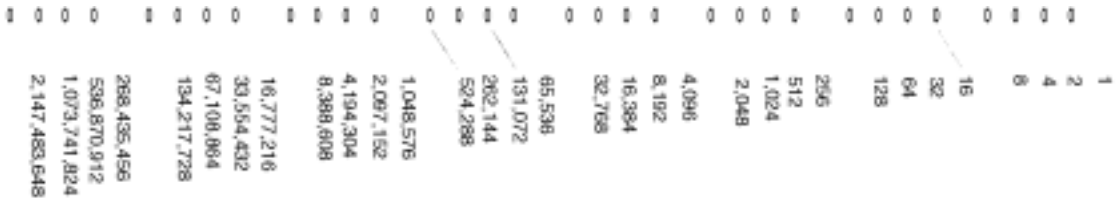
8-, 16-, or 32-bit Output Access

Outputs may be set in groups of 8, 16, or 32 by storing a number to one of the following special purpose registers:

- For 32-bit access, store a number (–2,147,483,648 to +2,147,483,647) to registers 10001 to 10032.
- For 16-bit access, store a number (0 to 65535) to registers 10101 to 10164.
- For 8-bit access, store a number (0 to 255) to registers 10201 to 10328.

For example, storing a number to register 10001 causes a binary representation of that number to appear on the first 32 outputs within the controller. Similarly, register 10002 affects outputs 33 through 64; register 10003 affects outputs 65 through 96; and so on.

In each instance, the smallest-numbered output receives the least-significant bit of the binary value. The binary weighting of each output is shown in the following illustration.



NOTE: The numeric value stored to registers 10001 through 10032 (32-bit access) must be expressed as a two's-complement signed integer from –2,147,483,648 to +2,147,483,647. For a description of two's complement notation, see page 3-54.

You can also read the current status of the outputs from these same registers.

8-, 16-, or 32-bit Input Access

The controller's inputs can also be accessed in groups of 8, 16, or 32 by using one of the following special-purpose registers:

- For 32-bit access to inputs, use registers 11001 to 11032.
- For 16-bit access to inputs, use registers 11101 to 11164.
- For 8-bit access to inputs, use registers 11201 to 11328.

For example, the number you read from register 11001 will be a 32-bit binary representation of the first 32 inputs within the controller. This value will be expressed in signed two's-complement. Reading a number from register 11002 accesses inputs 33 through 64; register 11003 accesses inputs 65 through 96; and so on. As with outputs, the inputs are read with the smallest-numbered input represented by the least-significant binary bit.

Masking Unused Bits

The number of inputs you wish to read, or outputs you wish to change, may not always be exactly equal to 8, 16, or 32. In such cases, you can use Boolean math operators (AND, OR, XOR, NAND, NOR, NXOR) to mask the bits you do not wish to affect. For example:

```
store reg_11101 and 4095 to reg_10
```

The above instruction first obtains the binary representation of inputs 1 through 16, but the upper 4 bits are then cleared by using the AND operator with the number 4095. The number 4095, expressed in binary, is
0000 1111 1111 1111.

You can also use Boolean operators to set, clear, or toggle certain outputs. For instance:

```
store reg_10201 or 15 to reg_10201
```

This instruction reads the current status of the first 8 outputs in your controller, then forces the lowest 4 bits (representing outputs 1 through 4) to be on. The number 15, in binary, is 0000 1111. The result of this instruction is stored immediately back to the first 8 outputs. Therefore, outputs 5 through 8 remain unchanged, but outputs 1 through 4 are forced on.

Similarly, you can toggle a set of outputs with an instruction such as:

```
store reg_10201 xor 8 to reg_10201
```

Since the effect of the XOR operator is to invert any bit that combines with 1, and to leave alone any bit that combines with 0, this instruction causes output 4's state to change The number 8, expressed in binary, is 0000 1000, a pattern with only the fourth bit set.

IMPORTANT! The above instructions will read, modify, and then write back a value from a bank of outputs. This will work in Quickstep because no other task or external communication is allowed to affect the outputs during the execution of an individual step. Use extreme caution, however, in programs that read a bank of outputs in one step, and then store the information back in another step. If any outside influence has changed an output state in the interim, the write operation will undo that change, which can potentially cause unexpected operations to occur.

Performing Boolean Operations on Binary Numbers

Performing Boolean Operation on Binary Numbers

Quickstep can perform bit-wise Boolean operations on binary numbers. Bit-wise Boolean instructions perform a bit-wise comparison between the identically positioned bits in two numeric expressions.

The STORE instruction can perform the following Boolean operations on 32-bit binary numbers.

- AND
- OR
- XOR
- NOR
- NXOR
- NAND
- ANDNOT

AND

Using a Boolean AND instruction to perform a bit-wise comparison gives the following results:

Bit in first operand	Bit in second operand	Result
0	0	0
0	1	0
1	0	0
1	1	1

Boolean AND instructions can be expressed in either of the following formats:

```
Store Reg_15 and Reg_16 to Reg_17
Store Reg_15 & Reg_16 to Reg_17
```

This instruction does the following:

1. Takes the first operand, the value in register 15, and expresses it as a 32-bit binary number using two's-complement notation. For a description of two's complement notation, see page 3-54.
2. Logically ANDs the first operand on a bit-wise basis to the second operand, the value in register 16 (also expressed as a 32-bit binary number using two's-complement notation).
3. Stores the result in register 17.

The value in register 15 is:	0000 0000 0000 0000 0000 0000 0110 0101
The value in register 16 is:	0000 0000 0000 0000 0000 0000 0111 0011
The value stored to register 17 is:	0000 0000 0000 0000 0000 0000 0110 0001

NAND

NAND means NOT AND. A Boolean NAND operation is the opposite of a Boolean AND operation and gives the following results:

Bit in first operand	Bit in second operand	Result
0	0	1
0	1	1
1	0	1
1	1	0

Boolean NAND instructions are expressed in the following format:

```
Store Reg_15 nand Reg_16 to Reg_17
```

The instruction expresses the values in registers 15 and 16 as 32-bit binary numbers using two's-complement notation and performs a logical NAND operation on them and then stores the result in register 17.

The value in register 15 is:	0000 0000 0000 0000 0000 0000 0110 0101
The value in register 16 is:	0000 0000 0000 0000 0000 0000 0111 0011
The value stored to register 17 is:	1111 1111 1111 1111 1111 1111 1001 1110

OR

Using a Boolean OR instruction to perform a bit-wise comparison gives the following results:

Bit in first operand	Bit in second operand	Result
0	0	0
0	1	1
1	0	1
1	1	1

Boolean OR instructions can be expressed in either of the following formats:

```
Store Reg_15 or Reg_16 to Reg_17
Store Reg_15 | Reg_16 to Reg_17
```

The instruction expresses the values in registers 15 and 16 as 32-bit binary numbers using two's-complement notation and performs a logical OR operation on them and then stores the result in register 17.

The value in register 15 is:	0000 0000 0000 0000 0000 0000 0110 0101
The value in register 16 is:	0000 0000 0000 0000 0000 0000 0111 0011
The value stored to register 17 is:	0000 0000 0000 0000 0000 0000 0111 0111

NOR

NOR means NOT OR. A Boolean NOR operation is the opposite of a Boolean OR operation and gives the following results:

Bit in first operand	Bit in second operand	Result
0	0	1
0	1	0
1	0	0
1	1	0

Boolean NOR instructions are expressed in the following format:

```
Store Reg_15 nor Reg_16 to Reg_17
```

The instruction expresses the values in registers 15 and 16 as 32-bit binary numbers using two's-complement notation and performs a logical NOR operation on them and then stores the result in register 17.

The value in register 15 is:	0000 0000 0000 0000 0000 0000 0110 0101
The value in register 16 is:	0000 0000 0000 0000 0000 0000 0111 0011
The value stored to register 17 is:	1111 1111 1111 1111 1111 1111 1000 1000

XOR

XOR means Exclusive OR. A Boolean XOR operation gives the following results:

Bit in first operand	Bit in second operand	Result
0	0	0
0	1	1
1	0	1
1	1	0

Boolean XOR instructions can be expressed in either of the following formats:

```
Store Reg_15 xor Reg_16 to Reg_17
Store Reg_15 ^ Reg_16 to Reg_17
```

The instruction expresses the values in registers 15 and 16 as 32-bit binary numbers using two's-complement notation and performs a logical XOR operation on them and then stores the result in register 17.

The value in register 15 is:	0000 0000 0000 0000 0000 0000 0110 0101
The value in register 16 is:	0000 0000 0000 0000 0000 0000 0111 0011
The value stored to register 17 is:	0000 0000 0000 0000 0000 0000 0001 0110

NXOR

NXOR means Not Exclusive OR. A Boolean NXOR operation gives the following results:

Bit in first operand	Bit in second operand	Result
0	0	1
0	1	0
1	0	0
1	1	1

Boolean NXOR instructions are expressed in the following format:

```
Store Reg_15 nxor Reg_16 to Reg_17
```

The instruction expresses the values in registers 15 and 16 as 32-bit binary numbers using two's-complement notation and performs a logical NXOR operation on them and then stores the result in register 17.

The value in register 15 is: 0000 0000 0000 0000 0000 0000 0110 0101

The value in register 16 is: 0000 0000 0000 0000 0000 0000 0111 0011

The value stored to register 17 is: 1111 1111 1111 1111 1111 1111 1110 1001

ANDNOT

ANDNOT is a combination of two boolean operations AND and NOT. This STORE instruction

```
Store Reg_15 andnot Reg_16 to Reg_17
```

does the following:

1. Takes the second operand, the value in register 16, expresses it as a 32-bit binary number using two's-complement notation, and then inverts that number.

   This means that 0000 0000 0000 0000 0000 0000 0111 0011 becomes 1111 1111 1111 1111 1111 1111 1000 1100.

2. Takes the first operand, the value in register 15, and expresses it as a 32-bit binary number using two's-complement notation

3. Logically ANDs the first operand on a bit-wise basis to the inverted second operand.

4. Stores the result in register 17.

The value in register 15 is: 0000 0000 0000 0000 0000 0000 0110 0101

The value in register 16 is: 0000 0000 0000 0000 0000 0000 0111 0011

The value stored to register 17 is: 0000 0000 0000 0000 0000 0000 0000 0100

Bit in first operand	Bit in second operand	Second operand Inverted	Result
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

NOTE: Two's complement is a method of representing signed binary numbers, where: 0000 0000 0000 0000 0000 0000 0000 0000 represents zero
and 0111 1111 1111 1111 1111 1111 1111 1111 represents 2,147,483, 647.

Negative one is represented by: 1111 1111 1111 1111 1111 1111 1111 1111 and the most negative number -2,147,483, 648 is represented by:
1000 0000 0000 0000 0000 0000 0000 0000

Using Bit-wise Boolean Algebra in Your Quickstep Program

You can use Boolean STORE instructions to perform bit-wise Boolean algebra on numbers in order to mask data from 8-, 16-, and 32-bit data sources. The following example reads the binary value represented by the state of the controller's first set of 32 inputs and performs a boolean AND operation with the number 4095. As a result of this instruction, the binary representation of the first 12 inputs only is stored in register 10.

```
store reg_11001 and 4095 to reg_10
```

The value in reg_11001 is:	0100 0110 1111 0100 0000 1111 1001 1101
4095 in binary is:	0000 0000 0000 0000 0000 1111 1111 1111
The resulting number stored in register 10 is:	0000 0000 0000 0000 0000 1111 1001 1101

NOTE: For additional information and examples, refer to the Application Note, *Bit Level Operators and CTC Controllers*.

Sample Programs

Contents

Introduction

This appendix contains a series of sample programs. Each program contains extensive comments describing the program and the functions preformed in each step.

The sample programs are as follows:

- Program to control a simple machine
- Using registers - cycle counting
- Using counters
- Multi-tasking example
- Using thumbwheels and displays
- Using analog inputs and outputs
- tepping motor control - using stepping motor instructions
- Servo motor control - using servo instructions
- Servo motor control - velocity mode example
- Using the Data Table in an iterative program (to store motor coordinates)
- Using a circular buffer and the phantom register
- Accessing multiple I/O points using the phantom register
- Program showing a multi-station Indexing table

Copies of the sample programs are available in the **QSWIN** directory and are in a self-extracting compressed file named **SAMPLES.EXE**. **SAMPLES.EXE** places the files in the directory that the EXE file is located in. If you want the samples in subdirectory, create the subdirectory and place **SAMPLES.EXE** in that directory before extracting the sample files.

IMPORTANT! These programs are not intended to be complete working programs for any specific machine. Instead, they are intended to illustrate programming concepts that can be used in certain programs and must be combined with sound engineering practices to achieve adequate levels of functionality and safety

Program to Control a Simple Machine

```
[1] INITIALIZE
    ;;; This program illustrates some of the basic
    ;;; concepts of writing automation programs using
    ;;; Quickstep.
    ;;;
    ;;; This text constitutes a comment for step 1.
    ;;; Comments may be used to explain what is happening
    ;;; in a step, to provide information about the status
    ;;; of your machine, and to document the use of
    ;;; registers, flags, etc.  Use comments; six months
    ;;; later you'll be glad you did.
    ;;;
    ;;; Notice the name at the top of this step: INITIALIZE.
    ;;; Use of step names is highly recommended; future
    ;;; versions of Quickstep will require step names.
    ;;; Later in the program, if you wish to return to
    ;;; this step, you can use the instruction goto INITIALIZE.
    ;;; Extensive use of such labels makes your program more
    ;;; readable.
    ________________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ________________________________________
    store 0 to Part_Counter
    delay 1 sec goto Next

[2] BEGIN_SEQUENCE
    ;;; The two horizontal lines below delineate any
    ;;; changes made to the controller's digital outputs
    ;;; during this step.  These output changes take place
    ;;; as soon as the controller enters the step.
    ;;;
    ;;; Outputs can have two symbolic names. One symbolic
    ;;; name is for turning the output on; the other is for
    ;;; turning it off.  In this program, outputs use the symbol
    ;;; names Output_n_On to indicate an instruction to
    ;;; turn an output, and Output_n_Off to indicate an
    ;;; instruction to turn an output off. (n is the
    ;;; number of the output.)
    ;;;
    ;;; Of course, normally you would name these output
    ;;; changes to reflect the operation they perform on
    ;;; your machine.
    ________________________________________
    Output_15_On
    ________________________________________
    delay 1 sec goto Next

[3] LOAD_PART
    ;;; In this step, an output is turned on to actuate a
    ;;; pneumatic cylinder, loading a new part into the
    ;;; machine being controlled.
    ;;;
    ;;; Two instructions are programmed below; one
    ;;; represents the normal path of the program (delay 1
    ;;; sec goto next), the other looks for a fault
    ;;; condition via a sensor:
    ;;; monitor Fault_Sensor goto FAULT_DETECTED.
    ;;;
    ;;; This further illustrates the value of step names.
```

```
    ;;; The name FAULT_DETECTED implies that this second
    ;;; instruction is for fault detection, making the
    ;;; program more self-explanatory.
    ____________________________________________
    Output_1_On
    ____________________________________________
    delay 1 sec goto Next
    monitor Fault_Sensor goto FAULT_DETECTED

[4] CLAMP_PART
    ;;; In this step, the output which was turned on in
    ;;; the previous step is now turned off, and five
    ;;; other outputs are turned on, simultaneously
    ;;; actuating five clamping cylinders.
    ;;;
    ;;; The monitor instruction below requires five
    ;;; separate limit switch inputs to be closed before
    ;;; proceeding to the next step.  These limit switches
    ;;; are located at the end of each clamping cylinder,
    ;;; and are used to confirm their full actuation.  The
    ;;; input names “Limit_A”, etc., may be programmed to
    ;;; refer to input as either normally-open or
    ;;; normally-closed.  Symbolic names are defined using
    ;;; the Symbol Browser.
    ____________________________________________
    Output_1_Off
    Output_5_On
    Output_6_On
    Output_7_On
    Output_8_On
    Output_9_On
    ____________________________________________
    monitor (and Limit_A Limit_B Limit_C Limit_D Limit_E) goto Next

[5] STAMP_PART
    ;;; In this step, a stamping ram is actuated by
    ;;; turning on output 10. Because we want the “on”
    ;;; time of this operation to be adjustable by the
    ;;; machine’s operator, we have programmed a time
    ;;; delay referring to a thumbwheel switch.
    ;;;
    ;;; In the instruction below, the four-digit thumbwheel
    ;;; array is interpreted as two digits of seconds and
    ;;; two digits of fractions of a second (ssff), allowing
    ;;; the setting of time delays from 1/100 second
    ;;; (0001) to 99.99 seconds (9999).
    ____________________________________________
    Output_10_On
    ____________________________________________
    delay thwl_1 ssff goto Next

[6] UNCLAMP
    ;;; This step simultaneously retracts the stamping ram
    ;;; (output 10) and retracts the clamps.  Notice how
    ;;; much more readable this step would be if these
    ;;; output operations had explanatory names.  Such
    ;;; names may be assigned using the Symbol Browser.
    ____________________________________________
    Output_5_Off
    Output_6_Off
    Output_7_Off
```

```
    Output_8_Off
    Output_9_Off
    Output_10_Off
    ____________________________________

    delay 1 sec goto Next

[7] EJECT
    ;;; This step actuates an eject cylinder using output 11,
    ;;; ejecting the completed part.
    ;;; In addition, the value in a numeric register named
    ;;; Part_Counter is incremented; this counts the part
    ;;; just produced.  The updated count is then stored
    ;;; to a display which has been named Part_Count_Display.
    ____________________________________

    Output_11_On
    ____________________________________

    store Part_Counter + 1 to Part_Counter
    store Part_Counter to Part_Count_Display
    delay 1 sec goto Next

[8] END_SEQUENCE
    ;;; In this step, the eject cylinder is retracted. We
    ;;; then return to the beginning of the cycle to start
    ;;; over again.
    ____________________________________

    Output_11_Off
    ____________________________________

    delay 1 sec goto BEGIN_SEQUENCE

[9] FAULT_DETECTED
    ;;; The controller only executes this step if a fault
    ;;; condition is sensed.  Here, we stop the controller
    ;;; and turn on a fault indicator light attached to
    ;;; output 12.
    ;;;
    ;;; When the controller is restarted by the operator, the
    ;;; program will return to the step called INITIALIZE.
    ____________________________________

    Fault_Indicator_On
    ____________________________________

    stop goto INITIALIZE
```

Using Registers - Cycle Counting

```
[1] INITIALIZE
    ;;; This program illustrates some of the uses of
    ;;; numeric registers.
    ;;;
    ;;; In this program, a register named Cycle_Ctr is
    ;;; used as a cycle counter; after 100 cycles, the
    ;;; machine will be told to shut down.
    ;;;
    ;;; In addition, a nonvolatile register named
    ;;; Lifetime_Cycle_Ctr is used as a lifetime cycle
    ;;; counter for the machine. This information will be
    ;;; retained even when power is removed from the machine,
    ;;; and is useful for maintenance, etc.
    ;;;
    ;;; In this first step, we will initialize the machine
    ;;; by turning all of the controller's outputs off,
    ;;; and storing 0 to register Cycle_Ctr. We do not
    ;;; initialize register Lifetime_Cycle_Ctr, because we
    ;;; want it to retain its previous count.
    ;;;
    ;;; NOTE:  the above initializations are not necessary
    ;;;        on power-up, as the controller automatically
    ;;;        turns its outputs off and zeros all volatile
    ;;;        registers when power is first applied. However,
    ;;;        we are occasionally jumping back to this step
    ;;;        from within the program, and these initializations
    ;;;        are necessary under those circumstances.
    ____________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________
    store 0 to Cycle_Ctr
    goto Next

[2] BEGIN_CYCLE
    ;;; This step will be the start of a cycle for the
    ;;; machine we are controlling. No detail will be
    ;;; given, as the machine is hypothetical.
    ____________________________________
    Output_1_On
    Output_18_On
    Output_5_On
    ____________________________________
    monitor (and Limit_Switch_C Limit_Switch_A Limit_Switch_B) goto Next

[3] MID_CYCLE
    ;;; The program for the hypothetical machine continues...
    ____________________________________
    Output_18_Off
    Output_15_On
    Output_5_Off
    ____________________________________
    delay 200 ms goto Next

[4] END_OF_CYCLE
    ;;; This is the last step of our machine's cycle. Here,
    ;;; we will count the finished part just produced by
    ;;; adding one to Cycle_Ctr, as well as
    ;;; Lifetime_Cycle_Ctr. Then, we'll test Cycle_Ctr to
    ;;; determine if more parts should be produced in this
```

```
    ;;; lot (“if Cycle_Ctr >= 100 goto END_OF_BATCH”). If
    ;;; we have not yet made 100 parts, then the program
    ;;; returns to the beginning of the cycle.
    ____________________________________________
    Output_15_Off
    Output_1_Off
    ____________________________________________
    store Cycle_Ctr + 1 to Cycle_Ctr
    store Lifetime_Cycle_Ctr + 1 to Lifetime_Cycle_Ctr
    if Cycle_Ctr >=100 goto END_OF_BATCH
    goto BEGIN_CYCLE

[10] END_OF_BATCH
    ;;; Having made a batch of 100 parts, we will now wait
    ;;; until the operator of the machine removes the
    ;;; finished parts and restarts the machine.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    stop goto INITIALIZE
```

Using Counters

```
[1] INITIALIZE
    ;;; This program illustrates the use of counters.
    ;;; Although any register may be used as a counter, the
    ;;; first 8 numeric registers may be configured to
    ;;; perform automatic counting of pulses appearing on
    ;;; any of the controller's inputs. To do this, the
    ;;; counters must first be initialized with a Start
    ;;; Counter instruction.
    ;;;
    ;;; The program uses counters in a conveyor/palletizer
    ;;; application. Counter 1 counts items on a conveyor
    ;;; belt; when 10 items have gone by, we trigger a
    ;;; palletizing sequence (not shown here).
    ;;;
    ;;; NOTE: when a counter is initialized, we may specify
    ;;; inputs for counting up and down, as well as for
    ;;; resetting the counter to zero.
    ____________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________
    start Counter_1 up (Item_Sense_Input) down (Manual_Down_Count_Input)
      reset (Manual_Count_Reset)
    goto Next

[2] WAIT_FOR_TEN
    ;;; In this step, we will start the conveyor by turning
    ;;; on output 1 (Conveyor_On), reset our counter to zero
    ;;; (via programmed instruction), and then wait for the
    ;;; counter to count ten items before jumping to the
    ;;; next step.
    ____________________________________
    Conveyor_On
    ____________________________________
    reset Counter_1
    if Counter_1 >=10 goto Next

[3] STOP_CONVEYOR
    ;;; Here, we have already sensed ten items on the
    ;;; conveyor, so we will stop the conveyor motor by
    ;;; turning output 1 off (Conveyor_Off), and then
    ;;; continue to our palletizing program (not shown here).
    ____________________________________
    Conveyor_Off
    ____________________________________
    goto PALLETIZE
```

```
[1] INITIALIZE
    ;;; This program illustrates one of the uses of
    ;;; multi-tasking.  As an example, we'll use a program
    ;;; for an automatic riveting machine.
    ;;;
    ;;; On this machine, a vibratory feed system is used to
    ;;; feed rivets into a singulator to feed exactly one
    ;;; rivet into the machine at a time. At the same time,
    ;;; using a pick-and-place robot, we are feeding a new
    ;;; workpiece into the machine to be riveted. The machine
    ;;; has been designed to allow these two operations to
    ;;; occur simultaneously, so we will use two
    ;;; simultaneous, independent tasks to control them.
    ____________________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________________
    goto Next

[2] START_CYCLE
    ;;; Here, we will just check to make sure the machine
    ;;; is not out of rivets, or out of workpieces, by
    ;;; checking two proximity sensors (Rivet_Supply and
    ;;; Parts_Supply) before proceeding.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    monitor (and Rivet_Supply Parts_Supply) goto Next
    goto SHUT_DOWN

[3] START_TASKS
    ;;; In this step, we will start two separate programs
    ;;; by using the do instruction to commence multi-
    ;;; tasking. The first program, starting at the step
    ;;; labelled FEED_RIVET, feeds a rivet into the
    ;;; machine. The second task, starting at a step
    ;;; labelled FEED_PART, operates the pick-and-place
    ;;; robot to feed a new workpiece into the machine.
    ;;;
    ;;; Only after both of these tasks are done, does the
    ;;; main program continue on to the next step.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    do (FEED_RIVET FEED_PART) goto Next

[4] CHECK_BEFORE_TRIP
    ;;; In this step, we'll perform a final check of the
    ;;; machine's status before tripping the riveting press.
    ;;; Two sensors, on inputs Rivet_Loaded and Part_Loaded,
    ;;; are checked to insure the presence of both a rivet
    ;;; and a workpiece in the press.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    monitor (and Rivet_Loaded Part_Loaded) goto Next
    goto SHUT_DOWN
```

```
[5] TRIP_PRESS
    ;;; In this step we fire the riveting press by turning
    ;;; on output 1, Rivet_Press_On.
    ____________________________________________
    Rivet_Press_On
    ____________________________________________
    delay 250 ms goto Next

[6] END_OF_CYCLE
    ____________________________________________
    Rivet_Press_Off
    ____________________________________________
    goto START_CYCLE

[10] FEED_RIVET
    ;;; This is the beginning of a short program (“task”) that
    ;;; feeds a rivet into the riveting head of our machine.
    ;;; This program operates a singulator, which separates a
    ;;; continuous line of rivets from a vibratory feed into
    ;;; an individual rivet to be fed. We start by operating
    ;;; a hold-back cylinder, actuated via a solenoid valve
    ;;; (output 2, Hold_Back_On)
    ____________________________________________
    Hold_Back_On
    ____________________________________________
    delay 250 ms goto Next

[11] DROP_RIVET
    ;;; The hold-back cylinder will retain all but the first
    ;;; rivet in line. Now, by actuating another valve, we
    ;;; will retract the drop cylinder, allowing the first
    ;;; rivet to fall into the riveting head.
    ____________________________________________
    Drop_Rivet_On
    ____________________________________________
    delay 350 ms goto Next

[12] RIVET_FED
    ;;; Now that the rivet has dropped, we can return both
    ;;; singulation cylinders to their original position.
    ;;; This is the end of the FEED_RIVET task, as indicated
    ;;; by the instruction “done”.
    ____________________________________________
    Hold_Back_Off
    Drop_Rivet_Off
    ____________________________________________
    done

[50] FEED_PART
    ;;; This is the beginning of a program to operate a
    ;;; pick-and-place robot which will feed a new workpiece
    ;;; into our riveter. In this step, we will operate a
    ;;; gripper to grasp a new workpiece.
    ____________________________________________
    Gripper_On
    ____________________________________________
    delay 200 ms goto Next
```

```
[51] EXTEND_PART
    ;;; In this step, we trigger a horizontal motion of the
    ;;; robot grasping the part.  We'll wait for a limit
    ;;; switch, Arm_Extended, to confirm the completion of
    ;;; this motion.
    ____________________________________
    Extend_Arm
    ____________________________________
    monitor Arm_Extended goto Next

[52] LOWER_PART
    ;;; In this step, we will lower the part into a holding
    ;;; fixture. Once again a limit switch, Part_Lowered,
    ;;; indicates completion of the motion.
    ____________________________________
    Lower_Arm
    ____________________________________
    monitor Part_Lowered goto Next

[53] RELEASE_PART
    ;;; Now, we can release the gripper, dropping the part
    ;;; into its fixture. Due to the design of our robot,
    ;;; we may simultaneously raise the gripper.
    ____________________________________
    Gripper_Off
    Raise_Arm
    ____________________________________
    monitor Part_In_Place goto Next

[54] RETRACT_ARM
    ;;; Having released the part, we may now retract the
    ;;; pick-and-place arm to its rest position.
    ____________________________________
    Retract
    ____________________________________
    done

[100] SHUT_DOWN
    ;;; A fault has been detected, so we'll turn off all
    ;;; outputs and stop the controller. First, we'll
    ;;; terminate multitasking by executing a cancel other
    ;;; tasks instruction.
    ;;;
    ;;; When the operator restarts the controller, the program
    ;;; will jump back to the beginning. Because we've ended
    ;;; multitasking, this will not cause recursion.
    ____________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________
    cancel other tasks
    stop goto INITIALIZE
```

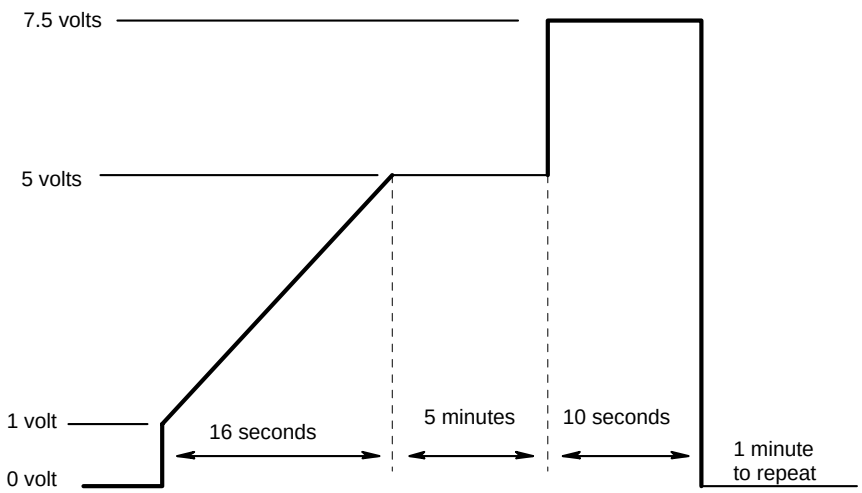
Using Thumbwheels and Displays

```
[1] INITIALIZE
    ;;; This program illustrates the use of thumbwheels and
    ;;; numeric displays.
    ;;;
    ;;; Thumbwheels allow manual entry of numeric parameters,
    ;;; while displays allow numeric information to be viewed
    ;;; by the operator of a machine.
    ;;;
    ;;; This sample program uses two thumbwheel arrays and
    ;;; a numeric display to control a cut-off machine.  A
    ;;; stepping motor is used to meter out a certain length
    ;;; of material, after which a cutter is triggered.  We
    ;;; want the operator to be able to dial in the desired
    ;;; length of material, as well as the quantity of parts
    ;;; to be cut before shutting down.
    ;;;
    ;;; The first step sets up the stepping motor (Feed_Motor)
    ;;; with a profile instruction, and initializes the
    ;;; Part_Counter (a numeric register) to zero.
    ________________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ________________________________________
    profile Feed_Motor (half) basespeed=100 maxspeed=7500 accel=420
      decel=420
    store 0 to Part_Counter
    goto Next

[2] FEED_MATERIAL
    ;;; The machine's operator dials the desired material
    ;;; length into thumbwheel 1, but this length is expressed
    ;;; in inches.  Since we know that the stepping motor must
    ;;; turn 50 steps in order to feed one inch of material,
    ;;; we will multiply this value by 50.  The resulting
    ;;; value is temporarily stored in Travel, for use in the
    ;;; following turn motor instruction.
    ________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ________________________________________
    store Material_Length * 50 to Travel
    turn Feed_Motor cw Travel steps
    monitor Feed_Motor:stopped goto Next

[3] CUT_OFF
    ;;; In this step, we will cut off the material fed in the
    ;;; previous step by actuating a cutter. A time delay is
    ;;; programmed which should be sufficient to allow the
    ;;; cutter to complete its travel.
    ________________________________________
    Cutter_On
    ________________________________________
    delay 500 ms goto Next

[4] RETRACT_CUTTER
    ;;; Now we will retract the cutter by turning off the
    ;;; output.  At the same time, we will count the part just
    ;;; cut (by adding one into Part_Counter), and display the
    ;;; current count to the operator (store Part_Counter to
    ;;; Parts_Count_Display).
    ;;;
    ;;; This count is then compared with a value dialed into
    ;;; a second thumbwheel (Batch_Total) to determine if the
```

```
    ;;; proper number of parts has been made.  If so, we will
    ;;; shut down the machine (END_OF_BATCH), otherwise we'll
    ;;; return to feed another part.
    ____________________________________
    Cutter_Off
    ____________________________________
    store Part_Counter + 1 to Part_Counter
    store Part_Counter to Parts_Count_Display
    if Part_Counter >=Batch_Total goto END_OF_BATCH
    goto FEED_MATERIAL

[5] END_OF_BATCH
    ;;; Now we have completed the desired number of parts; the
    ;;; stop instruction causes the controller to stop until
    ;;; the operator presses the START switch connected to the
    ;;; controller's input 1.
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    stop goto INITIALIZE
```

```
[1] SET_INITIAL_CONDITIONS
    ;;; This program demonstrates the use of analog I/O
    ;;; instructions, using two independent tasks.
    ;;;
    ;;; The first task controls a temperature by taking an
    ;;; analog output through a timed ramp.  This ramp is
    ;;; generated by a series of programmed commands which
    ;;; determine the initial value, the rate of increase,
    ;;; and the terminal value.
    ;;;
    ;;; The other task continuously monitors a pressure
    ;;; transducer connected to an analog input.  The input
    ;;; value is ranged into engineering units from 0 to 100
    ;;; PSIG, and is displayed on a numeric display.  In
    ;;; addition, the peak pressure value is maintained and
    ;;; displayed on a second numeric display. If the pressure
    ;;; goes above a fixed limit (75 PSIG), the temperature
    ;;; control value is sent to zero, and the entire process
    ;;; is stopped.
    ;;;
    ;;; This step contains three instructions that perform
    ;;; certain initialization functions within the
    ;;; controller.  The analog output, used to control the
    ;;; process temperature, is initially set to zero volts.
    ;;; Both the numeric register used to accumulate the peak
    ;;; pressure reading and the numeric display that displays
    ;;; the actual pressure reading are also initialized to
    ;;; zero. The last instruction in this step sends the
    ;;; controller to step 2.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store 0 to Temp_Control
    store 0 to Peak_Value
    store 0 to Current_Pressure_Display
    goto Next

[2] START_TASKS
    ;;; This step begins multi-tasking and starts two tasks:
    ;;; TEMP_PROCESS and MONITOR_PRESSURE.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    do (TEMP_PROCESS MONITOR_PRESSURE) goto SET_INITIAL_CONDITIONS

[10] TEMP_PROCESS
    ;;; This task controls the ramping of the analog output
    ;;; (see the ramping sequence shown in the accompanying
    ;;; illustration).  Storing 1000 to the register called
    ;;; Process_Temp sets the initial value at 1 volt. This
    ;;; value will be stored to the analog output controlling
    ;;; temperature, Temp_Control, and incremented in
    ;;; subsequent steps to create the desired ramp.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store 1000 to Process_Temp
    goto Next
```

Ramp Sequence Used to Generate Ramp




```
[11] BEGIN_ANALOG_RAMP
    ;;; This step creates the initial analog ramp. The
    ;;; controller updates the temperature control value
    ;;; by storing the Process_Temp setting to the analog
    ;;; output called Temp_Control.
    ;;;
    ;;; This step goes on to increase the Process_Temp
    ;;; setting via a store instruction which adds 25 to the
    ;;; register.  This does not change the voltage at the
    ;;; analog output yet.
    ;;;
    ;;; The next instruction in this step tests the value in
    ;;; the register to determine if our ramp is complete,
    ;;; having reached a value of 5000 volts.  If so, the
    ;;; program will proceed to the next step.
    ;;;
    ;;; Otherwise, the time delay instruction will ultimately
    ;;; time out.  However, the destination of the time delay
    ;;; instruction is this same step, so the controller
    ;;; executes it as if it were a new step. This time the
    ;;; new, updated value in Process_Temp is stored to the
    ;;; analog output, resulting in an increase in voltage.
    ;;;
    ;;; Then as before the controller increments the value in
    ;;; Process_Temp and the time delay begins.  This process
    ;;; continues, with the voltage increasing by 25 mv each
    ;;; 100 ms, until the if instruction is satisfied,
    ;;; indicating that the 5 volt level has been reached.
    ;;;
    ;;; Two factors dictate the slope of the ramp being
    ;;; generated; the time delay duration (which determines
    ;;; the update interval) and the amount added to the
    ;;; voltage value each time the step is re-executed.  By
    ;;; modifying the instructions within this step, you can
    ;;; create ramps of various slopes.
```

```
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store Process_Temp to Temp_Control
    store Process_Temp + 25 to Process_Temp
    if Process_Temp > 5000 goto Next
    delay 100 ms goto BEGIN_ANALOG_RAMP

[12] DELAY
    ;;; After the voltage of analog output Temp_Control has
    ;;; ramped from 1 to 5 volts, the step uses a delay
    ;;; instruction to maintain this level for a period of
    ;;; five minutes. Such an operation is sometimes called
    ;;; a soak interval.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    delay 5 min goto Next

[13] STEP_FUNCTION
    ;;; As shown in the ramp illustration, the program
    ;;; executes a step function which increases the voltage
    ;;; to a level of 7.5 volts for a period of ten seconds.
    ;;; The store instruction shown below sends the number
    ;;; 7500, representing 7.5 volts, to the analog output.
    ;;; The delay instruction generates the required ten
    ;;; second time delay.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store 7500 to Temp_Control
    delay 10 sec goto Next

[14] TEMP_PROCESS_END
    ;;; This is the last step in the ramp generation.  It
    ;;; completes the process by returning the analog output
    ;;; to a value of zero volts and, after a one minute time
    ;;; delay, returns to step TEMP_PROCESS.
    ;;;
    ;;; Note the absence of a “done” instruction.  In this
    ;;; case, each task simply loops back upon itself,
    ;;; continuously executing its intended function.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store 0 to Temp_Control
    delay 1 min goto TEMP_PROCESS

[100] MONITOR_PRESSURE
    ;;; This task operates in parallel with the ramping
    ;;; program.  It continuously monitors a pressure
    ;;; transducer connected to analog input Transducer_Input.
    ;;; The current and peak pressures are displayed for the
    ;;; system’s operator, and the pressure is tested to
    ;;; insure that a predefined limit is never exceeded.
    ;;;
    ;;; The pressure transducer provides an output signal
    ;;; ranging from 0.5 to 5.5 volts in response to an
    ;;; applied pressure of 0 to 100 PSIG.  The analog input
    ;;; reports this range as a number from 500 (0.5 V.) to
    ;;; 5500 (5.5 V.).  The conversion math necessary to
    ;;; convert these values into units of PSIG is:
```

```
    ;;;
    ;;; Pressure_PSIG = (Transducer_Input - 500) / 50
    ;;;
    ;;; Two successive instructions are used to perform this
    ;;; math.  The first instruction reads the current value
    ;;; of Transducer_Input, subtracts the offset of 500,
    ;;; and stores the result in register Intermediate_Val.
    ;;; The second instruction divides that result by 50, then
    ;;; stores the finished value in register Pressure_PSIG.
    ;;;
    ;;; Note that the intermediate value could have been
    ;;; stored in Pressure_PSIG, since it would have been
    ;;; immediately replaced with the final value.
    ;;;
    ;;; The third instruction updates the reading on the
    ;;; Current_Pressure_Display by storing our new pressure
    ;;; value to it.
    ;;;
    ;;; The “if” instruction will then test to see if the
    ;;; new pressure exceeds the Peak_Value we’ve previously
    ;;; stored.  If so, the program will continue to the
    ;;; next step to store the new Peak_Value.
    ;;;
    ;;; Otherwise, the instruction “goto MONITOR_PRESSURE”
    ;;; will cause the current step to be re-executed.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store Transducer_Input - 500 to Intermediate_Val
    store Intermediate_Val / 50 to Pressure_PSIG
    store Pressure_PSIG to Current_Pressure_Display
    if Pressure_PSIG > Peak_Value goto Next
    goto MONITOR_PRESSURE

[101] UPDATE_PEAK_VALUE
    ;;; If we’ve reached this step, the current pressure
    ;;; exceeds the previous peak reading stored in Peak_Value.
    ;;; Therefore, we will update Peak_Value by storing the
    ;;; current Pressure_PSIG to it, and also update the
    ;;; Peak_Pressure_Display with the same value.
    ;;;
    ;;; Each time a new peak pressure is sensed, it is also
    ;;; checked to determine if it has exceeded a maximum
    ;;; limit of 75 PSIG. If the pressure is over this limit,
    ;;; the controller jumps to step OVER_PRESSURE.
    ;;;
    ;;; WARNING:  No single device, including a programmable
    ;;; controller, should be the sole device responsible
    ;;; for detecting a fault condition that could result
    ;;; in human injury or substantial economic damage.
    ;;; Proper design practice dictates the use of appropriate
    ;;; judgment in the design of backup safety systems.
    ;;; This is the responsibility of the machine designer.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store Pressure_PSIG to Peak_Value
    store Peak_Value to Peak_Pressure_Display
    if Pressure_PSIG >=75 goto OVER_PRESSURE
    goto MONITOR_PRESSURE
```

WARNING! No single device, including a programmable controller, should be the sole device responsible for detecting a fault condition that could result in human injury or substantial economic damage. Proper design practice dictates the use of appropriate judgment in the design of backup safety systems. This is the responsibility of the machine designer.

```
[250] OVER_PRESSURE
    ;;; This step cancels the other tasks and sets the
    ;;; temperature control to zero. The program then goes to
    ;;; the next step.
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    cancel other tasks
    store 0 to Temp_Control
    goto Next

[251] PRESSURE_CHECK
    ;;; This step returns the controller to MONITOR_PRESSURE
    ;;; where the controller will continue to update the
    ;;; operator displays. The operator will then have to
    ;;; press a reset switch to return to the first step in
    ;;; the program and restart the sequence.
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    goto MONITOR_PRESSURE
```

Programming Hints

Because of the `goto MONITOR_PRESSURE` instruction in step MONITOR_PRESSURE, the display program in step continuously cycles and constantly updates the pressure reading. Without this instruction, the pressure value would only be displayed once, upon first entering the step and not be updated as the pressure changed.

```
[1] INITIALIZE
    ;;; This program illustrates the use of the stepping motor
    ;;; instructions.  In this step, we'll execute a profile
    ;;; instruction to establish the initial motion parameters
    ;;; for the motor.  A turn instruction may not be executed
    ;;; until this initial profile has been established.
    ;;;
    ;;; Next, a "search ccw and zero" instruction tells the
    ;;; motor to begin turning slowly in the counter-clockwise
    ;;; direction, searching for a "home" limit switch.  This
    ;;; limit switch, connected to the stepping motor control
    ;;; board, establishes a reference point for future
    ;;; motions.  The motor automatically stops when this
    ;;; home position has been sensed.
    ____________________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________________
    profile Motor_1 (half) basespeed=100 maxspeed=7500 accel=400 decel=400
    search ccw and zero Motor_1
    monitor Motor_1:stopped goto Next

[2] FIRST_POSITION
    ;;; In this step, we send the stepping motor to its first
    ;;; position, using an absolute turn motor instruction.
    ;;; This instruction sends the motor to a coordinate
    ;;; position based on the previously established "home"
    ;;; zero position.
    ;;; The "monitor motor#1:stopped" instruction will take
    ;;; the controller to the next step only when the stepping
    ;;; motor board has finished sending pulses to the motor.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    turn Motor_1 to 12500
    monitor Motor_1:stopped goto Next

[3] SECOND_POSITION
    ;;; This step sends the motor to a second position, still
    ;;; based on the originally-established home position.
    ;;;
    ;;; After the motor motion is complete, the controller
    ;;; returns to the previous step to repeat the motor's
    ;;; two-position sequence.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    turn Motor_1 to 5000
    monitor Motor_1:stopped goto FIRST_POSITION
```

Using Servo Motor Instructions

Programming a Servo

```
[1] INITIALIZE
    ;;; This program illustrates the programming of SERVO
    ;;; instructions by setting up a simple back-and-forth
    ;;; servo motion.
    ;;;
    ;;; The speed of this motion is determined by a
    ;;; numeric register called Servo_Speed, allowing the
    ;;; speed to be tuned while the program is executing.
    ;;; An additional task will continuously update a
    ;;; numeric display with the instantaneous servo error,
    ;;; allowing us to view the servo's accuracy.
    ;;;
    ;;; In this first step, an initial value is stored in
    ;;; the register Servo_Speed, to be used further on
    ;;; in the program.  Also, a profile instruction
    ;;; establishes initial motion parameters, including
    ;;; a slow max_speed setting to be used for homing the
    ;;; servo.
    ;;;
    ;;; The search and zero instruction causes the servo to
    ;;; turn until a signal is received from a limit sensor,
    ;;; attached to the servo control module's HOME input.
    ;;; Refer to the module's Installation Guide for specific
    ;;; information.  Finally, the monitor instruction senses
    ;;; when the servo has reached its home position and come
    ;;; to a stop.
    ____________________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________________
    store 20000 to Servo_Speed
    profile Servo_1 servo at position maxspeed=1000 accel=250000 P=10 I=253
      D=237
    search and zero Servo_1
    monitor Servo_1:stopped goto Next

[2] START_TASKS
    ;;; This step starts two tasks running; one task will
    ;;; sequence the servo motions, while the other will
    ;;; monitor servo error.
    ;;;
    ;;; NOTE: Neither of the tasks contain Done instructions;
    ;;; rather, they each loop back on themselves in normal
    ;;; operation.  Therefore, the destination of this
    ;;; instruction, "goto INITIALIZE", will never be taken.
    ;;; It is there simply to satisfy the form of the Do
    ;;; instruction.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    do (MAIN_PROGRAM FAULT_MONITOR) goto INITIALIZE

[10] MAIN_PROGRAM
    ;;; This begins a simple 2-step task, sequencing the
    ;;; servo back-and-forth between two positions.
    ;;;
    ;;; Before commencing each new motion, we re-profile
    ;;; the servo to allow any changes which may have been
    ;;; made to the register Servo_Speed to take effect.
    ;;; This register may be modified using the CTCMON
```

```
    ;;; controller monitoring utility, or by using an
    ;;; operator interface terminal or HMI software
    ;;; communicating with the controller.  Any such change
    ;;; will not take effect, however, until the next cycle
    ;;; when the profile instruction is executed.
    ;;;
    ;;; The turn instruction commences a 100,000 step
    ;;; motion.  The monitor instruction will sense when
    ;;; the motion is complete and send the controller to
    ;;; the next step of the program.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    profile Servo_1 maxspeed=Servo_Speed
    turn Servo_1 cw 100000 steps
    monitor Servo_1:stopped goto Next

[11] SERVO_RETURN
    ;;; This step works similarly, except that the motion
    ;;; is initiated in the counter-clockwise direction.
    ;;;
    ;;; Once again, we allow any changes in the register
    ;;; Servo_Speed to take effect.
    ;;;
    ;;; When the motion is complete, this task will jump
    ;;; back to step MAIN_PROGRAM to begin another cycle.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    profile Servo_1 maxspeed=Servo_Speed
    turn Servo_1 ccw 100000 steps
    monitor Servo_1:stopped goto MAIN_PROGRAM

[50] FAULT_MONITOR
    ;;; This task normally remains looping at this one
    ;;; step.  The Store instruction updates a numeric
    ;;; display, named “Servo_Error_Disp”, with the
    ;;; instantaneous error of Servo_1.  This error value
    ;;; is also tested and, if it is excessive, the task
    ;;; will jump to the step called SHUT_DOWN.
    ;;;
    ;;; The Store instruction would normally only execute
    ;;; once upon entering this step.  To create a continuous
    ;;; update, the “goto FAULT_MONITOR” instruction causes
    ;;; the step to re-execute, thus updating the display
    ;;; again.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store Servo_1:error to Servo_Error_Disp
    if Servo_1:error >=100 goto SHUT_DOWN
    goto FAULT_MONITOR

[51] SHUT_DOWN
    ;;; In this step, a servo error of 100 or greater has
    ;;; been sensed, so we shut down the servo and halt
    ;;; the controller.
    ;;;
    ;;; First, however, we must cancel the other task to
    ;;; avoid having it restart a new servo motion.
    ;;;
```

```
    ;;; When the controller is restarted, the program will
    ;;; jump back to the step INITIALIZE.  Note that,
    ;;; because we have cancelled multitasking at this point,
    ;;; this will not cause recursion.
    ————————————————————————————————————
    <NO CHANGE IN DIGITAL OUTPUTS>
    ————————————————————————————————————
    cancel other tasks
    stop (hard) Servo_1
    stop goto INITIALIZE
```

Velocity Mode Example

```
[1] INITIALIZE
    ;;; This program illustrates the control of a servo in
    ;;; velocity mode.
    ;;;
    ;;; The primary goal of this mode of operation is to
    ;;; control the servo speed through a long or continuous
    ;;; motion.  Quickstep supports this mode with a
    ;;; continuous “turn” command, as well as the capability
    ;;; to modify servo parameters on-the-fly.
    ;;;
    ;;; In this initial step, we’ll set starting parameters
    ;;; for the servo using a Profile command, then search
    ;;; for a home reference position.
    ————————————————————————————————————
    <TURN OFF ALL DIGITAL OUTPUTS>
    ————————————————————————————————————
    profile Servo_1 servo at position maxspeed=20000 accel=250000 P=10 I=253
      D=237
    search and zero Servo_1
    monitor Servo_1:stopped goto Next

[2] START_MOTION
    ;;; This step starts a continuous servo motion using
    ;;; a velocity-mode servo instruction:
    ;;;
    ;;;   turn Servo_1 cw
    ;;;
    ;;; We will remain in this step until the servo reaches
    ;;; a position 25000 steps from its zero or home position.
    ————————————————————————————————————
    <NO CHANGE IN DIGITAL OUTPUTS>
    ————————————————————————————————————
    turn Servo_1 cw
    if Servo_1:position >=25000 goto Next

[3] SLOW_DOWN
    ;;; Having reached position 25000, we will now
    ;;; decelerate the servo to 10000 steps/sec by
    ;;; re-profiling.
    ;;;
    ;;; We then remain in this step until the servo
    ;;; reaches a position 75000 steps from its home
    ;;; position.
    ————————————————————————————————————
    <NO CHANGE IN DIGITAL OUTPUTS>
    ————————————————————————————————————
    profile Servo_1 maxspeed=10000
    if Servo_1:position >=75000 goto Next
```

```
[4] SPEED_UP
    ;;; Now, we'll accelerate the servo to a rate of
    ;;; 100000 steps/sec.
    ;;;
    ;;; We then remain in this step until the servo
    ;;; reaches a position 200000 steps from its zero or home
    ;;; position. Once this position has been attained,
    ;;; we'll go on to the next step.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    profile Servo_1 maxspeed=100000
    if Servo_1:position >=200000 goto Next

[5] OUTPUT_ON
    ;;; In this step we'll cause some action to take place
    ;;; on our machine by turning on Output 1.  We are not
    ;;; changing the servo speed at this point.
    ;;;
    ;;; This step illustrates how you can program various
    ;;; events at specific servo positions.  Analog output
    ;;; voltages, secondary motor instructions, and many
    ;;; other types of events could be triggered at various
    ;;; points in the servo's motion.
    ;;;
    ;;; Once the servo reaches a position 250,000 steps from
    ;;; home, our program continues at the following step.
    ____________________________________________
    Output_1_On
    ____________________________________________
    if Servo_1:position >=250000 goto Next

[6] ALL_DONE
    ;;; Having reached servo position 250000, we begin
    ;;; decelerating to a stop through the use of a soft
    ;;; stop instruction.
    ;;;
    ;;; The second stop instruction will stop the
    ;;; controller's execution of the program, until the
    ;;; operator restarts it.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    stop (soft) Servo_1
    stop goto INITIALIZE
```

Using the Data Table

```
[1] INITIALIZE
    ;;; This program uses a DATA TABLE to store information
    ;;; for an iterative (repeating) program. The program
    ;;; controls a drilling machine, where the workpiece is
    ;;; held on an X-Y table allowing it to be automatically
    ;;; positioned.
    ;;;
    ;;; We wish to drill a number of holes in the workpiece,
    ;;; and have the speed of each move independently
    ;;; programmable.  We also want a programmable delay
    ;;; after each drilling operation.
    ;;;
    ;;; To accomplish these goals, we'll store the data for
    ;;; these holes, including speeds and delays, in a data
    ;;; table.  This will result in a much shorter and simpler
    ;;; program, as well as keeping all of this data in one
    ;;; place.
    ;;;
    ;;; Each data table row will contain the data for one
    ;;; drilling operation, as follows:
    ;;;         column 1 = X-axis position (motor 1)
    ;;;         column 2 = Y-axis position (motor 2)
    ;;;         column 3 = Time delay (1/100ths second)
    ;;;         column 4 = Motor speed (steps/second)
    ;;;
    ;;; In this step, we point to the first row of the
    ;;; data table by storing 1 to the register named
    ;;; Data_Tbl_Ptr (register 126). We also set up the
    ;;; initial motion profile for each motor, and have
    ;;; both motors search for their home positions.
    ;;;
    ;;; Note that if the motors do not find home within
    ;;; 20 seconds, the delay instruction shuts the machine
    ;;; down automatically.
    ____________________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________________
    store 1 to Data_Tbl_Ptr
    profile X_Axis (half) basespeed=100 maxspeed=2500 accel=400 decel=400
    profile Y_Axis (half) basespeed=100 maxspeed=2500 accel=400 decel=400
    search ccw and zero X_Axis
    search ccw and zero Y_Axis
    monitor (and X_Axis:stopped Y_Axis:stopped) goto Next
    delay 20 sec goto ALL_DONE

[2] NEW_POSITION
    ;;; This is the beginning of the drilling loop. In this
    ;;; step, we establish a profile for both motors, with
    ;;; their speeds based on column #4 of the Data Table.
    ;;;
    ;;; Then, we tell the motors to turn to the positions
    ;;; specified in columns 1 and 2 of the current row of
    ;;; the Data Table.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    profile X_Axis (half) basespeed=100 maxspeed=Col_4 accel=400 decel=400
    profile Y_Axis (half) basespeed=100 maxspeed=Col_4 accel=400 decel=400
    turn X_Axis to Col_1
    turn Y_Axis to Col_2
    monitor (and X_Axis:stopped Y_Axis:stopped) goto Next
```

```
[3] DRILL_HOLE
    ;;; Here, we are at a new position to be drilled, so
    ;;; the drill is sent down by actuating an air cylinder,
    ;;; controlled by one of the controller's digital outputs.
    ____________________________________________
    Extend_Drill
    ____________________________________________
    delay 3 sec goto Next

[4] RETRACT
    ;;; In this step, we retract the drill.
    ;;;
    ;;; We also add one to the register Data_Tbl_Ptr
    ;;; (register 126), so that we are pointing to the next
    ;;; row of the data table.  This points to the next set
    ;;; of coordinates to be drilled.
    ;;;
    ;;; The If instruction below will test to determine if the
    ;;; last row has been  encountered; if so, the program
    ;;; jumps out of its loop and stops.
    ;;;
    ;;; If the program has not encountered the last row of the
    ;;; Data Table, it returns to step NEW_POSITION to drill
    ;;; another hole.  Note that the delay instruction in this
    ;;; step derives its value from column 3 of the data table.
    ____________________________________________
    Retract_Drill
    ____________________________________________
    delay Col_3 sec/100 goto NEW_POSITION
    store Data_Tbl_Ptr + 1 to Data_Tbl_Ptr
    if Data_Tbl_Ptr >=26 goto ALL_DONE

[10] ALL_DONE
    ;;; In this step, we have either completed a workpiece, or
    ;;; have shut down because the motors never found
    ;;; their home position. In case the motors never found
    ;;; home, we will execute a stop motor instruction for
    ;;; each motor.
    ____________________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________________
    stop X_Axis
    stop Y_Axis
    stop goto INITIALIZE
```

Data in Data Table

Row	Columns 1	2	3	4
1	100	2500	20	2000
2	100	2600	10	2000
3	200	2100	20	2000
4	200	2500	50	2000
5	200	2600	20	2000
6	200	3000	25	2000
7	375	1500	25	2000

Row	Columns 1	2	3	4
8	375	2500	25	2000
9	375	3500	25	2000
10	500	1500	20	2000
11	500	2500	20	2000
12	500	3500	20	2000
13	500	4500	10	2000
14	750	500	20	1000
15	750	750	20	1000
16	750	1000	20	1000
17	750	1250	50	1000
18	750	1500	20	1000
19	750	1750	20	1000
20	750	2000	20	1000
21	750	2250	20	1000
22	750	2500	20	1000
23	750	2750	20	1000
24	750	3000	20	1000
25	750	3250	20	1000

Using the Phantom Register

Using the Phantom Register to Create a Circular Buffer

```
[1] INITIALIZE
    ;;; This program demonstrates the use of the PHANTOM
    ;;; REGISTER to create a circular buffer.  This circular
    ;;; buffer is used to store a continuous stream of data
    ;;; relating to a continuous flow of parts through a
    ;;; machine.
    ;;;
    ;;; The circular buffer consists of a series of numeric
    ;;; registers; this series must be long enough to
    ;;; accommodate the maximum possible number of parts
    ;;; within the machine at one time. Two additional
    ;;; registers are used as pointers. These pointers keep
    ;;; track of which register to use next (i.e.; the next
    ;;; empty register) and which register contains the oldest
    ;;; active data (i.e.; where to find the data applicable
    ;;; to the oldest part on the machine).
    ;;;
    ;;; Each time the data for a new part is stored in the
    ;;; next empty register the Next_Empty_Ptr is incremented
    ;;; to point to the following register. Because we do not
    ;;; have an infinite number of registers available, we
    ;;; must always test the pointer after incrementing it to
    ;;; make sure that it is not pointing to a register
    ;;; outside of the circular buffer. If it is, we set the
    ;;; pointer to point back at the beginning of the circular
    ;;; buffer, where we will then start overwriting old data.
    ;;; As long as we have made the circular buffer large
    ;;; enough, this will not be a problem, because this data
    ;;; will no longer be in use.
    ;;;
    ;;; The Oldest_Ptr is maintained so as to point to the
    ;;; register containing the data for the oldest part on
    ;;; the machine. Presumably, this is the data which will
    ;;; next be used by the process for which we have
    ;;; originally stored the data. When our process has made
    ;;; use of the data, it is no longer needed, and the
    ;;; Oldest_Ptr may be moved to point to the next
    ;;; register location. Once again, if the pointer has
    ;;; moved beyond the end of the circular buffer, we must
    ;;; set it to point back to the beginning of the buffer.
    ;;;
    ;;; In this manner, the two pointers chase each other
    ;;; around the circular buffer. A continuous flow of
    ;;; information is accommodated, and a variable number of
    ;;; data elements may be contained between the two
    ;;; pointers.
    ;;; In this first step, we will initialize both of the
    ;;; pointers to point to the beginning of the circular
    ;;; buffer. At this point, the buffer contains no data.
    ;;;
    ;;;         register 10 = Bake_Time
    ;;;         register 127 = Phantom_Ptr
    ;;;         register 128 = Phantom_Reg
    ;;;         registers 501 through 550 are used for the
    ;;;           circular buffer.
    ;;;         register 600 = Oldest_Ptr
    ;;;         register 601 = Next_Empty_Ptr
```

```
    ____________________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________________
    store 501 to Oldest_Ptr
    store 501 to Next_Empty_Ptr
    goto Next

[2] START_TASKS
    ;;; In this step, we will start two separate tasks:
    ;;;
    ;;; The first task, called MEASURE, takes a
    ;;; measurement of new parts coming into the machine.
    ;;; Each measurement will be stored in the next empty
    ;;; location of the circular buffer.
    ;;;
    ;;; The second task, called PROCESS, bakes the part
    ;;; for a time duration proportionate to the
    ;;; measurement originally taken of that part. The
    ;;; measurement data will be drawn from the circular
    ;;; buffer location indicated by Oldest_Ptr.
    ;;;
    ;;; There may be a variable number of parts in transit
    ;;; between the Measure station and the Process
    ;;; station.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    do (MEASURE PROCESS) goto INITIALIZE

[10] MEASURE
    ;;; This is the beginning of the part measuring task.
    ;;;
    ;;; In this step, we'll wait for a part to appear (as
    ;;; sensed by a limit switch), then proceed to the
    ;;; next step where the measurement will be taken.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    monitor Part_In_Place goto Next

[11] STORE_DATA
    ;;; In this step, the measurement of the new part is
    ;;; stored in the next empty location of the
    ;;; circular buffer. We prepare for this by storing the
    ;;; number of the next empty register (Next_Empty_Ptr)
    ;;; to Phantom_Ptr. The measurement (Transducer) data is
    ;;; then stored into the phantom register, and lands in
    ;;; the next empty location of the circular buffer.
    ;;;
    ;;; We then must increment Next_Empty_Ptr to point to the
    ;;; next register, and test it to see if it has moved
    ;;; beyond the limits we have established for the
    ;;; circular buffer. If so, we'll reset the pointer
    ;;; to point to register 501,  which is the first
    ;;; register of the circular buffer.
    ;;;
    ;;; NOTE: In many applications, it would be advisable to
    ;;; test that the Next_Empty_Ptr never overruns the
    ;;; Oldest_Ptr. This would mean that the buffer has
    ;;; overflowed and data is being lost.
```

```
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store Next_Empty_Ptr to Phantom_Ptr
    store Transducer to Phantom_Reg
    store Next_Empty_Ptr + 1 to Next_Empty_Ptr
    if Next_Empty_Ptr <=550 goto Next
    store 501 to Next_Empty_Ptr
    goto Next

[12] PART_GONE
    ;;; In this step, we'll simply wait for the part to move
    ;;; out of the measurement station, at which time we can
    ;;; return to the beginning of this task to wait for
    ;;; another part.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    monitor No_Part_In_Place goto MEASURE

[20] PROCESS
    ;;; This is the beginning of the task which controls
    ;;; the baking of the parts in the machine. In this
    ;;; step, we'll wait for a part to enter the baking
    ;;; area, as sensed by a limit switch.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    monitor Part_In_Bake goto Next

[21] BAKE_DATA
    ;;; Here, we'll extract the data originally stored for
    ;;; the part now in the baking area.
    ;;;
    ;;; We have carefully maintained Oldest_Ptr, so we know
    ;;; that the part which has now entered the baking area
    ;;; is the part whose data is in the register being
    ;;; pointed to. To extract this data, we'll store
    ;;; Oldest_Ptr into the phantom register pointer, then
    ;;; read the data itself from the phantom register
    ;;; (register 128). This data is stored temporarily in
    ;;; register Bake_Time, to be used in the next step.
    ;;;
    ;;; Having used this data, we must then adjust the
    ;;; Oldest_Ptr to point to the next location. As with
    ;;; the Next_Empty_Ptr, we must test Oldest_Ptr after we
    ;;; have incremented it to insure that it has not gone
    ;;; beyond the limits of our circular buffer. If it has,
    ;;; we'll reset Oldest_Ptr to point to the beginning of
    ;;; the buffer (i.e.; register 501).
    ;;;
    ;;; NOTE: In some applications, it may be advisable to
    ;;; test the new value of Oldest_Ptr to insure that it
    ;;; never passes Next_Empty_Ptr. This would mean that
    ;;; one or more parts have been "lost" and the measurement
    ;;; data is now out of sync.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store Oldest_Ptr to Phantom_Ptr
    store Phantom_Reg to Bake_Time
```

```
    store Oldest_Ptr + 1 to Oldest_Ptr
    if Oldest_Ptr <=550 goto Next
    store 501 to Oldest_Ptr
    goto Next

[22] BAKE
    ;;; Here, we'll turn on the heating elements, and wait for
    ;;; an amount of time proportionate to the data which we
    ;;; extracted in the previous step and stored in register
    ;;; Bake_Time. NOTE: the data is first ranged by division
    ;;; and addition; this just illustrates the implementation
    ;;; of a possible transfer function between measurement
    ;;; and bake duration.
    ________________________________________
    Bake_Part_On
    ________________________________________
    store Bake_Time / 55 to Bake_Time
    store Bake_Time + 12 to Bake_Time
    delay Bake_Time min goto Next

[23] AFTER_BAKE
    ;;; In this step, we'll just wait for the part to
    ;;; clear the limit switch with which we originally
    ;;; sensed it.
    ________________________________________
    Bake_Part_Off
    ________________________________________
    monitor No_Part_In_Bake goto PROCESS
```

Using the Phantom Register to Access multiple I/O Points

```
[1] INITIALIZE_REGISTERS
    ;;; This program illustrates the use of the PHANTOM
    ;;; REGISTER to perform the same operation on ten
    ;;; different groups of I/O points, using a single copy
    ;;; of a program. These I/O points could relate to ten
    ;;; separate but identical workstations.
    ;;;
    ;;; The program controls the workstations via the phantom
    ;;; register, changing pointers to point to each
    ;;; successive workstation, and then repeating the cycle.
    ;;; We have created an arbitrary control program,
    ;;; consisting of:
    ;;;
    ;;;   * Turning on two outputs
    ;;;   * Waiting for a limit switch before proceeding
    ;;;   * Turning off both outputs
    ;;;   * Waiting for a second limit switch.
    ;;;
    ;;; Since this task accesses more than one I/O point
    ;;; during each cycle and there is only one set of pointer
    ;;; and phantom registers, we need to maintain a series of
    ;;; pointers in general-purpose numeric registers. Just
    ;;; prior to the required access to a given I/O point the
    ;;; program transfers the pointer value from the general-
    ;;; purpose register into the pointer register.
    ;;;
    ;;; This step stores initial pointer values in registers
    ;;; 10, 11, 12, and 13. The value being stored in register
    ;;; Output_A_Ptr, 1010, will point to output number 10
```

```
    ;;; when this value is stored to the phantom register
    ;;; pointer.  Similarly, the value 1020 (stored in
    ;;; Output_B_Ptr) will point to output number 20, the
    ;;; value 2010 (stored in Input_A_Ptr) will point to input
    ;;; number 10, and the value 2020 (stored in Input_B_Ptr)
    ;;; will point to input number 20.
    ;;;
    ;;; Pointer = Register 127
    ;;; Phantom = Register 128
    ;;; Output_A_Ptr = Register 10
    ;;; Output_B_Ptr = Register 11
    ;;; Input_A_Ptr  = Register 12
    ;;; Input_B_Ptr  = Register 13
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store 1010 to Output_A_Ptr
    store 1020 to Output_B_Ptr
    store 2010 to Input_A_Ptr
    store 2020 to Input_B_Ptr
    goto OUTPUTS_ON

[50] OUTPUTS_ON
    ;;; This step will turn on two outputs, then wait for
    ;;; a limit switch input, all using indirect references
    ;;; through the phantom register.
    ;;;
    ;;; The first instruction transfers Output_A_Ptr to the
    ;;; phantom register pointer. If this is the first time
    ;;; through the loop, this means the phantom register is
    ;;; now pointing to output number 10, since Output_A_Ptr
    ;;; was previously initialized to the value 1010. The
    ;;; second instruction then turns this output ON by
    ;;; storing the value 1 to the phantom register.
    ;;;
    ;;; The next two instructions work similarly, except that
    ;;; this time we will turn on output number 20, because
    ;;; Output_B_Ptr was initialized to the value 1020 above.
    ;;;
    ;;; The last two instructions use the phantom register to
    ;;; monitor a limit switch input. As before, we transfer
    ;;; the pointer value (Input_A_Ptr) to the phantom
    ;;; register pointer.  Then, however, we'll use an If
    ;;; instruction to test the phantom register.  This
    ;;; actually will indirectly test the input being pointed
    ;;; to.  The phantom register will read as 1 if the input
    ;;; has a closure on it, otherwise it will read as zero.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store Output_A_Ptr to Pointer
    store 1 to Phantom
    store Output_B_Ptr to Pointer
    store 1 to Phantom
    store Input_A_Ptr to Pointer
    if Phantom=1 goto Next

[51] MONITOR_INPUT
    ;;; In this step, a series of instructions similar to the
    ;;; previous step turns off both outputs, using the same
    ;;; pointer values used to turn them on in the previous
```

```
    ;;; step. The last two instructions monitor a different
    ;;; limit switch, as pointed to by Input_B_Ptr. When it is
    ;;; turned on, the program proceeds to the next step.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    store Output_A_Ptr to Pointer
    store 0 to Phantom
    store Output_B_Ptr to Pointer
    store 0 to Phantom
    store Input_B_Ptr to Pointer
    if Phantom=1 goto Next

[52] NEXT_OUTPUT
    ;;; Here we will adjust all the pointer values so they
    ;;; will point to the I/O points of the next workstation.
    ;;;
    ;;; First, however, we must test to see if we have
    ;;; completed the entire series of ten workstations. If
    ;;; so, we'll return back to the initialization step to
    ;;; begin again.
    ;;;
    ;;; Otherwise, after pointing to the new I/O points, we'll
    ;;; return to the beginning of our program loop, starting
    ;;; with step OUTPUTS_ON.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    if Output_A_Ptr >=1019 goto INITIALIZE_REGISTERS
    store Output_A_Ptr + 1 to Output_A_Ptr
    store Output_B_Ptr + 1 to Output_B_Ptr
    store Input_A_Ptr + 1 to Input_A_Ptr
    store Input_B_Ptr + 1 to Input_B_Ptr
    goto OUTPUTS_ON
```

Using a Multi-station Indexing Table

```
[1] INITIALIZE_SYSTEM
    ;;; This program shows a multi-station rotary indexing
    ;;; table assembly machine. There are eight stations on
    ;;; a rotary index. Each station performs a specific
    ;;; function on a workpiece. The machine also tests each
    ;;; workpiece during certain critical points of the
    ;;; process. If a test fails, all further assembly on that
    ;;; workpiece ceases, resulting in it being off-loaded
    ;;; into the “bad” bin.
    ;;;
    ;;; The program tracks good parts/bad workpieces using a
    ;;; series of flags in a shift register. The flag’s
    ;;; status, either set or clear, represents a good or bad
    ;;; workpiece. As the table indexes workpieces from one
    ;;; station to the next, the Shift Flag instruction
    ;;; transfers the flag status to the next station on the
    ;;; table. The task that controls a station monitors its
    ;;; flag to determine if the part is good or not.
    ____________________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________________
    goto Next

[2] LAUNCH_TASKS
    ;;; This step starts start two tasks. MAIN_PROGRAM will
    ;;; start eight tasks and run the indexing table. The
    ;;; other task continuously monitors for fault conditions.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    do (MAIN_PROGRAM FAULT_MONITOR) goto LAUNCH_TASKS

[3] MAIN_PROGRAM
    ;;; Here, the position of the index table, controlled by
    ;;; the servo called Table, is synchronized by searching
    ;;; for a home reference position.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    profile Table servo at position maxspeed=2000 accel=500000 P=50 I=30
      D=50
    search and zero Table
    monitor Table:stopped goto Next

[4] AWAIT_START
    ;;; After home position is established, the program waits
    ;;; for the operator to press the Start_Switch before
    ;;; proceeding.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    monitor Start_Switch goto Next

[5] INDEX_TABLE
    ;;; The distance the table moves is defined by a register
    ;;; named Index. We’ll use this value in a relative
    ;;; clockwise Turn instruction to index the table.
    ;;;
    ;;; This step also shifts the flags so that, as the
    ;;; workpieces are indexed, any previously stored good-
```

```
    ;;; part/bad-part information is also indexed to follow
    ;;; the associated workpiece.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    profile Table servo at position maxspeed=System_Speed accel=System_Accel
    turn Table cw Index steps
    shift S1_Part >> S8_Part
    monitor Table:stopped goto Next

[6] DO_STATIONS
    ;;; This step starts eight tasks. Each task controls one
    ;;; station on the table and performs an operation on the
    ;;; part.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    do (LOAD_PART FILL_PART FORM_PART TEST_PART WELD_COVER FINAL_TEST
      OFFLOAD_BAD OFFLOAD_GOOD) goto INDEX_TABLE

[10] LOAD_PART
    ;;; This step pushes a core part onto the table and
    ;;; monitors to see if the part is in position.
    ____________________________________________
    Push_Part_On
    ____________________________________________
    monitor Part_Positioned goto Next

[11] LOAD_PART_RETRACT
    ;;; Once the part is correctly in place, we retract the
    ;;; push arm. When the push arm hits a limit switch, we
    ;;; go to the next step.
    ;;; This step also sets the first flag in the shift
    ;;; register, indicating that a part is present.
    ____________________________________________
    Retract_Pusher
    ____________________________________________
    set S1_Part
    monitor Pusher_Retracted goto Next

[12] LOAD_PART_SUCCESS
    ;;; This is the end of the task controlling the first
    ;;; station.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    done

[20] FILL_PART
    ;;; This step checks if there is part on the table. If the
    ;;; flag is set, it continues to the next step and
    ;;; performs the duties of this station. If it is clear,
    ;;; the task is complete for this cycle.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    monitor S2_Part:set goto Next
    goto FILL_DONE
```

```
[21] FILL_PART_DROP
    ;;; This step opens the fill valve and waits for the scale
    ;;; to read full. If the part hasn't filled within one
    ;;; second, it alerts the operator that the reservoir is
    ;;; empty.
    ____________________________________
    Fill_Open
    Alert_Operator_Off
    ____________________________________
    if Weigh_Scale >=Full_Weight goto Next
    delay 1 sec goto RESERVOIR_EMPTY

[22] FILL_PART_CLOSE
    ;;; This step closes the fill valve and waits for the gate
    ;;; to close.
    ____________________________________
    Fill_Closed
    ____________________________________
    monitor Gate_Closed goto FILL_DONE

[23] RESERVOIR_EMPTY
    ;;; This step alerts the operator that the hopper is empty.
    ;;; After 500 ms it returns to FILL_PART_DROP and tests
    ;;; the scale again.
    ____________________________________
    Alert_Operator_On
    Fill_Closed
    ____________________________________
    delay 500 ms goto FILL_PART_DROP

[24] FILL_DONE
    ;;; This is the end of the FILL_PART task.
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    done

[30] FORM_PART
    ;;; This step checks if there is part on the table. If the
    ;;; flag is set, it continues to the next step and
    ;;; performs the duties of this station. If it is clear,
    ;;; the task is complete for this cycle.
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    monitor S3_Part:set goto Next
    goto FORM_DONE

[31] FORM_PART_COMPRESS
    ;;; This step turns on an output and extends a cylinder to
    ;;; compress the part. When the cylinder hits a limit
    ;;; switch, we go to the next step.
    ____________________________________
    Compress
    ____________________________________
    monitor Part_Compressed goto Next
```

```
[32] FORM_PART_DECOMPRESS
    ;;; This step turns off the output and retracts the
    ;;; compression cylinder. When the cylinder hits a limit
    ;;; switch, we go to the next step.
    ____________________________________________
    Compress_Up
    ____________________________________________
    monitor Compress_Ret goto Next

[33] FORM_DONE
    ;;; This is the end of the FORM_PART task.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    done

[40] TEST_PART
    ;;; This step checks if there is part on the table. If the
    ;;; flag is set, it continues to the next step and
    ;;; performs the duties of this station. If it is clear,
    ;;; the task is complete for this cycle.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    monitor S4_Part:set goto Next
    goto TEST_DONE

[41] TEST_PART_MEASURE
    ;;; This step turns on an output that applies air pressure
    ;;; to the formed part. If a sensor detects a leak, the
    ;;; part fails.
    ____________________________________________
    Pressure_On
    ____________________________________________
    monitor Pressure_Leak goto FAILED_PRESSURE_TEST
    delay 300 ms goto Next

[42] TEST_PART_PASSED
    ;;; This step turns of the output for the pressure tester.
    ____________________________________________
    Pressure_Off
    ____________________________________________
    goto TEST_DONE

[43] FAILED_PRESSURE_TEST
    ;;; This step is only executed when part fails the
    ;;; pressure test. It also increments the Pressure_Fail
    ;;; counter so we may keep track of problem areas on the
    ;;; machine. We also clear the station flag so no further
    ;;; work will be performed on this part as it indexes
    ;;; through the rest of the machine.
    ____________________________________________
    Pressure_Off
    ____________________________________________
    store Pressure_Fail + 1 to Pressure_Fail
    clear S4_Part
    goto Next
```

```
[44] TEST_DONE
    ;;; This is the end of the TEST_PART task.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    done

[50] WELD_COVER
    ;;; This step checks if there is part on the table. If the
    ;;; flag is set, it continues to the next step and
    ;;; performs the duties of this station. If it is clear,
    ;;; the task is complete for this cycle.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    monitor S5_Part:set goto Next
    goto WELD_DONE

[51] WELD_COVER_PLACE
    ;;; This step activates an output which places a cover
    ;;; onto the part.  When the cover is in place, it
    ;;; proceeds to the next step.
    ____________________________________________
    Place_Cover_On
    ____________________________________________
    monitor Cover_Positioned goto Next

[52] WELD_COVER_RETRACT
    ;;; This step activates two outputs. The first retracts
    ;;; the mechanism that placed the cover on. The second
    ;;; activates the welder arm. The Boolean monitor
    ;;; instruction waits until the cover pusher has retracted
    ;;; and the weld arm is in position.
    ____________________________________________
    Place_Cover_Off
    Weld_Arm_On
    ____________________________________________
    monitor (and Place_Cover_Ret Weld_Arm_Positioned_On) goto Next

[53] WELD_COVER_WELD_PART
    ;;; This step turns on the output which controls the
    ;;; welder. After a 250 ms delay it proceeds to the next
    ;;; step.
    ____________________________________________
    Welder_On
    ____________________________________________
    delay 250 ms goto Next

[54] WELD_COVER_PULL_BACK
    ;;; This step turns off two outputs. One retracts the
    ;;; welding arm and the other turns the welder off.  The
    ;;; monitor instruction waits until the weld arm has
    ;;; retracted before moving to the next step.
    ____________________________________________
    Weld_Arm_Off
    Welder_Off
    ____________________________________________
    monitor Weld_Arm_Positioned_Off goto Next
```

```
[55] WELD_DONE
    ;;; This is the end of the WELD_COVER task.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    done

[60] FINAL_TEST
    ;;; This step checks if there is part on the table. If the
    ;;; flag is set, it continues to the next step and
    ;;; performs the duties of this station. If it is clear,
    ;;; the task is complete for this cycle.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    monitor S6_Part:set goto Next
    goto FINAL_TEST_DONE

[61] FINAL_TEST_ENGAGE
    ;;; This step activates an output that clamps the part for
    ;;; an electrical test. A sensor indicates that the part
    ;;; is secure and proceeds to the next step.
    ____________________________________________
    Clamp_Part
    ____________________________________________
    monitor Part_Clamped_On goto Next

[62] FINAL_TEST_POWER_ON
    ;;; This step activates an electrical tester and checks to
    ;;; see if the part passes or fails. It also monitors the
    ;;; results of the test.
    ____________________________________________
    Power_On
    ____________________________________________
    monitor Good_Part goto Next
    monitor Bad_Part goto FINAL_TEST_FAILED

[63] FINAL_TEST_PASSED
    ;;; If the part has passed the electrical test, the task
    ;;; proceeds to this step. It turns off the output that
    ;;; controls the electrical power and the clamp. It also
    ;;; monitors a sensor to determine when the part is no
    ;;; longer clamped.
    ____________________________________________
    Power_Off
    Unclamp_Part
    ____________________________________________
    monitor Part_Clamped_Off goto FINAL_TEST_DONE

[64] FINAL_TEST_FAILED
    ;;; This step is only executed when a part fails the
    ;;; electrical test. It turns off both outputs, increments
    ;;; the counter for parts failing the electrical test, and
    ;;; clears the flag for this station, indicating that a
    ;;; bad part is present. When the sensor indicates the
    ;;; part is not longer clamped, the task concludes.
    ____________________________________________
    Power_Off
    Unclamp_Part
    ____________________________________________
    store Electrical_Fail + 1 to Electrical_Fail
```

```
    clear S6_Part
    monitor Part_Clamped_Off goto FINAL_TEST_DONE

[65] FINAL_TEST_DONE
    ;;; This is the end of the FINAL_TEST task.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    done

[70] OFFLOAD_BAD
    ;;; This step tests for a bad part on the table. If the
    ;;; flag corresponding to the part in this station is
    ;;; clear (indicating that it had failed a test), the part
    ;;; is pushed into the bad part bin.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    monitor S7_Part:clear goto Next
    goto OFFLOAD_BAD_DONE

[71] OFFLOAD_BAD_PUSH
    ;;; This step activates an output which pushes the bad
    ;;; part off the table and monitors a sensor to make sure
    ;;; it is gone.
    ____________________________________________
    Bad_Part_On
    ____________________________________________
    monitor Bad_Part_Gone goto Next

[72] OFFLOAD_BAD_RETRACT
    ;;; This step turns off the output turned on in the
    ;;; previous step and monitors a limit switch to see that
    ;;; the push arm has retracted.
    ____________________________________________
    Bad_Part_Off
    ____________________________________________
    monitor Bad_Part_Pusher_Home goto Next

[73] OFFLOAD_BAD_DONE
    ;;; This is the end of the OFFLOAD_BAD task.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    done

[80] OFFLOAD_GOOD
    ;;; This task pushes good parts into the good part bin.
    ;;; It checks to see if the flag is set, indicating the
    ;;; presence of a good part.
    ____________________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________________
    monitor S8_Part:set goto Next
    goto OFFLOAD_GOOD_DONE

[81] OFFLOAD_GOOD_PUSH
    ;;; This step activates an output which pushes the good
    ;;; part off the table and monitors a sensor to make sure
    ;;; it is gone.
```

```
    ____________________________________
    Good_Part_On
    ____________________________________
    monitor Good_Part_Gone goto Next

[82] OFFLOAD_GOOD_RETRACT
    ;;; This step turns off the output turned on in the
    ;;; previous step and monitors a limit switch to see that
    ;;; the push arm has retracted.
    ____________________________________
    Good_Part_Off
    ____________________________________
    monitor Good_Part_Pusher_Home goto Next

[83] OFFLOAD_GOOD_DONE
    ;;; This is the end of the OFFLOAD_GOOD task.
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    done

[100] FAULT_MONITOR
    ;;; If the servo motor error exceeds 2000 counts from its
    ;;; intended position or if the light-curtain is broken,
    ;;; the index table is shut down.
    ____________________________________
    <NO CHANGE IN DIGITAL OUTPUTS>
    ____________________________________
    if Table:error >=2000 goto Next
    if Table:error <=-2000 goto Next
    monitor Light_Curtain_Intrusion goto Next

[101] DISABLE_MACHINE
    ____________________________________
    <TURN OFF ALL DIGITAL OUTPUTS>
    ____________________________________
    cancel other tasks
    monitor Start_Switch goto INITIALIZE_SYSTEM
```

Default Symbolic Names

Contents

The default symbolic names are located in a file called **DEFAULTS.SYM**. This File is automatically placed in your **QSWIN** directory when you installed Quickstep. **DEFAULTS.SYM** contains default symbolic names for controller resources, analog and digital inputs/outputs, motion control devices, and some special registers.

Default Symbolic Names for Controller Resources

Default Names for Registers

Register Number	Symbolic Name
Register 10	reg_10
Register 11	reg_11
Register 12	reg_12
Register 13	reg_13
Register 14	reg_14
Register 15	reg_15
Register 16	reg_16
Register 17	reg_17
Register 18	reg_18
Register 19	reg_19
Register 20	reg_20
Register 501	reg_501
Register 502	reg_502
Register 503	reg_503
Register 504	reg_504
Register 505	reg_505
Register 506	reg_506
Register 507	reg_507
Register 508	reg_508
Register 509	reg_509
Register 510	reg_510
Register 511	reg_511
Register 512	reg_512
Register 513	reg_513
Register 514	reg_514
Register 515	reg_515
Register 516	reg_516
Register 517	reg_517
Register 518	reg_518
Register 519	reg_519
Register 520	reg_520
Register 990	reg_990

Register Number	Symbolic Name
Register 991	reg_991
Register 992	reg_992
Register 993	reg_993
Register 994	reg_994
Register 995	reg_995
Register 996	reg_996
Register 997	reg_997
Register 998	reg_998
Register 999	reg_999
Register 1000	reg_1000

Default Names for Counters

Counter Number	Symbolic Name
Counter 1	ctr_1
Counter 2	ctr_2
Counter 3	ctr_3
Counter 4	ctr_4
Counter 5	ctr_5
Counter 6	ctr_6
Counter 7	ctr_7
Counter 8	ctr_8

Default Names for Data Table Columns

Data Table Column Number	Symbolic Name
Column 1	col_1
Column 2	col_2
Column 3	col_3
Column 4	col_4

Default Names for Flags

Flag Number	Symbolic Name
Flag 1	flag_1
Flag 2	flag_2
Flag 3	flag_3
Flag 4	flag_4
Flag 5	flag_5
Flag 6	flag_6
Flag 7	flag_7
Flag 8	flag_8

Default Symbolic Names for Numbers

Default Names for Numbers

Number	Symbolic Name
0	clear
0	false
1	true
1	on

Default Symbolic Names for Steps

Default Names for Steps

Undefined Step	Symbolic Name
0	TOP

Default Symbolic Names for Inputs and Outputs

Default Names for Inputs

Input Number	Symbolic Name	State
Input 1	input_1_off	Closed (1)
Input 1	input_1_on	Open (0)
Input 2	input_2_off	Closed (1)
Input 2	input_2_on	Open (0)
Input 3	input_3_off	Closed (1)
Input 3	input_3_on	Open (0)
Input 4	input_4_off	Closed (1)
Input 4	input_4_on	Open (0)
Input 5	input_5_off	Closed (1)
Input 5	input_5_on	Open (0)
Input 6	input_6_off	Closed (1)
Input 6	input_6_on	Open (0)
Input 7	input_7_off	Closed (1)
Input 7	input_7_on	Open (0)
Input 8	input_8_off	Closed (1)
Input 8	input_8_on	Open (0)

Default Names for Outputs

Output Number	Symbolic Name	State
Output 1	output_1_off	Off (0)
Output 1	output_1_on	On (1)
Output 2	output_2_off	Off (0)
Output 2	output_2_on	On (1)
Output 3	output_3_off	Off (0)
Output 3	output_3_on	On (1)
Output 4	output_4_off	Off (0)
Output 4	output_4_on	On (1)
Output 5	output_5_off	Off (0)
Output 5	output_5_on	On (1)
Output 6	output_6_off	Off (0)
Output 6	output_6_on	On (1)
Output 7	output_7_off	Off (0)
Output 7	output_7_on	On (1)
Output 8	output_8_off	Off (0)
Output 8	output_8_on	On (1)

Default Symbolic Names for Specialized I/O Devices

Default Names for Displays

Display Number	Symbolic Name
Display 1	display_1
Display 2	display_2
Display 3	display_3
Display 4	display_4

Default Names for Thumbwheels

Thumbwheel Number	Symbolic Name
Thumbwheel 1	wheel_1
Thumbwheel 2	wheel_2
Thumbwheel 3	wheel_3
Thumbwheel 4	wheel_4

Default Names for Analog Inputs

Input Number	Symbolic Name
Analog Input 1	ain_1
Analog Input 2	ain_2
Analog Input 3	ain_3
Analog Input 4	ain_4

Default Names for Digital Outputs

Output Number	Symbolic Name
Analog Output 1	aout_1
Analog Output 2	aout_2
Analog Output 3	aout_3
Analog Output 4	aout_4

Default Symbolic Names for Motion Control Devices

Default Names for Stepping Motors

Stepping Motor Number	Symbolic Name
Stepping Motor 1	motor_1
Stepping Motor 2	motor_2
Stepping Motor 3	motor_3
Stepping Motor 4	motor_4

Default Names for Servo Motors

Servo Motor Number	Symbolic Name
Servo Motor 1	servo_1
Servo Motor 2	servo_2
Servo Motor 3	servo_3
Servo Motor 4	servo_4
Servo Motor 5	servo_5
Servo Motor 6	servo_6
Servo Motor 7	servo_7
Servo Motor 8	servo_8

Default Symbolic Names for Special Registers

Default Names for Special Registers

Register Number	Symbolic Name
Register 127	resource_pointer
Register 128	resource_access
Register 131	dt_row_pointer
Register 132	dt_column_pointer
Register 1001	output_01
Register 1002	output_02
Register 1003	output_03
Register 1004	output_04
Register 1005	output_05
Register 1006	output_06
Register 1007	output_07
Register 1008	output_08
Register 1009	output_09
Register 1010	output_10
Register 1011	output_11
Register 1012	output_12
Register 1013	output_13
Register 1014	output_14
Register 1015	output_15
Register 1016	output_16
Register 2001	input_01
Register 2002	input_02
Register 2003	input_03
Register 2004	input_04
Register 2005	input_05
Register 2006	input_06
Register 2007	input_07
Register 2008	input_08
Register 2009	input_09
Register 2010	input_10
Register 2011	input_11
Register 2012	input_12
Register 2013	input_13

Register Number	Symbolic Name
Register 2014	input_14
Register 2015	input_15
Register 2016	input_16
Register 3001	disp_1
Register 3002	disp_2
Register 3003	disp_3
Register 3004	disp_4
Register 4001	disp8_1
Register 4002	disp8_2
Register 5001	high_speed_ctr_1
Register 5002	high_speed_ctr_2
Register 6001	decimal_point_disp_1
Register 6500	snapshot_control
Register 7001	2205_1_pos
Register 7002	2205_2_pos
Register 7003	2205_3_pos
Register 7004	2205_4_pos
Register 8001	analog_out_1
Register 8002	analog_out_2
Register 8003	analog_out_3
Register 8004	analog_out_4
Register 8005	analog_out_5
Register 8006	analog_out_6
Register 8007	analog_out_7
Register 8008	analog_out_8
Register 8501	analog_in_1
Register 8502	analog_in_2
Register 8503	analog_in_3
Register 8504	analog_in_4
Register 8505	analog_in_5
Register 8506	analog_in_6
Register 8507	analog_in_7
Register 8508	analog_in_8

Register Number	Symbolic Name
Register 9000	dt_access
Register 9001	analog_1_gain
Register 9002	analog_2_gain
Register 9003	analog_3_gain
Register 9004	analog_4_gain
Register 9501	analog_1_resolution
Register 9502	analog_2_resolution
Register 9503	analog_3_resolution
Register 9504	analog_4_resolution
Register 10001	group1_32_outs
Register 10002	group2_32_outs
Register 10003	group3_32_outs
Register 10004	group4_32_outs
Register 10101	group1_16_outs
Register 10102	group2_16_outs
Register 10103	group3_16_outs
Register 10104	group4_16_outs
Register 10201	group1_8_outs
Register 10202	group2_8_outs
Register 10203	group3_8_outs
Register 10204	group4_8_outs
Register 10205	group5_8_outs
Register 10206	group6_8_outs
Register 10207	group7_8_outs
Register 10208	group8_8_outs
Register 11001	group1_32_ins
Register 11002	group2_32_ins
Register 11003	group3_32_ins
Register 11004	group4_32_ins
Register 11101	group1_16_ins
Register 11102	group2_16_ins
Register 11103	group3_16_ins
Register 11104	group4_16_ins

Register Number	Symbolic Name
Register 11201	group1_8_ins
Register 11202	group2_8_ins
Register 11203	group3_8_ins
Register 11204	group4_8_ins
Register 11205	group5_8_ins
Register 11206	group6_8_ins
Register 11207	group7_8_ins
Register 11208	group8_8_ins
Register 12000	comm_status
Register 12001	dt_row_xmit
Register 12300	comm_mode_control
Register 12301	comm_baud_control
Register 12302	comm_inchar_count
Register 12303	comm_parse_control
Register 12304	incoming_message_parser
Register 12305	all_flags
Register 12306	comm_switch_control
Register 12307	comm_switch_delay
Register 12309	comm_switch_output
Register 12310	comm_format_control
Register 13002	millisecond_timer
Register 13003	ctc_rev
Register 13004	ctc_type
Register 13008	ctc_model
Register 13009	software_fault_output
Register 13010	analog_input_range
Register 13011	super_task
Register 13012	current_task_num
Register 14001	servo_1_position
Register 14002	servo_2_position
Register 14003	servo_3_position
Register 14004	servo_4_position
Register 14005	servo_5_position

Register Number	Symbolic Name
Register 14006	servo_6_position
Register 14007	servo_7_position
Register 14008	servo_8_position
Register 14101	servo_1_error
Register 14102	servo_2_error
Register 14103	servo_3_error
Register 14104	servo_4_error
Register 14105	servo_5_error
Register 14106	servo_6_error
Register 14107	servo_7_error
Register 14108	servo_8_error
Register 14201	servo_1_velocity
Register 14202	servo_2_velocity
Register 14203	servo_3_velocity
Register 14204	servo_4_velocity
Register 14205	servo_5_velocity
Register 14206	servo_6_velocity
Register 14207	servo_7_velocity
Register 14208	servo_8_velocity
Register 14301	servo_1_status
Register 14302	servo_2_status
Register 14303	servo_3_status
Register 14304	servo_4_status
Register 14305	servo_5_status
Register 14306	servo_6_status
Register 14307	servo_7_status
Register 14308	servo_8_status
Register 14501	servo_1_vel_feed_fwd
Register 14502	servo_2_vel_feed_fwd
Register 14503	servo_3_vel_feed_fwd
Register 14504	servo_4_vel_feed_fwd
Register 14505	servo_5_vel_feed_fwd
Register 14506	servo_6_vel_feed_fwd

Register Number	Symbolic Name
Register 14507	servo_7_vel_feed_fwd
Register 14508	servo_8_vel_feed_fwd
Register 14601	servo_1_decel
Register 14602	servo_2_decel
Register 14603	servo_3_decel
Register 14604	servo_4_decel
Register 14605	servo_5_decel
Register 14606	servo_6_decel
Register 14607	servo_7_decel
Register 14608	servo_8_decel
Register 14701	servo_1_inputs
Register 14702	servo_2_inputs
Register 14703	servo_3_inputs
Register 14704	servo_4_inputs
Register 14705	servo_5_inputs
Register 14706	servo_6_inputs
Register 14707	servo_7_inputs
Register 14708	servo_8_inputs
Register 14801	servo_1_accel_feed_fwd
Register 14802	servo_2_accel_feed_fwd
Register 14803	servo_3_accel_feed_fwd
Register 14804	servo_4_accel_feed_fwd
Register 14805	servo_5_accel_feed_fwd
Register 14806	servo_6_accel_feed_fwd
Register 14807	servo_7_accel_feed_fwd
Register 14808	servo_8_accel_feed_fwd

Glossary

Glossary

Controller Resources

CTC controllers provide the following internal controller resources you can use when writing your Quickstep program: special and general purpose numeric registers, counters, flags, and Data Table.

Counters

Counters allow the automatic counting of pulses from the controller's inputs. They work in the background and, once started, operate much like an independent device within the controller.

Data Table

The Data Table is a two-dimensional array of numbers that can be stored in the controller's memory along with your Quickstep program. Storing this information in the Data Table instead of within the body of a program makes the program easier to maintain. The size of the Data Table depends on the controller model.

Dedicated Inputs

Dedicated inputs are functions that can be programmed for certain controller inputs. They are called Start, Stop, Reset, and Step.

Flags

Flags are memory elements within a controller that can be either set or clear and are used to store yes/no types of information.

Multi-tasking

Multi-tasking programs in Quickstep execute multiple program modules simultaneously. Each module can control a separate sequence of events.

Nesting

Any task in a multi-tasking program can contain other tasks inside of it. Tasks contained within a task are called nested tasks. Nested tasks must start and end during the execution of its parent task and follow the rules for multiple tasks.

Numeric Registers

Numeric Registers are storage locations for numbers within your controller. Special purpose registers perform specific functions, depending on the register number and the value stored in it.

For the storage capacity of the general purpose registers and a list of the special purpose registers and their functions, refer to *Register Reference Guide*, and the installation and applications guide for your controller model.

Parameter Editor

Use the Parameter editor to specify the following information:

- The model of your controller
- The number of rows and columns in data table
- Which, if any, of the first four inputs are used for dedicated functions

Recursion

Recursion occurs when a task (or one of its nested subtasks) includes an instruction to restart the same task. Eventually the controller has to keep track of so many tasks that it crashes.

Registers (see Numeric Registers)

Specialized I/O Devices

Quickstep supports the following specialized input/output devices:

- Analog inputs and outputs
- Thumbwheel arrays
- Numeric displays

Specialized Motion Control Devices

Quickstep supports the following specialized motion control devices:

- Servo motors
- Stepping motors

Step

A Quickstep program uses *steps* to define each new state of a machine. A complete program is composed of a series of steps executed in a defined pattern. Steps usually contain the following two elements:

- An action that establishes the machine's new state
- One or more instructions for leaving the step. These instructions establish the duration of the state.

Symbol Browser

Use the Symbol Browser to specify symbolic names for steps, numeric constants and the following controller resources and special devices:

- Analog inputs
- Analog outputs
- Counters
- Data Table columns
- Displays
- Flags
- Inputs
- Outputs
- Stepping motors and servos
- Numeric registers
- Thumbwheels

Symbolic Names

Symbolic names are names given to resources, such as registers, inputs, or motors. Starting with Quickstep 2.0 you can give resources like these symbolic names. Symbolic names can identify the function that the resource performs. For example, a series of servo motors can be called Traverse, Rotate, and Spindle, rather than Servo_1, Servo_2, and Servo_3.

Glossary Glossary-3

Control Technology Corporation proprietary. Reproduction or distribution forbidden.

Index

Index

Symbols

A

B

C

D

E

F

G

H

I

Q

R

S

T

V

Z