# Enough to be Dangerous:

## A Programming Primer

**Comments in a program:** Comments can be used anywhere in the program. Comments start with a semi-colon (;) and anything after the semi-colon is considered a comment. This can be used to document the program or to quickly remove a line of code from execution. In this document, every line of code will be commented to help with understanding the commands used.

**Motion profile parameters:** Before you can move the motor, you need to define a series of parameters to define HOW it will move. This includes setting up the units of the machine and setting up the velocity and acceleration for the move. These commands are actually just setting internal variables called "system variables" in the drive.

**UNITS:** This system variable is used to define how many user units the system will move per revolution of the motor. The user units (UU) can be anything the customer desires: Millimeters, inches, degrees, furlongs, radians, etc. For example, if the customer has the motor attached to a ballscrew that has a pitch of 10mm, and he wants his UU to be millimeters, then the system variable will be set using the following command:

```
UNITS=10 ; One revolution of the motor will be 10 UU, in this case 10mm.
```

**MAXV:** This system variable sets the maximum velocity of the motion profile. This is the speed at which the motor will rotate during the motion profile (unless, of course, the acceleration and deceleration prevent it from ever spinning that fast). This variable is always in the units of UU/second. If the customer wants to have his motor move at a maximum velocity of one meter per second, and his UU is mm, then the command would be:

```
MAXV=1000 ; 1000mm/second, or 1m/s
```

If a program requires different speeds for different moves, simply change the MAXV value before the individual moves.

**ACCEL and DECEL:** These system variables are used, obviously, to set the acceleration and deceleration ramp for the motion profile. The units for these variables are UU/sec$^2$. For the above example, if the customer wants to accelerate to (and decelerate from) the maximum velocity (1000mm/s) in 0.2 seconds, then apply the following command:

```
ACCEL=MAXV/0.2 ;MAXV is the maximum speed, 0.2 is the time for acceleration.
DECEL=ACCEL ; Set the deceleration equal to the acceleration
```

Also, two new concepts were introduced in these commands: Firstly, when declaring a variable you can use other variables. Secondly, it's possible to do math when setting a variable, like dividing the velocity by the deceleration time.

## Causing Motion:

**ENABLING THE MOTOR:** To enable the motor (provide current to the motor so it has torque), just use the command ENABLE. To disable the motor, use the command DISABLE.

**JOGGING:** Jogging (moving at a certain speed with no set distance) is accomplished through the MOVE UNTIL or MOVE WHILE commands. MOVE UNTIL will jog forward until a condition becomes true. To move in the reverse direction, use the command MOVE BACK UNTIL. These commands are often used while homing. To jog in the reverse direction until an input comes on (more on inputs later), use the command:

```
MOVE UNTIL IN_A1 ;Move forward at speed defined by MAXV until input A1 is TRUE.
```

**MOVING A DISTANCE:** This is a relative move, which is a move that moves a certain distance from the start point. This type of move is typically used when indexing forward repeatedly, such as a feed system. The command to do this is MOVED x, where x is the distance (in UU) to be move from the starting point. So, if you want to move forward five units, the command is:

```
MOVED 5 ; Move a distance of 5 units forward from wherever the motor is now.
```

If the actual starting position is 2 units from 0, the end position will be 7 units. To move backwards, just make the distance a negative (i.e. MOVED –5). The velocity, accel, and decel are determined by the MAXV, ACCEL, and DECEL variables, respectively.

**MOVING TO A POSITION:** This is an absolute move, which is a move that goes to a certain distance from zero. This type of move is best used for back-and-forth movement, such as a pick-and-place. If the user wants to move the position 5 units from zero, apply the following command:

```
MOVEP 5 ; Move to the position that is five units from the zero position.
```

Using this command, if the starting position is 2, the motor will move 3 units forward to end at a final position of 5. Similarly, if the starting position is 99, the motor will move backwards 94 units and again end up at a position of 5 units.

## Using I/O and User Variables:

**INPUTS:** An input is a simple true or false evaluation of whether an input is on or off. There are three groups of inputs (A, B and C), each with four individual inputs (1, 2, 3 and 4). To evaluate the first input of the A group use the variable IN_A1. If it's ON, then IN_A1 will be 1, whereas if it's off, it will be 0.

**OUTPUTS:** Outputs are also true or false values. There are four programmable outputs which use the variables OUT1 to OUT4. To turn an output ON, set it equal to 1. To turn an output off, set it equal to 0.

```
OUT1=IN_A1 ; Set Output 1 ON if Input A1 is ON, and OFF if the input is OFF.
```

**NEGATING AN INPUT:** Some applications require you to evaluate if an input is OFF instead of when it is on. This is called "negating" an input, or a NOT condition. You can negate an input (or any logical evaluation) by preceding the variable with an exclamation point (!).

```
OUT2=!IN_A2 ; Set Output 1 ON if Input A2 is OFF, and OFF if the input is ON.
```

**USER VARIABLES:** In addition to the various system variables, there are of course variables that the user can define. There are 32 of these variables, labeled V0 through V31. These variables are inherently floating point (that is, they are capable of storing decimal variables), but can also store integers and bit-type data as well.

**DEFINING CONSTANTS OR VARIABLES:** When looking at a complex program, variable names like IN_A1 or V0 mean very little. It is advantageous to use "real-world" descriptions in place of variable names. Therefore it is useful to use the DEFINE command to "rename" these variables. For instance, if the input IN_B3 is an input used to start a move, and OUT1 is used to confirm the status of this input, the programmer might use the code:

```
DEFINE StartInput IN_B3
DEFINE ConfirmStartOutput OUT1
ConfirmStartOutput=StartInput
```

This code is equivalent to the command "OUT1=IN_B3", which is a shorter to type method, but is not intuitive when reading the code. A similar method can be used to define a constant value. For instance, if a move starts at a certain point and ends at another point, the programmer might use:

```
DEFINE StartPoint 0 ; The start point for the move is 0 UU
DEFINE EndPoint 10 ; The end point for the move is 10 UU
```

If the move parameters change for a different product or for a different machine, the user only needs to change these two lines instead of hunting through the program and trying to figure out which values need to be changed.

When writing a complex program, it is advisable to use the DEFINE statement as much as possible so the program is easy to follow and change when necessary. Note, however, that all DEFINE statements must be at the beginning of the program.

## Waiting and Looping:

WAITING: In many instances, it is necessary to wait for certain conditions to occur before continuing on with the program. To accomplish this, use the WAIT command. This can be done in a few ways:

- WAIT UNTIL is used to wait until a certain condition is true. For instance, if the user wants to start a relative once an input has come on, the code would be:

  ```
  WAIT UNTIL IN_A1 ;Do not proceed with the program until Input A1 is ON
  MOVED 10 ;Move to a distance 10 UU from where it started at
  ```

- WAIT WHILE is used to wait while a certain condition is true. It is the same as using a WAIT UNTIL with a negated condition (i.e. WAIT UNTIL IN_A1 is the same command is WAIT WHILE !IN_A1).

- WAIT TIME is used to wait for a certain time, defined in milliseconds. To delay for ½ second, use the command:

  ```
  WAIT TIME 500 ; Wait for 500 milliseconds
  ```

- WAIT MOTION COMPLETE is used to wait until all motion that has been commanded is finished.

**DO LOOPS:** In some cases, it is desirable to perform an operation several times while certain conditions are true. This can be accomplished using the WHILE and DO UNTIL loops. As they are not commonly used, it is recommended that the programmer read more about them in the 940 Programming Manual.

**UNCONDITIONAL LOOPS:** It is common for a program to repeat itself when in operation. This is accomplished using labels and the GOTO command. A label is a text field followed by a colon. The label text cannot be the same as a command, variable, or user text defined using the DEFINE command. The GOTO command is used to jump to one of the labels in the program. Here is a simple program that moves the motor forward five units, waits ½ second, and then repeats:

```
JumpPoint:     ;This is the label. Note that it is followed by a colon.
MOVED 5        ;Move forward 5 units.
WAIT TIME 500  ;Wait 500 milliseconds
GOTO JumpPoint ;Go to the defined label. Notice the colon is NOT included.
```

## Using what's been explained so far:

What follows is a sample exercise using the concepts and commands we've learned so far. In this example, the user has a rotary table that's attached to the motor through a 50:1 gearbox. When an input has a rising edge, the table should move forward 30° in about one second. When the table is in it's final position, an output will be turned on. When the move is done, the program should wait for the next input and repeat the process.

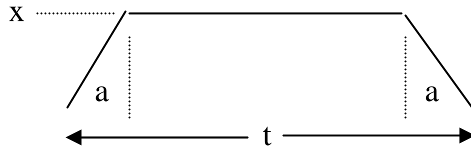
```
DEFINE GearboxRatio 50   ;The ratio of the gearbox.
DEFINE MoveDistance 30   ;The distance to be moved (degrees).
DEFINE MoveTime 1000     ;The time of the move (ms).
DEFINE AccelTime 100     ;The portion of the time to accel up to speed (and decel).
DEFINE StartInput IN_A1  ;The input used to trigger the start of the move
DEFINE CompleteOutput OUT1;The output used to indicate that the move is complete
DEFINE TRUE 1            ;Instead of setting the outputs to 1 and 0…
DEFINE FALSE 0           ; this program will use TRUE and FALSE

UNITS=360/GearboxRatio   ;1 g'box rev is 360deg;One motor rev is 1/50 of g'box rev

MAXV=MoveDistance/(MoveTime-AccelTime)
                         ;This command is a little more complicated, but remember
                         ;that velocity is distance over time. The move distance is
                         ;defined. The time is determined by taking the total move
                         ;time and subtracting the acceleration time. There is an
                         ;explanation below.
ACCEL=MAXV/AccelTime     ;The acceleration time is defined above, and the velocity
                         ;was just calculated. Acceleration is velocity divided by
                         ;time.
DECEL=ACCEL              ;Set the deceleration equal to the acceleration.

MoveStart:               ;This is a label for the loop being used in this program.
CompleteOutput=FALSE     ;Turn off the indication that the move is complete.
WAIT UNTIL !StartInput   ;Wait for the input to be OFF, then…
WAIT UNTIL StartInput    ;Wait for the input to turn ON. These two commands combine
                         ;to form an "edge detection" for a rising edge.
MOVED 30                 ;Move 30 units
CompleteOutput=TRUE      ;Turn on the indication that the move is complete.
GOTO MoveStart           ;Jump to the label to wait for the next trigger signal.
```

A couple of notes about this program: First, note that there is no WAIT MOTION COMPLETE command after the move. This is because, by default, the MOVED and MOVEP commands will automatically wait for the motion to complete before continuing to the next command. Also note that because the important parameters for the program are defined at the beginning, it is very simple to change the program. If the move needs to be 45° instead of 30°, it is only necessary to change the second line to `DEFINE`

`MoveDistance 45` and everything else (velocity and acceleration rates, specifically) will take care of itself.

This explanation of setting the MAXV command has little to do with programming and more with general motion control concepts, but it is important to understand.

Take the following motion profile as an example:




The time a is the acceleration and the total move time is t. For our program a is 100ms, and t is 1000ms. The distance moved is the area under the curve. X is the speed we want to calculate.

Knowing that the area under the curve is the distance traveled and using simple geometry, we know that the distance moved during acceleration is da=1/2*a*x, and the deceleration is the same. The distance traveled during the constant-speed part of the move is dt=(t-2a)*x. So the whole distance traveled is dtotal=dt+da+da, or dtotal=(t-2a)*x+2(1/2*a*x), or dtotal=tx-2ax+ax, or dtotal=tx-ax. Solving for x (which, again, is the velocity), we get x=dtotal/(t-a). Plugging in the variable names from the program, the result is the equation used in the program: MAXV=MoveDistance/(MoveTime-AccelTime).

## Conditional Operations, comparatives and multiple conditions:

**IF/ELSE STATEMENTS:** An IF/ELSE statement is used to make a decision. It must be closed with an ENDIF command. For instance, if a user wants to set a variable to 15 if an input is on, and 30 if it's not, then the code would be:

```
IF IN_A1      ;If the input is ON,
  V1=15       ;Set the variable to 15
ELSE          ;Otherwise,
  V1=30       ;Set the variable to 30
ENDIF         ;The end of the IF statement
```

You can have multiple commands for IF and ELSE statements. ELSE statements are optional. IF statements can also be "nested" to check for several conditions.

**COMPARATIVES:** Comparatives are used to compare one value to another. Comparatives in the 940 are: "greater than" (A>B), "less than" (A<B), equality (A==B), inequality (A<>B), "greater or equal" (A>=B), and "less or equal" (A=<B). Note that equality uses a double-equal when it is *comparing* values (like A==B, where the result is true or false) to differentiate it from a command that *sets* the values equal (such as A=B, where A takes the value of B). For instance, if the program has a certain variable that has a maximum value of 15, then the code might be:

```
IF V1>15      ;If the variable is greater than the maximum value,
  V1=15       ; set the variable to the maximum value.
ENDIF         ;Close the IF statement.
```

**MULTIPLE CONDITIONS (AND/OR):** More than one condition can be used when making a logical decision. The two types that are used are AND and OR conditions. For an AND condition, the symbol && is used; for the command "IF A&&B&&C " then A, B, and C must all be true for the IF statement to evaluate as true. With the OR condition, the symbol || is used. (The | is called a "pipe" and is typically

shift-\ on the keyboard.) For the command "IF A||B||C", then one or more of A, B, and C must be true for the IF statement to evaluate as true.

If the user wants to move 5 units if input B1 is on AND either input B2 OR B3 is on:

```
IF IN_B1&&(IN_B2||IN_B3)   ;If B1 and either B2 or B3 is on,
   MOVED 5                 ; Move 5 units
ENDIF                      ;Close the IF statement
```

## Registration:

Registration is moving to a distance from an input. For instance, a candy bar needs to be cut in the right place in order for the packaging to be centered on the product. To do this, a sensor reads a mark on the packaging, and the product is moved to a certain distance beyond that sensor trigger. In the 940 drive, the MOVEDR and MOVEPR commands are used. The registration input is always tied into input IN_C3.

**RELATIVE MOVE WITH REGISTRATION:** This type of move is just like the MOVED relative move command that was discussed earlier, except with a registration. There are two operands for the MOVED command: The default distance and the registration distance. The default distance is the distance that the motor will travel if no registration signal is received. The registration distance is the distance past the registration signal that the motor will move. The command is:

```
MOVEDR DefaultDistance, RegistrationDistance.
```

Imagine a situation where a product is typically moved about 10 units, but to compensate for any slip or error in the system, a registration signal is used. This registration mark is detected ½ unit from the end of the product. The programmer might use the following code:

```
MOVEDR 11, 0.5 ;Move 11 units or 0.5 from the registration input.
```

11 is used instead of the distance of 10 because there may be slip in the system. However, if the registration input is detected during that 11 units of travel, it will change the final position to ½ unit from the point the motor is at when the sensor is triggered.

**ABSOLUTE MOVE WITH REGISTRATION:** Absolute moves with registration are accomplished by the MOVEPR command. This is just like the MOVEDR command, but the default move is an absolute move instead of a relative move.

**Other Important System Variables:**

**APOS:** This is the actual position that the motor is currently at. You can see where the motor is at by monitoring this variable, or you can change the definition of the current position by redefining this variable. For instance, after a machine is homed you want to set the actual position counter to zero. To do this, the code is:

```
APOS=0 ;Set the actual position to zero.
```

Note that this doesn't move the motor at all. It simply tells the drive that the motor's current position is to be considered zero.

**PHCUR:** This variable indicates the amount of current being used by the motor measured in Amps. It is a read-only variable. In many applications, the homing process needs to be accomplished by (lightly) running the motor to a hard stop to find the end of travel. To do this, the PHCUR variable can be compared to a preset value. For instance:

```
MOVE BACK UNTIL PHCUR>2    ;Move the motor backwards until more than 2A
                           ;of current is being used by the motor.
```

To determine the appropriate level of current, a trial-and-error method is best.

**AIN:** The AIN input is the analog input value in volts from –10 to 10. If there is 2.76V on the analog input, then AIN will have a value of 2.76. It is a read-only value.

**AOUT:** The AOUT is the analog output in volts from –10 to 10. To put a value of 3.54V on the analog output, use the code:

```
AOUT=3.54 ;Set the analog output to 3.54V
```

## Events:

**What are events?** The processor in the 940 drive is actually doing several things at once. There is the main program execution, of course, which is what this document has focused on so far. In addition, there's the actual motion (which is triggered by the main program, but really a separate processor task), communication, fault handling, events, and other processes. Events are a task that runs independent of the main program execution. So, in essence, an event is something that's happening in the background.

Every 256μs, the event scanner checks all active events and activates the ones that should be activated. Events can be triggered by an input, a time period, or a true/false expression. When the criteria for the event are met, the event executes.

There are some operations that events cannot directly do. For instance, an event cannot directly cause motion because only the main program triggers motion. However, the JUMP statement can be used in an event to "redirect" the program execution to a new point in the program, allowing the programmer to trigger motion.

**Defining events:** Events are defined by using the EVENT command followed by the name and condition for the event. For instance, if the user wants to have a programmable limit switch function where an output is activated when the position of the motor is between 50 and 100, the code might be:

```
EVENT PLS TIME 10          ;Perform this event (named “PLS”) every 10ms.
   IF APOS>50&&APOS<100    ;If the actual position is between 50 and 100…
      OUT1=1               ;Turn the output on
   ELSE                    ;Otherwise…
   OUT1=0                  ;Turn the output off
ENDEVENT                   ;End the event definition
```

This *defines* the event, but the event is not active until it is turned on in the main program (see next section). It is also important to remember that the events must be defined just after any DEFINE statements at the beginning of the program.

**Turning on events:** The EVENT statement is also used to activate it in the main program. To turn on the event we defined in the previous section, the code is:

```
EVENT PLS ON ;Turn event named “PLS” on.
```

Turn an event off with:

```
EVENT PLS OFF ;Turn the event named “PLS” off.
```

## Putting it all together to write a sample program:

A customer has an application where the motor is driving a sheet of paper forward. The motor is attached to a wheel that has a diameter of 6″. This wheel rides on the paper and pushes it forward. There is a mark on the paper every 15″, and a sensor attached to input C3 that reads that mark. When input B2 is triggered, the customer wants to move to a position 1.5″ after the sensor reads the mark on the paper. When it is in position, output 2 is tuned on to start an outside process. The motor will wait until the input B2 is triggered again, and then repeat the process.

A sample program to demonstrate this application:

```
DEFINE WheelDiameter 6            ;The wheel has a diameter of six inches
DEFINE DefaultDistance 18         ;The wheel should move 15”, but might have slip
DEFINE RegistrationDistance 1.5   ;The paper will move 1.5” past the registration
                                  ; sensor input
DEFINE MoveSpeed 30               ;The speed was not defined, so 30 in/s is
                                  ; arbitrarily used
DEFINE MoveAccel 300              ;The accel is not defined, so 300 in/s/s is
                                  ; arbitrarily used
DEFINE SensorInput IN_B1          ;The sensor is attached to input C3
DEFINE GoInput IN_B2              ;The “go” input is attached to input B2
DEFINE DoneOutput OUT1            ;The done output is attached to output 1
DEFINE pi 3.141592                ;The numerical constant pi

UNITS=WheelDiameter*pi            ;In one revolution of the motor, the paper will
                                  ; move by the circumference of the wheel, which
                                  ; is the diameter times pi.
MAXV=MoveSpeed                    ;Set the maximum velocity
ACCEL=MoveAccel                   ;Set the move acceleration.
DECEL=ACCEL                       ;Set the move deceleration equal to the accel
DoneOutput=0                      ;Initialize the motion completion output
ENABLE                            ;Enable the motor

MainLoop:                         ;A label to jump to later in the program

WAIT UNTIL GoInput                ;Wait until the start input is activated
DoneOutput=0                      ;Turn off the completion output
MOVEDR DefaultDistance, RegistrationDistance
                                  ;Do a registration move
DoneOutput=1                      ;Turn on the completion output
GOTO MainLoop                     ;Loop infinitely
```

This is not a complete program for an industrial environment, but it serves as a sample to prove the concept to the customer. In addition, using so many DEFINE statements for such a simple program might seem like overkill, but it’s good programming practice and makes the program much easier to follow. It can be used as the basis for the full program once the product is sold.

## Appendix: Compiling, downloading, and executing a user program:

**COMPILING:** The compiler is a tool that checks the code of the program and makes sure that there are no errors. To compile a program, press F6 from the programming screen. If the compiler can not understand your program due to errors in syntax, undefined variable tags, etc., a dialog box will appear on your screen indicating that the compiling has failed. The status screen at the bottom of the MotionView window will display the type of error and the line number where the error was found.

**DOWNLOADING:** The 'Compile and Load' function (Shift-F6 from the programming screen) causes MotionView to compile the user program and, if the compilation is successful, download it to the drive. If an error occurs during compilation, the program will not be downloaded to the drive.

**RESTARTING THE PROGRAM:** The restart function (Shift-F5 from the programming screen) resets any drive alarms and sets the program to restart from the beginning.

**EXECUTING THE PROGRAM:** To execute the program, the F5 button is used. The program can also be automatically executed on drive power-up by using the "Autoboot" option in the Parameter menu. If Autoboot is enabled, then program starts as soon as the 940 drive is powered on.

**STOPPING THE PROGRAM:** To stop the program, the Alt-F5 key is used.

**MONITORING PROGRAM VARIABLES:** The debug view can be used to monitor system variables in the drive. To open the debug view, press the F10 key. A list of system and user variables appears on the right side of the dialog box. Move these over to the monitor area to view their value. For inputs and outputs, indicators at the bottom of the dialog box are used to show the status of the input or output.

**Typical procedure for downloading and running a program:**

1. From the MotionView programming screen, press Shift-F6 to compile the program and load it to the drive. If any errors occur during compilation, correct them.
2. Press Shift-F5 to reset any faults on the drive and restart the program from the beginning.
3. Press F5 to execute the program.
4. After testing, press Alt-F5 to stop the program from executing.

**Toolbar buttons in MotionView programming environment:**