Logic and program flow control

As you know, the general flow (order of execution) for a standard CNC program is from beginning to end. When a program is activated, the control will read, interpret, and execute the very first command in your program. Then it will proceed to the next command. Read, interpret, and execute. Then on to the next command. It is by this sequential order that CNC programs are normally executed.

With parametric programming, you have the ability to change the order of program execution. You can have your parametric programs make tests to determine which of two or more possible outcomes it will follow. You can even make the control repeat a series of commands until some criteria related to the series of commands is satisfied. We’ll call this powerful ability to change the order of program execution program flow control.

Most parametric programming languages offer several commands related to program flow control. While the actual syntax for each command varies from one version to another, the need for each command type remains remarkably similar. In this chapter, we introduce program flow command types and show examples of when they are used.

Statement labels

Most program flow commands require unique pointers or markers to designate positions in the program that are to be used to control program flow. We call these markers statement labels. Statement labels specify points in the program where the control is to continue executing the program. The use of statement labels will be clarified in greater detail as we discuss unconditional branching just a little later.

Statement labels in Fanuc’s custom macro B

Fanuc uses sequence numbers (N words) for statement labels. Though this text shows a sequence number in almost every command of most programs for the purpose of documentation, to conserve memory space it is a good idea to use sequence numbers in only those commands that require statement labels. One good technique is to take advantage of the numerical value of the sequence number to help you find the statement label in the program. This is especially helpful in very long programs.

For example, use sequence numbers under ten for statement labels close to the beginning of the custom macro program. Use sequence numbers labels around fifty right in the middle of the custom macro program. And use sequence numbers over ninety close to the end of the custom macro program. If you use this technique, you can easily find the statement label being referenced.

Keep in mind that while sequence numbers are quite flexible in normal CNC programming, there are two strict rules about their use as statement labels. First of all, if a command references a statement label, the statement label must exist in program. Otherwise an alarm will sound. Second, you must be careful not to use the same statement label twice in your program. While duplicate sequence numbers will not affect the flow of normal CNC programs, duplicate statement labels in parametric programs can result in unpredictable program flow or alarm generation.
Statement labels in Okuma’s user task 2
User task 2 also allows the use of sequence numbers for statement labels. And if you use sequence numbers as statement labels, we recommend that you follow the suggestions given in the previous discussion of custom macro B statement labels.

User task 2 provides the additional ability to use a more logical label name as a statement label. While the label name must also begin with the letter N (just like a sequence number), the next four characters allow much more flexibility in statement label naming.

If using statement label names (instead of numbers), the first character (following the N) must be a letter of the alphabet. The next three characters can be letters or numbers. Examples of allowable statement label names include NSTRT, NHERE, NPOS1, and NEXIT.

Note that statement labels cannot be duplicated in the program. If you use the statement label NSTRT at one point in the program, for example, you cannot use it in any other position in the same program. Additionally, if you reference a statement label that does not exist, an alarm will sound.

Statement labels in Fadal’s macro
Statement labels in macro must begin with a colon (:). Any series of letters and numbers can follow, meaning you have a great deal of flexibility in statement label naming. You’ll want to take advantage of this flexibility, and choose statement label names that make sense. Examples of allowable statement label names include :HERE, :LOOPBEGIN, :LOOPEND, and :EXIT. In Fadal’s macro, note that the statement label is the only word that can appear in the macro command.

Note that statement labels cannot be duplicated in the program. If you use the statement label :EXIT at one point in the program, for example, you cannot use it in any other position in the same program. Additionally, if you reference a statement label that does not exist, an alarm will sound.

Unconditional branching
As the name implies, the unconditional branching command will cause the control to branch or jump from one location in the program to another location in the same program. The location to which branching is done must be marked with a statement label. Most versions of parametric programming (including all addressed by this text) specify unconditional branching with a GOTO statement. The syntax for GOTO changes just slightly from one version to another. Here are three examples.

In Fanuc’s custom macro B:

GOTO 50 (Branch to statement label N50. Note no N word)

N50 G00 X . . . (Statement label N50)
In Okuma’s user task 2:

GOTO NSTRT (Branch to statement label NSTRT)

NSTRT G00 X . . . (Statement label NSTRT)

In Fadal’s macro:

#GOTO :EXIT ‘Branch to statement label :EXIT

#:EXIT ‘Only the statement label is allowed on this line!

G00 X . . .

Applications for unconditional branching

While most applications for unconditional branching are related to parametric programming applications, there are some times when it can be helpful to know about unconditional branching even with standard CNC programs.

Changing machining order

For example, have you ever made a processing mistake with the sequence of machining operations when developing a complicated program? Consider this incorrect turning center process:

1) Rough face and turn
2) Finish face and turn
3) Drill 2” diameter hole
4) Rough bore
5) Finish bore

As you develop the CNC program, say it does not occur to you that you are breaking an important rule of basic machining practice that states: rough everything before you finish anything. So you write the program (shown here in Fanuc format) with the incorrect process.
Incorrect program shown in Fanuc format:

O0001 (Program number)
N005 T0101 (Index to rough turn and face tool)
N010 G96 S400 M03 (Select constant surface speed, start spindle)
N015 G00 X3.2 Z0.005 (Rapid to approach position)
N020 G01 X-0.062 F0.010 (Rough face)
N025 G00 X2.79 Z0.1 (Rapid to rough turn diameter)
N030 G01 Z-0.995 (Rough turn)
N035 X3.1 (Feed up face)
N040 G00 X6.0 Z5.0 (Rapid to tool change position)
N045 M01 (Optional stop)
N050 T0202 (Index to finish face and turn tool)
N055 G96 S500 M03 (Start spindle)
N060 G00 X3.2 Z0 (Rapid to approach position)
N065 G01 X2.15 F0.006 (Face just past diameter to be bored)
N070 G00 X2.75 Z0.1 (Rapid to finish turn diameter)
N075 G01 Z-1.0 (Finish turn)
N080 X3.1 (Feed up face)
N085 G00 X6.0 Z5.0 (Rapid to tool change position)
N090 M01 (Optional stop)
N095 T0303 (Index to 2” drill)
N100 G97 S400 M03 (Start spindle)
N105 G00 X0 Z0.1 (Rapid to approach position)
N110 G01 Z-1.35 F0.008 (Drill hole)
N115 G00 Z0.1 (Rapid out of hole)
N120 G00 X6.0 Z5.0 (Rapid to tool change position)
N125 M01 (Optional stop)
N130 T0404 (Index to rough boring bar)
N135 G96 S400 M03 (Start spindle)
N140 G00 X1.21 Z0.1 (Rapid to approach position)
N145 G01 Z-0.995 F0.010 (Rough bore hole)
N150 X1.9 (Feed down face)
N155 G00 Z0.1 (Rapid out of hole)
N160 X6.0 Z5.0 (Rapid to tool change position)
N165 M01 (Optional stop)
N170 T0505 (Index to finish boring bar)
N175 G96 S500 M03 (Start spindle)
N180 G00 X1.25 Z0.1 (Rapid to finish turn diameter)
N185 G01 Z-1.0 F0.006 (Finish bore)
N190 X1.9 (Feed down face)
N195 G00 Z0.1 (Rapid out of hole)
N200 X6.0 Z5.0 (Rapid to tool change position)
N205 M30 (End of program)

It is not until after you take this program out to the machine and run it that you discover the mistake. Now your very expensive CNC turning center will be sitting idle while you change the process of this program. While some CNC controls have very nice cut, copy, and paste,
functions for CNC program segments, many do not. Most programmers will have to go back to their personal computer based CNC text editor or CAM system to fix the program.

The unconditional branching (GOTO) command will make it possible to easily change this program’s execution order right at the CNC control by adding three simple lines. Here is the corrected process followed by the modified program.

1) Rough face and turn
2) Drill 2” diameter hole
3) Rough bore
4) Finish bore
5) Finish face and turn

Modified program shown in Fanuc format:

O0001 (Program number)
N005 T0101 (Index to rough turn and face tool)
N010 G96 S400 M03 (Select constant surface speed, start spindle)
N015 G00 X3.2 Z0.005 (Rapid to approach position)
N020 G01 X-0.062 F0.010 (Rough face)
N025 G00 X2.79 Z0.1 (Rapid to rough turn diameter)
N030 G01 Z-0.995 (Rough turn)
N035 X3.1 (Feed up face)
N040 G00 X6.0 Z5.0 (Rapid to tool change position)
N045 M01 (Optional stop)
N048 GOTO 095 (Branch to beginning of drilling operation)
N050 T0202 (Index to finish face and turn tool)
N055 G96 S500 M03 (Start spindle)
N060 G00 X3.2 Z0 (Rapid to approach position)
N065 G01 X2.15 F0.006 (Face just past diameter to be bored)
N070 G00 X2.75 Z0.1 (Rapid to finish turn diameter)
N075 G01 Z-1.0 (Finish turn)
N080 X3.1 (Feed up face)
N085 G00 X6.0 Z5.0 (Rapid to tool change position)
N090 M01 (Optional stop)
N093 GOTO 205 (Branch to end of program)
N095 T0303 (Index to 2” drill)
N100 G97 S400 M03 (Start spindle)
N105 G00 X0 Z0.1 (Rapid to approach position)
N110 G01 Z-1.35 F0.008 (Drill hole)
N115 G00 Z0.1 (Rapid out of hole)
N120 G00 X6.0 Z5.0 (Rapid to tool change position)
N125 M01 (Optional stop)
N130 T0404 (Index to rough boring bar)
N135 G96 S400 M03 (Start spindle)
N140 G00 X1.21 Z0.1 (Rapid to approach position)
N145 G01 Z-0.995 F0.010 (Rough bore hole)
N150 X1.9 (Feed down face)
N155 G00 Z0.1 (Rapid out of hole)
N160 X6.0 Z5.0 (Rapid to tool change position)
N165 M01 (Optional stop)
N170 T0505 (Index to finish boring bar)
N175 G96 S500 M03 (Start spindle)
N180 G00 X1.25 Z0.1 (Rapid to finish turn diameter)
N185 G01 Z-1.0 F0.006 (Finish bore)
N190 X1.9 (Feed down face)
N195 G00 Z0.1 (Rapid out of hole)
N200 X6.0 Z5.0 (Rapid to tool change position)
N203 GOTO 050 (Branch to beginning of finish face and turn)
N205 M30 (End of program)

Note the addition of the three unconditional branching statements in lines N048, N093, and N203. These commands cause the control to machine the workpiece in the desired order.

*Using optional block skip to exit a program*

As you know, optional block skip (also called block delete) is specified by the slash code (/). When the control reads the slash code, it looks to the position of the optional block skip switch on the control panel. If the switch is on, the control skips the command that begins with the slash code. If the switch is off, the control executes the command.

There are times when it can be helpful to repeat a series of commands. Of course, subprogramming easily allows this. And as long as the number of times the subprogram must be executed is known, programming is very simple. However, if the programmer is unsure about how many times the subprogram must be executed, programming becomes more difficult.

Consider the operation of trial boring on a machining center. With trial boring, the programmer gives the operator a series of special commands (in a subprogram) to perform the trial boring operation. However, the hole may or may not be perfectly to size after the first try. It may take several tries to get the hole right to size. In this case, the programmer does not know precisely how many times the subprogram must be executed. Consider these commands in custom macro B format.

```
/N065 M95 P1000 (Jump to trial boring subprogram)
/N070 GOTO 065 (Go back to line N065)
```

As long as the optional block skip switch is turned off, the subprogram will be executed. When the operator eventually gets the hole on size, he or she will turn on the optional block skip switch and the control ignores line N070 and proceeds with the balance of the program.

*With conditional branching*

While unconditional branching can sometimes be helpful even with standard CNC programs, from this point on we will concentrate on true parametric programming applications. By far, the most popular application for unconditional branching is in conjunction with conditional branching, which is the topic of the next discussion.
Conditional branching

Just like conditional branching in any computer programming language, conditional branching in parametric programming allows a decision-making capability. The parametric programmer can have commands in the program perform tests. The program will be executed differently based upon the results of tests.

The purposes for testing are extremely varied and we cannot over-stress the importance of conditional branching for testing purposes. You can test just about anything right within your parametric program. And based upon the results of your test, you can make the program execute differently. This feature has fantastic implications!

Syntax for conditional branching statements

Before we show specific applications for conditional branching, you must understand its general syntax. All versions of parametric programming addressed by this text use an IF statement for conditional branching. However, some versions of parametric programming are more flexible than others in the use of this statement. Rest assured that all applications discussed in this text can be handled with conditional branching, regardless of how flexible the IF statement.

The conditional expression

All IF statements require a conditional expression. The result of the conditional expression will be either true or false. If the result is true, the command to the right of the IF statement is executed. If the result is false, the command following the IF statement will be executed.

Conditional expressions require conditional operators. There are six conditional operators with most versions of parametric programming. Conditional operators include less than (LT), less than or equal to (LE), equal to (EQ), greater than or equal to (GE), greater than (GT), and not equal to (NE). Note that in Fadal’s macro, you can also use the same conditional operator symbols used in BASIC (< for less than, <= for less than or equal to, = for equal to, >= for greater than or equal to, > for greater than, and <> for not equal to).

The six conditional operators allow great flexibility in the method by which you develop conditional branching commands. In most cases, the same test can be written in several ways. While this flexibility is good, it can sometimes be confusing, especially when you are testing with the not equal to (NE or <>) conditional operator. Always remember that the test within the conditional branching statement is to determine whether the conditional expression is true or false. When testing against a negative condition, it is easy to make mistakes.

Conditional branching in Fanuc’s custom macro B

While newer Fanuc controls do allow a little more flexibility with conditional branching (and you will want to consult your Fanuc manual for more information), we limit our presentation to include only those functions of conditional branching that will work with all Fanuc control models equipped with custom macro B.

As mentioned earlier, if the conditional branching statement is true, the command to the right of the statement is executed. With custom macro B, you are limited to placing a GOTO statement to the right of the IF statement (newer controls additionally allow other custom macro B statements). Here is a specific example.

Say you wish to write a grooving custom macro for your turning center that can take advantage of right hand or left hand tooling. You wish to include a special argument in the call statement,
say H for hand, to specify the tooling style. Your criteria states that if H is set to one, right hand tooling is being used. If H is set to zero, left hand tooling is being used.

If, of course, right hand tooling is being used, the spindle must be running in a forward (M03) direction. If left hand tooling is being used, the spindle must be running in reverse (M04). Here is a portion of the custom macro program that performs the test and starts the spindle in the appropriate direction. (Note that letter address H must be referenced by local variable #11 in the custom macro.)

\[ \text{IF } \#11 \text{ EQ } 1. \text{ GOTO 5} \]
\[ \text{M04} \]
\[ \text{GOTO 6} \]
\[ \text{N5 M03} \]
\[ \text{N6 \ldots} \]

Note that brackets ([ ]) must be used to enclose the conditional expression. This conditional branching statement uses the equal to (EQ) logical operator, which makes it quite easy to evaluate the results of the test (true or false). If the programmer using this custom macro includes an H1.0 in the call statement, the control will evaluate the conditional branching statement as true. In this case, the GOTO statement next to the IF statement will be executed. In N5, notice that the M03 will start the spindle in the forward direction (as required for right hand tools).

On the other hand, if the person using this parametric program sets the value of H to zero (or anything else), the IF statement’s conditional expression will be evaluated as false. In this case, the command following the IF statement will be executed, starting the spindle in the reverse direction (as required for left hand tools).

Finally, note the need for the GOTO 6 statement. If the conditional expression is evaluated as false, and the spindle starts in the reverse direction, we need the control to skip the other possibility. This GOTO statement is needed to keep the control from immediately starting the spindle in the forward direction. While this use of the GOTO statement does not make for very structured programming, just about any situation requiring conditional branching can be handled with the IF statement in custom macro B.

Earlier we said that the six conditional operators allow a great deal of flexibility. This right hand versus left hand tooling test can be handled in several ways. Consider these commands that will render the same results:

\[ \text{IF } \#11 \text{ NE } 1. \text{ GOTO 5} \]
\[ \text{M03} \]
\[ \text{GOTO 6} \]
\[ \text{N5 M04} \]
\[ \text{N6 \ldots} \]
Or these:

```
IF [#11 GT 0] GOTO 5
M04
GOTO 6
N5 M03
N6 . . .
```

Which is best? In some cases, it truly does not matter which logical operator you choose to test with, meaning you can setup up the test in any way that works, and seems comfortable to you. However, there are many times when you will really want to think through your test to search for the hidden benefit/s of one logical operator over another. In the previous left hand versus right hand tooling example, for example, say you almost always run right hand tooling. Less than ten percent of your jobs require a left hand tool. In this case, you might elect to set up the test in such a way that the person using your custom macro must only specify the argument for tooling hand (H in our case) if they are using left hand tooling. If right hand tooling is used, the argument H can be omitted. Consider these commands:

```
IF [#11 EQ 0] GOTO 5
M03
GOTO 6
N5 M04
N6 . . .
```

Remember that if H is omitted from the call statement, the value of #11 will be vacant (not zero). In this case, the logical expression of the IF statement will be evaluated as false and the spindle will be started in the forward direction.

If this is your first exposure to conditional branching, do not be overly concerned with developing the best possible logical expression for every IF statement you write. With time and experience, setting up more appropriate conditional branching statements will get easier. Also, if you follow the suggestion given earlier to read a beginner’s book on BASIC, you will find much information about how to write effective conditional branching statements.
Conditional branching in Okuma’s user task 2
As mentioned earlier, if the conditional expression in the IF statement is evaluated as true, the command to the right of the statement is executed. With user task 2, you are limited to placing a GOTO statement to the right of the IF statement. Here is a specific example to show the syntax.

Say you wish to write a grooving custom macro for your turning center that can use right hand or left hand tooling. You wish to include a special local variable argument in the call statement, say TLHD for tool hand, to specify the tooling style. If TLHD is set to one, right hand tooling is being used. If TLHD is set to zero, left hand tooling is being used.

If, of course, right hand tooling is being used, the spindle must be running in a forward (M03) direction. If left hand tooling is being used, the spindle must be running in reverse (M04). Here is a portion of the user task program that makes the test and starts the spindle in the appropriate direction.

```
IF [TLHD EQ 1.] GOTO NFWD
M04
GOTO NCONT
NFWD M03
NCONT . . .
```

Not that brackets ([ ]) must be used to enclose the conditional expression. This conditional branching statement uses the equal to (EQ) logical operator, which makes it quite easy to evaluate the results of the test (true or false). If the programmer using this user task program includes an TLHD=1.0 in the CALL command, the control will evaluate the conditional branching statement as true. In this case, the GOTO statement next to the IF statement will be executed. In the line beginning with statement label NFWD, notice that the M03 will start the spindle in the forward direction (as required for right hand tools).

On the other hand, if the person using this parametric program sets the value of TLHD to zero (or anything else), the IF statement’s conditional expression will be evaluated as false. In this case, the command following the IF statement will be executed, starting the spindle in the reverse direction (as required for left hand tools).

Finally, note the need for the GOTO NCONT statement (we’re using NCONT to represent continue). If the conditional expression is evaluated as false, and the spindle starts in the reverse direction, we need the control to skip the other possibility. This GOTO statement is needed to keep the control from immediately starting the spindle in the forward direction. While the use of this GOTO statement does not make for very structured programming, just about any situation requiring conditional branching can be handled with the IF statement in user task 2.

Earlier we said that the six conditional operators allow a great deal of flexibility. This right hand versus left hand tooling test can be handled in several ways. Consider these commands that will render the same results:

```
 .
 .
```

IF [TLHD NE 1.] GOTO NREV
M03
GOTO NCONT
NREV M04
NCONT . . .
.
.
Or these:
.
.
IF [TLHD GT 0] GOTO NFWD
M04
GOTO NCONT
NFWD M03
NCONT . . .
.
.
Which is best? In some cases, it truly does not matter which logical operator you choose to test with, meaning you can set up the test in any way that works, and seems comfortable to you. However, there are many times when you will really want to think through your test to search for the hidden benefit/s of one logical operator over another.

For example, user task 2 provides a slight variation to the IF statement for testing whether a given local variable argument is currently active (whether it has a value). The command

IF TLHD GOTO NCONT

for example, simply tests to see if local variable argument TLHD is currently active (note that brackets are not required in this command). If it is, the control will execute the GOTO NCONT command. If it is not active (not specified in the CALL command), the control will proceed to the very next command in the user task program.

In the previous left hand versus right hand tooling example, say you almost always run right hand tooling. Only ten percent of your jobs require a left hand tool. In this case, you might elect to set up your test in such a way that the person using your user task program must only specify the argument for tooling hand (TLHD in our case) if they are using left hand tooling. If right hand tooling is used, the argument TLHD can be omitted. Consider these commands in the user task program:

.
.
IF TLHD GOTO NTST1
M03
GOTO NCONT
NTST1 IF[TLHD EQ 1] GOTO NFWD
M04
GOTO CONT
NFWD M03
M04
In the first IF statement, we determine whether local variable TLHD is active. If it is, we still have to determine how it is set, so we branch to the second test. If it is not active, however, we can assume that the person using this user task program is using right hand tooling (TLHD is left out of the CALL command), and start the spindle in the forward direction. Then we skip the unneeded test.

If this is your first exposure to conditional branching, do not be overly concerned with developing the best possible logical expression for every IF statement you write. With time and experience, setting up more appropriate conditional branching statements will get easier. Also, if you follow the suggestion given earlier to read a beginner’s book on BASIC, you will find information about how to write effective conditional branching statements.

**Conditional branching in Fadal’s macro**

As mentioned earlier, if the conditional branching statement is true, the command to the right of the statement is executed. Of the three versions of parametric programming addressed by this text Fadal’s macro provides the most flexibility regarding how the conditional expressions within the IF statement can be developed. With macro, you can even include multiple conditional expressions in one IF statement. Additionally, if the expression is evaluated as true, you are not limited to specifying a GOTO statement to the right of the IF statement. Any macro command can be given. In this introductory discussion, we will limit our presentation to show the most important functions of Fadal’s IF statement.

Let’s begin with the simplest form of the IF statement. Say you wish to write a grooving macro for your turning center that can take advantage of right hand or left hand tooling. You wish to include a special argument in the call statement, say V1, to represent the tooling style. If V1 is set to one, right hand tooling is being used. If V1 is set to zero, left hand tooling is being used. If, of course, right hand tooling is being used, the spindle must be running in a forward (M03) direction. If left hand tooling is being used, the spindle must be running in reverse (M04). Here is a portion of the macro that makes the test and starts the spindle in the appropriate direction.

```
#IF V1=1. THEN GOTO :FWD
M04
#GOTO :CONT
#:FWD
M03
#:CONT
```

This conditional branching statement uses the equal to (= or EQ) logical operator, which makes it quite easy to evaluate the results of the test (true or false). If the programmer using this macro includes an V1.0 in the call statement, the control will evaluate the conditional branching statement as true. In this case the GOTO statement next to the IF statement will be executed. In
the command after the statement label :FWD, notice that the M03 will start the spindle in the forward direction (as required for right hand tools).

On the other hand, if the person using this parametric program sets the value of V1 to zero (or anything else), the IF statement’s conditional expression will be evaluated as false. In this case, the command following the IF statement will be executed, starting the spindle in the reverse direction (as required for left hand tools).

Finally, note the need for the GOTO :CONT statement. If the conditional expression is evaluated as false, and the spindle starts in the reverse direction, we need the control to skip the other possibility. This GOTO statement is needed to keep the control from immediately starting the spindle in the forward direction. Note that this GOTO statement does not make for very structured programming. With the flexibility allowed by Fadal’s macro programming, there are other, more structured ways to develop this test.

Earlier we said that the six conditional operators allow a great deal of flexibility. This right hand versus left hand tooling test can be handled in several ways. Consider these commands that will render the same results:

```
#IF V1<>1. THEN GOTO :REV
M03
#GOTO :CONT
#:REV
M04
#:CONT
```

Or these:

```
#IF V1>0 THEN GOTO :FWD
M04
#GOTO :CONT
#:FWD
M03
#:CONT . . .
```

Which is best? In some cases, it truly does not matter which logical operator you choose to test with, meaning you can setup up the test in any way that works, and seems comfortable to you. However, there are many times when you will really want to think through your test to search for the hidden benefit/s of one logical operator over another. And with all the power allowed by Fadal’s macro, the choices are almost endless.

First of all, the macro IF statement allows you to include macro commands to the right of the IF statement (not just the GOTO statement). Consider these commands.
Notice that if the conditional expression is true, the equality statement will be read. (One of these statements must be true.) In either case, the value of R1 is set to the appropriate spindle on M code number.

Second, you are allowed to set up multiple conditional expressions with macro using AND, OR, and NOT logical operators. AND requires that all of the conditional expressions be true before the statement to the right of the IF statement will be executed. OR requires but one of the conditional expressions be true before the statement to the right of the IF statement will be executed. NOT simply reverses the conditional expression being performed. Though they have nothing to do with the right hand versus left hand tooling example, they should nicely illustrate what is possible with this very powerful command.

#IF (V1>V2) AND (V3=0) THEN GOTO :LOOP

Notice that both conditional expressions must be enclosed in parentheses. With AND, only if both conditional expressions are true will the GOTO statement be executed.

#IF (V1>V2) OR (V3=0) THEN GOTO :LOOP

If either of the conditional expressions is true, the GOTO statement will be executed.

#IF NOT(V1=0) THEN V2=33.25

Only if V1 is not equal to zero will the equality statement be executed.

#IF (V1>V2) AND (V3=0) AND (V4<>5) THEN GOTO :LOOP

Only if all three conditional expressions are true will the control execute the GOTO statement.

If this is your first exposure to conditional branching, do not be overly concerned with developing the best possible logical expression for every IF statement you write. With time and experience, setting up more appropriate conditional branching statements will get easier. Also, if you follow the suggestion given earlier to read a beginner’s book on BASIC, you will find information about how to write effective conditional branching statements.
Applications for conditional branching

As stated, conditional branching is used for many purposes. Here we will expose you to as many applications as possible. Generally speaking, all applications will fit into one of these six categories.

Argument flags

There will be many times when you will need to set up criteria within your parametric program for two or more possibilities. The user of your parametric program will then use a flag to tell the parametric program which of the possibilities they want at the present time. In essence, the programmer is flagging down the parametric program, specifying how it should behave with a flag.

In the previous introduction to conditional branching syntax, our example involved a tooling style flag. The flag argument just discussed specifies which hand of tooling is being used. The conditional branching statement/s in the parametric program examples sets up criteria for both possibilities, and the flag specifies which possibility is required at the present time.

Though it is not mandatory, it is a good idea to set up your flags in such a way that a setting of 1.0 represents a yes, on, or in other way positive answer to the condition. Use a setting of zero to represent a no, off, or in other way negative answer to the condition. This is but one of many times when argument flags are helpful. Here are some other possibilities.

To control coolant

Say for example, you are developing a user-created canned cycle and you know that sometimes the material being machined will be steel (which requires coolant) and other times the material will be cast iron (which does not require coolant). You could easily set a flag argument in the main program’s call statement to specify whether or not coolant is required. Consider these commands given in three different versions of parametric programming.

In custom macro B:

This example uses letter address argument C (represented by local variable #3) as the coolant flag argument. In the call statement (G65), if C is equal to 1.0, coolant is required. If C is equal to zero, coolant should be left off. Here is one way to handle this application with custom macro commands:

```
  IF[#3 EQ 1] GOTO 5
  M09
  GOTO 6
  N5 M08
  N6 . . .
```

In user task 2:

This example uses local variable argument CTFL (for coolant flow) as the coolant flag argument. In the CALL command, if CTFL is equal to 1.0, coolant is required. If CTFL is equal to zero, coolant should be left off. Here is one way to handle this application with user task commands:

```
IF[CTFL EQ 1] GOTO NCYES
M09
GOTO NCONT
NCYES M08
NCONT . . .
.
.

*In Fadal’s macro:*

This example uses V series variable argument V1 as the coolant flag argument. In the main program, if V1 is set equal to 1.0, coolant is required. If V1 is equal to zero, coolant should be left off. Here is one way to handle this application with macro commands:
.
.
#IF V1=1 THEN GOTO :COOLANTON
M09
#GOTO :CONT
#:COOLANTON
M08
#:CONT
.
.

*To climb or conventional mill*

You may want to set up criteria in you parametric program to allow both climb and conventional milling. If the flag argument is set to one, the parametric program should climb mill. If the argument is set to zero, the parametric program should conventional mill.

*In custom macro B:*

This example uses letter address argument M (represented by local variable #13) as the milling type flag argument. In the call statement (G65), if M is equal to 1.0, climb milling is required. If M is equal to zero, conventional milling is required. Here is one way to handle this application with custom macro commands:
.
.
IF[#13 EQ 1] GOTO 5
(Set up calculations and/or motions for conventional milling)
.
.
GOTO 6
N5 (Set up calculations and/or motions for climb milling
.
.
N6 . . .
.
.
In user task 2:
This example uses local variable argument MILL (for milling type) as the climb milling flag argument. In the CALL command, if MILL is equal to 1.0, climb milling is required. If MILL is equal to zero, conventional milling is required. Here is one way to handle this application with user task commands:

```
IF [MILL EQ 1] GOTO NCLIM
(Set up calculations and/or motions for conventional milling)
```

```
GOTO NCONT
NCLIM (Set up calculations and/or motions for climb milling)
```

```
NCONT . . .
```

In Fadal’s macro:
This example uses V series variable argument V2 as the milling type flag argument. In the main program, if V2 is set equal to 1.0, climb milling is required. If V2 is equal to zero, conventional milling is to be done. Here is one way to handle this application with macro commands:

```
#IF V2=1 THEN GOTO :CLIMMILL
(Set up calculations and/or motions for conventional milling)
```

```
#GOTO :CONT
#:CLIMMILL
(Set up calculations and/or motions for climb milling)
```

```
#:CONT
```

As stated, there are many applications for argument flags. Here we describe just a few more.

If developing a user-created canned cycle for circle milling, for example, you may wish to use a flag to specify whether there is a hole currently in the workpiece. If there is, the milling cutter can rapid to depth. If there is not, it must feed to depth.

If developing a peck drilling cycle, you may wish to include a flag argument to specify whether deep hole or chip breaking peck drilling is required. Truly, any time you wish set up for two or more possibilities is probably a good time to use an argument flag.

If developing a threading parametric program for a turning center, you may wish to use a flag to specify whether or not the thread ends in a thread relief. If it does, you will have the threading
tool pull straight out of the workpiece. If not, you will have the threading tool feed out at a 45 degree angle, minimizing the stress on the threading tool.

**Global flags**
As you know, most arguments (with the exception of Fadal’s V series variables) are local. They are forgotten as soon as the control finishes executing the parametric program. However, there may be times when you want a given flag to be global, available to any program at any time.

Say for example, you have a machine tool (machining center or turning center) that does not allow complete manual control of coolant during a program’s execution. If coolant is commanded from within a CNC program (with M08), the coolant will come on. While verifying the CNC program, this can be messy if the machine is not fully guarded, especially if the operator has no convenient way to get the coolant to stop flowing.

Even though you may have CNC machines that allow adequate manual control of coolant, this is a great application for a global flag. A special non-volatile (global) variable (a #500 series in custom macro, or a V series variable in user task 2 or macro) can be set by the setup person when a dry run is to be done. This can be done right at the machine through the display screen and keyboard. Whenever coolant is activated in the CNC program, a special test can be performed to determine the value of the global variable. If the variable is set to allow coolant to come on, the coolant command will be given. If not, the coolant command will be skipped.

---

**Example in custom macro B format:**

We will show but one example of this application. You should be able to easily convert it to any version of parametric programming. We’ll use permanent common variable #500 as the global coolant flag variable. The setup person will set #500 to 1.0 prior to doing a dry run to keep coolant from coming on. After the program is verified and production is to be run, they will set #500 back to zero. Consider these commands.

```
IF[#500 EQ 1.0] GOTO 5
M08
N5 . . .
```

If you think about it, this is very similar to using the optional block skip function. If, for example, the command

```
/M08
```

is programmed, the coolant will only come on if the optional block skip switch is turned off. However, most controls only allow one optional block skip function. If you need more, you’re out of luck. Global flags have no such limitation. And anything you can use the optional block skip to control, you can use a global flag to control as well. And you can have as many global flags as you need.

**Call statement error checking**
Even a perfectly written parametric program will fail if the person using it does not set arguments properly. If, for example, the user leaves out a very important argument, it can be difficult to predict just how the parametric program will behave. In similar fashion, if a basic
mistake in an argument setting is made, it can result in disaster. While a beginning parametric programmer will be most concerned with simply getting their first few programs to work, a well written parametric program will use tests to determine if certain mistakes have been made or other problems exist.

**Testing mandatory arguments**
Any argument that must be supplied to the parametric program is called a mandatory argument. Mandatory arguments are extremely important to the success of the parametric program. These are arguments that commonly change from one use of the parametric program to the next. We recommend that you test all mandatory arguments for the purpose of confirming that they are included in the call statement. If they are not, most versions of parametric programming give you the ability to actually generate an alarm and/or print a message that halts execution of the program. Here are some examples.

*Generating alarms in Fanuc’s custom macro B:*
In order to confirm that mandatory arguments are assigned with custom macro B, you must understand the meaning of vacancy. Remember from chapter two, if a letter address argument is not assigned in the call statement, its corresponding local variable is set to vacant (having no value). The custom macro B representation for vacancy is #0. You can use #0 to test for vacancy in the body of the custom macro.

To generate the actual alarm, a special system variable is used. While we devote an entire chapter to CNC features (including all kinds of system variables), you must be familiar with this particular system variable to understand our example.

The alarm generating system variable is #3000. If the control reads a #3000 command, an alarm will be generated. Like any program storage (PS) alarm, the alarm generated by #3000 will put the machine in a true alarm state. The operator must clear the alarm (by pressing reset) to clear the custom macro generated alarm. Here is the syntax for the #3000 system variable:

#3000 = 100 (ARGUMENT A MISSING IN CALL)

Again, if the control reads this command, it will generate an alarm. The value (100) is an alarm number of your choosing and can range from 100 to 255 (some control models allow other alarm numbers as well). The message in parentheses (all capitols) is the alarm documenting message. It must be 26 characters or less. If this command is read, the control will switch to the alarm page and this alarm message will appear on the control screen.

**MC-100 ARGUMENT A MISSING IN CALL**

With an understanding of vacant variables and alarm generation, you should understand this example of how to test whether mandatory arguments have been left out of the call statement. Here is the right side milling custom macro call statement shown in chapter two. For this example, we’ll say every argument in the call statement is a mandatory argument.

N025 G65 P1000 X0. Y0. Z0. W5.0 H3.0 T0.5 D1.0 F5.0 (Mill right side)

Remember X is represented by #24, Y by #25, Z by #26, W by #23, H by #11, T by #20, D by #7, and F by #9. One very explicit way of testing mandatory arguments is to set up a specific alarm generating command for each argument. At the beginning of custom macro O1000, you can have this series of commands to test that each local variable representing an argument has a value.
O1000 (Program number)
IF[#24 NE #0] GOTO 1
#3000 = 100 (X MISSING IN CALL)
N1 IF[#25 NE #0] GOTO 2
#3000 = 101 (Y MISSING IN CALL)
N2 IF[#26 NE #0] GOTO 3
#3000 = 102 (Z MISSING IN CALL)
N3 IF[#23 NE #0] GOTO 4
#3000 = 103 (W MISSING IN CALL)
N4 IF[#11 NE #0] GOTO 5
#3000 = 104 (H MISSING IN CALL)
N5 IF[#20 NE #0] GOTO 6
#3000 = 105 (T MISSING IN CALL)
N6 IF[#7 NE #0] GOTO 7
#3000 = 106 (D MISSING IN CALL)
N7 IF[#9 NE #0] GOTO 8
#3000 = 107 (F MISSING IN CALL)
N8 . . .
.
.
.
GOTO 99
N98 #3000 = 107 (DATA MISSING IN CALL)
N99 M99 (End of custom macro)

In this example, if any of the IF statements is true, the generic alarm DATA MISSING IN CALL is generated.
Generating alarms in Okuma’s user task 2:
Keep in mind that if you try to reference an inactive local variable with user task 2 (one that has not been assigned), an alarm will sound. This in itself may provide enough protection against forgetting to assign mandatory arguments. However, the machine may be right in the middle of a machining operation when it reads a command including an inactive local variable and sounds the alarm. It is wiser to test whether mandatory arguments are assigned before you allow machining to begin.

In order to confirm that mandatory arguments are assigned with user task two, you must understand the special conditional branching command that is used to determine whether or not a local variable is active. The syntax for this command is as follows:

IF DIA1 GOTO N1

In this command, local variable DIA1 is being tested. If it is active, the command GOTO N1 is executed. If it is not, the next command in the user task program is executed.

To generate the actual alarm, two special system variables are used. While we devote an entire chapter to CNC features (including all kinds of system variables), you must be familiar with these particular system variables to understand the example we give.

The alarm generating system variable is VDOUT. Actually, VDOUT is used for several purposes, including alarm generation. If the value set to VDOUT is 991, alarm type C is generated (which is appropriate for program related alarms). You are also allowed to specify an alarm code number within the VDOUT command.

The actual message printed on the display screen is commanded by another system variable called VUACM (VUACM stands for Variable for User Alarm CoMment). You are allowed up to sixteen characters in the message.

If the control reads these system variables, an alarm will be generated. Like any program storage alarm, the alarm generated will put the machine in a true alarm state. The operator must clear the alarm (by pressing reset) to clear the user task generated alarm. Here is the syntax for VDOUT and VUACM:

VUACM[1] = `DIA1 IS MISSING'
VDOUT[991] = 100

The number 1 in the brackets of the VUACM system variable command specifies the character number of the first character of the message (from one to sixteen). Note that the message must be enclosed in single quotation marks. Again, the 991 in the VDOUT system variable setting specifies alarm type C and the alarm number included in the message will be 100. Here is precisely how the message will appear on the display screen:

3202 Alarm C User reserve code 100 DIA1 IS MISSING

The 3202 in this message is Okuma’s alarm number for type C user reserved alarm codes.

With an understanding of the special conditional branching command for local variables and alarm generation, you should understand this example of how to test whether mandatory arguments have been left out of the call statement. Here is the right side milling custom macro call statement shown in chapter two. For this example, we’ll say every argument in this call statement is a mandatory argument.
Parametric Programming Lesson Four: Logic and Program Flow Control

N025 CALL O1000 XZER = 0  YZER = 0  ZZER = 0  WIDT = 5.0  HGHT = 3.0  THK = 0.5  TDIA = 1.0  FEED = 5.0 (Mill right side)

One very explicit way of testing mandatory arguments is to set up specific alarm generating commands for each argument. At the beginning of user task program O1000, you can have this series of commands to test that each local variable has a value.

O1000 (Program number)
IF XZER THEN GOTO N1
VUACM[1] = 'XZER IS MISSING'
VDOUT[991] = 100
N1 IF YZER THEN GOTO N2
VUACM[1] = 'YZER IS MISSING'
VDOUT[991] = 101
N2 IF ZZER THEN GOTO N3
VUACM[1] = 'ZZER IS MISSING'
VDOUT[991] = 102
N3 IF WIDT THEN GOTO N4
VUACM[1] = 'WIDT IS MISSING'
VDOUT[991] = 103
N4 IF HGHT THEN GOTO N5
VUACM[1] = 'HGHT IS MISSING'
VDOUT[991] = 104
N5 IF THK THEN GOTO N6
VUACM[1] = 'THK IS MISSING'
VDOUT[991] = 105
N6 IF TDIA THEN GOTO N7
VUACM[1] = 'TDIA IS MISSING'
VDOUT[991] = 106
N7 IF TDIA THEN GOTO 8
VUACM[1] = 'TDIA IS MISSING'
VDOUT[991] = 107
N8 . . .
.
.
.

Notice in each case, a specific alarm is generated for each missing argument. Some programmers feel this may be a little too specific. To minimize commands, you may elect to generate one generic alarm if any of the arguments is missing. Here is another example that shows how.

O1000 (Program number)
IF XZER THEN GOTO N1
GOTO NALM
N1 IF YZER THEN GOTO N2
GOTO NALM
N2 IF ZZER THEN GOTO N3
In this example, if any of the IF statements is false, the generic alarm DATA IS MISSING is generated.

*In Fadal’s macro:*

While macro does not have a specific command designed for the purpose of generating alarms, it does have a very flexible PRINT command that allows messages of all kinds to be printed on the display screen. This, combined with the ability to halt the machine, should suffice to let the operator know that something is wrong.

To actually halt the program, you could use a simple M00 program stop. However, the operator could simply press the cycle start button and the control would continue with the (incorrect) program. For this reason, we recommend using the INPUT command with some special logic that keeps the machined stopped if the operator tries to activate the cycle after getting an alarm message.

One other problem that presents itself with macro when considering the testing of whether mandatory arguments have been assigned is the fact that V series variables are global. They never lose their values as local variables do. There are two ways to handle this problem.

First, you can use the CLEAR function prior to setting V series variable arguments (usually in the main program). CLEAR lets you a set series of specified variables to zero. Here are some examples:

```
#CLEAR 'Set all V series variables to zero
#CLEAR V1 'Set V1 to zero
#CLEAR V1-V8 'Set V series variables V1 through V8 to zero
```

Once all V series variable arguments have been CLEARed in this manner, you and after you assign their values, it can be assumed that if the person using your macro program forgets to
assign an argument, its value will still be zero. In the macro program itself, you can simply test each argument to find out if it is zero. If it is, you can generate the alarm message and halt the machine.

Unfortunately, there may be those arguments that must sometimes be set to zero. In the right side milling example from chapter two, for example, we used V series variable V3 to represent the top surface location of the workpiece in Z. This value is commonly set to zero, meaning if we apply the test just discussed, we’ll get an incorrect alarm message.

For this reason, you may elect to use a second method of testing each V series variable argument. Though it involves more work, it is much more fail-safe. It involves determining a range of allowable setting for each argument. If any argument is out of its range, the alarm message will be given. You can actually combine these two methods to come up with the most appropriate test/s for each argument.

The trick is to come up with an appropriate range for each variable based upon your own machining criteria. Here is the range for each V series variable argument used in our example.

V1, V2, & V3 - minus 10.0 through plus 10.0 (Since these are absolute positions in X, Y, and Z, they have a broad range.)

V4 & V5 - plus 1.5 though plus 15.0 (The workpieces this company machines are never smaller than 1.5 in or bigger than 15.0 in.

V6 - plus 0.25 through plus 1.5.

V7 - plus 0.375 through plus 1.5

V8 - plus 0.5 through 15.0

While coming up with appropriate ranges takes a little time, this method is quite fail-safe and well worth the time it takes. Here is the beginning of the macro program that performs the mandatory argument testing. Note that an explicit alarm message is generated for each mandatory argument.

O1000 (Program number)
#IF (V1>-10.0) AND (V1<10.0) THEN GOTO :NXT1
#PRINT “V1 IS OUT OF RANGE”
#INPUT V99 ‘This command stops program
#GOTO :NOCONT ‘Confirm that program cannot continue
#:NXT1
#IF (V2>-10.0) AND (V2<10.0) THEN GOTO :NXT2
#PRINT “V2 IS OUT OF RANGE”
#INPUT V99 ‘This command stops program
#GOTO :NOCONT ‘Confirm that program cannot continue
#:NXT2
#IF (V3>-10.0) AND (V3<10.0) THEN GOTO :NXT3
#PRINT “V3 IS OUT OF RANGE”
#INPUT V99
#GOTO :NOCONT
#:NXT3
#IF (V4>1.5) AND (V4<15.0) THEN GOTO :NXT4
Parametric Programming Lesson Four: Logic and Program Flow Control

```plaintext
#PRINT "V4 IS OUT OF RANGE"
#INPUT V99
#GOTO :NOCONT
#:NXT4
#IF (V5>1.5) AND (V5<15.0) THEN GOTO :NXT5
#PRINT "V5 IS OUT OF RANGE"
#INPUT V99
#GOTO :NOCONT
#:NXT5
#IF (V6>0.25 AND (V6<1.5) THEN GOTO :NXT6
#PRINT "V6 IS OUT OF RANGE"
#INPUT V99
#GOTO :NOCONT
#:NXT6
#IF (V7>0.375) AND (V7<1.5) THEN GOTO :NXT7
#PRINT "V7 IS OUT OF RANGE"
#INPUT V99
#GOTO :NOCONT
#:NXT7
#IF (V8>0.5) AND (V8<15.0) THEN GOTO :NXT8
#PRINT "V8 IS OUT OF RANGE"
#INPUT V99
#GOTO :NOCONT
#:NXT8
.
.
.
.
.
.

#GOTO :CONT ‘Skip no continue segment
#:NOCONT ‘Eliminate possibility of continuing
#PRINT “YOU MUST RESET CONTROL!”
#INPUT V99
#GOTO :NOCONT
#:CONT
M99 (End of macro program)
```

Notice in each case, a specific alarm is generated for each mis-assigned argument. Some programmers feel this may be a little too specific. To minimize commands, you may elect to generate one generic alarm if any of the arguments is out of range. Here is another example that shows how.

O1000 (Program number)
#IF (V1<-10.0) OR (V1>10.0) THEN GOTO :ALARM
#IF (V2<-10.0) OR (V2>10.0) THEN GOTO :ALARM
#IF (V3<-10.0) OR (V3>10.0) THEN GOTO :ALARM
#IF (V4<1.5) OR (V4>15.0) THEN GOTO :ALARM
#IF (V5<1.5) OR (V5>15.00) THEN GOTO :ALARM
#IF (V6<0.25) OR (V6>1.5) THEN GOTO :ALARM
#IF (V7<0.375) OR (V7>1.5) THEN GOTO :ALARM
#IF (V8<0.5) OR (V8>15.00) THEN GOTO :ALARM

. . . .

#.GOTO :CONT ‘Skip alarm
#:ALARM ‘Alarm generation commands
#PRINT “AN ARGUMENT IS OUT OF RANGE”
#INPUT V99 ‘This command will halt the program
#GOTO :ALARM
#:CONT
M99 (End of macro program)

In this example, if any of the IF statements is true, the generic alarm message AN ARGUMENT IS OUT OF RANGE is printed and the program will halt. While it may be difficult for the operator to determine which of the arguments is incorrectly set, this technique minimizes the number or required commands.

Testing for erroneous argument settings
Leaving out a mandatory argument is but one mistake a person using your parametric program could make when specifying arguments. And while you can get carried away with error trapping routines, the lower the experience level of the people using your parametric program, the more important it is to make your program as fail-safe as possible. At the very least, you should guard against the most common mistakes. And as your parametric program is put into production, be on the lookout for mistakes your people are making. You can guard against almost any potential mistake with conditional branching. Here is one simple example.

Say you are developing a user-created canned cycle for multiple pass grooving on a turning center. As you are evaluating the error trapping functions of this application, it occurs to you that the grooving tool width must be equal to or smaller than the groove width. If it is not, you cannot predict how your parametric program will behave and you wish to generate an error trap for this potential mistake.

In custom macro B. For this example, T (#20) is the tool width and W (#23) is the groove width.

. . .

IF [#20 LE #23] GOTO 1
#3000 = 100 (TOOL TOO WIDE)
N1 . . .
In user task 2. For this example, TWID is the tool width and GWID is the groove width.

IF [TWID LE GWID] GOTO N1
VUACM[1] = `TOOL TOO WIDE'
VDOUT[991] = 100
N1 . . .

In Fadal’s macro. For this example, V1 is the tool width and V2 is the groove width.

#IF V1<=V2 THEN GOTO :CONT
#:REPEAT
#:PRINT “TOOL IS TOO WIDE”
#:INPUT V99 ‘Halts program activation
#:GOTO :REPEAT ‘Do not allow continuation
#:CONT . . .

What about execution time? As you have been reading this section on error trapping, you may have been questioning what effect error trapping will have on cycle time. Even with newer 32 bit controls, you will likely notice pauses in the program's execution while these logic commands are executed. With very low production quantities, the added cycle time may not be a problem. However, as production quantities grow, the effects of logic and arithmetic commands on cycle time can be devastating, and you will want to do something about it.

One way is to use the optional block skip function to skip any calculation and logic commands after the first workpiece is machined. The operator will leave the optional block skip switch turned off for the first workpiece and all calculations and error checking will be done once. After the first workpiece, the optional block skip switch will be turned on for the balance of the production run.

A better, more automatic way to minimize wasted execution time is to devise a special test (we’ll call a check sum test) to find out if any pertinent data has changed since the last execution of the program. If it has, all calculations and error checking will be done. If not, these commands are skipped. We will elaborate on both methods later in part one.

Setting default values

If you have ever worked with any kind of computer program application software (word processing, desk top publishing, drawing programs, etc.), you have likely seen pull down menus. Menus in software applications are commonly used to get data from the computer user. You have likely appreciated the fact that many of the actual entries on certain menus have default
settings, meaning you must only enter data for those entries that are not currently set correctly. Default settings dramatically reduce the amount of data you must enter.

By incorporating conditional branching techniques, you can assign default values for arguments in most versions of parametric programming. This can dramatically reduce the number of arguments a programmer must assign. To be able to perform this function, however, you must have a firm way to determine whether an argument has been assigned or not, as is possible in custom macro B and user task 2.

Say for example, you are developing a user-created canned cycle for pocket milling. As you look at your series of prints to evaluate the application, you notice that almost all workpieces require the pocket to be milled into the top (Z zero) surface of the workpiece. Just a very small percentage have the pocket at a different Z surface. Instead of forcing the person using your parametric program to specify the Z value for all programs, you want your program to assume the Z surface location is at zero unless the programmer specifies the Z surface argument.

*In custom macro B:* The letter address argument Z (#26) is used to pass the Z surface value to the custom macro.

\[
\text{IF} [\#26 \text{ NE } \#0] \text{ GOTO 1 (If Z is not vacant, skip default)} \\
\#26 = 0 \text{ (Set default value for Z to zero)} \\
\text{N1 . . . (Continue with program)} 
\]

*In user task 2:* Local variable argument ZSUR is used to pass the Z surface to the user task program.

\[
\text{IF ZSUR GOTO N1 (If ZSUR is active, skip default)} \\
\text{ZSUR} = 0 \text{ (Set default value for Z to zero)} \\
\text{N1 . . . (Continue with program)} 
\]

In both cases, if the programmer leaves out the argument assigning the Z surface position, the control will assume it to be zero.

If you think about it, setting default values for arguments is just the opposite of generating an alarm based on a missing mandatory argument. Instead of testing whether or not an argument has been assigned for the purpose of generating an alarm, you are testing whether it has been assigned for the purpose of setting its default value. We will call arguments that have default values set in the parametric program defaulted arguments. In well written parametric programs, all arguments supplied to the parametric program are mandatory arguments or defaulted arguments.
Other kinds of error checking
Using conditional branching to test for missing or erroneous arguments is but one kind of error checking. With most versions of parametric programming, you can do much more. In fact, most allow you to set up an error trap for just about any kind of problem. One way to determine whether you have need for this kind of error checking is to track the specific problems your operators have had running machines. What mistakes are they most prone to making?

Do your operators sometimes miss a decimal point when entering offset values (typing 1.0 instead of 0.1, for example)? You can test the values of all offsets to confirm they are in an acceptable range (more on how when we discuss system variables). If they are not, you can generate an alarm to halt the program.

Do your operators sometimes start the machine at an improper location? Possibly you have a large fixture on the table, over which the automatic tool changer cannot be activated without a collision. You can test the machine’s position (more on how when we discuss system variables) to determine if the ATC is above the fixture. If it is, you can generate an alarm.

On your multi-pallet machining centers, do your operators sometimes activate a program with the wrong pallet in position? Using a simple global flag that specifies the active pallet, you can test if the correct pallet is in position at the beginning of each program. If it is not, you can generate an alarm.

On your CNC turning centers, do your operators sometimes forget to size workpieces and update offsets at regular intervals? You could set up a counter that will cause the machine to go into an alarm state after a specified number of workpieces have been machined, telling the operator to perform sizing and tool maintenance for the appropriate tools.

These are but a few of the countless tests you can perform within a parametric program that will make your machines easier and safer to run. While not every detail of machine utilization can be tested, do not be too quick to give up on an important test (especially one related to a safety issue). With a little ingenuity, you may be surprised at just how much can be done!

Loops
The last application for conditional branching is looping. Looping is the act of repeating commands in a program a specified number of times. While some versions of parametric programming have a special command word for looping, most use conditional branching for this purpose. With custom macro B for example, there is a special WHILE statement for looping. However, it provides no special advantages over creating loops with conditional branching. For this reason, we exclusively stress the use of conditional branching to generate loops in this text. Looping is the topic of discussion in the next lesson.
**Conditional branching example**

Before going on to the final section of this chapter, we wish to show you an example program that includes many of the conditional branching capabilities just discussed. It is an extension of the user-created canned cycle for thread milling shown at the end of chapter three. We are going to give this thread milling canned cycle application much more power, make it safer to use, and in general, make it easier to work with. All thanks to the conditional branching! Figure 4.2 shows the same drawing used for the example at the end of chapter three. Here is how the criteria will change.

![Figure 4.2](image)

1) Set default for workpiece thickness to 0.75 - We’ll say well over half the workpieces to be machined are of this thickness. Note that setting default values is only possible in parametric programming languages like custom macro B and user task 2 that allow firm testing of whether an argument has been specified in the call statement or not, which is not possible in Fadal’s macro.

2) Set default for tool diameter to 1.0 in - We’ll say that almost all workpieces require a 1.0 diameter cutter. Again, this is only possible with parametric programming languages like custom macro B and user task 2 that allow firm testing of whether an argument has been specified in the call statement or not, which is not possible in Fadal’s macro.

3) All other arguments are mandatory arguments. Test them to confirm that they have been set.

4) Add logic for top-to-bottom or bottom-to-top milling. This will be done with a special flag (see documentation for each program for the flag name). This kind of logic can be helpful if milling blind and through holes. For through holes it is acceptable to mill from top to bottom. For blind holes, it is better to mill from bottom to top. Note that the actual direction of milling (cw versus ccw) must also change. If the flag is set to one (1.0) we have a blind hole, and milling must occur from bottom to top. If this flag is set to zero (0), milling must occur from top to bottom.
In custom macro B:

Figure 4.3 shows the letter address arguments related to this application. Note the addition of argument B, which controls the milling direction.

Figure 4.3

Here is a portion of the main program that calls the thread milling custom macro. Note in line N170, the T and Z arguments are left out, assuming the custom macro will use their preset values. Also notice that argument B is set to 1.0, meaning we have a blind hole to thread mill.

O0006 (Program number)
.
.
(Machine hole to get ready for thread milling)
.
.
N150 T04 M06 (Place thread milling cutter in spindle)
N155 G90 G54 S400 M03 T05 (Select absolute mode, coordinate system, start spindle, get next tool ready)
N160 G00 X2.0 Y2.0 (Rapid over to center of hole)
N165 G43 H04 Z.1 (Rapid up to workpiece in Z)
N170 G65 P1006 X2.0 Y2.0 Q0.125 M3.0 A1.0 F5.0 B1.0 (Call custom macro and mill thread)
N175 G91 G28 Z0 M19 (Return to tool change position)
N180 M01 (Optional stop)
.
.
.
Here is the modified custom macro that mills the thread:

```
O1006 (Program number)
(PRESET Z AND T)
IF[#26 NE #0] GOTO 5 (Test if Z is vacant)
#26 = 0.75 (Preset Z to 0.75)
N5 IF[#20 NE #0] GOTO 6 (Test if T is vacant)
#20 = 1.0 (Preset T to 1.0)
(TEST MANDATORY VARIABLES & GENERATE ALARM IF MISSING)
N6 IF[#24 EQ #0] GOTO 98 (If X is missing, generate alarm)
IF[#25 EQ #0] GOTO 98 (If Y is missing, generate alarm)
IF[#17 EQ #0] GOTO 98 (If Q is missing, generate alarm)
IF[#13 EQ #0] GOTO 98 (If M is missing, generate alarm)
IF[#1 EQ #0] GOTO 98 (If A is missing, generate alarm)
IF[#9 EQ #0] GOTO 98 (If F is missing, generate alarm)
IF[#2 EQ #0] GOTO 98 (If B is missing, generate alarm)
#20 = #20 / 2 (Set #20 to radius of tool instead of diameter)
#13 = #13 / 2 (Set #13 to radius of major diameter)
IF[#2 EQ 1.0] GOTO 20 (Test if blind hole)
(CALCULATIONS FOR TOP TO BOTTOM)
#101 = #24 - #1 + #20 (X position of tool center at approach position)
#102 = #24 + #1 - #20 (X position of tool center at retract position)
#103 = #25 + #13 - #20 (Y position of tool center at points 3 & 5)
#104 = #25 + #13 - #1 (Y position of tool center at points 2 & 6)
#105 = -[#26 + #17/4] (Z position of point 3)
#106 = #105 - #17/2 (Z position of point 4)
#107 = #106 - #17/2 (Z position of point 5)
#108 = #107 - #17/4 (Z position of point 6)
#109 = #1 - #20 (Radius of approach tool path)
#110 = #13 - #20 (Radius of tool path for milling thread)
#111 = #25 - #13 + #20 (Y position of tool center at point 4)
#112 = 2 (Set G code number for circular motion [G02])
GOTO 60
(CALCULATIONS FOR BOTTOM TO TOP)
N20 #101 = #24 + #1 - #20 (X position of tool center at approach position)
#102 = #24 - #1 + #20 (X position of tool center at retract position)
#103 = #25 + #13 - #20 (Y position of tool center at points 3 & 5)
#104 = #25 + #13 - #1 (Y position of tool center at points 2 & 6)
#105 = -[#26 - #17/4] (Z position of point 3)
#106 = #105 + #17/2 (Z position of point 4)
#107 = #106 + #17/2 (Z position of point 5)
#108 = #107 + #17/4 (Z position of point 2)
#109 = #1 - #20 (Radius of approach tool path)
#110 = #13 - #20 (Radius of tool path for milling thread)
#111 = #25 - #13 + #20 (Y position of tool center at point 4)
#112 = 3 (Set G code number for circular motion [G03])
```
Note the impact these relatively small changes have on this program. About fifteen commands are required for testing and setting default values. Additionally, the flag for blind versus through hole has a dramatic impact on this program. Notice how, based on the findings of the flag test, calculations are made and common variables are assigned for motions from bottom to top or from top to bottom. Even the G code number (2 or 3) is manipulated in this manner. Based upon how these variables are assigned, the same series of motion commands can be used to handle both possibilities.

In user task 2:

Figure 4.4 shows the local variable arguments related to this application. Note the addition of argument BHLE, which controls the milling direction.

Figure 4.4

Here is a portion of the main program that calls the thread milling user task program. Note in line N170, the TDIA and THK arguments are left out, assuming the custom macro will use their
preset values. Also notice that argument BHLE is set to 1.0, meaning we have a blind hole to thread mill.

O0006 (Program number)

(Machine hole to get ready for thread milling)

N145 G15 H01 (Select coordinate system)
N150 T04 M06 (Place thread milling cutter in spindle)
N155 G90 S400 M03 T05 (Select absolute mode, start spindle, get next tool ready)
N160 G00 X2.0 Y2.0 (Rapid over to center of hole)
N165 G56 H04 Z.1 (Rapid up to workpiece in Z)
N170 CALL O1006 XCTR=2.0 YCTR=2.0 PICH=0.125 MDIA=3.0 ARAD=1.0 FEED=5.0 BHLE=1.0 (Call user task program and mill thread)

Here is the modified user task program that mills the thread:

O1006 (Program number)
(PRESET THK AND TDIA)
IF THK GOTO N5 (Test if THK is active)
THK = 0.75 (Preset THK to 0.75)
N5 IF TDIA GOTO N6 (Test if TDIA is vacant)
TDIA = 1.0 (Preset TDIA to 1.0)
(TEST MANDATORY VARIABLES & GENERATE ALARM IF MISSING)
N6 IF XCTR GOTO N7 (If XCTR is active, GOTO N7)
GOTO N98 (If not, generate alarm)
N7 IF YCTR GOTO N8 (If YCTR is active, GOTO N8)
GOTO N98 (If not, generate alarm)
N8 IF PICH GOTO N9 (If PICH is active, GOTO N9)
GOTO N98 (If not, generate alarm)
N9 IF MAJR GOTO N10 (If MAJR is active, GOTO N10)
GOTO N98 (If not, generate alarm)
N10 IF ARAD GOTO N11 (If ARAD is active, GOTO N11)
GOTO N98 (If not, generate alarm)
N11 IF FEED GOTO N12 (If FEED is active, GOTO N12)
GOTO N98 (If not, generate alarm)
N12 IF BHLE GOTO N13 (If BHLE is active, GOTO N13)
GOTO N98 (If not, generate alarm)
N13 TRAD = TDIA / 2 (Set TRAD to radius of tool)
MRAD = MDIA / 2 (Set MRAD to radius of major diameter)
IF[BHLE EQ 1.0] GOTO N20 (Test if blind hole)
(CALCULATIONS FOR TOP TO BOTTOM)
Parametric Programming Lesson Four: Logic and Program Flow Control

XAPP = XCTR - ARAD + TRAD (X position of tool center at approach position)
XRET = XCTR + ARAD - TRAD (X position of tool center at retract position)
YP35 = YCTR + MRAD - TRAD (Y position of tool center at points 3 & 5)
YP26 = YCTR + MRAD - ARAD (Y position of tool center at points 2 & 6)
ZPT3 = -[THK + PICH/4] (Z position of point 3)
ZPT4 = ZPT3 - PICH/2 (Z position of point 4)
ZPT5 = ZPT4 - PICH/2 (Z position of point 5)
ZPT6 = ZPT5 - PICH/4 (Z position of point 6)
TAAR = ARAD - TRAD (Radius of approach tool path)
TMPR = MRAD - TRAD (Radius of tool path for milling thread)
YPT4 = YCTR - MRAD + TRAD (Y position of tool center at point 4)
GCD = 2 (Set G code number for circular motion [G02])
GOTO N60
(CALCULATIONS FOR BOTTOM TO TOP)
N20 XAPP = XCTR + ARAD - TRAD (X position of tool center at approach position)
XRET = XCTR - ARAD + TRAD (X position of tool center at retract position)
YP35 = YCTR + MRAD - TRAD (Y position of tool center at points 3 & 5)
YP26 = YCTR + MRAD - ARAD (Y position of tool center at points 2 & 6)
ZPT3 = -[THK - PICH/4] (Z position of point 3)
ZPT4 = ZPT3 + PICH/2 (Z position of point 4)
ZPT5 = ZPT4 + PICH/2 (Z position of point 5)
ZPT6 = ZPT5 + PICH/4 (Z position of point 6)
TAAR = ARAD - TRAD (Radius of approach tool path)
TMPR = MRAD - TRAD (Radius of tool path for milling thread)
YPT4 = YCTR - MRAD + TRAD (Y position of tool center at point 4)
GCD = 3 (Set G code number for circular motion [G03])
(MOTIONS)
N60 G00 X=XAPP Y=YP26 (Move in XY to approach point)
Z=-THK (Rapid to start of thread milling in Z)
G=GCD X=XCTR Y=YP35 Z=ZPT3 R=TAAR F=FEED (Approach to point 3)
Y=YPT4 Z=ZPT4 R=TMPR (Mill half way around circle to point 4)
Y=YP35 Z=ZPT5 R=TMPR (Mill the rest of the way to point 5)
X=XRET Y=YP26 Z=ZPT6 R=TAAR (Retract from thread)
G00 X=XCTR Y=YCTR (Move to hole center)
Z0.1 (Rapid out of hole)
GOTO N99
(ALARM IF DATA IS MISSING)
N98 VUACM[1] = `DATA MISSING`
VDOUT[991] = 100
N99 RTS (End of user task program)

Note the impact these relatively small changes have on this program. About twenty-five commands are required for testing and setting default values. Additionally, the flag for blind versus through hole has a dramatic impact on this program. Notice how, based on the findings of the flag test, calculations are made and local variables are assigned for motions from bottom to top or from top to bottom. Even the G code number (2 or 3) is manipulated in this manner.
Based upon how these variables are assigned, the same series of motion commands can be used to handle both possibilities.

In Fadal’s macro:
Figure 4.5 shows the V series variable arguments related to this application. Note the addition of argument V9, which controls the milling direction.

Since there is no way to firmly identify whether a variable has or has not been assigned (all variables are global in macro; they never loose their values), there is no way to set default values in a fail-safe manner. For this reason, we must bypass the setting of default values. For the same reason, we cannot test for the purpose of generating alarms if V series variables have not been assigned. However, we can (and will) test to ensure that all arguments fall in an appropriate range.

Here is a portion of the main program that calls the thread milling macro program. Note in line N168, that argument V9 is set to 1.0, meaning we have a blind hole to thread mill.

O0006 (Program number)
.
.
(Machine hole to get ready for thread milling)
.
.
N150 T04 M06 (Place thread milling cutter in spindle)
N155 G90 G54 S400 M03 T05 (Select absolute mode, coordinate system, start spindle, get next tool ready)
N160 G00 X2.0 Y2.0 (Rapid over to center of hole)
N165 G43 H04 Z.1 (Rapid up to workpiece in Z)
N170 #V1=2.0 'Assign argument V1
N175 #V2=2.0 'Assign argument V2
N180 #V3=0.75 'Assign argument V3
N185 #V4=0.125 'Assign argument V4
N190 #V5=1.0 'Assign argument V5
N195 #V6=3.0 'Assign argument V6
N200 #V7=1.0 'Assign argument V7
N205 #V8=5.0 'Assign argument V8
N210 #V9=1.0 'Assign argument V9
N215 M98 P1006 (Call macro program and mill thread)
N220 G91 G28 Z0 M19 (Return to tool change position)
N225 M01 (Optional stop)
.
.
.
Here is the modified macro program to mill the thread:

O1006 (Program number)
(TEST ARGUMENTS)
#IF (V1>-10.0) AND (V1<10.0) THEN GOTO :NXT1 'Test range of V1
#PRINT “V1 IS OUT OF RANGE” 'Print message
#INPUT V99 'This command halts program
#GOTO :NOCONT 'Confirm that program cannot continue
#:NXT1 'Continuation point
#IF (V2>-10.0) AND (V2<10.0) THEN GOTO :NXT2 'Test range of V2
#PRINT “V2 IS OUT OF RANGE”
#INPUT V99 'This command stops program
#GOTO :NOCONT 'Confirm that program cannot continue
#:NXT2
#IF (V3>0) AND (V3<2.5) THEN GOTO :NXT3 'Test range of V3
#PRINT “V3 IS OUT OF RANGE”
#INPUT V99
#GOTO :NOCONT
#:NXT3
#IF (V4>0) AND (V4<0.25) THEN GOTO :NXT4 'Test range of V4
#PRINT “V4 IS OUT OF RANGE”
#INPUT V99
#GOTO :NOCONT
#:NXT4
#IF (V5>0.5) AND (V5<2.0) THEN GOTO :NXT5 'Test range of V5
#PRINT “V5 IS OUT OF RANGE”
#INPUT V99
#GOTO :NOCONT
#:NXT5
#IF (V6>1.0 AND (V6<5.0) THEN GOTO :NXT6 'Test range of V6
#PRINT "V6 IS OUT OF RANGE"
#INPUT V99
#GOTO :NOCONT
#:NXT6
#IF (V7>0.375) AND (V7<V6) THEN GOTO :NXT7 'Test range of V7
#PRINT "V7 IS OUT OF RANGE"
#INPUT V99
#GOTO :NOCONT
#:NXT7
#IF (V8>0.5) AND (V8<30.0) THEN GOTO :NXT8 'Test range of V8
#PRINT "V8 IS OUT OF RANGE"
#INPUT V99
#GOTO :NOCONT
#:NXT8
#IF (V9=0) AND (V9>1.0) THEN GOTO :NXT9 'Test range of V9
#PRINT "V9 IS OUT OF RANGE"
#INPUT V99
#GOTO :NOCONT
#:NXT9
#IF V9 < 1.0 THEN GOTO :BLINDHOLE 'Test if blind hole
(MOVEMENTS FOR THROUGH HOLE)
#V5=#V5/2 'Set V5 to the radius of the cutter instead of its diameter
#R9=V3 'Initial Z position
#R1=V1 'Set R1 to X center of hole
#R2=V2+V6/2-V7 'Set R2 to Y center of approach radius
G00 X+R1 Y+R2 (Rapid to center of approach radius, point 1) Z-R9 (Rapid to beginning Z position)
#R9=R9+V4/4 (Calculate Z position at end of approach radius)
#R1=V1-V7+V5 'Set R1 to X position approach radius
#R2=V8*5 'Set R2 to fast moving feedrate
G01 X+R1 F+R2 (Fast feed to beginning of approach radius, point 2)
#R1=V2-V6/2-V5 'Set R2 to Y end point of approach radius
#R3=V7-V5 'Set R3 to tool path radius for circular motion
#R4=V8 'Set R4 to milling feedrate
G02 X+R1 Y+R2 Z-R9 R+R3 F+R4 (Circular approach to beginning of thread, point 3)
#R9=R9+V4/2 'Calculate Z position half way around thread
#R1=V2-V6/2+V5 'Set R1 to Y position half way around thread
#R2=V6/2-V5 'Calculate radius for circular move
G02 Y+R1 Z-R9 R+R2 (Mill half way around thread, point 4)
#R9=R9+V4/2 'Calculate Z position at end of thread
#R1=V2+V6/2-V5 'Set R1 to Y end of circular move
G02 Y+R1 Z-R9 R+R2 (Mill balance of thread, point 5)
#R9=R9+V4/4 'Calculate Z position at end of retract radius
#R1=V1+V7-V5 'Calculate X end point of retract radius
#R2=V2+V6/2-V7 'Calculate Y end point of retract radius
Parametric Programming Lesson Four: Logic and Program Flow Control

#R3=V7-V5 ‘Calculate radius of retract
G02 X+R1 Y+R2 Z-R9 R+R3 (Retract to point 6)
#R1=V1 ‘Store X center position in R1
G00 X+R1 (Return to center of approach radius)
Z0.1 (Rapid to clearance position above workpiece)
GOTO :CONT ‘Skip blind hole possibility
#.BLINDHOLE
(MOVEMENTS FOR BLIND HOLE)
#V5=#V5/2 ‘Set V5 to the radius of the cutter instead of its diameter
#R9=V3 ‘Initial Z position
#R1=V1 ‘Set R1 to X center of hole
#R2=V2+V6/2-V7 ‘Set R2 to Y center of approach radius
G00 X+R1 Y+R2 (Rapid to center of approach radius, point 1)
Z-R9 (Rapid to beginning Z position)
#R9=R9-V4/4 (Calculate Z position at end of approach radius)
#R1=V1+V7-V5 ‘Set R1 to X position approach radius
#R2=V8*5 ‘Set R2 to fast moving feedrate
G01 X+R1 F+R2 (Fast feed to beginning of approach radius, point 2)
#R1=V1 ‘Set R1 to X center of hole
#R2=V2+V6/2-V5 ‘Set R2 to Y end point of approach radius
#R3=V7-V5 ‘Set R3 to tool path radius for circular motion
#R4=V8 ‘Set R4 to milling feedrate
G03 X+R1 Y+R2 Z-R9 R+R3 F+R4 (Circular approach to beginning of thread, point 3)
#R9=R9-V4/2 ‘Calculate Z position half way around thread
#R1=V2-V6/2+V5 ‘Set R1 to Y position half way around thread
#R2=V6/2-V5 ‘Calculate radius for circular move
G03 Y+R1 Z-R9 R+R2 (Mill half way around thread, point 4)
#R9=R9-V4/2 ‘Calculate Z position at end of thread
#R1=V2+V6/2-V5 ‘Set R1 to Y end of circular move
G03 Y+R1 Z-R9 R+R2 (Mill balance of thread, point 5)
#R9=R9-V4/4 ‘Calculate Z position at end of retract radius
#R1=V1-V7+V5 ‘Calculate X end point of retract radius
#R2=V2+V6/2-V7 ‘Calculate Y end point of retract radius
#R3=V7-V5 ‘Calculate radius of retract
G03 X+R1 Y+R2 Z-R9 R+R3 (Retract to point 6)
#R1=V1 ‘Store X center position in R1
G00 X+R1 (Return to center of approach radius)
Z0.1 (Rapid to clearance position above workpiece)
GOTO :CONT ‘Skip no continue segment
#:NOCONT ‘Eliminate possibility of continuing
#PRINT “YOU MUST RESET CONTROL!”
#INPUT V99 ‘Halt program
#GOTO :NOCONT ‘Go back to message
#:CONT ‘Continuation point
M99 (End of macro program)