

Fanuc CNC Custom Macros

Fanuc CNC Custom Macros

Programming Resources for *Fanuc Custom Macro B* Users

Peter Smid

Industrial Press, Inc.
200 Madison Avenue
New York, NY 10016-4078, USA
<http://www.industrialpress.com>

Industrial Press Inc.
200 Madison Avenue
New York, New York 10016-4078

Copyright © 2005. Printed in the United States of America

All rights reserved

This book or parts thereof may not be reproduced,
stored in a retrieval system or transmitted in any form without the
permission of the publisher

Cover Design: Janet Romano
Managing Editor: John Carleo

10 9 8 7 6 5 4 3 2 1

*To
Joan, Michael and Michelle*

Thank you for everything

Acknowledgments

In this first edition of the *Fanuc Custom Macros*, the author would like to express much deserved and sincere thanks and repeated appreciation to Peter Eigler for never turning away from a challenge and always being able to provide working solution to a problem. Many thanks are also reserved for Eugene Chishow, who can rightly claim many macros bearing his name.

My family has always provided a great support to me - thanks to you all.

In the handbook, there are references to several manufacturers, software developers and some trade names. It is only fair to acknowledge their names:

- ❑ **FANUC and CUSTOM MACRO or USER MACRO or MACRO B are registered trademarks of Fujitsu-Fanuc, Japan**
- ❑ **GE FANUC is a registered trademark of GE Fanuc Automation, Inc., Charlottesville, VA, USA**
- ❑ **MASTERCAM is the registered trademark of CNC Software Inc., Tolland, CT, USA**
- ❑ **WINDOWS is a registered trademarks of Microsoft, Inc., Redmond, WA, USA**
- ❑ **FADAL, OKUMA, MAKINO, YASNAC, MITSUBISHI, MELDAS, MAZAK, MAZATROL - are also trade names that appear in the handbook**

About the Author

Peter Smid, the author of the best-selling *CNC Programming Handbook* and a number of other publications, is a professional consultant, educator and speaker, with many years of practical, hands-on experience, in the industrial and educational fields. During his career, he has gathered an extensive experience with CNC and CAD/CAM applications on all levels. He consults to manufacturing industry and educational institutions on practical use of Computerized Numerical Control technology, CNC part programming, CAD/CAM, advanced machining, tooling, setup, and many other related fields. His comprehensive industrial background in CNC programming, machining and company oriented training has assisted several hundred companies to benefit from his wide-ranging knowledge.

Mr. Smid's long time association with advanced manufacturing companies and CNC machinery vendors, as well as his affiliation with a number of Community and Technical College industrial technology programs and machine shop skills training, have enabled him to broaden his professional and consulting skills in the areas of CNC and CAD/CAM training, computer applications and needs analysis, software evaluation, system benchmarking, programming, hardware selection, software customization, and operations management.

Over the years, Mr. Smid has developed and delivered hundreds of customized educational programs to thousands of instructors and students at colleges and universities across the United States, Canada and Europe, as well as to a large number of manufacturing companies and private sector organizations and individuals.

He has actively participated in many industrial trade shows, conferences, workshops and various seminars, including submission of papers, delivering presentations and a number of speaking engagements to professional organizations. He is also the author of many magazine columns and articles as well as in-house publications on the subject of CNC and CAD/CAM. During his many years as a professional trainer in the CNC industrial and educational field, he has developed tens of thousands of pages of high quality training materials.

Peter Smid is currently completing a new book *CNC Programming Techniques*, scheduled for release by Industrial Press, Inc., in the Spring of 2005.

The author welcomes comments, suggestions and other input from educators,
students and industrial users

You can e-mail him from the **Fanuc CNC Custom Macros** page at www.industrialpress.com

Disclaimer

Industrial Press (the *Publisher*) and Peter Smid (the *Author*) provide this publication and the included CD files in the form of 'as is', without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The author may make improvements or changes in this publication and/or the included CD files, or in the program examples used in this publication, at any time and without notice.

Neither the Publisher nor the *Author* assumes any responsibility for any error that may appear in the publication or the CD files.

Use of names of companies and products in this publication does not reflect an endorsement by either those companies or by the *Publisher* or the *Author*.

Preface

For more than twenty five years, control systems for CNC machines have been designed with many more features than necessary just to process a manually written part program. A whole group of conversational programming systems has been offered by several control manufacturers for many years. From the original Fanuc FAPT system to the modern on-machine programming available from companies such as Mazak and their Mazatrol system, this method has proven very successful for CNC lathes, and to a smaller degree, to CNC milling systems as well.

While most conversational programming offers a great number of benefits for a variety of suitable parts, it does not offer the flexibility of most CAD/CAM systems, within the area of so called CAM programming. The majority of CAM systems offer off-machine CNC programming using graphical toolpath generation combined with many features to produce programs of excellent quality. For this reason, they have become the most popular method of part program generation.

With all the benefits and some inevitable disadvantages, the traditional programming methods offer thousands of CNC users the choice between the three common options - manual programming, on-machine conversational type of programming, or a CAM software, in order to develop the part program. Program development using macros offer an additional method, complementing - *not competing with* - the other methods.

The purpose of this handbook is not to compare between the part program development methods, but to bring attention to the type of part program development that has not been used as often as it should - macros (known as *Custom Macros* or *User Macros*).

CNC program development using macros does not replace any other programming method. In fact, it belongs to the category of manual programming - and as an extension - it offers much higher level of sophistication. This handbook is all about macros - what they are and how to develop them and how to use them. It offers many do's and don'ts, and it covers all the popular Fanuc control systems. Although there are other controls offering macros, this handbook covers Fanuc macros exclusively. Macros for different controls share the common approach and mainly differ in their syntax. Learning macros for one control will become a benefit when learning a macro for a different control. Macros present an extremely wide and rich field of programming tools that a professional CNC programmer or CNC technical person can explore in great depth.

This handbook has been designed as a training and reference text that can be used in a production environment - not as a production oriented text that can be used for training. In no way the handbook is intended to replace manuals supplied with the machine tool or the control system - they are vital part of the learning process.

Peter Smid, January 2005

TABLE OF CONTENTS

1 - FANUC MACROS	1
<hr/>	
General Introduction	1
Review of G-codes, M-codes and Subprograms	2
System Parameters	2
Data Setting	2
Custom Macros	3
Probing Applications	3
Overall View	3
Macro Programming	4
Macro Option Check	4
What is a Macro Programming?	4
Typical Features	5
Main Program with Macro Features	5
Using Macros	6
Groups of Similar Parts	7
Offset Control	8
Custom Fixed Cycles	8
Nonstandard Tool Motions	8
Special G-codes and M-codes	8
Alarm and Message Generation	8
Replacing Control Options	9
Hiding and Protecting Macro Programs	9
Probing and Gauging	9
Various Shortcuts and Utilities	9
Skills Requirements	10
2 - BASIC PROGRAM CODES	11
<hr/>	
Preparatory Commands	11
Default Settings	11
Modal Values	12
Programming Format	12
Miscellaneous Functions	12
Programming Format	12
M-codes with a Motion	12
Custom M-codes	12
Reference Tables	13
G-codes for Milling	13
Three-Digit G-codes	15
M-codes for Milling	16
G-codes for Turning	16
M-codes for Turning	19
Standard Program Codes	20
Optional Program Codes	20

3 - REVIEW OF SUBPROGRAMS	21
Subprogram Example - Mill	21
Rules of Subprograms	25
Subprogram Repetition	27
Subprogram Nesting	27
Subprogram Documentation	29
Subprograms vs. Macros.	30
Unique Features.	30
CNC Lathe Applications	31
Subprogram Development	32
4 - SYSTEM PARAMETERS	33
What are Parameters ?	33
Saving Parameters	34
Backing Up Parameters	34
Parameter Identification	35
Numbering of Parameters	35
Parameter Classification	35
Parameters Grouping	36
Parameter Display Screen	37
Parameter Data Types.	37
Bit-Type Data Type	37
Relationship of Parameters	40
Byte Data Type	41
Word Data Type.	42
2-Word Data Type	42
Axis Data Type	43
Important Observations.	44
Binary Numbers.	45
Setting and Changing Parameters	46
Protection of Parameters	46
Battery Backup	46
Changing Parameters	47
System Defaults	47
Default Values Settings	48
5 - DATA SETTING	49
Input of Offsets	49
Data Setting Command	50
Coordinate Mode	50
Absolute Mode	50
Incremental Mode	50
Work Offsets.	51
Standard Work Offset Input.	51
Additional Work Offset Input	52
External Work Offset Input	52

Offset Memory Types - Milling	53
Geometry Offset	53
Wear Offset	53
Which Offset to Update?	54
Memory Type A.	55
Memory Type B.	56
Memory Type C.	57
Memory Type and Macros	57
Offset Memory Types - Turning.	58
Adjusting Offset Values	59
Absolute Mode	59
Incremental Mode	59
Tool Offset Program Entry	60
L-Address	60
G10 Offset Data Settings - Milling Examples	61
Valid Input Range	62
Lathe Offsets.	62
P-Offset Number	63
Tip Number Q	63
G10 Offset Data Settings - Turning Examples	64
Data Setting Check in MDI	65
Programmable Parameter Entry	65
Modal G10 Command	66
N-address in G10 L50 Mode	67
P-address in G10 L50 Mode	67
R-address in G10 L50 Mode	67
Program Portability	67
Setting Machine Axes to Zero	70
Bit Type Parameter Example	70
Differences Between Control Models	72
Effect of Block Numbers	72
Block Skip	72

6 - MACRO STRUCTURE

Basic Tools	73
Variables.	74
Functions and Constants	74
Logical Functions	74
Defining and Calling Macros.	75
Macro Definition	75
Macro Call	75
Arguments	77
Visual Representation	78
Macro Program Numbers	79
Macro Program Protection	79
Setting Definitions	79
Program Numbers - Range O0001 to O7999	80
Program Numbers - Range O8000 to O8999	80
Program Numbers - Range O9000 to O9999	81
Program Numbers - Range O9000 to O9049	82
Difference Between the O8000 and O9000 Program Numbers	82

7 - CONCEPT OF VARIABLES	83
Types of Macro Variables	83
Variables in Macros	84
Definition of Variables	84
Calculator Analogy	84
Variable Data	84
Variable Declaration	85
Real Numbers and Integers.	85
Variable as an Expression	86
Usage of Variables	86
Decimal Point Usage	87
Metric and English Units	88
Least Increment	88
Positive and Negative Variables	89
Syntax Errors	90
Restrictions	90
Custom Machine Features	92
8 - ASSIGNING VARIABLES	93
Local Variables	93
Defining Variables	93
Clearing Local Variables	94
Assigning Local Variables	94
Assignment List 1 - Method 1	94
Assignment List 2 - Method 2	95
Missing Addresses	97
Disallowed Addresses	98
Simple and Modal Macro Calls	98
Selection of Variables	99
Main Program and Local Variables	101
Local Variables and Nesting Levels	105
Common Variables	106
Volatile and Nonvolatile Memory Groups.	106
Input Range of Variables	107
Out-of-Range Values	107
Calculator Analogy.	107
Set Variable Name Function SETVN	108
Protection of Common Variables	108
9 - MACRO FUNCTIONS	109
Function Groups	109
Definition of Variables Revisited	110
Referencing Variables	110
Vacant or Empty Variables	111
Axis Motion Commands and Null Variables	111
Terminology	112
Arithmetic Functions.	113
Nesting.	113
Arithmetic Operations and Vacant Variables	114
Division by Zero	115

Trigonometric Functions	116
Conversion to Decimal Degrees	116
Available Functions	116
Rounding Functions	117
Rounding to a Fixed Number of Decimal Places	119
FUP and FIX Functions	121
Miscellaneous Functions	122
SQRT and ABS Functions	122
LN, EXP and ADP Functions	124
Logical Functions	124
Boolean Functions	124
Binary Numbers Functions	125
Boolean and Binary Examples.	125
Conversion Functions	126
Evaluation of Functions - Special Test	126
Order of Function Evaluation	128
Approach to Practical Applications	129
Using Local Variables	129
Using Common Variables	133
Speeds and Feeds Calculation	134

10 - SYSTEM VARIABLES 137

Identifying System Variables	137
System Variables Groups	138
Read and Write Variables	138
Displaying System Variables	138
System Variables for Fanuc Series 0	139
Fanuc Model 0 Compared to Other Models	140
System Variables for Fanuc Series 10/11/15	140
System Variables for Fanuc Series 16/18/21	141
Organization of System Variables.	144
Resetting Program Zero.	145

11 - TOOL OFFSET VARIABLES 147

System Variables and Tool Offsets	147
Tool Offset Memory Groups	148
Tool Offset Memory - Type A	148
Tool Offset Memory - Type B	149
Tool Offset Memory - Type C	149
Tool Offset Variables - Fanuc 0 Controls	150
Milling Control FS-0M	150
Turning Control - FS-0T	151
Tool Offset Variables - FS 10/11/15/16/18/21 for Milling.	152
Assignments for 200 Offsets or Less - Memory Type A	152
Assignments for 200 Offsets or Less - Memory Type B	153
Assignments for 200 Offsets or Less - Memory Type C	154
Assignments for More than 200 Offsets - Memory Type A	155
Assignments for More Than 200 Offsets - Memory Type B	156
Assignments for More than 200 Offsets - Memory Type C	157

Tool Offset Variables - FS 10/11/15/16/18/21 for Turning	158
Tool Setting	158
Assignments for 64 Offsets or Less - Memory Type A	159
Assignments for 64 Offsets or Less - Memory Type B	160
Assignments for More than 64 Offsets - Memory Type A	161
Assignments for More than 64 Offsets - Memory Type B	162
12 - MODAL DATA	163
System Variables for Modal Commands	163
Fanuc 0/16/18/21 Modal Information	163
Fanuc 10/11/15 Modal Information	163
Preceding and Executing Blocks	164
Modal G-codes	164
Fanuc 0/16/18/21	165
Fanuc 10/11/15.	166
Saving and Restoring Data	167
Saving Modal Data.	167
Restoring Modal Data	168
Other Modal Functions.	168
Fanuc 0/16/18/21	169
Fanuc 10/11/15.	170
13 - BRANCHES AND LOOPS	171
Decision Making in Macros.	171
IF Function	172
Conditional Branching	172
Unconditional Branching	173
IF-THEN Option	174
Single Conditional Expressions	175
Combined Conditional Expressions	176
Concept of Loops	177
Single Process	177
Multiple Process	177
WHILE Loop Structure	179
Single Level Nesting Loop.	179
Double Level Loop.	180
Triple Level Loop	180
General Considerations	181
Restrictions of the WHILE Loop	181
Conditional Expressions and Null Variables	182
Formula Based Macro - Sine Curve	184
Clearing Common Variables	186
14 - ALARMS AND TIMERS	187
Alarms in Macros	187
Alarm Number	187
Alarm Message	187
Alarm Format	188
Embedding Alarm in a Macro	188
Resetting an Alarm	190
Message Variable - Warning, Not an Alarm	190

Table of Contents	xvii
Timers in Macros	191
Time Information	191
Timing an Event	191
Dwell as a Macro	192
15 - AXIS POSITION DATA	193
Axis Position Terms	193
Position Information	194
16 - AUTO MODE OPERATIONS	195
Controlling Automatic Operations	195
Single Block Control	195
M-S-T Functions Control	196
Feedhold, Feedrate, and Exact Check Control	197
Example of Special Tapping Operation	198
Systems Settings	199
Mirror Image Status Check	199
Interpreting System Variable #3007	200
Controlling the Number of Machined Parts.	202
17 - EDITING MACROS	203
Editing Units	203
Program Comments	203
Abbreviations of Macro Functions	204
18 - PARAMETRIC PROGRAMMING	205
What is a Parametric Programming ?	205
Variable Data	205
Benefits of Parametric Programming	206
When to Program Parametrically	206
Planned Approach to Macro Development.	207
19 - FAMILY OF SIMILAR PARTS	209
Macro Development in Depth - Location Pin	209
Drawing Evaluation	210
Objective of the Macro	210
Part Setup, Tooling and Machining Method	210
Drawing Sketch	211
Standard Program	211
Identify Variable Data	212
Creating Arguments	215
Using Variables	216
Writing the Macro	217
Final Version	218
Macro Improvements	220
20 - MACROS FOR MACHINING	221
Angular Hole Pattern - Version 1	221
Variable Data for Angular Hole Pattern	223
Angular Hole Pattern - Version 2	224

Frame Hole Pattern	226
Variable Data for Frame Hole Pattern	227
Bolt Hole Circle Pattern	229
Variable Data for Bolt Hole Circle Pattern	231
Arc Hole Pattern	233
Variable Data for Arc Hole Pattern	234
Circular Pocket Roughing	236
Variable Data for Circular Pocket Roughing	237
Amount of Stock Left	239
Circular Pocket Finishing	240
Variable Data for Circular Pocket Finishing	241
Slot Machining Macro	244
Variable Data for Slot Machining	245
Circular Groove with Multiple Depth	247
From Subprograms to Macros	248
Macro Version Development	249
Rectangular Pocket Finishing	251
21 - CUSTOM CYCLES	255
Special Cycles	255
Options Available	256
G-code Macro Call	256
M-functions Macro Call	258
G13 Circle Cutting Cycle.	260
Macro Call - Normal	262
Macro Call - as a Special Cycle	262
Detailed Evaluation of Offset Value	264
Counterboring Application	266
22 - EXTERNAL OUTPUT	267
Port Open and Port Close Commands	267
Data Output Functions	268
BPRNT Function Description	268
DPRNT Function Description	269
Parameter Settings - Fanuc 10/11/12/15.	269
Metric vs. Inch Format.	270
Parameter Settings - Fanuc 16/18/21.	271
Structure of External Output Functions	272
Output Examples	273
Blank Output Line	274
Columns Formatting	274
DPRNT Practical Examples	274
Date	274
Time	274
Work Offset	274

23 - PROBING WITH MACROS	275
What is Probing ?	275
Touch Probes	276
Probing Technology Today	276
Probe Calibration	277
Feedrate and Probing Accuracy	277
Probing Devices on CNC Machines	278
In-Process Gauging Benefits	278
Types of Probes	278
Probe Size	279
Probe Selection Criteria	279
Machined Part	279
Control System Capabilities	280
Expected Tolerances	280
Additional and Optional Features	280
Associated Costs	280
CNC Machine Probe Technology	280
Optical Signal Transmission	281
Inductive Signal Transmission	282
Radio Signal Transmission	282
In-Process Gauging	282
Features to be Measured	283
Center Location Measurement	284
Measuring External or Internal Width	286
Measuring Depth	287
Measuring External Diameter	287
Measuring Internal Diameter	287
Measuring Angles	288
Changing of Set Values	288
Calibration Devices	288
Calibrating device - Type 1	288
Calibrating device - Type 2	288
Checking the Calibration Device	289
Centering Macro Example	289
Probe Length Calibration	291
Skip Command G31	293
24 - ADDITIONAL RESOURCES	295
Limitations During Macro Execution	295
Single Block Setting	295
Block Number Search	295
Block Skip Function	295
MDI Operation	296
Edit Mode	296
Control Reset	296
Feedhold Switch	296

Knowledge for Macro Programming	297
General Skills	297
Manual Programming Experience.	298
Math Applications	298
Setup Practices	298
Machining Practices	298
Control and Machine Operation	298
Complementary Resources.	299
Industrial Press, Inc.	299
Internet.	299
Practical Programming Approach.	299
Macro Programming Tips	300
25 - MACRO COURSE OUTLINE	301

Macro Course Outline	301
Closing Comments	306
Index	307
27 - WHAT'S ON THE CD-ROM ?	313

1

FANUC MACROS

This handbook has been developed as a resource material for CNC programming at its highest level, using Fanuc and compatible Computer Numerical Control systems (CNC systems). Techniques described in the handbook are still part of the manual programming process, in the sense that no external CAD/CAM software or hardware is required. Although the main topic of this handbook is application of *Fanuc Custom Macros* in CNC programming (known as *Fanuc Custom Macro B*), several related topics have been added, mainly for coherence and comparison, but mainly as a refresher of some basic CNC programming skills required as a prerequisite.

The subject matters deal with several major topics, and the handbook is organized in the suggested order of learning. More experienced users can start at any section within the handbook:

- General Introduction**
- Review of G-codes and M-codes**
- Review of subprograms**
- System parameters**
- Data setting**
- Custom macros**
- Probing applications**

Numerous examples and sample programs are used throughout the handbook. Their purpose is to serve not only as practical applications of the techniques explained, but - for many of them - as the basis for ready-to-run macro programs.

Although all the topics covered in the handbook are critical, they are discussed here for the single purpose of learning one subject, commonly known as *Custom Macros*, *User Macros*, *Fanuc Macros*, *Macro B*, or - just *Macros*. Several non-Fanuc controls also offer their version of macros, for example Fadal and Okuma, but only Fanuc macros are covered in this handbook.

General Introduction

This is the general introduction to the subject of macros. Its purpose is to make you aware of what macros are, what related subjects are important, and to identify several other helpful items to get you started in this important, exiting and often underestimated, field of CNC programming.

Knowledge of macros is becoming more and more essential, as companies large and small look towards more efficient ways of CNC program development, particularly for certain type of parts. Although CAD/CAM programming systems have become very popular and are on the rise, they do not and cannot always replace macro programming, for various reasons. Macros often serve as a special solution to special requirements.

The following brief descriptions provide some ideas of major subjects covered in the handbook.

Review of G-codes, M-codes and Subprograms

It may appear that any discussion of preparatory commands (G-codes), and miscellaneous functions (M-codes), as well as subprogram topics is too basic and should not be included in a handbook on custom macro programming. Before you get interested in macros or actually work with them, there are certain prerequisites of knowledge and experience. CNC program structure is composed of a number of features, such as positioning data (machine axes), cutting data (speed and feeds), offsets, comments, cycles, etc. Developing a CNC program requires knowledge and discipline. Before getting into the field of macro programming, you should be well experienced in the usage of the preparatory commands - the *G-codes*, and the miscellaneous functions - the *M-codes*. You should also understand the structure and development of subprograms, including multiple level nesting applications. These topics form the corner stone of macro development. They are included in this handbook strictly for a review, as a refresher material and for reference only, in a somewhat condensed form.

System Parameters

In a little play on words, you may say that *parameters control the control*. That means, parameters are part of the control system and make it function in a harmonious way with the machine tool. Good knowledge of only a few parameters are necessary for an average CNC user, and not all control parameters are necessary for macro program development. Parameters do, however, form the environment in which macros are developed and operate. Both terms *parameters* and *system parameters* are frequently used throughout the handbook, and so is the term *parametric programming*. Although both terms are covered in this handbook and they are linguistically connected, they *do not* share the same meaning in CNC programming.

Parameters or *System Parameters* are settings of the control system. They can be thought of as various registers that store machine and program data. On the other hand, *Parametric Programming* is a method of programming often known as the *programming the family of similar parts*.

Data Setting

In order for a CNC machine to execute a program correctly, it requires more than just the part setup on the machine. We are dealing with technology called *numerical* control, therefore with interpretation of *numbers* - we need many settings of data in numeric form. The three offset groups required for a complete machine setup are the largest part of this topic. They are:

- Offsets relating to work position ... work offsets (G54, G55, G56, G57, G58, G59)
- Offsets relating to tool length ... tool length offsets (G43, G44, G49)
- Offsets relating to tool radius ... tool radius offsets (G40, G41, G42)

Various offset data can be set through the program with the **G10** preparatory command, without using any macros at all. Offset data can also be changed *through* the macros for even more flexibility - in this case, the system parameters and data setting techniques are important prerequisites. Keep in mind that various control systems in the Fanuc family may require a slightly different format of programming, even when the final results are the same. It is important to know each control in the shop as well as the machine tool using this control. Do not assume that a macro developed for one control model will work with another control model.

Throughout the handbook, there will be reminders that any changes to the control system will affect its operation. It is extremely important that all changes to parameters or any stored data are done by qualified and authorized professionals !

Custom Macros

The subject of *Custom Macros* - or the *User Macros* - is the major topic of this handbook. In the order of suggested learning, you will discover many valuable programming methods and techniques, procedures, tips, and suggestions of how to develop a macro program from the scratch. Many practical examples are included to help you to start or to be used as a reference later.

With increasing work experience, you will be able to make macros faster, macros more powerful and more efficient. You will be able to develop macros for various machine tool activities you would have never thought of before. Macros may take some time to develop properly, but it is a time well invested.

Probing Applications

Macros are the backbone for any automatic *probing* and *gauging* on the CNC machine (and many other automated procedures). Although some users may want to distinguish between the two terms, we will use them interchangeably for the same purpose. Probing allows an *in-process* inspection of the machined part, including offsets corrections and many other adjustments. Probing has no real equivalent in standard programming.

Macro programming for probing devices requires more than just the mandatory availability of macro option in the control system - it also requires additional hardware installed on the machine tool, plus the necessary software interfaces. Many manufacturers of probing devices may offer their own generic macros, but you still need to develop custom probing macros for specific purposes, as they relate to your work.

Overall View

Except the brief refresher topics, the last few topics may not be the easiest subject to learn for the beginner, but all topics are very logical in their nature. Also, they are presented here in the order you should learn them. What you need as a good background is the knowledge of the basic CNC programming, the syntax of word address format, understanding of the program structure (flow), and at least the basic operation of the CNC machine tool on a production level. Detailed knowledge (as opposed to a superficial one) of the G-codes and the M-codes is imperative, and for macro development, the knowledge of subprograms and their structure, including nesting levels, is equally imperative.

To learn the advanced programming methods efficiently, you almost *must* have an access to a CNC control that has the required macro option installed. Learning how to develop macros is like learning how to swim. A great number of books will describe the many techniques of swimming, but you learn the most only when you learn in the water. There are no shortcuts to success - you have to *understand* what is going on, and you have to try it - and that may take a little time.

Macro Programming

In the handbook, the short term *macro* will be used, referring to an optional feature of Fanuc control systems called the *Custom Macro* or the *User Macro*. Typically, a letter *B* is added to the description, such as *Custom Macro B* or *User Macro B*. That is just an indication of a level more advanced from the original version. Virtually all Fanuc controls now offer the optional *Macro B* version, even if it is not specified directly in the control description. Other control systems offer a similar method of flexible programming and the logic and general approach you learn here for the Fanuc control can be adapted for control systems other than Fanuc (Fadal, Okuma, etc.). As the name suggests, *custom macros* are available to the CNC user, to serve as an additional tool for unique and specific applications of a machine tool.

Keep in mind that macros are *optional* feature of the control system, and unless your company has purchased this option, you will have no access to them. However, it is easy to have them activated by a qualified Fanuc technician on request, and upon a payment, of course.

Macro Option Check

Do I have the macro option installed? This is a common question of many users of CNC equipment. Even if you have absolutely no idea about macros at this time, it is very important to know whether the control system you are using has the macro option installed before you write a macro program. There is very a simple way to find out, and no special program is necessary to do that.

Set the control to *MDI mode (Manual Data Input)* and type in the following command:

```
#101 = 1
```

When you press the *Cycle Start* button, one of two possibilities will happen. If the control system accepts the command without issuing an alarm or error condition, it means the macro option *is* installed. On the other hand, if the control system returns an alarm (error) message (usually indicating a syntax error or address not found), the macro option is *not* installed on that control. Make sure to enter the data as shown in the example, including the # symbol that identifies the number that follows as a *variable* number 101 with an assigned value of 1. Other commands can be entered as well, but the one shown is a harmless way to make the macro availability check.

What is a Macro Programming?

In a few words, macro programming is a part programming technique that combines standard CNC programming methods with additional control features for more power and flexibility. Macro for all CNC systems is the closest method of programming to a true language based programming, using the CNC system directly. Generally available high-level languages, such as *C++™* or *Visual Basic™*, and many of their forms and derivatives, are used by computer software professionals everywhere to develop sophisticated software for various computer applications. Fanuc macro is not a language itself by a strict definition - it is a special purpose software used for CNC machines only. However, a CNC macro program uses many features found in high level computer languages.

Typical Features

Typical features found in Fanuc macros are:

- Arithmetic and algebraic calculations
- Trigonometric calculations
- Variable data storage
- Logical operations
- Branching
- Looping
- Error detection
- Alarm generation
- Input and Output
- ... and many other features

A macro program resembles a standard CNC program to a certain extent, but includes many features not found in regular programming. Essentially, a macro program is structured as a regular subprogram. It is stored under its own program number (O-), and it is called by the main program or by another macro, using a G-code (typically **G65**). However, in a very simple form, macro features can be used in a single program as well, without the macro call command.

Main Program with Macro Features

Here is a simple example of a normal part program that cuts four slots (roughing cuts only):

```

N1 G21
N2 G17 G40 G80
N3 G90 G00 G54 X25.0 Y30.0 S1200 M03
N4 G43 Z2.0 H01 M08
N5 G01 Z-5.0 F100.0
N6 Y80.0 F200.0           (SLOT 1)
N7 G00 Z2.0
N8 X36.0
N9 G01 Z-5.0 F100.0
N10 Y30.0 F200.0         (SLOT 2)
N11 G00 Z2.0
N12 X47.0
N13 G01 Z-5.0 F100.0
N14 Y80.0 F200.0         (SLOT 3)
N15 G00 Z2.0
N16 X58.0
N17 G01 Z-5.0 F100.0
N18 Y30.0 F200.0         (SLOT 4)
N19 G00 Z2.0 M09
N20 G28 Z2.0 M05
N21 M30
%
```

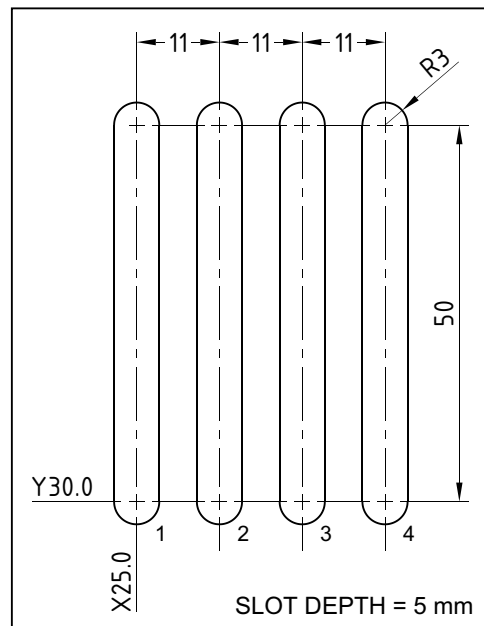


Figure 1

Simple job to illustrate feedrate as a macro function

Note the repetitive use of the two feedrates:
F100.0 for plunging and F200.0 for slot cutting.

In the program, each feedrate appears once per slot. The more slots, the more programmed feedrate. If you need to change one or both feedrates in the program, the change has to be done for each slot individually. With many slots, this could be a time consuming task. A macro feature used in the program greatly simplifies the job. The key is to define the two feedrates as variables, at the beginning of the program. A variable definition is preceded by the # symbol and used instead of the 'real' value:

```

N1 G21
N2 G17 G40 G80
N3 #1=100.0 (PLUNGING FEEDRATE)
N4 #2=200.0 (CUTTING FEEDRATE)
N5 G90 G00 G54 X25.0 Y30.0 S1200 M03
N6 G43 Z2.0 H01 M08
N7 G01 Z-5.0 F#1
N8 Y80.0 F#2 (SLOT 1)
N9 G00 Z2.0
N10 X36.0
N11 G01 Z-5.0 F#1
N12 Y30.0 F#2 (SLOT 2)
N13 G00 Z2.0
N14 X47.0
N15 G01 Z-5.0 F#1
N16 Y80.0 F#2 (SLOT 3)
N17 G00 Z2.0
N18 X58.0
N19 G01 Z-5.0 F#1
N20 Y30.0 F#2 (SLOT 4)
N21 G00 Z2.0 M09
N22 G28 Z2.0 M05
N23 M30
%
```

By changing the variable #1, *all plunging* feedrates will be changed automatically, and by changing the variable #2, *all cutting* feedrates will be changed automatically. This is just a small example of the power of macros - it should provide at least basic appreciation of their benefits.

Another example of this technique will be described in *Chapter 8*, with additional details.

Using Macros

Knowing what macros are and what they are capable of will help you to use them in an effective and profitable way. There are many areas where macros can be an essential part of CNC programming, whether a manual method or a CAD/CAM method is used. If used wisely, macros, as a programming tool can coexist with any CAD/CAM system - *but they do not replace it* - they serve as just another method to achieve a certain goal. Macros can be used for general CNC machining, but also more and more they are used to control modern manufacturing equipment and its many automated features, such as material handling, tool breakage inspection, special cycles, etc.

Certain control features are normally accessed by a skilled CNC operator, typically done during the job setup. Characteristic in this category are the common offsets (for work position, tool length and radius). The CNC operator makes the appropriate measurements and enters the offsets into the control system. Macros allow the programmer to automate both the measurement *and* the input of offset values. Special *measuring* equipment (also known as *probing* or *gauging* equipment) is required for such activities, but many others do not require any additional equipment.

Another very common application of macros is a group of parts that are similar in some respects, for example, their basic shape. All parts in such a group can use a single master program (in the form of a macro) that can be called with different data input values for each group member.

The following list highlights some of the most common applications of macros:

- Groups of similar parts
- Offset control
- Custom fixed cycles
- Nonstandard tool motions
- Special G-codes and M-codes
- Alarm and message generation
- Replacing control options
- Hiding and protecting macro programs
- Probing and gauging
- Various shortcuts and utilities

Groups of Similar Parts

In the old days (1970's) of off-machine language programming (using programming languages such as *Compact II*[™], *Split*[™], *APT*[™], *ADAPT*[™], and many others), it was common to program parts that are similar in shape (meaning '*similar, but not exactly the same*'), and also similar in the general machining process. For example, a bolt circle of equally spaced holes is a common machining operation for many machine shops and without macros, each bolt circle has to be calculated individually for the *XY* coordinates and the relating cutting data. Yet, the same formulas are used over and over again, each time a new bolt circle is required. Once there is a suitable bolt circle macro available, it can be called repeatedly by supplying only the data that change. The calculations of the bolt circle hole positions and the appropriate machining will be determined by this data. Other hole patterns, such as arc, row, grid and frame patterns are also good candidates for a macro (see the *Parametric Programming* section of the handbook). Developing a macro will usually require more time than an individual part program, but this time is well invested. Once a macro exists and is used, it eliminates any other programming - all that is needed is the change of the parameters (for example, speeds, feedrates, dimensions, depths, etc.).

Many other repetitive machining tasks and similar toolpaths can also benefit from proper application of macro programs. The group of similar parts is often called the *family of similar parts* or, more generically, *parametric programming*, where the user supplies parameters (changeable values) to an existing macro. However, a true parametric programming is not restricted to the similar parts only. Many machining operations, such as pocketing, are very common. Rectangular and circular pockets, with straight and tapered walls can benefit from a macro application. Do not confuse parametric programming with system parameters, also described in the handbook.

Offset Control

Programmable offsets used for CNC work are of three kinds:

- Work offset** - G54 to G59 commands (standard) + G54 . 1 P1 to G54 . 1 P48 commands (option)
- Tool length offset** - using the G43 or G44 commands, usually with an H-address
- Cutter radius offset** - using the G41 or G42 commands, usually with a D-address

In addition, there are different *versions* of the available offsets, such as *geometry* and *wear*, external or common offset, and three different types of control memory. Using macros, offsets can be entered, cleared, checked, adjusted and manipulated through the program, without interference from the CNC machine operator. Some offset changes require a probing device, others can be changed arbitrarily, depending on the work situation. Knowledge of offsets and the way they interact with the CNC program is absolutely essential for most macro applications.

Custom Fixed Cycles

Fixed cycles have been part of programming for a long time. They are used every day and they work very well. Occasionally, there is a need for a special cycle that will do something unusual, less common, yet important for a certain application. For example, there are fixed cycles that feed-in and feed-out of a hole. The feedrate is always the same for both directions. You may develop a new cycle, where the cutting feedrate will change in one direction only. Another example is a cycle that can peck drill with a decreasing peck depth of each subsequent cut. G83 and G73 cycles cannot do that. Many special cycles can be developed, and not just for machining holes.

Nonstandard Tool Motions

The three common tool motions, *rapid*, *linear* and *circular*, are suitable for most CNC jobs. Other motion types are often needed, yet impossible to achieve without special control software. They include curves based on mathematical formulas, such as a straight helix, tapering helix, parabola, hyperbola, sine curve, etc. Custom macros can be developed to accurately simulate such a toolpath, using a mathematical formula and resulting in a very complex tool motion calculation.

Special G-codes and M-codes

Manufacturers of special equipment may want to control certain operations by a *G-code* or an *M-code*. These will be nonstandard functions and can be developed with a macro. Later in the handbook, there is a circular pocket cycle macro using a call G13 - a new Fanuc G-code that can be used in everyday programming (already available on some non-Fanuc controls).

Alarm and Message Generation

Macros can also be used to detect a number of erroneous conditions (faults) and allow the part programmer to communicate the fault to the CNC operator in the form of an *alarm* or an *error* condition. Alarm (error) can have its own number and a brief description of the cause. Instead of alarms indicating the possible cause of a fault, instructional messages to the machine operator can be generated, describing what is happening or what activity needs to be done.

Replacing Control Options

Fanuc controls offer many special features that are only available as an option. Typical optional features are *Scaling Function*, *Coordinate System Rotation*, *Polar Coordinates*, *Additional Offsets*, etc. With macros, you can develop a program performing exactly the same function what these options offer, without the extra cost, which often is quite high.

Hiding and Protecting Macro Programs

There will be macro programs that you create, than use them over and over again. After all, that is the main reason for their development in the first place. If something goes wrong with the macro in the control system, for example, an accidental deletion, its loss would cause a significant problem to the production process at that point. Macros can be *protected* within the control software, so they cannot be accidentally deleted or changed without forced additional steps. Macros with sensitive contents can also be *hidden* from the directory display.

Probing and Gauging

Probing and gauging are a very important areas of using custom macros. A section on probing, with examples, is also a significant part of this handbook. Using probes and similar devices, custom macros can be utilized as an 'on-machine inspection station', using a method commonly known as *in-process gauging*. Measured values (actual values) can be compared with the expected values (drawing values), and various offsets can be automatically adjusted.

Custom macros used in probing can be applied to different types of drawing specifications, such as corner locations, center locations, angles, diameters, depths, widths, automatic centering, boring measurements, and many others.

Various Shortcuts and Utilities

Many small utility programs can also be written into a macro form, to make the programming job (and the operator's job) easier and safer. Utilities are usually small programs that do not actually machine a part, but are used for certain common operations. Typical applications may include safe tool call, table or pallet indexing, management of tool life for unmanned operations, detection of worn out or broken tools, redefining the program zero (origin) for uneven castings, boring jaws on a lathe, counting the parts already machined, automating part-off operations on a CNC lathe, automatic tool changing, and many other possibilities. All these programs share a common feature - they are very effective shortcuts for repeatable activities that occur in CNC programming.

The field of macro applications is very extensive. In addition to the many possibilities already described, macros can be used to check spindle speeds, feedrates, and tool numbers, they can control the I/O (input and output) of data, check and register the active G-code command in a particular group, and disable the feedhold feature, spindle and feedrate overrides and a single block operation. The applications are virtually endless, and depend not only on your particular needs but your skills as well.

Skills Requirements

Like any human endeavor, a successful custom macro programming requires certain skills, and not only in the field of machine shop work and related technology. When a macro programming is compared to a conventional CNC programming, all the skills required for the standard CNC programming will be needed, plus many others.

For the standard CNC programming, the programmer in a typical machine shop environment must understand all the items already mentioned, plus many new ones. Work experience is a definite asset, and all issues relating to skills can be summed up into the following areas:

- CNC machines and controls - operation and programming**
- Machining skills - how to machine a part**
- Basic mathematics skills - calculations, formulas**
- Program structure development skills - convenience and consistency**
- Offsets and Compensations applications skills - various adjustments**
- Fixed cycles in depth - how they work, in detail**
- Subprograms in depth, including multi nesting applications**
- System parameters, their purpose and functions**

To become a successful CNC custom macro programmer, a good working knowledge of a high level language is not absolutely necessary, but it is of a primary benefit. Languages mentioned earlier, such as various forms of *Visual Basic*[™], *C++*[™], the old *Pascal*[™], *Delphi*[™], *Lisp*[™] - including *AutoLISP*[™] from the makers of Autocad (the most popular CAD software for personal computers), and many others, offer excellent platform for learning.

One skill that is very important to understanding macros, is the deep knowledge of the *Preparatory Commands* (G-codes) and the *Miscellaneous Functions* (M-codes) in a part program (both are reviewed in this handbook). Keep in mind that *G-codes* are reasonably consistent between different Fanuc control models (and compatible controls), although a lot of them are special options. *M-codes* will vary a great deal between different Fanuc controlled machined tools, depending on the machine tool manufacturer. The chapter covering this topic lists the M-codes for reference only, and knowledge of all M-codes (and G-codes) on a particular CNC machine tool is absolutely essential. Another very important background skill, also reviewed here, is the knowledge of *subprograms* in depth. ***Subprograms are the first logical step into the macro development.***

Finally, a list of skills that are not exclusively confined to the CNC programming area but that are also very useful are, for example:

- Problem solving skills**
- Analytical skills**
- Logical thinking**
- Organizational skills**
- Patience (a lot of it)**

There are no simple solutions, but one advice may be useful - *always try to work towards a particular goal*. Establish a specific practical project, evaluate it, then work toward its 100% completion. Test, test again and then, test one more time. *Just do not give up!*

2

BASIC PROGRAM CODES

In CNC part programming, all address codes (letters in the program) are equally important and should not be underestimated, but there are two addresses that stand out as very important to macro programming. The *G-codes* and the *M-codes* are the major feature of every CNC program and the key to successful macro program development. In macros, they are used the same way as in standard programs, but also gain additional attributes. It is important to know these codes for every CNC machine and control system equipped with macros. In this chapter, all these important codes in a CNC program are reviewed. The chapter also includes a typical reference list for both types of the basic G and M programming codes.

Preparatory Commands

The G-codes in a CNC program are called the *preparatory commands*. The purpose of the preparatory commands is to *prepare* - or to *preset* - the control system to a certain mode of operation. For example, the CNC program can use *English* or *metric* units of measurement. The control system has to be *preset* to that mode before any dimensional value appears in the program. Normally, we use the **G20** command to select English units (inches) and the **G21** command to select metric units (millimeters). Other common examples of preparatory commands include the *type* of tool motion (**G00**, **G01**, **G02**, **G03**), *absolute* and *incremental* mode (**G90**, **G91**), and many others. The key to programming any G-codes is that the desired mode has to be selected *before* it is used. If you do not select the mode in the program, the control system has many default settings.

Default Settings

When the power to the control system is turned on, there was no program that could influence the internal settings of the control system. That means the built-in settings, the *default settings*, will take effect. Although most controls have the same defaults, it is important that you know them for *each* control individually, because they can be permanently changed by the vendor or the user. Typical default settings are identified with the ♦ (diamond) symbol in the reference tables. Defaults of several G-codes may be set by the vendor or the user, and may differ from the typical list. They are **G00/G01**, **G17/G18/G19**, **G90/G91**, and several others. To set a different default, you have to use system parameters, described in a separate chapter.

Be very cautious when making permanent changes to the control system settings !

An incorrect parameter setting can permanently damage
the control system and / or the machine tool !

Modal Values

Preparatory commands can be either *modal* or *non-modal*. Modal commands are programmed only once, and remain in their selected mode until changed or canceled by another command. Most - but not all - of the G-codes are modal. The typical non-modal commands are **G04**, **G09** and the machine zero return commands **G27–G30**. These are often called 'one shot' commands. Some preparatory commands remain in effect, *even if the power of the control has been turned off*. The most typical is the **G20** or **G21** command that selects the units mode.

Programming Format

Any number of G-codes from different groups (see table) can be programmed in a single block, providing they do not conflict with each other. If a conflicting G-code appears in the same block, *the one specified later will be effective - Fanuc system will not cause an error condition!*

Miscellaneous Functions

The M-codes in a CNC program are called the *Miscellaneous Functions*. Most of them control the hardware functions of the machine tool, for example **M08** turns the coolant pump motor on and **M09** turns it off. They also control the program flow, for example **M01** is an optional program stop, **M30** is the program end, etc. Many M-codes are designed by the manufacturer of the machine tool, and are unique to that machine only - *they are nonstandard* and can only be found in the machine tool manual.

Programming Format

Normally, only one M-code can be used in any block. Some latest controls (Fanuc 16/18/21) now allow up to three M-codes in a single block, providing they do not conflict with each other. If a conflicting M-code appears in the same block or too many M-codes are in the block, the system will return an error condition.

M-codes with a Motion

If an M-code is programmed together with an axis motion, it is important to know when the M-code takes effect. For example, **M03** will start *simultaneously* with the motion, but **M05** will take effect after the motion has been *completed*. Every machine tool manual should include information on how the M-codes behave when programmed with a motion.

Custom M-codes

The M-codes are the *least standard* from one control or machine to another, even from the same manufacturer. Only a small number of M-codes can claim to be standard. Machine tool manufacturers assign an M-code to any unique option the machine tool may have. Some manufacturers may assign hundreds of unique M-codes for a particularly complex machine tool. Always know the special M-codes for every machine you work with.

Reference Tables

The following tables list the typical *preparatory commands* (G-codes) and *miscellaneous functions* (M-codes). Both milling and turning applications are included and the typical default preparatory commands are marked with the ♦ symbol (subject to change by the vendor or the user).

In case of discrepancy between the included tables and the CNC machine tool manual, always use the codes listed by the machine tool manufacturer

G-codes for Milling

The following table is a fairly comprehensive reference listing of all standard as well as the most common G-codes (preparatory commands) used for CNC milling programs (CNC milling machines and machining centers). All inter-dependent G-codes belong to the same group number and are modal, unless they belong to the Group 00, which identifies all non-modal commands:

G-code	Group	Description
G00	01	Rapid positioning mode
G01	01	Linear interpolation mode ♦
G02	01	Circular interpolation mode - clockwise direction
G03	01	Circular interpolation mode - counterclockwise direction
G04	00	Dwell function (programmed as a separate block)
G07	00	Hypothetical axis interpolation
G09	00	Exact stop check for one block
G10	00	Data setting mode (programmable data input)
G11	00	Data setting mode cancel
G15	17	Polar coordinate mode cancel ♦
G16	17	Polar coordinate mode
G17	02	XY plane designation ♦
G18	02	ZX plane designation
G19	02	YZ plane designation
G20	06	English units of input
G21	06	Metric units of input
G22	04	Stored stroke check <i>ON</i> ♦
G23	04	Stored stroke check <i>OFF</i>
G25	25	Spindle speed fluctuation detection <i>ON</i>
G26	25	Spindle speed fluctuation detection <i>OFF</i> ♦

G-code	Group	Description
G27	00	Machine zero return position check
G28	00	Machine zero return - primary reference point
G29	00	Return from machine zero
G30	00	Machine zero return - secondary reference point
G31	00	Skip function
G33	01	Threading function
G37	00	Tool length automatic measurement
G40	07	Cutter radius compensation mode cancel ◆
G41	07	Cutter radius compensation mode to the left
G42	07	Cutter radius compensation mode to the right
G43	08	Tool length offset - positive
G44	08	Tool length offset - negative
G45	00	Position compensation - single increase
G46	00	Position compensation - single decrease
G47	00	Position compensation - double increase
G48	00	Position compensation - double decrease
G49	08	Tool length offset cancel ◆
G50	11	Scaling function mode cancel ◆
G51	11	Scaling function mode
G52	00	Local coordinate system setting
G53	00	Machine coordinate system setting
G54	14	Work coordinate offset 1 ◆
G54.1	14	Additional work coordinate offset
G55	14	Work coordinate offset 2
G56	14	Work coordinate offset 3
G57	14	Work coordinate offset 4
G58	14	Work coordinate offset 5
G59	14	Work coordinate offset 6
G60	00	Single direction positioning
G61	15	Exact stop mode
G62	15	Automatic corner override mode
G63	15	Tapping mode
G64	15	Cutting mode ◆
G65	00	Custom macro call

G-code	Group	Description
G66	12	Custom macro modal call
G67	12	Custom macro modal call cancel
G68	16	Coordinate system rotation mode
G69	16	Coordinate system rotation mode cancel
G73	09	High speed deep hole drilling cycle (peck drilling)
G74	09	Left hand tapping cycle
G76	09	Precision boring cycle
G80	09	Fixed cycle cancel
G81	09	Drilling cycle
G82	09	Spot drilling cycle
G83	09	Deep hole drilling cycle (peck drilling)
G84	09	Right hand tapping cycle
G85	09	Boring cycle
G86	09	Boring cycle
G87	09	Back boring cycle
G88	09	Boring cycle
G89	09	Boring cycle
G90	03	Absolute input of motion values
G91	03	Incremental input of motion values
G92	00	Coordinate system setting (tool position register)
G94	05	Feedrate per minute - <i>in/min</i> or <i>mm/min</i>
G95	05	Feedrate per revolution - <i>in/rev</i> or <i>mm/rev</i>
G98	10	Retract motion to the initial level in a fixed cycle
G99	10	Retract motion to R-level in a fixed cycle

One G-code in a modal group replaces another G-code from the same group

Three-Digit G-codes

Some machines and control systems also provide G-codes that have *three* digits instead of the standard two digits, for example, **G102**. This is a good indication that the machine manufacturer has included some special time-saving cycles (internal macros). These are not standard codes and usually vary from one machine to another. As you will learn later, a macro can also be called by a G-code other than the standard **G65**.

M-codes for Milling

The following table is a fairly comprehensive reference listing of the most typical and common M-codes (miscellaneous functions) used for CNC milling programs (CNC milling machines and machining centers). Only a very few M-codes are industry standard, so check the manual of your machine for details and usage:

M-code	Description
M00	Mandatory program stop
M01	Optional program stop
M02	End of program (usually no reset and rewind)
M03	Spindle rotation normal - clockwise
M04	Spindle rotation reverse - counterclockwise
M05	Spindle rotation stop
M06	Automatic tool change (<i>ATC</i>)
M07	Coolant mist <i>ON</i> <i>(machine option)</i>
M08	Coolant pump motor <i>ON</i>
M09	Coolant pump motor <i>OFF</i>
M19	Programmable spindle orientation
M30	End of program with reset and rewind
M48	Feedrate override cancel <i>OFF</i> - feedrate override switch effective
M49	Feedrate override cancel <i>ON</i> - feedrate override switch ineffective
M60	Automatic pallet change (<i>APC</i>)
M78	B-axis clamp <i>(non-standard)</i>
M79	B-axis unclamp <i>(non-standard)</i>
M98	Subprogram call
M99	Subprogram end <i>or</i> Macro end

G-codes for Turning

The following table is a fairly comprehensive reference listing of the standard and the most common G-codes (preparatory commands) used for CNC turning (CNC lathes). All dependent G-codes belong to the same group number and are modal, unless the Group is 00, which identifies all non-modal commands.

NOTE: Fanuc offers an option of *three G-code types* (called A, B and C). The most common in North America is the *A-type*. Type is selected by a system parameter. Small differences between control units should be expected - check the Fanuc reference manual for your application!

G-code types cannot be mixed !

G-code Type			Group	Description
Type A	Type B	Type C		
G00	G00	G00	01	Rapid positioning mode
G01	G01	G01	01	Linear interpolation mode ◆
G02	G02	G02	01	Circular interpolation mode - clockwise direction
G03	G02	G03	01	Circular interpolation mode - counterclockwise direction
G04	G04	G04	00	Dwell function (programmed as a separate block)
G09	G09	G09	00	
G10	G10	G10	00	Data setting mode (programmable data input)
G11	G11	G11	00	Data setting mode cancel
G18	G18	G18	16	ZX plane designation ◆
G20	G20	G70	06	English units of input
G21	G21	G71	06	Metric units of input
G22	G22	G22	09	Stored stroke check <i>ON</i> ◆
G23	G23	G23	09	Stored stroke check <i>OFF</i>
G25	G25	G25	08	Spindle speed fluctuation detection <i>ON</i>
G26	G26	G26	08	Spindle speed fluctuation detection <i>OFF</i> ◆
G27	G27	G27	00	Machine zero return position check
G28	G28	G28	00	Machine zero return - primary reference point
G29	G29	G29	00	Return from machine zero
G30	G30	G30	00	Machine zero return - secondary reference point
G31	G31	G31	00	Skip function
G32	G33	G33	01	Threading function - constant lead thread
G34	G34	G34	01	Threading function - variable lead thread
G35	G35	G35	01	Circular threading CW
G36	G36	G36	01	Circular threading CCW <i>or:</i>
G36	G36	G36	00	Automatic tool compensation for the X-axis
G37	G37	G37	00	Automatic tool compensation for the Z-axis
G40	G40	G40	07	Tool nose radius compensation mode cancel ◆
G41	G41	G41	07	Tool nose radius compensation mode to the left
G42	G42	G42	07	Tool nose radius compensation mode to the right
G50	G92	G92	00	Coordinate system setting (tool position register) <i>and / or:</i>

G-code Type			Group	Description
Type A	Type B	Type C		
G50	G92	G92	00	Maximum spindle <i>rpm</i> setting for the G96 mode
G52	G52	G52	00	Local coordinate system setting
G53	G53	G53	00	Machine coordinate system setting
G54	G54	G54	14	Work coordinate offset 1
G55	G55	G55	14	Work coordinate offset 2
G56	G56	G56	14	Work coordinate offset 3
G57	G57	G57	14	Work coordinate offset 4
G58	G58	G58	14	Work coordinate offset 5
G59	G59	G59	14	Work coordinate offset 6
G61	G61	G61	15	Exact stop mode
G62	G62	G62	15	Automatic corner override mode
G64	G64	G64	15	Cutting mode ◆
G65	G65	G65	00	Custom macro call
G66	G66	G66	12	Custom macro modal call
G67	G67	G67	12	Custom macro modal call cancel ◆
G68	G68	G68	04	Mirror image for double turrets <i>ON</i>
G69	G69	G69	04	Mirror image for double turrets <i>OFF</i> ◆
G70	G70	G72	00	Profile finishing cycle
G71	G71	G73	00	Profile roughing cycle - turning and boring
G72	G72	G74	00	Profile roughing cycle - facing
G73	G73	G75	00	Pattern repeating cycle
G74	G74	G76	00	Deep hole drilling cycle along the Z-axis
G75	G75	G77	00	Grooving / drilling cycle along the X-axis
G76	G76	G78	00	Multiple thread cutting cycle
G80	G80	G80	10	Fixed cycle for drilling cancel ◆
G83	G83	G83	10	Cycle for face drilling
G84	G84	G84	10	Cycle for face tapping
G86	G86	G86	10	Cycle for face boring
G87	G87	G87	10	Cycle for side drilling
G88	G88	G88	10	Cycle for side tapping
G89	G89	G89	10	Cycle for side boring
G90	G77	G20	01	Simple diameter cutting cycle
G92	G78	G21	01	Simple thread cutting cycle

G-code Type			Group	Description
Type A	Type B	Type C		
G94	G79	G24	01	Simple face cutting cycle
G96	G96	G96	02	Constant surface speed control - (<i>CSS mode</i>)
G97	G97	G97	02	Constant surface speed control cancel - (<i>rpm mode</i>) ♦
G98	G94	G94	05	Feedrate per minute - <i>IPM</i> or <i>mm/min</i>
G99	G95	G95	05	Feedrate per revolution - <i>ipr</i> or <i>mm/rev</i> ♦
-	G90	G90	03	Absolute input of motion values
-	G91	G91	03	Incremental input of motion values ♦

Note that some G-codes, for example **G36** or **G50** may have different meaning. Check your machine tool manual for any discrepancies in this reference list.

M-codes for Turning

The following table is a fairly comprehensive reference listing of the most typical and common M-codes (miscellaneous functions) used for CNC turning (CNC lathes). Only a very few M-codes are industry standard and common to all controls:

M-code	Description
M00	Mandatory program stop
M01	Optional program stop
M02	End of program (usually no reset and rewind)
M03	Spindle rotation normal - clockwise
M04	Spindle rotation reverse - counterclockwise
M05	Spindle rotation stop
M07	Coolant mist <i>ON</i> <i>(machine option)</i>
M08	Coolant pump motor <i>ON</i>
M09	Coolant pump motor <i>ON</i>
M10	Chuck or collet open
M11	Chuck or collet close
M12	Tailstock quill <i>IN</i> <i>(non-standard)</i>
M13	Tailstock quill <i>OUT</i> <i>(non-standard)</i>
M17	Turret indexing forward <i>(non-standard)</i>
M18	Turret indexing reverse <i>(non-standard)</i>
M19	Programmable spindle orientation <i>(machine option)</i>

M-code	Description
M21	Tailstock body forward <i>(non-standard)</i>
M22	Tailstock body reverse <i>(non-standard)</i>
M23	Gradual pull-off from thread <i>ON</i>
M24	Gradual pull-off from thread <i>OFF</i>
M30	End of program with reset and rewind
M41	Gear range selection - low gear <i>(if available)</i>
M42	Gear range selection - medium gear 1 <i>(if available)</i>
M43	Gear range selection - medium gear 2 <i>(if available)</i>
M44	Gear range selection - high gear <i>(if available)</i>
M48	Feedrate override cancel <i>OFF</i> - feedrate override switch effective
M49	Feedrate override cancel <i>ON</i> - feedrate override switch ineffective
M98	Subprogram call
M99	Subprogram end <i>or</i> Macro end

Standard Program Codes

Most G-codes and M-codes used in macros are standard codes. These are available to every user and when used in macros, are usually quite portable from one control to another. Unfortunately, there is no established convention as to what codes are standard, and some may vary for the same control used with different machines. Caution is advised here:

Always check each control/machine combination for G-codes and M-codes

Optional Program Codes

The main reason for including several non-standard M-codes in the tables is that they should serve as an example of available M-codes. Hopefully, they will help you to find the actual function code for the same activity, done on your CNC machine. The standard program codes are fairly common across different Fanuc models, but there are well known differences between older and newer control versions.

The focus of this handbook is mainly on the higher level of Fanuc controls, which are more likely to be used with the custom macro feature. Lower level controls (such as Fanuc 0), with a macro feature, will have limited features, including G-codes and M-codes.

3

REVIEW OF SUBPROGRAMS

To review the subject of subprograms, you have to understand first what a subprogram is, what it can be used for and what are its benefits. Comprehensive knowledge of subprograms is essential for macro program development.

In CNC programming, a subprogram is very similar in structure to a conventional program. What makes it different is its *content*. Typically, a subprogram is a separate program containing only unique repetitive tasks, such as a common contouring toolpath, a hole pattern or similar machining operations. For example, the task is to program a certain pattern of holes, in which the holes have to be spot-drilled, drilled and tapped. In standard part programming, the *XY* point coordinates for each hole will have to be calculated and repeated for each tool, using the appropriate fixed cycle. In a subprogram, the hole locations can be calculated only once, then stored in a separate program (subprogram) and retrieved many times, as needed, for different operations using different fixed cycles.

A subprogram is always called by another program (main program or another subprogram).

Subprogram must only contain data common to all parts or operations

Subprogram Example - Mill

To illustrate the concept of subprograms with a practical example, a very simple pattern of five holes is shown in the following illustration - *Figure 2*:

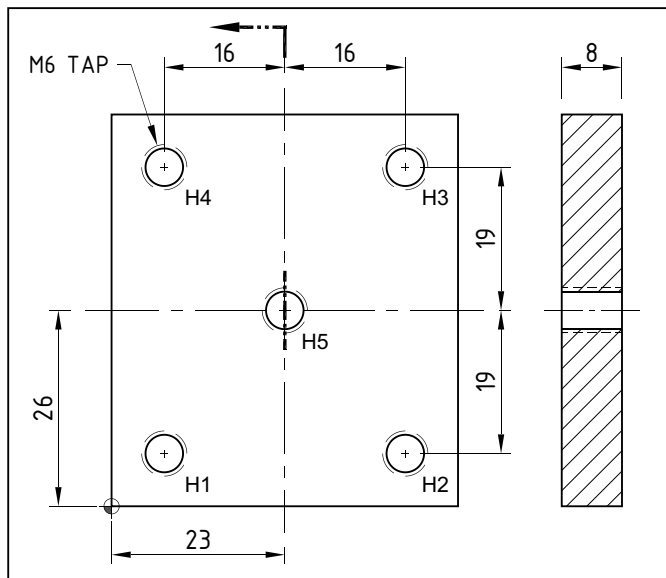


Figure 2

Sample drawing for subprogram example - mill application

Five holes have to be machined with three tools. Without a subprogram, the program is fairly long, and all hole locations will be repeated for each tool. All three examples will follow the holes in the same order, and the first tool is assumed to be in the spindle. Program *Example 1* shows the program *without a subprogram* - this is the longest version:

```

O0001
(EXAMPLE 1 OF 3 - MAIN PROGRAM ONLY - PETER SMID)
(PROGRAM ZERO IS AT LOWER LEFT CORNER AND TOP OF PART)
(T01 - 90-DEG SPOT DRILL)
N1 G21
N2 G17 G40 G80
N3 G90 G54 G00 X7.0 Y7.0 S1200 M03 T02           (H1)
N4 G43 Z25.0 H01 M08
N5 G99 G82 R2.5 Z-3.4 P200 F200.0
N6 X39.0                                         (H2)
N7 Y45.0                                         (H3)
N8 X7.0                                           (H4)
N9 X23.0 Y26.0                                   (H5)
N10 G80 G00 Z25.0 M09
N11 G28 Z25.0 M05
N12 M01

(T02 - 5 MM TAP DRILL)
N13 T02
N14 M06
N15 G90 G54 G00 X7.0 Y7.0 S950 M03 T03         (H1)
N16 G43 Z25.0 H02 M08
N17 G99 G81 R2.5 Z-10.5 F300.0
N18 X39.0                                         (H2)
N19 Y45.0                                         (H3)
N20 X7.0                                           (H4)
N21 X23.0 Y26.0                                   (H5)
N22 G80 G00 Z25.0 M09
N23 G28 Z25.0 M05
N24 M01

(T03 - M6X1 TAP)
N25 T03
N26 M06
N27 G90 G54 G00 X7.0 Y7.0 S600 M03 T01         (H1)
N28 G43 Z25.0 H03 M08
N29 G99 G84 R5.0 Z-11.0 F600.0
N30 X39.0                                         (H2)
N31 Y45.0                                         (H3)
N32 X7.0                                           (H4)
N33 X23.0 Y26.0                                   (H5)
N34 G80 G00 Z25.0 M09
N35 G28 Z25.0 M05
N36 G28 X23.0 Y26.0
N37 M30
%
```

Even if the program length does not matter, it is a bad programming practice to repeat common data. The main reason is a possible drawing change. For example, if only one hole location is revised on the drawing, at least *three* changes will have to be done in the part program. Using a subprogram will not only shorten the program length, but also enables much more efficient editing. Program *Example 2* shows the *same* machining process using a *subprogram call*:

```

O0002
(EXAMPLE 2 OF 3 - MAIN PROGRAM WITH A SUBPROGRAM - PETER SMID)
(PROGRAM ZERO IS AT LOWER LEFT CORNER AND TOP OF PART)
(T01 - 90-DEG SPOT DRILL)
N1 G21
N2 G17 G40 G80
N3 G90 G54 G00 X7.0 Y7.0 S1200 M03 T02
N4 G43 Z25.0 H01 M08
N5 G99 G82 R2.5 Z-3.4 P200 F200.0 L0 (OR K0 ON SOME CONTROLS)
N6 M98 P1001
N7 G80 G00 Z25.0 M09
N8 G28 Z25.0 M05
N9 M01

(T02 - 5 MM TAP DRILL)
N10 T02
N11 M06
N12 G90 G54 G00 X7.0 Y7.0 S950 M03 T03
N13 G43 Z25.0 H02 M08
N14 G99 G81 R2.5 Z-10.5 F300.0 L0 (OR K0 ON SOME CONTROLS)
N15 M98 P1001
N16 G80 G00 Z25.0 M09
N17 G28 Z25.0 M05
N18 M01

(T03 - M6X1 TAP)
N19 T03
N20 M06
N21 G90 G54 G00 X7.0 Y7.0 S600 M03 T01
N22 G43 Z25.0 H03 M08
N23 G99 G84 R5.0 Z-11.0 F600.0 L0 (OR K0 ON SOME CONTROLS)
N24 M98 P1001
N25 G80 G00 Z25.0 M09
N26 G28 Z25.0 M05
N27 G28 X23.0 Y26.0
N28 M30
%

O1001 (5 HOLE LOCATIONS SUBPROGRAM - VERSION 1)
N101 X7.0 Y7.0 (H1)
N102 X39.0 (H2)
N103 Y45.0 (H3)
N104 X7.0 (H4)
N105 X23.0 Y26.0 (H5)
N106 M99 (SUBPROGRAM END)
%
```

Note that the subprogram contains the hole locations only (XY coordinates), and nothing else. Also note the L0 or K0 added to the fixed cycle block for all tools. L0 or K0 is a fixed cycle parameter, meaning that the cycle is not executed in the current block. The data programmed in the current block are stored in memory and will be used when the subprogram is processed. The hole locations in the subprogram will use any fixed cycle data that are active (passed from the main program, including the parameters). The cycle repetition address L is used on Fanuc models 10/11/12/15 and the address K is used on Fanuc models 0/16/18/21.

There might be other ways to structure the main program and the subprogram. The clue may often be found in the main program. Anytime you see several consecutive blocks that are *identical* either before or after the subprogram call, you may consider their inclusion in the subprogram. In the shown *Example 2*, each time the M98 P1001 is programmed (for the three tools), it is always followed by two identical blocks:

```
G80 G00 Z25.0 M09
G28 Z25.0 M05
```

Could these blocks be added to the subprogram? *Yes*. The program *Example 3* of the complete program will be a bit shorter than the previous version (see reservations following the program):

```
O0003
(EXAMPLE 3 OF 3 - MAIN PROGRAM WITH A SUBPROGRAM - PETER SMID)
(PROGRAM ZERO IS AT LOWER LEFT CORNER AND TOP OF PART)
(T01 - 90-DEG SPOT DRILL)
N1 G21
N2 G17 G40 G80
N3 G90 G54 G00 X7.0 Y7.0 S1200 M03 T02 (H1)
N4 G43 Z25.0 H01 M08
N5 G99 G82 R2.5 Z-3.4 P200 F200.0 L0 (OR K0 ON SOME CONTROLS)
N6 M98 P1002
N7 M01

(T02 - 5 MM TAP DRILL)
N8 T02
N9 M06
N10 G90 G54 G00 X7.0 Y7.0 S950 M03 T03 (H1)
N11 G43 Z25.0 H02 M08
N12 G99 G81 R2.5 Z-10.5 F300.0 L0 (OR K0 ON SOME CONTROLS)
N13 M98 P1002
N14 M01

(T03 - M6X1 TAP)
N15 T03
N16 M06
N17 G90 G54 G00 X7.0 Y7.0 S600 M03 T01 (H1)
N18 G43 Z25.0 H03 M08
N19 G99 G84 R5.0 Z-11.0 F600.0 L0 (OR K0 ON SOME CONTROLS)
N20 M98 P1002
N21 G28 X23.0 Y26.0
N22 M30
%
```

```

O1002 (5 HOLE LOCATIONS SUBPROGRAM - VERSION 2)
N101 X7.0 Y7.0 (H1)
N102 X39.0 (H2)
N103 Y45.0 (H3)
N104 X7.0 (H4)
N105 X23.0 Y26.0 (H5)
N106 G80 G00 Z25.0 M09 (CANCEL CYCLE AND CLEAR)
N107 G28 Z25.0 M05 (Z-AXIS HOME RETURN)
N108 M99 (SUBPROGRAM END)
%
```

Do you like the program better than the previous version? In a strict technical definition, there is nothing wrong with the program - it will work well as is. Yet, there is a problem of different kind - the program uses a structure that many experienced programmers should and will avoid. Although the program itself is somewhat shorter, it is also much harder to interpret. Look at the reasons. When the subprogram is completed, the processing returns to the main program. Study the main program and you will see that it is impossible to tell whether the fixed cycle had been canceled or not. Also difficult is to see what other data may have been passed to the main program from the subprogram. You have look deep into the subprogram to find out these important details, which may be many printed pages away from the main program. In conclusion, while the shown *Example 3* is correct, it is definitely *not* recommended to be used, because of its poor structure.

Rules of Subprograms

From the last two examples for the five holes, you can see how a subprogram is defined, how it is ended and how it is called from another program. In a summary, there are two miscellaneous functions associated with subprograms:

M98	Subprogram call (followed by the subprogram number)
M99	Subprogram end

The **M98** function must always be followed by the subprogram number, for example,

```
M98 P1001
```

The subprogram must be stored in the control system under the assigned number, for example, as *O1001*. The miscellaneous function **M99** is usually programmed as a separate block - and also as the last block in the subprogram. This function will cause the transfer of the processing from the subprogram back to the program it *originated from*. That may be the main program or another subprogram.

The *end of record* symbol (the % sign) follows the **M99** function, the same way it follows the **M30** function in the main program. The % symbol represents a flag to stop transmission of the program, typically in DNC mode. When the processing returns to the program of origin, it will always be to the block immediately following the program call. For instance, look at the earlier *Example 2*:

```
N6 M98 P1001
N7 G80 G00 Z25.0 M09
```

When the subprogram O1001 is completed, the program processing returns to the block N7 of the main program (which is the program of origin in the example).

There are times when the program processing has to return to a block *other* than the one immediately following the subprogram call. This is not a common occurrence and is used for special purposes only. In such a case, the **M99** will have a *P-address*, indicating which block number to return to in the program of origin. Note that the P-address in this case has a totally different meaning than the P-address in the **M98** function. For example, a subprogram O1003 has the following end block:

```
N108 ...
N109 M99 P47
%
```

In the main program, the subprogram call may look something like this:

```
N43 ...
N44 M98 P1003
N45 ...
N46 ...
N47 ... (THE BLOCK TO RETURN TO FROM THE SUBPROGRAM)
...
```

The normal block to return to would be N45. However, because of the P-address included with the **M99** function, the processing will return to the block N47, skipping two blocks. One area of programming where this technique has a good application is bar feeding on a CNC lathe.

Figure 3 below, shows a typical subprogram application (in a structured form), with processing returning to the next block of the calling program:

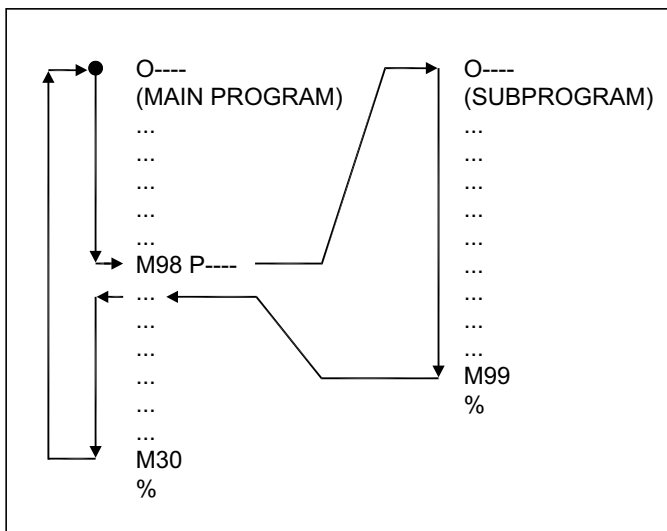


Figure 3

Typical program flow - a subprogram called by the main program - this is the most common application of subprograms in CNC programming

Subprogram Repetition

It is not unusual - in fact, it is quite common - to call a particular subprogram more than once from the same program of origin. Normally, when a subprogram is called, it is automatically processed (executed) only once. That is the most common default condition, when no other instruction has been issued. If the subprogram has to be repeated more than once, either a special address L or K has to be added or the *number of repetitions* has to be included in the subprogram call. The choice depends on the control system. Look at this example, where a subprogram stored as O4321 has to be repeated three times:

Method 1 **M98 P4321 L3** ... uses the address L - common to 6/10/11/12/15 controls

Method 2 **M98 P4321 K3** ... uses the address K - common to 0/16/18/21 controls

Method 3 **M98 P0034321** ... uses the combined structure in the same block

These three methods have the same result - the subprogram O4321 will be repeated three times. What are the differences? There is a simple answer - the control systems used. The address L or K specify the number of repetitions directly and separately from the subprogram number call - those are the first two examples. The third example uses a combined structure - the first three digits identify the number of subprogram repetitions (003), and the last four digits identify the subprogram number being called (4321). Check the control system user manual to find which method is supported for your control unit. If the number of repetitions is not specified, the system will process the called subprogram only once.

Subprogram Nesting

The most common application of a subprogram is to call it only once and process it only once. After that, the program of origin (usually the main program) continues normally. Although several subprogram calls can be made from the main program, once the subprogram is completed, the processing continues in the main program. This is called a *single level nesting*, and is also the most common application of subprograms.

Fanuc controls allow for up to *four levels* of subprogram nesting (also called *four levels fold*). Nesting means that one subprogram may call another subprogram, which may call still another subprogram, up to four levels deep. As the number of calling levels increase, the programming becomes increasingly more complex as well and can be quite difficult to develop. It is very unusual to program more than two levels deep nesting. In all cases, there is one important rule in programming to observe:

In a nested program environment, a subprogram will always return to the program it originated from

The program of origin may be the main program or another subprogram. All four illustrations that follow show a graphical program flow for the four levels of subprogram nesting:

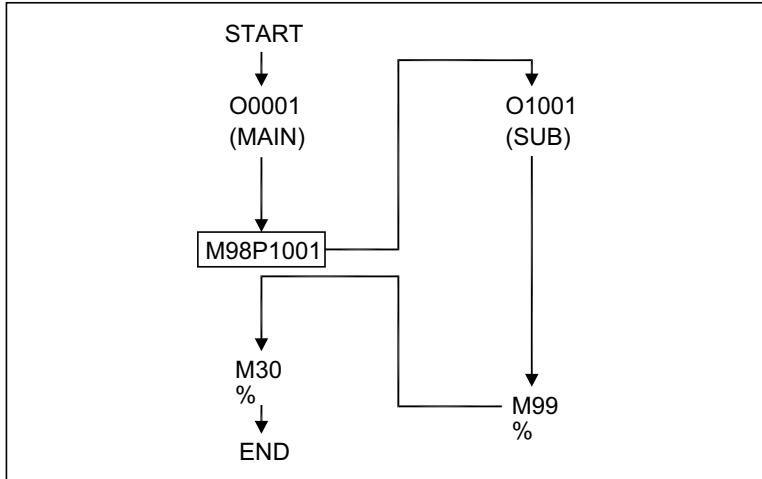


Figure 4

Single level of a subprogram nesting

The illustration in *Figure 4* shows a general schematic representation of the previous example - a *single level* of a subprogram nesting. This is the most common application of subprograms.

More complex (multi-level) subprogram nesting brings an extra power to the CNC programming process, but at the price of more time required for development, as well as some clarity and convenience built into the program. That is not to say the multi-level subprogram nesting should be discouraged or even avoided altogether. It simply means that although you may develop a very sophisticated program flow, but you may also be the only one who understands it.

Subprograms that use the three or four-level nesting are very rare in practice. By careful planning, the design of a control system must always be a step ahead of the design of machine tools. For example, a 250 000 rev/min spindle speed may not be available on machine tools at this time, but the control system can still support it, in case a particular manufacturer comes forward with exactly that kind of a spindle. Four-level nesting has been designed for the same reason. The three levels of subprogram nesting are illustrated as schematic graphics (*Figures 5 to 7*):

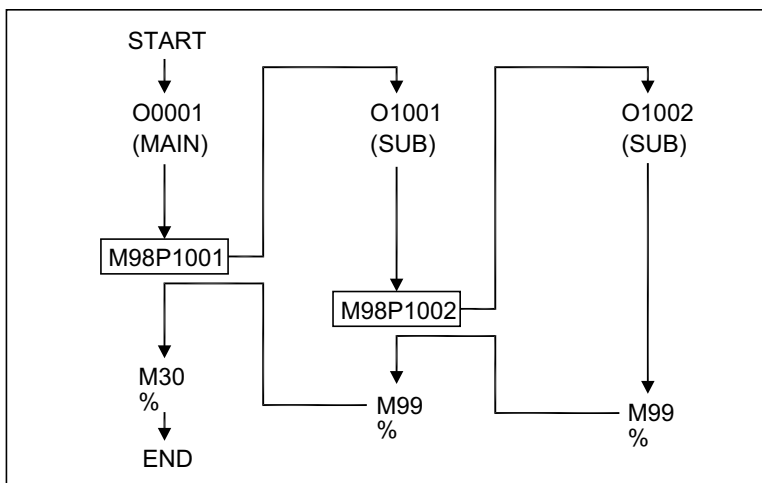


Figure 5

Two-level subprogram nesting

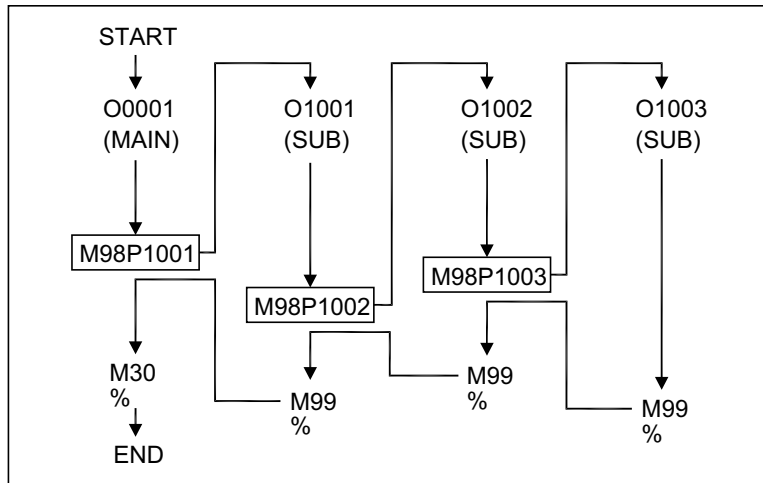


Figure 6

Three-level subprogram nesting

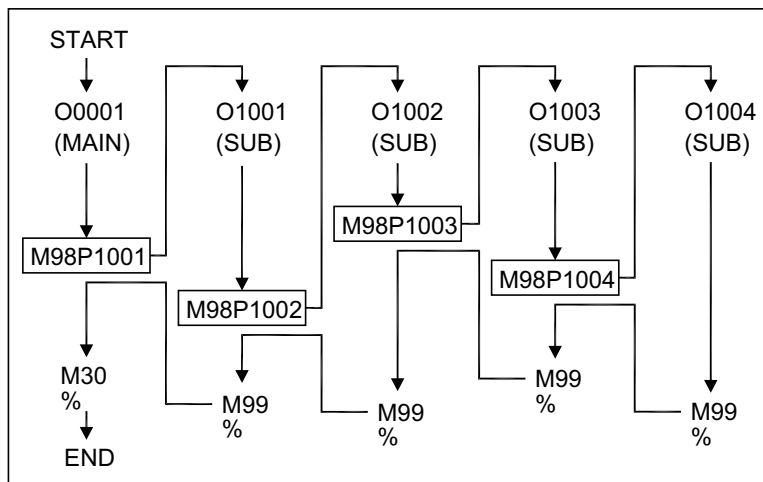


Figure 7

Four-level subprogram nesting

Subprogram Documentation

Any complex part program (subprograms and macros included) should always be well documented. Documenting CNC programs has been largely ignored by many users, often because of the perceived need to do a job fast. Although somewhat forgivable for simple and easy programs, the practice of *not* documenting programs is definitely not acceptable for subprograms and is also not acceptable for macros. Good program documentation is the key part of any CNC program development. Look at the schematic drawings of the four levels of subprogram nesting and you will see how complex the program can become with each increasing level of nesting. A good documentation will help the user in orientation and program 'decoding', therefore becomes a mandatory part of the programming process.

For both, the subprograms and macros, the program documentation should be internal as much as possible. This can be achieved by including important comments in the program body (main program, subprogram, or a macro). Program comments are typically enclosed in parentheses, for example, as (DRILLING HOLE NUMBER 5). Provide only those comments that are relevant.

Subprograms vs. Macros

One main purpose of this handbook is to emphasize the custom macro option of Fanuc controls. Since the program development of subprograms is critical as the basic knowledge for macro development, this chapter has so far reviewed the basic concepts of subprograms, their structure and their applications in a typical CNC program.

In the terms of purpose, custom macros are direct extensions of subprograms or similar subroutines. They are treated virtually the same way as subprograms - they are normally stored under a separate program number (O----- or O-----), and they always end the same way, using the **M99** function. Macros are called in a similar way, using the **G65** preparatory command, along with the specified parameters.

A typical CNC program can mix both, the conventional method of programming (with or without subprograms) and macros - or use at least some macro features. Of course, the control system must support the macro option.

The major difference between the two unique programming methods is the flexibility macros offer. Unlike subprograms, macros can be used with *variable* data (using the so called *variables*), they can perform many mathematical operations and they can store current values of various machine settings (current machine status). A very important part of macros is their ability to use conditional testing, branching and looping for a very flexible program flow. The use of looping features alone, so called *iteration*, adds much desired extra programming power. Overall, these are the three items that are the most significant in programming macros:

- Variable data input
- Mathematical functions and calculations
- Storage and retrieval of current machine values

Unique Features

Macros have their own unique features, not found in normal subprograms or, for that matter, in any other conventional method of part programming.

Typical features that are classified as unique to macros are mostly related to flexibility:

- Program data can be changed
- Program flow can be altered
- Data can be passed from one program to another
- Repetitions can be looped
- Measurement (probing) can be incorporated
- Special equipment can be fully controlled

These are only some items that distinguish the major differences between subprograms and custom macros. Do not think of macros just as a better replacement for subprograms. There are many uses of macros that cannot be compared with anything similar to subprograms. The main - and most unique - features of typical macros are their flexibility and ease of use, once you master the basic issues associated with macro development.

Unlike subprograms, macros include many special functions that can be found on a typical scientific calculator (**TAN**, **COS**, **SIN**, **SQRT**, etc.). Not only simple or more complex algebraic functions can be used, macros can also be applied for trigonometric calculations, square roots, powers of a number x^2 or x^3 , inverse functions, nested parentheses, rounding values, using many other features (just like the calculator functions). Specific constants, such as the π function ($\pi = 3.141592654\dots$), are not available, but can be defined. Calculated values can be stored into a memory register and used in the current program, or any other program.

There is no doubt that macros can elevate CNC programming to the levels never before possible with only main programs or subprograms alone. Knowledge of subprograms, how they work, how they are structured and how they interact with main program, is the key knowledge required for any programmer trying to unravel the mystery of custom macros.

CNC Lathe Applications

Macros are useful for any type of a machine tool. Although machining centers (used so far as illustrative examples) have become the most likely sources of macro programs, that does not mean other machine tools are excluded. The one machine tool that is widely used in everyday production, the one machine tool that will benefit from custom macros is - the *CNC lathe*.

In *Figure 8* shows a drawing of a lathe part with three identical grooves.

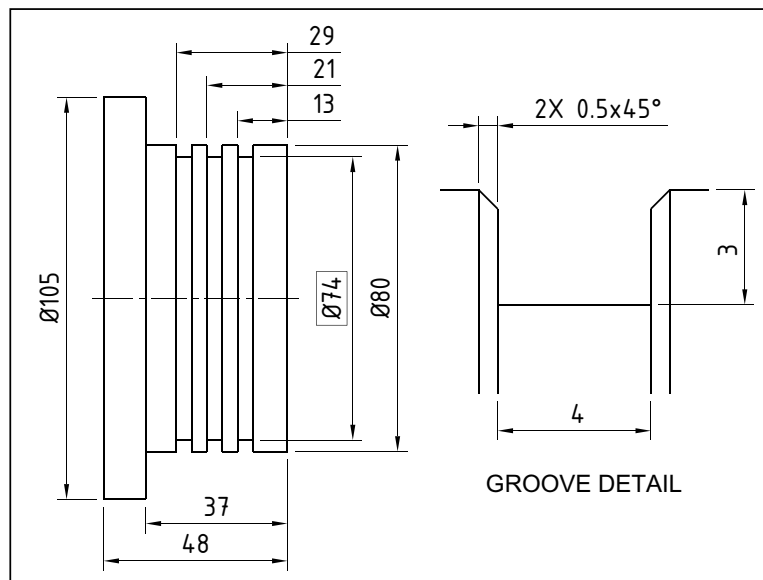


Figure 8

Sample drawing for
subprogram example
- lathe application

Groove detail shows a 4x3x0.5 mm groove. Although all three grooves are located at the same diameter, it is quite likely that this groove - or any standard groove - can still have the same overall dimensions but be placed at different diameters. Whether developing a subprogram or a macro, this is a very important consideration. While grooves that do not change diameters can be programmed in absolute mode along the X-axis and incremental mode along the Z-axis, grooves that will be at various diameters have to be programmed incrementally along both axes.

Subprogram Development

The basic approach for developing a macro is the same as for developing a subprogram. In this case, a 3 mm wide grooving tool will be selected (only the grooving operations are shown), following this machining procedure:

1. In the main program, the tool will move to the left side of the groove wall and 0.5 mm above the part diameter (= initial position for each groove)
2. In the subprogram, the tool shifts 0.5 mm to the right (middle of the groove), plunges to the depth but leaves 0.1 mm stock at the bottom
3. Tool retracts 0.5 mm above the part diameter, shifts to the start of the left chamfer, cuts it, then it cuts the left groove wall - still leaving 0.1 mm stock at the bottom
4. Tool retracts to the start position, shifts to the start of the right chamfer, cuts it, then it cuts the right groove wall to the full depth and sweeps the bottom towards the left groove wall
5. Tool retracts above the diameter to the initial position - subprogram ends

The main program and the subprogram are listed with comments:

```
(MAIN PROGRAM)
N1 G21 T0100          <... facing and turning ...>
...
N14 T0500
N15 G96 S120 M03
N16 G00 X81.0 Z-13.0 T0505 M08      Start for Groove 1 - initial position
N17 M98 P8001                    Cut Groove 1
N18 G00 Z-21.0 M98 P8001          Start for Groove 2 + cut Groove 2
N19 G00 Z-29.0 M98 P8001          Start for Groove 3 + cut Groove 3
N20 G00 X150.0 Z100.0 T0500 M09
N21 M30
%
O8001 (4 X 3 X 0.5 GROOVE SUBPROGRAM - 3 MM WIDE GROOVING TOOL)
N101 G00 W0.5                Shift to the middle of the groove
N102 G01 U-6.8 F0.125        Feed-in - leave 0.1 at the bottom
N103 G00 U6.8                Rapid out to the start position
N104 W-1.5                    Shift to the start of the left chamfer
N105 G01 U-2.0 W1.0 F0.08    Cut the left chamfer
N106 U-4.8 F0.1              Feed in along the left wall - leave 0.1 at the bottom
N107 U6.8 W0.5 F0.25        Rapid feed to the start position
N108 G00 W1.5                Shift to the start of the right chamfer
N109 G01 U-2.0 W-1.0 F0.08   Cut the right chamfer
N110 U-5.0 F0.1              Feed in along the right wall to full depth
N111 W-1.0 F0.08            Sweep the bottom to the left wall
N112 U7.0 W0.5 F0.25        Rapid feed to the start position
N113 G00 W-0.5                Shift back to the initial position
N114 M99                      End of subprogram
%
```

The included comments should be sufficient to understand the subprogram design.

4

SYSTEM PARAMETERS

The machine related information that establishes the connection between the CNC system and the machine tool, is stored as special data in internal control system registers, called the *system parameters*, or *control parameters*, or just *CNC parameters*. As an English word, its meaning is oriented towards mathematics, and is defined in a rather fancy sentence - '*A parameter is a quantity which may have various values, each fixed within the limits of a stated case*'. The sentence shows that the dictionary definition is right on for the purpose of defining parameters for a CNC system. Do not confuse parameters of the control system with the method of programming called *parametric programming* - except linguistically, they are *not* related. If you are a part programmer with limited experience, you should not be concerned about system parameters too much. Their original factory settings are generally quite sufficient for most work.

For specialized work like macro development, with all its related activities, such as probing and gauging, automatic offset changes, special methods of input and output, etc., a good in-depth knowledge of the system parameters is extremely important. There are hundreds of parameters available for any control system, and the majority of them you will never use.

Parameters are critical to the CNC machine operation - be careful when working with them !

What are Parameters ?

When the machine tool manufacturers design a CNC machine, they have to connect it with the CNC system, mostly designed by a *different* manufacturer. For example, a *Makino*™ CNC machining center should be connected with a *Fanuc* CNC system - two independent manufacturers of two different products are involved in the process. The process of connection and configuration is often known as *interfacing*. The control system of Fanuc units has been designed with great internal flexibility and many parameters have to be set before the CNC machine tool is operational. Normally, it is the machine tool manufacturer (often called the *vendor*) who supplies the end user with all settings - *the interfacing* - a typical user seldom makes any changes. Even when a system parameter is changed by the program (standard or macro), the change is often only a temporary one, designed for a particular purpose. When the purpose is achieved, the program (or macro) normally resets the parameter to its original contents. Parameters are often changed intentionally, in order to optimize the performance of the machine tool.

The majority of control parameters relate to the specifics of a particular CNC machine tool, known to the machine tool manufacturer or vendor as various *defaults*. They include such items as all machine tool specifications, functions and characteristics that are in the exclusive domain of the manufacturer. Typical examples include rapid traverse rate, spindle speed ranges, length of axis motions, rapid or cutting feedrate ranges, clearances, various timers, data transfer baud rates, and many others. These parameters do not change and any attempt to make any changes severally endangers smooth machine operation.

The settings, collectively known as *parameters*, are means of matching the control system with the unique features of the particular CNC machine tool. That way, a single control system may be used with many CNC machine tools, from different machine tool manufacturers, in many shops, and each with a setting unique to the particular machine.

Many CNC users are not familiar with the function of parameters at all and only a handful know how the parameters can be used to an optimum performance of a CNC machine. The reason is that when a company purchases a CNC machine, everything is set to work properly and without additional settings or changes. These factory settings work well for the majority of users and there is a little need to make any changes.

For macros - or for any serious CNC work - the programmer should be thoroughly familiar with the way system parameters work. Even if major changes to the default settings are planned, the knowledge of 'how-to-do-it' can be very useful.

Saving Parameters

There is a simple rule that should be followed by any CNC programmer and operator. The rule is so simple and so much a common sense, that it is surprising how often it is *not* followed. This rule recommends that all system parameters should be *backed up*. The same rule that applies to any computer data and is fairly well followed in an office environment, is mostly ignored in machine shops. There are several reasons for the backup, but two of them are the most important.

The first reason is that all current settings of the parameters need *power* (energy) to be retained in the memory of the CNC system. The energy is supplied through the main power, whenever the machine tool is operational. When the main power is turned off, a built-in battery power takes over. Each control has a battery backup that provides power to the important settings, even when the main power is not supplied. However, battery power is not permanent - batteries do lose power and with loss of power, data is lost as well.

Number two reason is *customized settings*. It may be fairly easy to replace lost parameters that had been supplied by the machine tool manufacturer. Either the manufacturer had provided the customer a copy or keeps a copy on file. It could also be the machine tool dealer that keeps a backup (no guarantees, of course). These backups may help to restore standard parameters. What about the parameters that have been modified by the user, or during a service, or even through some customized program? These parameters are strictly in the domain of the user and without a backup strategy, they are irreplaceable.

Backing Up Parameters

Control system parameters can be backed-up (saved externally) by several means. The most popular is to save the parameters as a *disk file*, using the DNC features of the control system. This operation requires a computer (laptop is ideal), with the proper communication software installed and configured. Saved parameters should also be printed as a *hard copy*, in case something happens to the disk itself. If you cannot save parameters in an efficient way, like the DNC method, use the old-fashioned pencil and paper and write them down. There may be quite a few of them, it may take time, but the effort may be well worth it in case of emergency.

Parameter Identification

Default parameters are supplied by the machine tool manufacturer or vendor and they are stored in the control system in a very orderly way - they are *numbered* and separated into special groups.

Numbering of Parameters

Parameters are typically numbered as four-digit integers within the 0000 to 9999 range. Since there are several models of Fanuc controls available, you should be aware that *different control models often use different parameter numbers*, even for the same type of setting. In addition, there are parameters having one meaning for a milling system and a different meaning for a turning system. Compound these differences with a number of different options available on each control, and you have a vast maze to explore. This can be confusing to anybody, including experienced service technicians.

Every CNC machine has literally several hundreds specifications that have to be set by the system parameters. However, for the purposes of the end user (the actual user of the CNC machine tool), the parameters belong to only two *general* groups:

- ◆ Parameters that *cannot be changed by the end user* (*machine oriented*)
- ◆ Parameters that *can be changed by the end user* (*program oriented*)

As a rule, any system parameter that is machine specific, should *never* be changed by the end user. On the other hand, many parameters relating to the CNC program (or some operational functions) may be changed by the end user, usually for the purpose of compatibility with other machine tools in the manufacturing environment or to maintain a consistent programming structure. Typical settings may include configurations of various machining cycles, input and output settings (I/O), various offsets and compensations, tooling data, cutting data, and many others.

Parameter Classification

Numbering the system parameters makes sense for the programming purposes, but the sheer number of parameters available to the user (CNC programmers are included here) makes it difficult to find a particular parameter quickly and efficiently. Just try to memorize a few dozen of particular four digit numbers.

Fanuc has been well aware of the potential problem and organized the parameters into about two and half dozen logical groups, based on the general purpose of the parameters. To illustrate what areas of the machine tool operation the CNC system parameters cover, the list on the next page is an itemized collection of parameter classification, by the groups, for a typical Fanuc control system. As a CNC programmer, you should be familiar with all of them, although most of them will be meaningful only to the service technicians.

In the Fanuc reference manual (typically called the *Parameter Manual*), each group has an assigned range of parameter numbers with detailed descriptions.

Parameters Grouping

The following groups are controlled by Fanuc. Do not expect all of them on all controls and do not expect them to be permanent. Control systems do develop, therefore they do change. Although the list is generally accurate, always consult the manual for your machine tool and the control system to get the parameter groups for a particular machine tool (changes happen constantly).

<i>Parameter Relation Group</i>	<i>10/11/12</i>	<i>15</i>	<i>16/18/21</i>
Parameters for Setting	0000-	0000-0032	0000-
Parameters for Axis Control Data	1000-	1000-1058	1000-
Parameters for Chopping		1191-1197	
Parameters for Coordinate System	1220-	1200-1260	1200-
Parameters for Feedrate	1400-	1400-1494	1400-
Parameters for Accel./Decel. Control	1600-	1600-1631	1600-
Parameters for Servo	1800-	1800-1980	1800-
Parameters for DI/DO	2000-	2000-2049	3000-
Parameters for MDI, EDIT, and CRT	2200-	2200-2388	3100-
Parameters for Programs	2400-	2400-2900	3400-
Parameters for Serial Spindle Output		3000-3303	
Parameters for Graphic Display		4821-4833	6500-
Parameters for I/O interface	5001-	5000-5162	0100-
Parameters for Stroke Limit	5200-	5200-5248	1300-
Parameters for Pitch Error Comp.	5420-	5420-5425	3600-
Parameters for Inclination Comp.		5461-5474	8200-
Parameters for Straightness Comp.		5481-5574	
Parameters for Spindle Control	5600-	5600-5820	3700-
Parameters for Tool Offset	6000-	6000-6024	5000-
Parameters for Canned Cycle	6200-	6200-6240	5100-
Parameters for Rigid Tapping			5200-
Parameters for Scaling & Coord. Rotation	6400-	6400-6421	5400-
Parameters for Automatic Corner Override	6820-	6610-6614	5480-
Parameters for Involute Interpolation		6620-6634	5610-
Parameters for Uni-directional Positioning	7000-	6820	5440-
Parameters for Polar Coord. Interpolation			5460-
Parameters for Indexing Index Table			5500-
Parameters for Custom (User) Macro	7200-	7000-7089	6000-
Parameters for External Data Input			6300-
Parameters for Oper. Time & Number of Parts			6700-
Parameters for Program Restart		7110	7300-
Parameters for High-Speed Skip Signal Input		7200-7214	6200-
Parameters for Automatic Tool Comp.	7300-	7300-7333	6240-
Parameters for Tool Life Management		7400-7442	6800-
Parameters for Position Switch Functions			6900-
Parameters for Turret Axis Control		7500-7557	
Parameters for High Speed Machining			7500-
Parameters for Axis Control by PMC			8001-
Parameters for Service	8000-	8000-8010	
Other parameters			7600-7794

As the list shows, many of the system parameter groups have nothing to do with CNC programming directly and are listed only as a matter of interest only, or for the electrical or electronic specialists, as well as service technicians. To the CNC programmer, the list should serve as a source of reference. *Do not confuse parameter numbers with system variables!*

Parameter Display Screen

A brief look at a typical display of parameter screen on a CNC unit (using the *SYSTEM/PARAM* menus of the keyboard), many display screen pages will be available. The screen pages can be quickly scrolled through, forward or backward. A particular parameter can be called by its number (see *Parameter Manual*), in order to speed up the search. The screen cursor (display indicator) will be positioned at the parameter number and the parameter data will be displayed in reverse colors (highlighted). For the purpose of this handbook, the actual procedures of the keyboarding is not important - the control system manual describes all necessary steps.

Parameter Data Types

The classification of parameters in the brief section listed on the previous page has only shown the description of the parameters by *function*. In a control system, the actual parameter values are entered for each parameter number, as needed. Depending on the individual parameter application, the parameters are also classified by their *data type*. Each data type group uses a different range of valid parameter data entry.

On all Fanuc controls, there are four data type groups available. They are listed here with appropriate data ranges for each group:

Parameter Data Type	Available Data Range
Bit or Bit axis	0 or 1
Byte or Byte axis	0 to 127 (byte) or 0 to 255 (byte axis)
Word or Word axis	0 to 32767
Two-word or Two-word axis	0 to 99999999

Bit-Type Data Type

Bits and *bytes* are common computer terms and should not be confused with each other. A *bit* is the smallest unit of a parameter input. Only *two* input values are allowed - the digit zero (0) and the digit one (1). The word *bit* is an abbreviation, derived from the full version of the words *binary digit*, as in *binary digit*. The English word *binary* has its origin in the Latin word *binarius*, meaning *something consisting of two parts*. Based on this definition, the two possible input values **0** and **1** represent a certain option, selected from no more than two conditions. Such a condition can be either *true* or *false*. *True* or *False* conditions can also be interpreted as *Yes* and *No*, *On* or *Off*, *Done* or *Not Done*, and so on. In the bit type entry, the selection represents one of only two possible alternatives.

A *byte* (described later) is a sequence of several adjacent bits (typically eight), that represents one alphanumeric character of data that is processed as a single unit of instruction.

Each bit type system parameter can be set with input values for up to eight registers (also known as *locations*). Be careful and watch the bit type parameters - a single data number is assigned eight (8) bits (*i.e.*, an eight-bit binary parameter). Each bit has a different meaning, so exercise care when changing only a single bit but not any other stored under the same parameter number. The safest way is to write down the original settings, make the required change, then compare the two.

The bit parameter entry (in all computer applications) has its own standard notation, in the way it appears on the display screen - note the numbering method for CNC units:

Parameter Number

#7	#6	#5	#4	#3	#2	#1	#0
----	----	----	----	----	----	----	----

There are three items of importance here - first item is the *parameter number* (graphically shown at the top), on the display screen usually shown to the left of the data (on the same line of display, depending on the control system). The second item is the *numbering* of the bit registers - it starts from *zero* to *seven*, *not* from one to eight. This is a standard computer method - computers start counting from 0, not from 1. The third item of importance is that the bit registers #0 to #7 are shown in *reverse* order, #7 being the first (left-most), and #0 being the last entry (right-most), when read from left to right. These three items are very important in human communications. For example, if the service technician suggests that you change the bit #2 of a particular parameter to such and such setting, you have to know *exactly* which register location it is - you have to speak the same language as the technician! This happens quite often when the conversation takes place on the phone, via fax, e-mail, or via similar means.

For example, the parameter #0000 for Fanuc 16/18/21 control systems uses only four of the available eight parameter registers (bits):

#0000

		SEQ			INI	ISO	TVC
0	0	0	0	1	0	1	0

If there is no heading, it means the bit is unassigned - it has no value, it does not exist. In the example, there are also four unassigned bits. If the bit is assigned, the two-line representation on the display screen of the control system shows abbreviated description of each relevant parameter (as a *bit* name). These descriptions are often hard to decode just by reading them, but they are well described and with good detail in the Fanuc manual that contains parameter descriptions and usage (*Parameter Manual*).

In the example above (parameter #0000), the four listed bit names represent settings for four different (and independent) bit-type parameters, related to settings:

SEQ **Automatic insertion of sequence numbers**
0: **Not performed**
1: **Performed**

INI	Unit of input
0:	Metric (millimeters)
1:	Inches (inches)
ISO	Code used for data output
0:	EIA code
1:	ISO code
TVC	TV check
0:	Not performed
1:	Performed

Based on the example entries of the parameter #0000, the shown bit settings for the illustrated control system are:

```

SEQ = 0    ... Automatic insertion of sequence numbers is not performed
INI  = 0    ... Unit of input is metric - mm
ISO  = 1    ... ISO code is used for output
TVC  = 0    ... TV check is not performed

```

Since only four registers - out of eight possible ones - are used (in Fanuc 16/18/21 controls) for the parameter setting of #0000, the remaining four registers are irrelevant, or to use a proper computer description, they are *unassigned*.

For comparison, the Fanuc 15 control system also uses parameter #0000 for settings, but with a totally different content (meaning of bits):

#0000

		DNC	EIA	NCR	ISP	CTV	TVC
--	--	------------	------------	------------	------------	------------	------------

#7 #6 #5 #4 #3 #2 #1 #0

The parameter #0000 shows the contents for the Fanuc 15 control system:

DNC	DNC operation with the remote buffer
0:	High speed distribution (HSD) enabled, if the HSD enable conditions are satisfied
1:	Not a high speed distribution but a normal distribution can be always performed
EIA	Punch code is
0:	ISO code
1:	EIA code
NCR	In ISO code, the end of block (EOB) code is punched as
0:	LF CR CR (LF = Line Feed, CR - Carriage Return)
1:	LF
ISP	Specifies whether the ISO code contains a parity bit
0:	ISO code with a parity bit
1:	ISO code without a parity bit

CTV	Specifies whether counting of characters for TV check is performed during control-out
0:	Check is performed
1:	Check is not performed
TVC	Specifies whether TV check is performed
0:	TV check is not performed
1:	TV check is performed

These descriptions of individual bits are taken directly from the Fanuc *Parameter Manual*. There is a very good chance that you may not understand some (or all) of the terms or functions in the descriptions. That is quite normal for a typical programmer or operator - only qualified service technicians should understand the many intricacies. It also leads to two important rules:

Always make sure you fully understand the exact meaning of each parameter

Do not change any parameter unless you know how it works

Relationship of Parameters

There are many cases when the setting of one system parameter is directly related to the setting of another system parameter. In such cases, there are *two or more* adjustments to different parameters to be done, in order to achieve a particular result. At other times, the setting of one parameter often *influences* meaning of another parameter. In the example shown below, such a situation does indeed exist. Let's explore it - it is a simple application and it is safe to try it on your own. The example is the same as shown earlier, for parameter #0000.

#0000

		SEQ			INI	ISO	TVC
0	0	0	0	1	0	1	0

Consider the parameter setting in the bit register #5 (SEQ), for Fanuc models 16/18/21. This bit sets the automatic insertion of sequence numbers into a part program, when entered directly into the control. In such a case, the part program is entered into the memory by using the keyboard of the control system (the slowest method of input, using only one finger typing, but not uncommon). This is typically useful only if you enter short CNC programs, using the control panel hard keys. If the parameter #5 is set to 1 (*Automatic insertion of sequence numbers is performed*), you can concentrate on the keyboarding of the program data only and forget the block numbers - they will be inserted automatically - because that is the purpose of the bit #5.

So far so good, the automatic block numbering will save some time - but there is one more small item to consider. What will be the actual increment of these sequence numbers? 1, 2, 5, 10? More? Less? Is there a default? Can it be changed? For many programmers, this is important.

Fanuc controls allow the use of *any* increment of the block numbers, in *any* order, so the possibilities are quite extensive. From the point of view of the software designer, no high end software should contain *assumed* (or preset) values. Such an approach would limit the user and make the software weak as a result. Fanuc engineers respect that view and allow settings of the block number increment by individual users. In fact, it is not enough just to say 'Set automatic insertion of sequence numbers' - we also have to say 'Set automatic insertion of sequence numbers with a particular increment'.

Parameter #0000 and bit #5, do not allow this setting at all. A *related parameter*, a parameter that is related to the #0000 and its bit #5 has to be used - and this parameter must contain the block increment amount. In the case of Fanuc 16/18/21 controls, the parameter is #3216, and is described in the Fanuc *Parameter Manual* as:

#3216
Increment in sequence numbers inserted automatically

In the above example, the block number increment can be set within the following limitations:

Data Type:	Word
Valid Data Range:	0 to 9999

The value of this parameter is the actual increment for the automatic block sequencing, within the range of 0 to 9999, defined as a *word type entry*. The word type entry will be discussed shortly. The selected amount will only be applied, if the bit #5 of parameter #0000 is set to 1, otherwise it will be ignored. This is a typical example of one parameter setting that is related to the setting of another system parameter. For different control systems (controls that claim Fanuc compatibility), the principle of setting will be the same, but the parameter numbers and the bit register numbers may be different. Always check the instructions for your control system.

Byte Data Type

The computer terms '*bit*' and '*byte*' have been already described somewhat briefly. These two words used in computing can be easily confused because they look similar. True, these words *are* similar, but they are *not* the same - they are unique words defined as *bit* and *byte*. Relating to the system parameters, the *bit type* parameter has already been described. The other type, the *byte type* system parameter accepts a range of values - from -127 to +127 for entries that require a signed value (plus and minus values), and the integer range from 0 to 255 for entries that do not require signed numbers. These ranges cover all eight bit entries, where each *byte digit* is the *bit*.

For example, many modern CNC machining centers are capable of the so called rigid tapping, rather than tapping using the floating tap tool holder. Most control systems will require a specific M-code for this function. This specific rigid tapping M-code is normally provided by the machine tool builder and must be interfaced to the control system - yes, you guessed it - *via a parameter setting*. The machine tool builder does all that. On Fanuc 16/18/21 controls for milling, the M-code for rigid tapping is specified by the parameter #5210:

#5210**Rigid tapping mode specification M code**

Data Type: Byte
Valid Data Range: 0 to 255

This parameter sets the M-code that specifies the rigid tapping mode. If the parameter **#5210** is set, for example, to 75, the rigid tapping mode in the CNC program will be activated by the miscellaneous function **M75**.

Only unassigned M-codes can be used and the machine tool design must be able to accept the rigid tapping mode (check the machine tool specifications). If the parameter **#5210** is set to the zero value (0), Fanuc 16/18/21 milling systems will assume that the entry is an arbitrary 29, programmed as **M29** - the default value.

Word Data Type

Another data type that applies to system parameters is the *word type*. The *Word Type* listing of a control system parameter covers the range from -32767 to +32767. This *long integer* range represents the sixteen bit data area of the parameter registry. The manufacturer's setting of parameter **#3772** (assigned the *Maximum Spindle Speed* function), illustrates the word type parameter:

#3772

8000

Data Type: Word
Unit Of Data: RPM
Valid Data Range: 0 to 32767

In the above example, the maximum spindle speed for a particular CNC machine tool has been set to 8000 rev/min. The maximum *rev/min* value is a typical setting that relates to the machine tool itself and represents a certain fixed machine tool specification. It also represents a setting that *is untouchable by the user!* It belongs to the machine tool only and it cannot be changed. Never attempt to change the spindle speed or feedrates, for example, because the appropriate settings are always at the discretion of the machine tool manufacturer.

2-Word Data Type

Two-word type parameter setting is similar to the *word type*, but accepts much larger values. In fact, the valid input values are between -99999999 and +99999999. An example that can be used to illustrate the 2-word system parameter type is for the setting of the maximum cutting feedrate - parameter **#1422** (*Specify the maximum cutting feedrate*):

Important Observations

From the previous explanations, some very useful conclusions can be drawn. Based on the system parameter storage entry type, all parameters can be separated into three general areas:

- ◆ **Binary codes**
- ◆ **Units input**
- ◆ **Setting values**

All features are equally important. Depending on the system parameter type, all parameters fall within a specified entry requirements, whether it is a single value or a range of values.

Each of the three parameter groups covers different input values or amounts. The *binary input* can only have an input of 0 or 1 for the *bit* data format, and 0 to ± 127 for the byte type. This covers the *bit* and *byte* sections. *Units input* section has a much broader scope - the units can be in either English or metric representations, expressed as a millimeter, inch, mm/min, inch/min, degree, millisecond, etc., depending on the kind of data entry and the parameter selection. A system parameter *value* can also be specified *within* a given range, for example a number within the range of 0 to ± 99 , or 0 to ± 99999 , or 0 to ± 127 , etc.

A typical example of a *binary input* parameter setting is a selection between *two available options*. For instance, a control feature called *dry run* can be set either as *effective* or *ineffective* for the rapid motion command - there are two possible and available system options. To select a particular preference, a predetermined bit number of a certain parameter would be set to 0 to make the dry run effective, and to 1 to make it ineffective. Non-Fanuc controls use similar inputs.

Units Input, for example, is used to set the *Increment System IS-x* (IS-A, IS-B, IS-C) - the system of dimensional units. Computers in general do not distinguish between inch and metric units, for example - to the computer, a number is a number. This applies equally to both the CNC and CAD/CAM software. It is up to the end user *and the parameter setting*, whether the control system will recognize the input of 0.001 or 0.0001 as the smallest motion increment or not. Another example of the units input would be a parameter setting that stores the maximum feedrate for each axis, the maximum spindle speed of the machine, rapid traverse rate and other features.

To illustrate the *setting value* of a control system parameter *within* a specified range, take a typical example of a CNC machine tool that has an indexing table, such as a general horizontal CNC machining center. A particular control system can be applied to a machine tool that supports one-thousandths of a degree increment, as well as to a machine that supports one degree increment or to a machine tool that supports five degree increments. Using the system parameter settings, the selected parameter can be set to the *smallest available* angle for the table indexing. Most users will have this parameter set to either 1 or 5 degrees as the least angle. For rotary axis (or advanced indexing tables), this setting will have a value of 0.01 or 0.001 of a degree. A parameter value cannot be set to a value that is lower or higher than the machine tool itself can support - these are physical limitations that should be considered at the time of machine tool purchase. For example, an indexing axis with the minimum increment of one degree, will *not* become a rotary axis with 0.001 increment, just because the system parameter is set to a lower value. This is an improper setting and will cause serious damage to the machine, if attempted!

Binary Numbers

In the preceding topics, the application of 0 and 1 entries was explained as a very common method of setting system parameter values. These are called the *bit settings* and are based on the system of *binary numbers*. Although many programmers have heard this term, not all understand its concept. In any CNC training program, binary numbers are not exactly the most appealing subject and are not generally covered. Strictly speaking, there is no need to know the binary numbers and how they work, but the knowledge does help in several special applications. Also, the subject of binary numbers may be interesting to those who would like to know more about it. The detailed description of binary numbers is beyond the scope of this handbook, but there are many excellent computer books describing the subject in detail. This section will only cover their main essence.

In everyday life, we use the decimal number system, which means we have ten digits available, from 0 to 9. The base of the decimal system is 10. In the decimal system, a number can be expanded, using the base of 10. For example, number 2763 can be represented as:

2	10³	7	10²	6	10¹	3	10⁰	2763
----------	-----------------------	----------	-----------------------	----------	-----------------------	----------	-----------------------	-------------

The base of binary system is not 10 but 2. For the purposes of a macro beginner, the binary system uses only the digits *0* and *1* (one choice out of two). The prefix *bi-* means 'of two'. Each symbol is known as a *bit* (**B**inary **digIT**). Many CNC system parameters are of the binary type. In computing, various components can only have two states - *ON* or *OFF*, *Open* or *Closed*, *Active* or *Inactive*, and several others. These *ON-OFF* states are represented by the digits *0* and *1*, in the system parameters of the bit type. In one of the earlier examples, a typical setting of such a parameter was shown. For example, in the following parameter, the 8-bit value is

00001010

which can be represented as:

0	0	0	0	1	0	1	0
#7	#6	#5	#4	#3	#2	#1	#0
2 ⁷	2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰
128	64	32	16	8	4	2	1

Note how each bit in the above example is represented by its own number (#0 to #7), its own exponential representation and its bit value. The sum of the bits is simple to calculate:

0	2⁷	0	2⁶	0	2⁵	0	2⁴	1	2³	0	2²	1	2¹	0	2⁰	10
----------	----------------------	----------	----------------------	----------	----------------------	----------	----------------------	----------	----------------------	----------	----------------------	----------	----------------------	----------	----------------------	-----------

In the chapter '*Automatic Operations*', an actual application of the binary numbers is shown in detail, relating to the subject of *mirror image status check*.

Setting and Changing Parameters

All control system parameters should be set or changed only by an authorized person, usually a qualified electronic service technician, an electrician, or an electronics professional. In all cases, always follow one simple rule relating to parameters:

Backup all system parameters and never modify their settings,
unless you are qualified **and** authorized to do so

As the general rule suggests, always keep a backup of the original system parameter settings. The backup should be stored on a disk, tape, or another computer and placed in a safe place - just in case. More than one copy kept at different locations enhances the quality of the safekeeping.

Protection of Parameters

Except a few system parameters related to setting, the majority of parameters cannot be accessed casually or by an accident. Parameters are automatically protected by the control system. In order to *set* or *change* a system parameter, the user has to *enable* the *Parameter Write* setting. This is usually done in the MDI mode, using the *SETTING* function on the control panel. Screen appears with the following line (Fanuc 16/18/21):

PARAMETER WRITE = 0 (0 : DISABLE 1 : ENABLE)

If the cursor is positioned on the currently set value, it can be changed, using the ON:1 soft key to enable parameter setting, or OFF:0 soft key to disable it. Note that while the setting is ON (changes enabled), the control system is in an alarm condition (non-operational). This is a normal and standard prevention from accidentally forgetting to disable the setting when finished. Some control systems require a user password to change parameters.

Other control systems, notably Fanuc 10/11/12 as well as Fanuc 15, use a special setting in parameter #8000, bit #0 (*PWE*). This parameter is of the *bit* type and accepts 0 or 1 input only. Setting the parameter to 0 (the normal state), will prohibit changes to the parameters, setting the parameter to 1 will allow the parameter setting changes.

For any CNC system, and in all cases when the parameter changes are enabled, the control system will enter into the *ALARM* state (error or fault condition). This is quite a normal situation - its purpose is *safety* - to guarantee that the machine tool cannot be operated while the system parameters can be changed.

Battery Backup

Another type of parameters protection is provided by the control manufacturer. It is called the *battery backup*. The system parameters (as well as many other settings) of the control system are maintained even when the main power to the machine tool is turned off (intentionally or by an accident), for example, during machine relocation.

The general intent is that a battery backup starts supplying the required power the moment the normal power to the CNC machine tool had been interrupted. Keep in mind that the power supplied by a backup battery system is only temporary. You should always have a current physical backup of all system parameters (as well as all other settings and the current part programs). The most common backup is downloading the settings to a disk, making two or three copies of the disk, printing out the file and storing each disk at a different location.

Changing Parameters

Many system parameters can be changed - this ability gives them the flexibility needed in a manufacturing environment. The method of setting or changing one or more control system parameter offers several choices:

- Through the external device, such as a tape or a disk file
- Through the control unit, using the *MDI* mode (Manual Data Input)
- Through the part program

The first and second methods are well described in various Fanuc manuals. The third method - *changing parameters through the part program* - is described in the next chapter in detail, under the subject heading of *programmable data input*.

Many control system parameters are periodically updated during program processing. The CNC operator, or even the CNC programmer, is often not aware that this activity is going on at all. There is no real need to monitor such activity. The safest rule to adhere to is that once the parameters have been set by a qualified CNC service technician, any temporary changes can - *and should* - be done through the CNC program itself. If permanent changes are required, only a qualified *and* authorized person should be assigned to do them - *and nobody else*.

System Defaults

Many control system parameter settings that are already in the control unit at the time of machine purchase have been entered by the machine tool manufacturer (or the vendor in some cases) as the *exclusive choices*, the *most suitable* choices, or the *common* selections. That does not mean these settings will become *your* customized settings - it only means that they were selected on the basis of their *common usage*. Quite a few changeable settings are rather conservative in their values, for various safety reasons. For example, the built-in clearance for the **G83** peck drilling cycle may be 0.5 mm or 0.02 inches. On Fanuc 16/18/21 controls, the clearance value is set in the parameter **#5115** - *Clearance of canned cycle G83*. Parameter **#6211** controls the same clearance on Fanuc 15 controls. This is a word-type parameter entry, that is usually set to 0.5 mm or 0.02 inches. Whatever the actual value is, it may never be known to the programmer or machine operator. The obvious reason is that the value is already *built-in* inside of the control system, as an initial or *default* value. It is established by the machine tool manufacturer as the most suitable general value, often taken from various surveys of end users. Such a value, established by the machine tool builder, is normally called the *default* value. In other words, the settings established by the factory (the machine manufacturer) are called the *default* settings that become active when the control system power is turned ON.

Default Values Settings

The English word '*default*' is a derivative of a French word '*defaut*', that can be loosely translated as '*assumed*'. When the main power to the control unit is turned *ON*, there are no set values passed to the parameters from any program, since no program has yet been processed. However, certain settings become active automatically, *without* any external CNC program. For instance, a cutter radius offset mode is automatically canceled at the startup of the control system. Also canceled are the fixed (canned) cycle modes or the tool length offset. The control system '*assumes*' that certain conditions are preferable to others. Most CNC machine tool operators will probably agree with the majority of these initial settings, although not necessarily with all of them. Many of these settings are customizable by a simple change in the parameter settings. Such settings will become permanent, and become the *new* 'default'.

We know that a computer is only a machine. It is fast and it is accurate, but it has no intelligence in human terms. Even artificial intelligence is just that - artificial. A human being, on the other side, is slow, makes mistakes, but has a unique ability - *human being thinks*. A computer is a very sophisticated machine and as such, it does not *assume* anything - computer does not consider, computer does not feel, computer does not think. Veteran computer users know that a computer does not do anything that a human ingenuity has not placed into it in the first place, during the design and development process.

In the terms of a CNC system operation, when its main power is turned on, the internal software automatically sets *certain* preprogrammed parameters to their default condition, as designed and decided by a human engineer. In the last sentence, the key word was '*certain*'. Not *all* system parameters, only *certain* parameters can have an assumed condition - a particular condition that is known as the *default value* or the *default condition*.

Consider an example that may be both practical and easy to understand at the same time. A tool motion can have three common modes - it can be a *rapid* motion, a *linear* or a *circular* motion. The default setting of such a motion is required within the control parameters. Which one of the three modes should be chosen? The rapid motion, the linear motion, or the circular motion? Only one of them can be active at the same time - but which one? The answer depends on the parameter settings. Many parameters have an option to be preset to a desired state. In this example, either the *rapid* or the *linear* mode will be effective automatically when the control power is turned on. Circular mode is definitely not practical under any circumstances.

To answer the question of '*which one*', look at the consequences of a choice. Many machines are sold with the *linear* motion - **G01** command - as a default, for safety reasons. When the machine axes are moved manually, the system parameter setting has no effect. If an axis command is entered manually, either through the program or in the MDI mode, a tool motion will happen. If the motion command is not specified, the system will use the command mode that had been preset as the *default* in the control parameters. Since the assumed mode, the default mode is, in this case, a linear motion - **G01** - an error condition will result. Why? Because there is *no cutting feedrate* in effect, which the **G01** does require. Had the default setting been the rapid motion **G00**, there would be a rapid motion, creating somewhat more risky situation in many applications, as it does not require feedrate designation - the rapid rate is controlled internally. Both, the CNC programmer and the CNC operator should be aware of the default settings of each control unit in the machine shop. Unless there is a compelling reason to do otherwise, defaults for several control units should be set identically, if possible, to create a consistent working environment.

5

DATA SETTING

In smaller machine shops, job shops or any other environment where stand alone CNC machines are used, the machine operator typically sets all offset values that have to be input into the control during the job setup, manually by simply typing them in the proper registers. This common method is very useful when the programmer *does not know* the setting values - in fact, this is the normal situation in most shops that the programmer or machine operator does not know the *actual* values of various offsets at the time of programming or machine setup.

Understanding the subject of offset data setting is important for many macro programs

Input of Offsets

In a manufacturing environment that has to be very tightly controlled during production, for example in agile and automatic manufacturing, or manufacturing of the same parts in very large volumes, the manual offset data entry during setup is very costly and inefficient. Also, this method does not provide an efficient means of adjusting the offsets values, for example, for tool wear. An agile or large volume manufacturing uses modern tools such as CAD/CAM systems for design and toolpath development, concept of multiple machine cells, robots, preset tools, automatic tool changing and tool life management, tool breakage detection, pallets, programmable auxiliary equipment, machine automation, and so on. Unknown elements cannot exist in such environment - relationships of all reference positions *must always be known*, and the need for offsets to be established and set at each individual machine is eliminated. All the initial offset values are - *and must be* - always known to the CNC programmer, well before the actual setup takes place on the CNC machine.

There is a great advantage in such information being known and used properly - the initial offset data can be included in the part program and be channeled into appropriate offset registers through the program flow. There is no operator's interference and the machining process is fully automated, including the maintenance of tools and related offsets. All offset settings are under the program control, *including their updates* required for position change, or a tool length change, or a radius change and other similar changes. The offsets are adjusted and updated from the information provided by an *in-process gauging system* that must be installed on the machine and interfaced with the control system.

All this automation is possible with several programming aids, often available as an optional feature of the control system. The main aid is the use of user macros and a feature called *Data Setting*. Before you can handle macros for the purpose of offset setting and adjustment, you have to understand the concept of the data setting. As a matter of fact, the control unit may have the data setting feature without even having the macro option. Don't get discouraged without trying it first. Even a small machine shop with a single stand alone CNC machine can benefit from this feature, if it is available. Fanuc control systems use a special preparatory command for data setting - **G10**.

Data Setting Command

To select the data setting option and to set offset data through the program, Fanuc controls offer a special preparatory *G* command:

G10	Data Setting (Programmable Data Input Function)
------------	---

In its basic form, this uncommon preparatory command **G10** is a *non-modal command, valid only for the block in which it is programmed*. If it is required in any subsequent blocks, it has to be repeated in every block.

By itself, the **G10** command does not do anything at all. It requires additional input. It has a simple format that is different for the CNC machining centers and CNC lathes. There are also some minor differences between formats of different Fanuc controls or Fanuc compatible controls, although the programming methods are logically identical. Formats also vary for the three different types of the offsets - the *work coordinate offset*, the *tool length offset*, and the *cutter radius offset*. The examples in this section are for typical Fanuc models and have been tested on Fanuc 16 MB (they apply to milling and turning control). They will work on many other control models as well.

Coordinate Mode

Selection of the absolute (**G90**) or incremental (**G91**) programming mode has a great impact on the input of all offset values using the CNC program with the **G10** command.

The **G90** or **G91** can be set anywhere in the program, changed from one to the other, as long as the block containing the required command is assigned *before* the **G10** data setting command is called. All types of available offsets can be set through the program, using the **G10** command:

- Work offsets** . . . **G54 to G59** (*and additional sets, if available*)
- Tool length offsets** . . . **G43 or G44** (*G49 to cancel*)
- Cutter radius offsets** . . . **G41 or G42** (*G40 to cancel*)

Absolute Mode

Absolute mode of programming uses the **G90** preparatory command for the milling controls and **XZ** addresses for turning controls. In the absolute mode, any one of the three programmed offset values *will replace* the offset value stored in the CNC system.

Incremental Mode

Incremental mode of programming uses the **G91** preparatory command for the milling controls and **UW** addresses for turning controls. In the incremental mode, any one of the three programmed offset values *will not replace* but only *update* the offset value stored in the CNC system.

Selection of ABSOLUTE or INCREMENTAL mode will affect data setting

All three offset groups can be used in macro programs using the data setting function. It is very important to understand how each offset group behaves in all conditions. If necessary, review subjects relating to the concept of the work offsets, tool length offset and cutter radius offset (see *Chapter 24* at the end of this handbook for suggestions).

Work Offsets

Standard Work Offset Input

The standard six work offsets **G54** to **G59** are generally available for both the milling and turning control systems. However, due to the unique machining requirements, they are normally associated with the milling controls. Their programming format is the same for both control types:

G10 L2 P- X- Y- Z- *Machining centers*
G10 L2 P- X- Z- *Turning centers*

The *L2* word is a *fixed mandatory offset group number* that identifies the offset input type as the *work coordinate setting*. The *P* address in this case can have a value from 1 to 6, assigned to the **G54** to **G59** selection respectively:

P1 = G54, P2 = G55, P3 = G56, P4 = G57, P5 = G58, P6 = G59

for example,

G90 G10 L2 P1 X-450.0 Y-375.0 Z0

will input X-450.0 Y-375.0 Z0 coordinates into the **G54** work coordinate offset register (all examples for this section are in millimeters).

G90 G10 L2 P3 X-630.0 Y-408.0

will input X-630.0 Y-408.0 coordinates into the **G56** work coordinate offset register. Since the Z-value has not been programmed, the *current* value of the Z-offset will be retained.

All examples above show the absolute programming mode **G90** (or *XZ*), where the offset setting in the control will be *replaced* with the one provided in the program. In incremental mode **G91** (or *UW*), the current offset setting will be *updated*:

G90 G10 L2 P1 X-450.0 Y-375.0 Z0
 (...MACHINING CONTINUES...)
G91 G10 L2 P1 X5.0

will first input X-450.0 Y-375.0 Z0 coordinates into the **G54** work offset, but after some machining, only the X-offset will be updated by adding 5 mm to its current value, to X-445.0.

Additional Work Offset Input

In addition to the standard six work coordinate settings for milling and turning controls, Fanuc offers an optional set of many additional offsets as an *extension* of the standard work offsets **G54** to **G59**. This option adds another forty eight work offsets (**G54.1 Px** or **G54 Px**), for the total of fifty four work offsets (**x=1 to 48**)! If so many work offsets seem excessive, just consider a complex job on a CNC horizontal machining center, where work offset may change with each axis indexing. These additional work offsets are identified in the CNC program by using the command **G54.1** or **G54** along with the address **P** - compare these three examples:

```
N3 G90 G54 G00 X50.0 Y75.0 S1200 M03           Uses the standard G54 offset
N3 G90 G54.1 P1 G00 X50.0 Y75.0 S1200 M03     Uses the first additional offset
N3 G90 G54 P1 G00 X50.0 Y75.0 S1200 M03      Exactly the same as the previous example
```

Note that for additional offsets, only the **G54.1** or **G54** command can be used, not any other. Whether to use **G54.1 P-** or **G54 P-** depends on the control - try them both to find out.

The **G10** data setting command can also be used to input offset values to any one of the 48 additional work offsets and the programming command is very similar to the previous one, except **L20** is used instead of the **L2**:

```
G10 L20 P- X- Y- Z-
```

Only the fixed mandatory offset group number has changed to another fixed offset group number, **L20**, which specifies the selection of one of the *additional* work offsets.

Do not confuse the address 'P' in the G10 block with the address 'P' in the G54 block !

External Work Offset Input

Another offset that belongs to the work coordinate system group is called either *External* or *Common*. This offset cannot be programmed normally, using any standard G-code. It can be set manually in the same work offset area as the **G54+** offsets - *at the control only* - it is the first offset on the screen, marked **00**, and either with the letters **EXT** or **COM**. The latest controls use the **EXT** letters, to eliminate confusion with letters related to communications settings. *External* (*Common*) offsets are always *added* to *all* other work offsets specified in the CNC program. With the **G10**, the external offset can be programmed, and it is normally used to update *all* work coordinate offsets at the same time (globally). This is a much more efficient way of updating all used work offsets by the *same* amount.

The programming format for the **G10** input of settings into the external offset uses the **L2** offset group and **P0** as the offset selection:

```
G90 G10 L2 P0 X-10.0
```

This program entry will place X-10.0 into the *external* work coordinate offset, while retaining all other settings (the Y-axis, the Z-axis, and any additional axes as well). As a result of this update, each work coordinate system used in the active CNC program will be shifted by 10 mm into the X-negative direction.

The other two offset groups, one for the tool length, the other for the cutter radius, can also be set by the **G10** command. However, there is an additional subject you have to fully understand, before using the **G10** command for those two offset groups. The subject covers the *memory types*.

Offset Memory Types - Milling

During the development of more advanced CNC technology, Fanuc controls have introduced three, progressively more advanced, types of memory to store tool length and tool radius offsets. The three stages are known as the *Memory Type A*, *Memory Type B*, and *Memory Type C*, referring to the *type of offset memory*.

They have these characteristics on the display screen of the control system:

Memory Type	Characteristics
Type A	One display column, shared by the length and radius offsets
Type B	Two display columns, one for the length offset and one for the radius offset
Type C	Four display columns, two for the length offset and two for the radius offset

In offsets are applied in macro programs, we deal with two important criteria - the size of the tool (length or radius) and the amount of wear on the tool (length change or a radius change). The terms *Geometry Offset* and *Wear Offset* are used frequently in this application, and may require some clarification.

Geometry Offset

For the *tool length*, the geometry offset stores the actual preset length of the tool, or the actual amount measured during setup. In a program, the length offset is called with the address H.

For the *tool radius*, the geometry offset stores the known (nominal) radius of the cutter (typically one half of the specified diameter). In a program, the radius offset is called with the address D but can also be called with the address H.

Wear Offset

As the name suggests, the wear offset is the *amount of deviation* from the measured value (*i.e.*, from the geometry offset). Do not take the name literally. Wear means tool wear, yes, but it also means an amount of deviation for *any other reason*, for example, a deviation due to resharpening of the tool or due to cutting pressures.

Which Offset to Update?

When a tool length or a tool radius offset value needs an adjustment for various reasons, the question is *which* offset to adjust. This question needs no answer for the memory *Type A*, but the answer is important for memory *Type B* and *Type C*. Both of these types offer *two* groups of adjustment, one for geometry, the other for wear.

➤ This example illustrates the offset update options (memory *Type B* is used):

In the program, tool length offset number *H03* is used. In the control system, the *Geometry* setting for offset 03 is -198.000 and the *Wear* setting is 0.000. The 198 mm tool-to-part distance has either been measured at the CNC machine or entered from a presetting device. This distance represents the initial conditions of the tool setup, the 'normal' condition. Since it represents the relationship between the machine and the cutting tool, it should be entered into the *Geometry* offset. Virtually all CNC operators will do that as a matter of routine.

While the tool provides correct cutting depth for many parts, eventually it may need a slight adjustment, because the depth becomes a bit shallow. The tool itself does not need sharpening, but the length offset needs an adjustment of 0.125 mm, in order to compensate for the shallow depth. The tool has to be forced to move down, into the material, by additional 0.125 mm. There are two options the CNC operator can choose from - either the *Geometry* offset is changed from -198.000 to -198.125 *or* the *Wear* offset is changed from 0.000 to -0.125. The result will be exactly same with either method - and hence the question - *which offset to update?*

Many CNC operators have the habit of changing (updating) the *Geometry* offset all the time, leaving the *Wear* offset intact and ignored. In automated machining, particularly when working with custom macros, this is the most undesirable approach, because the original setting - *the one that was physically measured* - the original offset value, will be lost. As a rule, *always* make adjustments to the *Wear* offset and leave the *Geometry* offset alone. This way, both values are preserved and offset settings are better controlled.

NOT RECOMMENDED			RECOMMENDED		
OFFSET NUMBER	GEOMETRY	WEAR	OFFSET NUMBER	GEOMETRY	WEAR
01	01
02	02
03	-198.125	0.000	03	-198.000	-0.125
04	04
05	05
...

Figure 9

Effect of adjustment to the *Geometry* and *Wear* offsets

The comparison between two offset entries is shown above (for memory *Type B*) - *Figure 9*.

In custom macros, selecting the wrong offset to be updated can cause serious problems

Memory Type A

This is the oldest memory type that reflects CNC technology of its time. Any length and radius offset setting is placed in the *same column* of offsets - the *same* register area of the control system. For the memory *Type A*, there is only a single column of offsets available - see *Figure 10* :

OFFSET NUMBER	GEOMETRY & WEAR
01	0.000
02	0.000
03	0.000
04	0.000
05	0.000
...	...

Figure 10

Offset memory *Type A*

Geometry and *Wear* offsets are shared in a single column.

Tool length and tool radius must use different offset numbers

Ideally, each program address should represent a unique meaning. That is not always possible, and this a good example of such a situation. When both tool length and tool radius offsets are included in the same program for a certain tool (a common occurrence), each should use a different address (letter). That also requires two registers to store the offset data in the control system. With a single memory registry, only one column offset memory is available (called *Type A*). The result is that the tool length offset *and* the tool radius offset will both use the H-address, with *two different offset* numbers. Several control models do allow the use of the H-address for the tool length offset number and the D-address for the tool radius offset number, but they cannot have the same number. Typically, the numbers are arbitrarily shifted by a certain amount, for example, by 25 or 50, depending on the total number of offsets available in the registry.

► Examples:

Tool T04 uses H04 for tool length and H54 for the tool radius offset (only H can be used):

```
G43 Z2.0 H04           Offset number 04 used for tool length - H address
...
G01 G41 X123.0 H54 F275.0   Offset number 54 used for tool radius - H address
```

Tool T04 uses H04 for tool length and D54 for the tool radius offset (both H and D can be used):

```
G43 Z2.0 H04           Offset number 04 used for tool length - H address
...
G01 G41 X123.0 D54 F275.0   Offset number 54 used for tool radius - D address
```

The selection of the actual offset number is the programmer's decision - most programmers choose the tool number *and* tool length offset number the same, for convenience. The offset memory *Type A* is still very common, not only on older machine tools, but on newer machine tools using the controls with limited features, such as Fanuc 0 model controls. Always check if H and D can be used in the same program, when only *Type A* offset memory is available at the control.

Memory Type B

The memory *Type B* has been a great improvement on the *Type A*, because it separates the geometry and the wear offsets into separate columns, for better offset management. Still, it does *not* separate the tool length and the tool radius entries into separate columns. *Figure 11* shows two columns for the offset entry of *Type B* offset memory, sharing the *same* offset number:

OFFSET NUMBER	GEOMETRY	WEAR
01	0.000	0.000
02	0.000	0.000
03	0.000	0.000
04	0.000	0.000
05	0.000	0.000
...

Figure 11

Offset memory *Type B*

Geometry and *Wear* offsets are separated into two columns.

Tool length and tool radius must use different offset numbers

Because of the separation of *geometry* and *wear* offsets into two columns, the control of offset data entries is somewhat simplified. At the same time, the actual programming input still does require two different offset numbers for the same tool - H-offset number and D-offset number.

The real benefit of this type of offset memory is more oriented towards the CNC operator, rather than the CNC programmer. At the machine, the CNC operator can make changes to the *wear* offset, without disturbing the *geometry* offset.

The programming structure itself (the program input) is exactly the same as for *Type A*:

➤ Examples:

Tool T04 uses H04 for tool length and H54 for the tool radius offset (only H can be used):

```
G43 Z2.0 H04           Offset number 04 used for tool length - H address
...
G01 G41 X123.0 H54 F275.0   Offset number 54 used for tool radius - H address
```

Tool T04 uses H04 for tool length and D54 for the tool radius offset (both H and D can be used):

```
G43 Z2.0 H04           Offset number 04 used for tool length - H address
...
G01 G41 X123.0 D54 F275.0   Offset number 54 used for tool radius - D address
```

The offset numbering method follows the same logic as for *Type A* (after all, it is almost identical). The part programmer decides the most convenient (comfortable) method. In all cases of offset numbering, it is important to carefully select the programming method first, then follow it consistently from one program to another, by all programmers in the company.

Memory Type C

The latest (and the greatest) type of offset memory is the *Type C*. It is a much improved input method, based on *Type B* - it offers extreme programming control and flexibility during machine operation. *Type C* offset memory contains geometry and wear offset registers that are *independent* of each other - they are *different* for the tool length offset *and* the tool radius offset data. In normal practice, that means the total of *four* display columns on the control system screen - *Figure 12*.

OFFSET NUMBER	H		D	
	GEOMETRY	WEAR	GEOMETRY	WEAR
01	0.000	0.000	0.000	0.000
02	0.000	0.000	0.000	0.000
03	0.000	0.000	0.000	0.000
04	0.000	0.000	0.000	0.000
05	0.000	0.000	0.000	0.000
...

Figure 12

Offset memory *Type C*

Geometry and *Wear* offsets are separated into two columns for each offset type. Tool length and tool radius may use the same offset numbers.

➤ Example:

Tool T04 uses H04 for tool length and D04 for the tool radius offset (both H and D can be used):

```
G43 Z2.0 H04           Offset number 04 used for tool length - H address
...
G01 G41 X123.0 D04 F275.0   Offset number 04 used for tool radius - D address
```

This is the most advanced method of CNC programming of length and radius offsets, because the control systems that support it - offer convenience and flexibility to both, the CNC programmer and the CNC machine tool operator. There is no need to *shift* one offset number by 25 or 50 numbers - they are both the same, both using *its own* offset register. There is no need to worry whether to use the H-address or the D-address - in *Type C* offset memory, the H-address will always be used for the tool length offset, the D-address will always be used for the radius offset - and can both have the *same* offset number.

Memory Type and Macros

It is very important to understand the offset memory types, because when writing a custom macro program that uses either the tool length or the tool radius offset (or both), the macro will have to reflect the differences of each offset type. That also means a particular macro will not be transferrable to a different machine control system, unless it also includes a multiple choice that covers the available offset memory types.

Offset memory type determines the macro structure for length and radius offsets

Offset Memory Types - Turning

There are only *two* types of offset memory available for the Fanuc turning controls (those used for CNC lathes and turning centers) - the *Type A* and the *Type B* memory offsets - they are similar to their milling counterparts with a few changes. As there is no tool length offset equivalent for turning applications, there is also no *Type C* offset memory available on turning controls. The memory *Type A* is considered old and impractical, particularly for lathe macros - it lacks many important features and is present on some old lathes only. Most CNC lathes and turning centers today use the offset memory *Type B*, which occupies *two* distinct screens on the CRT display. One screen is the *Geometry* offset setting, the other is the *Wear* offset setting. They are very similar, and CNC operators often make the mistake of inputting data into the wrong screen (as often happens on milling controls as well). All control models remind the user of the active screen displayed by the words *GEOMETRY* or *WEAR* at the screen top, and some controls even precede the offset number with the letter *G* for *Geometry* offset number and the letter *W* for the *Wear* offset number. As always, make sure to check the control system manual for relevant details - changes in the control software happen fairly frequently and are usually included only in the latest control models - controls come in many varieties, some rather very small, yet important.

In the next illustration - *Figure 13* - a typical screen view displays the contents of the *Geometry* offset for a CNC lathe unit, using the *Type B* offset memory settings.

GEOMETRY OFFSET NUMBER	X-AXIS GEOMETRY OFFSET	Z-AXIS GEOMETRY OFFSET	TOOLNOSE RADIUS GEOMETRY	TOOL TIP NUMBER
G 01	0.000	0.000	0.000	0.000
G 02	0.000	0.000	0.000	0.000
G 03	0.000	0.000	0.000	0.000
G 04	0.000	0.000	0.000	0.000
G 05	0.000	0.000	0.000	0.000
...

Figure 13

Lathe offset memory *Type B*
... GEOMETRY

The *Tool Nose Radius* is normally identified by the letter *R*, and the imaginary *Tool Tip Number* is identified by the letter *T*

A virtually identical screen display - *Figure 14* - shows the *Geometry* offset screen. It contains the lathe *Wear* offset screen, also using the *Type B* offset memory setting.

WEAR OFFSET NUMBER	X-AXIS WEAR OFFSET	Z-AXIS WEAR OFFSET	TOOLNOSE RADIUS WEAR	TOOL TIP NUMBER
W 01	0.000	0.000	0.000	0.000
W 02	0.000	0.000	0.000	0.000
W 03	0.000	0.000	0.000	0.000
W 04	0.000	0.000	0.000	0.000
W 05	0.000	0.000	0.000	0.000
...

Figure 14

Lathe offset memory *Type B*
... WEAR

The *Tool Nose Radius* is normally identified by the letter *R*, and the imaginary *Tool Tip Number* is identified by the letter *T*

Adjusting Offset Values

A brief mention of how the currently set coordinate modes influence any offset setting, was already mentioned earlier in this chapter. During a typical CNC machine operation, the operator regularly adjusts the current offset values. This adjustment is based on the result of a reliable measurement, regardless of the reason that may have caused the adjustment in the first place. The same adjustment can be done automatically, using macros and some additional machine hardware.

The most important key to understanding the offset value adjustment is the mode of data entry - is it *absolute* or is it *incremental*? Both offer certain advantages, and it is very important to understand them well, particularly for macro program development.

In the following few paragraphs, there are several examples that will illustrate the differences.

Absolute Mode

In the *absolute* mode of programming (**G90** for milling or **XZ** for turning), the selected offset will be *REPLACED* by the entered value in the program (or manually, at the control).

➤ Absolute mode example	(G90 or XZ):
Current offset value:	345.000
Input value:	350.000
New offset value:	350.000

If you make a mistake, the original setting is lost forever - be careful. Remembering the original value may be difficult, but writing it down *before* the change may prevent many problems.

Incremental Mode

In the *incremental* mode of programming (**G91** for milling or **UW** for turning), the selected offset will be *UPDATED* by the entered value in the program (or manually, at the control).

➤ Incremental mode example	(G91 or UW):
Current offset value:	345.000
Input value:	5.000
New offset value:	350.000

In case of an input error, the odds are little better for incremental input than absolute input of offsets. As long as you remember the incremental value used for the offset adjustment, you can always revert to it, in case of an erroneous input.

Macro programs will accept either method of coordinate input and will behave exactly the same, as if the part program were written in the so called 'normal' mode, without using macro statements. Keep in mind that macro programming only amplifies the slow manual process - it uses the same logical tools and procedures available in manual programming.

Tool Offset Program Entry

Tool length offset value for a CNC milling control system that supports the **G10** command, can be programmed using the **G10** command together with the L offset group. Depending on the memory type a particular control system offers, the L offset group may have a different number. The three types of offset memory on Fanuc controls, for the tool offsets (length and radius), have different entries and are illustrated in the following examples:

◆ Memory A (only one column for length offset and radius offset)

Offset entry:	Combined Geometry + Wear offset amount	Program
entry:	Offset amount set by G10 L11 P- R- block	

◆ Memory B (two columns for length offset and radius offset)

Offset entry 1:	Separate Geometry offset amount	Program entry 1:
Offset amount set by G10 L10 P- R- block		

Offset entry 2:	Separate Wear offset amount	Program entry 2:
Offset amount set by G10 L11 P- R- block		

◆ Memory C (two columns for length offset and two columns for radius offset)

Offset entry 1:	Separate Geometry offset amount - applied to H-address
Program entry 1:	Offset amount set by G10 L10 P- R- block

Offset entry 2:	Separate Geometry offset amount - applied to D-address
Program entry 2:	Offset amount set by G10 L12 P- R- block

Offset entry 3:	Separate Wear offset amount - applied to H-address
Program entry 3:	Offset amount set by G10 L11 P- R- block

Offset entry 4:	Separate Wear offset amount - applied to D-address
Program entry 4:	Offset amount set by G10 L13 P- R- block

L-Address

In all examples illustrated, the L address number is the arbitrary (fixed) offset group number (that means it is fixed within the particular Fanuc control system by the manufacturer), and the P address is the offset register number (used by the CNC system) - the R value is the actual amount of the selected offset to be transferred *into* the selected offset number registry. Absolute and incremental modes have the same effect on the tool length and radius programmed input, as for the work offset, described earlier in this chapter. Some additional practical examples of various offset data settings, using the **G10** preparatory command, should illustrate this topic even further.

NOTE: Older Fanuc controls used the address *L1*, instead of the newer *L11*. These controls did not have a wear offset as a separate entry. For the compatibility with several older controls, *L1* is accepted on all modern controls in lieu of *L11*.

G10 Offset Data Settings - Milling Examples

This section illustrates some of the common examples of **G10** offset data settings used in a program (standard or macro) for CNC machining centers. Block numbers are used for convenience.

➤ Example 1:

Block N50 will input the amount of negative 468.0 mm into the tool length offset register 5:

```
N50 G90 G10 L10 P5 R-468.0
```

➤ Example 2:

If this offset needs an adjustment to cut *0.5 mm less depth*, using the tool length offset 5, the **G10** block will have to be changed to incremental mode:

```
N60 G91 G10 L10 P5 R0.5
```

Note the **G91** incremental mode - if the blocks N50 and N60 are used in the listed order, then the registered amount of offset number 5 will be -467.5 mm.

➤ Example 3:

For memory *Type C*, the cutter radius value D may be passed to the selected offset register from the CNC program, using the **G10** command with *L12 (geometry)* and *L13 (wear)* offset groups:

```
N70 G90 G10 L12 P7 R5.0 ... inputs 5.0 mm radius amount into the geometry offset register 7
```

```
N80 G90 G10 L13 P7 R-0.03 ... inputs -0.03 mm radius amount into the wear offset register 7
```

The combined effect of the two entries will be the equivalent of cutter radius 4.97 mm.

➤ Example 4:

To increase or decrease a stored offset amount, use the incremental programming mode **G91**. The example in block N80 will be updated, by adding 0.01 mm to the current wear offset amount:

```
N90 G91 G10 L13 P7 R0.01
```

The new setting in the *wear* offset register number 7 will be 0.02 mm and the combined effect of both offsets number 7 will be the equivalent of cutter radius 4.98 mm (after blocks N70, N80 and N90 were processed). Always be careful with the **G90** or **G91** modes - it is recommended to reinstate the proper mode immediately after use, for any subsequent sections of the program.

Valid Input Range

On most CNC machining centers, the range of the *tool length* or the *tool radius* offset values is limited to a range specified by the Fanuc control. The input range of the offset input values is very wide and sufficient for all work. Note that the metric and English amounts vary only by a shifted decimal point, not by actual units conversion:

Offset Input	Lowest amount	Highest amount
GEOMETRY - Metric	-999.999 mm	+999.999 mm
WEAR - Metric	-99.999 mm	+99.999 mm
GEOMETRY - English	-99.9999 inches	+99.9999 inches
WEAR - English	-9.9999 inches	+9.9999 inches

The number of available offsets in the control system is also limited, with a typical minimum number of offset usually no less than 32. Optionally, the CNC system can have 64, 99, 200, and 400 offsets available - and more - most of them as a special option. It is important that to know the maximum number of offsets for each control system in the shop. As a simple rule, there will be (or should be) always more offsets available than the maximum number of tools the machine has.

Lathe Offsets

Tool length offset is a feature found on CNC machining centers and normally does not apply to the CNC lathes, because of a different tool and offset structure. Most CNC lathes use the *Group A* of G-codes (XZR for absolute input, UWC for incremental input) for programming and data setting. For such a CNC lathe, the **G10** command can be used to set offset data, using the following program formats (one or more axis specification):

G10 P- X- Y- Z- R- Q- *Absolute mode of programming*
 G10 P- U- V- W- C- Q- *Incremental mode of programming*

☞ where ...

G10 Programmable data input command
 P Offset number to set (P + 10000 = geometry, P + 0 = wear)
 X Absolute value of the offset register - X-axis
 U Incremental value of the offset register - X-axis
 Y Absolute value of the offset register - Y-axis (if available)
 V Incremental value of the offset register - Y-axis (if available)
 Z Absolute value of the offset register - Z-axis
 W Incremental value of the offset register - Z-axis
 R Absolute value of the offset register - tool nose radius
 C Incremental value of the offset register - tool nose radius
 Q Tool tip number for radius offset (imaginary tool tip number)

P-Offset Number

Note the comment next to the P-address - the P-address represents either the *geometry offset* number or the *wear offset* number to be set. To distinguish between the *geometry offset* and the *wear offset*, the *geometry offset* number is increased by an arbitrary amount of 10000:

➤ P10001 ... represents the selection of geometry offset number 1

➤ P10012 ... represents the selection of geometry offset number 12

If the 10000 value is *not* added, the P-address represents the number of the *wear offset*:

➤ P6 ... represents the selection of wear offset number 6

➤ P11 ... represents the selection of wear offset number 11

The number of available offsets depends on the control system. For example, Fanuc 16/18/21 controls have up to 64 offsets available.

Tip Number Q

The imaginary tool tip number (sometimes called the *virtual tip number* or just *tip number*) has been arbitrarily defined by Fanuc. For the rear type CNC lathes, the *Figure 15* illustrates the fixed numbers assigned to nine possible tool tips:

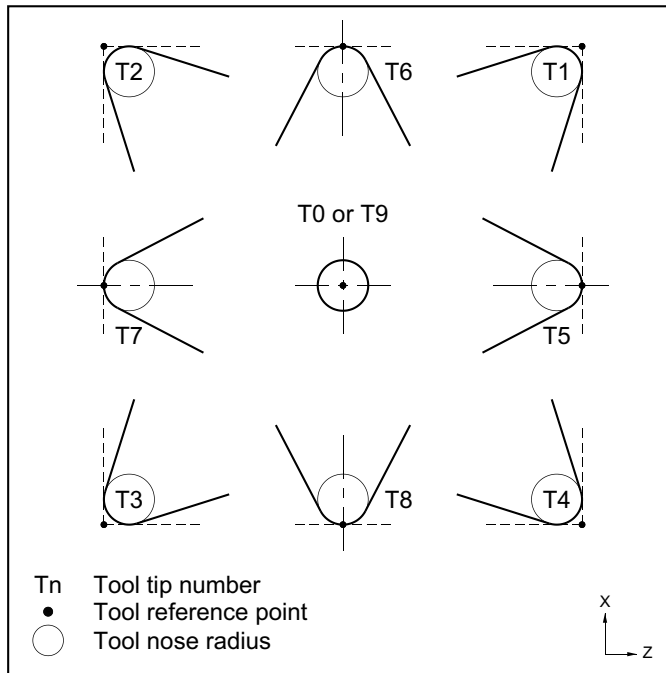


Figure 15

Tool nose tip numbers 0-9, programmed in the Q-address of the G10 command

The above tip numbers refer only to the *rear* lathe type, as shown by the axis orientation symbol.

G10 Offset Data Settings - Turning Examples

Data setting for lathe controls has a little different structure, but the control system processes it with the same built-in logic. Typical examples of offset data setting for a CNC lathe are shown, along with expected results. The listing is based on the *order of input* in the program:

➤ Example 1:

Block N10 will clear all *geometry* offsets for *G 01* group (the geometry offset 1 register)

```
N10 G10 P10001 X0 Z0 R0 Q0
```

Block N20 will clear all *wear* offsets for *W 01* group (the wear offset 1 register)

```
N20 G10 P1 X0 Z0 R0 Q0
```

☞ Using Q0 in G10 block will cancel BOTH tool GEOMETRY and tool WEAR tip numbers

➤ Example 2:

Block N30 will set the contents of *G 01* group to **X-200.0 Z-150.0 R0.8 T3**. It will also set the tool tip number 3 in the *wear* group *W 01* - automatically !

```
N30 G10 P10001 X-200.0 Z-150.0 R0.8 Q3
```

➤ Example 3:

Block N40 will set the *wear* group *W 01* tool nose radius to the value of 0.8, while the current tool tip number 3 is still active (not listed means not changed):

```
N40 G10 P1 R0.8 Current tool tip number setting IS assumed
```

In professional programming, it is much safer to program *without* assumptions of current values and include the required tip number in the block, just in case - compare block N50 to N40:

```
N50 G10 P1 R0.8 Q3 Current tool tip number setting IS NOT assumed
```

➤ Example 4:

Block N60 sets the *wear* group *W 01* to X-0.12, regardless of its previous setting:

```
N60 G10 P1 X-0.12
```

while the block N70 updates the current value of X-0.12 by the increment of U+0.05, to the new offset register value of X-0.07:

```
N70 G10 P1 U0.05
```

Note that the tool tip number (programmed in the **G10** application as the Q entry) will always change the geometry offset and the wear offset simultaneously, whatever the value or the offset type is. There is a simple and very logical reason for it - it is a control *built-in safety* that attempts to eliminate data entry error (manually or automatically). It is impossible to have a different geometry *and* wear tool tip numbers for the same physical tool. Data related to axes or the tool nose radius *may* have different geometry and wear offset values, because they relate to dimensions.

Data Setting Check in MDI

Programming offset values through the standard or macro program requires full understanding of the data input format for a particular control system. It is too late when an incorrect setting causes a damage to the machine or the part. One way to make sure the offset data setting is correct is *to test it*. Such a test is very easy to perform in the MDI (Manual Data Input) mode of the control system. An single word or the whole block can be entered in the MDI mode to test the input, before committing the data into the program. Select the *Program* mode and the MDI mode at the control unit, then insert the input data to check - for example:

```
G90 G10 L10 P12 R-106.475
```

- Press INSERT
- Press CYCLE START

To verify the accuracy of the input, check the tool length offset H12 - it should contain the stored amount of -106.475. Still in the MDI mode, *update* the preset data - for example:

```
G91 G10 L10 P12 R-1.0
```

- Press INSERT
- Press CYCLE START

To verify the accuracy of the input, check the tool length offset H12 again -it should contain the stored amount of -107.475.

Other tests should follow the same process. Always select the test data and offset numbers carefully, so they cannot cause any damage.

Programmable Parameter Entry

This section covers yet another aspect of programming the **G10** data setting command - this time as a *modal* command. It is used to change a *system parameter* through the standard or macro program. This application is sometimes called the '*Write to parameter function*', and is not very common in everyday programming, even in macros. Before you attempt to use this method, make sure you fully understand the concept of control system parameters section, described earlier in this handbook - see previous chapter for details.

Incorrect setting of CNC system parameters may cause irreparable damage to the machine tool

Typical uses of the **G10** command are common to changes of machine or cutting conditions, for example, spindle speed and feedrate time constants, pitch error compensation data, and others. This command often appears in the *custom macros*, normally called by the **G65** command, and its sole purpose is to control the machine operations. The subject and detailed explanation of the *Custom Macros* and their structure is covered in the next chapter - **G10** is just a small item of the basic knowledge before using macros.

Modal G10 Command

When the **G10** command for the offset data setting was first introduced in this chapter, it had to be repeated in each block, if a series of setting were required. By definition, **G10** data setting command for the entry of offsets can only be used as a *non-modal* command. Modern Fanuc controls also allow to do another type of data change through the program - the change of the *CNC system parameters*.

Without the machine user even knowing, many program entries are automatically converted to various system parameters by the control unit. For example, if the program contains **G54** work offset, its current setting can be found on the work offset screen. This display is for the operator's convenience only - the actual storage of the offset amount takes place in a system parameter, identified by a parameter number, as assigned by Fanuc. Even if the **G54** setting is changed manually, through the offset screen, or through a parameter change (using the proper parameter number), it is *always* stored in the system parameter. Some parameters cannot be changed as easily - some cannot be changed at all - the modal **G10** command can be very useful for changing several parameters at the same time. In fact, to achieve this goal, *two* related commands are required - **G10** to start the settings and **G11** to cancel the settings:

G10 L50	<i>Selecting parameter setting mode ON</i>
...	<i>Data setting single block or a series of blocks (typical use)</i>
G11	<i>Selecting parameter setting mode OFF</i>

The data setting block for a programmable parameter setting has three entries:

G10 L50	
N.. P.. R..	<i>Data setting block - more than one block is allowed between G10 and G11</i>
G11	

☞ where ...

G10	Data setting mode - <i>ON</i>
L50	Programmable parameter entry mode (fixed)
N.. P.. R..	Data entry specification (N=parameter number, P=axis number, R=setting value)
G11	Data setting mode - <i>OFF</i> (cancel)

Between the **G10 L50** and **G11** blocks is the list of parameters to be set, one per block. The parameter number uses the N-address and the data uses P and R address respectively.

N-address in G10 L50 Mode

The first of the three addresses, the N-address identifies the parameter number to be changed. Not all parameters can be changed. Fanuc provides *Parameter Manual* for every control model available, listing all parameters and their various states. Examples of typical application of the N-address (as well as the P and R addresses) are described in the next section.

P-address in G10 L50 Mode

The P-address is used *only* for parameters relating to one of the four available axis inputs:

- Bit axis
- Byte axis
- Word axis
- Two-word axis

If the parameter does not relate to an axis, the P-address is redundant and does *not* have to be programmed in the block. If more than one axis is required to be set at the same time, use multiple N.. P.. R.. entries between **G10** and **G11** (see examples further in this chapter).

R-address in G10 L50 Mode

The address R in the **G10 L50** mode contains the *new* value to be registered into the selected parameter number and must *always* be entered (no defaults). The valid range listed above must always be observed. The R-address may also define pitch error data. Note the lack of decimal points in all examples shown.

Program Portability

Program portability refers to the structure and contents of a program as it relates to its application on different machines and/or control systems. CNC programs containing even a single programmable parameter entry should be used only with the machine tool and control unit for which they were designed. *Use extreme care before using such a program on different machines.*

Parameter numbers and their meaning on different control models are not always the same, so the exact model and its parameter numbers must be known during the program development. For example, on *Fanuc 15* control, the parameter controlling the meaning of an address without a decimal point is **#2400 (Bit #0)**. The parameter that controls the same setting on *Fanuc 16/18/21* control models is **#3401 (Bit #0)** - 0=the least input increment is assumed, 1=the applicable unit is assumed.

The following examples illustrate various programmable parameter entries and have been tested on a Fanuc 16B control - both lathe and mill versions. The selected parameters are used for illustration only, not necessarily as typical applications or even common applications.

Testing these parameters on your machine *is generally not recommended because of their potentially harmful nature.*

➤ Example 1 :

This example changes the baud rate setting of the *Input/Output* device (RS-232 interface), if the I/O Channel is set to 0:

```
G10 L50
N0103 R10
G11
```

Parameter that controls the *baud rate* setting for a selected device has a number **#103**, identified as N0103 in the above example. Baud rate specifies the speed of program data transfer rate in characters per second (cps). From a table supplied in the Fanuc operation reference manual, the required R-value can be entered within the specified range - range of 1 to 12 selections is shown:

1:	50 baud	5:	200 baud	9:	2400 baud
2:	100 baud	6:	300 baud	10:	4800 baud
3:	110 baud	7:	600 baud	11:	9600 baud
4:	150 baud	8:	1200 baud	12:	19200 baud

In the above example, the baud rate setting of 4800 characters per second has been selected, because R10 in the sample block refers to the selection number 10. This type of baud rate setting is fairly common, and when working with several machine tools, it should also be the common setting for *all* CNC machine tools in the shop that require the RS-232C interface for program uploading and downloading (DNC). Always choose the fastest baud rate that guarantees 100% transfer accuracy of the CNC program or other settings between the CNC system and an external computer. Note absence of the P-address - as the parameter **#103** does *not* relate to a machine axis, the P-address is not required.

➤ Example 2 :

In another example, parameter **#5130** controls the chamfering distance for thread cutting cycles **G92** and **G76** (lathe controls only). The data type is a *non-axis byte*, unit if the data is 0.1 of a pitch and the range is 0 to 127.

```
G10 L50
N5130 R1
G11
```

This program segment will change the parameter **#5130** to 1. It does not matter what the current setting is, it will become 1 or remains as 1, if that happens to be the current value. The chamfering amount will be equivalent to *one pitch* of the thread, in the increments of 0.1 pitch. As a reminder, do not confuse a *byte* with a *bit* - byte is a value 0 to 127 or 0 to 255 for the byte axis type, bit is a certain state only (*0* or *1*, *OFF* or *ON*, *DISABLED* or *ENABLED*, *Open* or *Closed*, *etc.*); *i.e.*, selection of *one of two* options available.

➤ Example 3 :

Another example of a system parameter change is for the entry of a *two-word* parameter type (long integer). It will change the work coordinate offset **G54** to X-250.000:

```
G90
G10 L50
N1221 P1 R-250000
G11
```

This is another method, one that differs from the one described earlier. Parameter **#1221** controls the **G54**, **#1222** controls the **G55**, and so on. P1 refers to the X-axis, P2 refers to the Y-axis, and so on, up to 8 axes. Because the valid range of a long integer (two-word type) is required, a decimal point cannot be used. Since the setting is in metric system, and one micron (0.001 mm) is the least increment, the value of -250.000 will be entered as -250000. Be careful with the input of zeros - one zero too many or one zero too few could cause a major problem. Speaking from experience, this type of error is not always easy to discover. The following version of the example is *NOT* correct, and *will result in an error*:

```
G90
G10 L50
N1221 P1 R-250.0           Decimal point is not allowed in the R-address
G11
```

Correct input is *without* the decimal point, as R-250000. An error condition (control alarm) will also be generated if the P-address is not specified. For example,

```
G90
G10 L50
N1221 R-250000
G11
```

will generate an error condition (alarm) - the parameter P is missing.

➤ Example 4 :

The last example is similar to the previous one, but modified for two axes values:

```
G90
G10 L50
N1221 P1 R-250000
N1221 P2 R-175000
G11
```

If this example is used on a lathe control, the address P1 is the X-axis, the address P2 is the Z-axis. On a machining center, the address P1 is the X-axis, the address P2 is the Y-axis, and the address P3 will be the Z-axis, if required. In either case, the first two axes of the **G54** setting will be -250.0 (X) and -175.0 (Y) respectively.

Setting Machine Axes to Zero

Sometimes it is necessary to set all available axes in the work offset to zero. This may be done with the standard offset setting - the three basic axes shown:

```
G90 G10 L2 P1 X0 Y0 Z0
```

Application for a milling control

the same setting can also be written directly to a parameter, also as applied to a milling control:

```
G90
G10 L50
N1221 P1 R0      (SET G54 X-COORDINATE TO 0)
N1221 P2 R0      (SET G54 Y-COORDINATE TO 0)
N1221 P3 R0      (SET G54 Z-COORDINATE TO 0)
G11
```

Note the difference in programming format for the two methods.

Bit Type Parameter Example

The following example has been already mentioned earlier, albeit briefly. It is quite harmless, and may be used as a test (as long as you are careful about settings for other parameters). Its purpose is to set automatic block sequencing *ON* (for example, N1, N2, N3, ...), when keyboarding a CNC program at the control. It also serves as a good illustration of a bit-type parameter and some general thoughts and considerations that go into the program preparation that includes programmable parameter entry mode.

On Fanuc 16/18/21 (and many of the other models as well) is a feature that allows automatic entry of sequence numbers, if the program is entered from the keyboard. This feature is intended as a time saving device for manual entry of program data. In order to enable this feature, the parameter that controls the *ON/OFF* status of the feature has to be known and selected. On Fanuc models 16/18/21 (Model B), it is the parameter number 0000 (same as B).

This is a *bit-type* parameter (*not* a byte-type), which means it only contains eight bits. Each bit has its own meaning. Bit #5 (*SEQ*) controls the state of the automatic sequence numbering (*ON* or *OFF* is the same as 1 or 0, but only a number may be input). An individual bit cannot be programmed, the single data number of all eight bits must be specified. That means all the other bits have to be known in order to change only a single one. In this example, the current setting of parameter 0 is listed in eight independent bits, four with assigned meaning:

0000			SEQ			INI	ISO	TVC
	#7	#6	#5	#4	#3	#2	#1	#0
	0	0	0	0	1	0	1	0

The meaning of other parameters is irrelevant for the example, although important in the control for *other* operations. The bit #5 is set to 0, which means the automatic block numbering is disabled. Remember the numbering of the parameter bits - from right to left, starting at 0.

The following example of a program segment is an entry that will turn ON the bit #5 of the parameter #0000, *without* changing the other bits:

```
G10 L50
NO R00101010
G11
```

The resulting entry in the parameter screen will reflect that change:

0000			SEQ				INI	ISO	TVC
	#7	#6	#5	#4	#3	#2	#1	#0	
	0	0	1	0	1	0	1	0	

Note that *all bits* had to be written. Even if it looks that way, the job is not done yet. Fanuc controls offer an additional feature - the *increment amount* for the numbering can also be selected, for example, selection of 10 will use N10, N20, N30 entries, selection of 1 will use N1, N2, N3, and so on. For the example, we will select the increment of 5, to appear as N5, N10, N15, etc., on the control screen. The increment has to be set in the control - yes, by *another* parameter number. On Fanuc 16/18/21, the parameter number that contains the automatic numbering value is #3216. This is a word type parameter, and the valid range is 0 to 9999. This parameter can only be activated by setting the bit #5 in parameter 0000 to 1. Program segment will look like this:

```
G10 L50
N3216 R5
G11
```

These examples demonstrate how some parameters are connected. All is done in quite a logical and simple way, but it does take a little time to get used to it. Once these settings are completed, there is no need to enter block numbers in any program that is entered via the control panel keyboard, usually in the *Program* mode. Anytime the *End-Of-Block* key (*EOB*) is pressed, the N-number will appear automatically, in the increments of 5, saving the keyboarding time during manual program input.

The idea behind the **G10** being modal in the programmable parameter entry mode is that *more than one parameter can be set as a group*. Since the two parameters in the example are logically connected, it makes sense to create a single program segment, with the *same final results* as the two smaller program segments described earlier:

```
G10 L50
N0000 R00101010
N3216 R5
G11
```

As neither parameter is the axis-type, the address P was not needed, therefore, it was omitted. The N0000 is the same as N0, and was used only for better legibility.

Differences Between Control Models

Although the model numbers do not indicate it, Fanuc 15 system is a *higher* control level than the Fanuc 16/18/21 systems. On Fanuc 15, the parameter number that selects whether the automatic sequencing will be enabled is #0010, bit #1 (*SQN*).

Being a higher control, there is also more flexibility on Fanuc 15 - for example, the initial sequence number can be controlled with parameter #0031 (there is no equivalent on Fanuc 16/18/21 model), and the parameter number that stores the increment amount is #0032, with the same program entry styles as already shown. Also, on Fanuc system 15, the allowable range of sequence numbers is higher up to 99999.

This is a typical example of a difference between two similar controls, even from the same control manufacturer.

Effect of Block Numbers

Many CNC programs include block numbers, identified by the address N. It would be perfectly natural to assign block numbers to the last example. After all, entry of data is a valid CNC program segment - for example:

```
. . .
N121 G10 L50
N122 N0000 R00101010
N123 N3216 R5
N124 G11
. . .
```

Will the program work as shown? One of the basic rules of block sequencing is that only one N-address can be in a block, as the first address. What do you think? Will it work?

There are now *two different* N-addresses in the blocks N122 and N123. How does the control handle this situation? Rest easy - *there will be no conflict* whatsoever!

In case of two N-addresses in a single block between **G10** and **G11**, the first N-address is always the block number (basic rule), the second N-address in the same block is the parameter number. The control system can interpret the apparent discrepancy without a problem. If there is only one N-address between **G10** and **G11** blocks, it *always* applies to the parameter number. If there are two N-addresses in the block, the first one is the *block* number, the second one is the system *parameter* number.

Block Skip

Normal block skip symbol (/) can be used to control data blocks processing, but be careful when this function is used in macros, particularly if the control allows block skip function in the middle of a block. See *Chapter 24* for details.

6

MACRO STRUCTURE

Developing macro programs is not much different from development of standard CNC programs, at least not in the general approach. Before macro programs can be developed, study carefully the many ‘tools of the trade’ and ask a question - what features do we work with? Macros have the potential of being extremely powerful and flexible. Macros can also shorten the programming time by many hours, literally. Yet, in spite of their great possibilities, macros are often the ‘forgotten gems’ available for CNC programming. Many companies do have macro capabilities, but avoiding them, considering them too difficult and time consuming.

Macro tools include many functions, techniques and procedures. Custom macro cannot be classified as a true programming language, but macros do share many elements with languages such as *Visual Basic™*, *C++™*, *Lisp™*, and many others, including the derivatives of the ‘early’ languages, such as *Pascal*. The most important tool for the start is to know the format of the macro, and its contents. When these two features are considered together, in the proper sequential order, we are talking about the *macro structure*.

Basic Tools

Every CNC programming technique that a typical part programmer has already learned can be - and are - used in macros and macro development. An in-depth knowledge of CNC programming, combined with a good practical experience (even machining helps), is an essential requirement to learning macros and learning them right from the beginning. Many programming aids not found in standard CNC programming are also available in macros, but they *enhance* and *extend* the traditional programming methods - *they do not replace them*.

There are three basic areas to understand for successful macro development:

- ◆ **Variables** *... three types of data*
- ◆ **Functions and Constants** *... mathematical calculations*
- ◆ **Logical Functions** *... loops and branches*

These three feature areas offer many powerful special functions that are used *within* the body of a macro, which is very similar to a body of a subprogram, except standard subprograms cannot use variable data, whereby macros can (and do so very extensively).

Just like a subprogram, a macro by itself is not much of a use - it has to be interwoven (interfaced) with another program, *called from another program*, by a previously assigned program number. The address (letter) O is used to store the macro programs, the address (letter) P is used to call it, applying the same logic as for subprograms.

Variables

Variables are the most noticeable feature in macros. They are the heart and soul of all macros. Variables give macros the necessary flexibility, by being what they are - *storage units for data that constantly changes* - the so called *variable data*. The name 'variable' is suggestive enough - variables are storage areas in the control system that can hold a certain supplied value. When a value is assigned to a variable, it is stored there for future use. Stored values are called the *defined values*, or *defined variables*.

In macros, variables can be used instead of real values and they can be acted upon, for example, by adding two variables together, to get yet another value. The possibilities are enormous and greatly depend on the skill of the part programmer - or the macro programmer.

Functions and Constants

There is a significant number of functions available for macros. Functions are program features that *calculate* something - they solve a mathematical calculation or a formula. For example, a + (plus) function will *sum* two or more values together. The **SQRT** function will calculate the *square root* of a given number. Many other functions are available, for arithmetic, algebraic, trigonometric, and many other calculations.

In addition to functions, constants can be defined in a macro as well, for example the constant with the value of 3.14159265359....

Logical Functions

Logical functions - also known as *logical operators* - are used in a macro program for *looping* and *branching* purposes, sometimes called a *divergence*. Looping and branching means a *change in the program flow* that is based on - and dependent on - a certain *condition* that has been previously defined.

We are quite familiar with the concept of logical operators in everyday life, we just don't call them that. In English language, there is a short word '*if*'. We use it very frequently to present a certain statement based on a conditional situation. For example, we may say, "*If I have time, I will visit you*". That statement means that I can only visit you, *if* I have time, otherwise, it will not be possible, and I cannot visit you at all. These outcomes are *conditional*. The '*if*' word implies a choice based on the result of a certain condition.

In macros, there are two functions that are used with a given condition. The given condition may be checked (some programmers may say 'tested' or 'evaluated') on several grounds, using the comparison operators, such as '*greater than*', '*equal to*', '*less than or equal to*', and several others, used together with the '*if*' function. These operators are called the *Boolean operators*, named after their inventor George Boole (1815-1864), an English mathematician. They are also called the *logical operators*. The given condition can be evaluated (tested) only once, using the '*if*' check. It may also be evaluated many times, progressively, using a loop function '*while*' the given condition is true - which means '*as long as*' the condition is true). The result of the evaluation will determine further flow of the program.

Defining and Calling Macros

In essence, a macro is a much more sophisticated subprogram. From that viewpoint, it is fair to make a comparison between a regular subprogram and a typical macro. There are always at least *two* individual programs involved in this type of programming environment - the *main program* and the *subprogram*. This is also true for macros - there is the *main program* and the *macro program*. In both cases, the main program calls the subprogram or the macro, by its number, which makes the subprogram or the macro program *subservient* to the higher level program that calls it. Just like a subprogram, a macro can be called not only by the main program (the program at the top), but also by any other subprogram or macro as well, up to a four-level depth. As expected, certain structures must be observed. In all cases, the subprogram or the macro contains specially selected repetitive data, such a contouring toolpath or a specific hole pattern, and in all cases these data are stored as separate programs, under their own unique program numbers.

The single major difference between a subprogram and a macro is the *flexibility* of the input data. Subprograms always use *fixed data*, these are values that cannot change. Macros use *flexible* data, using variable values, that can be changed (defined or redefined) very quickly. Of course, macros may use fixed data as well, but that is not their main purpose.

Macro Definition

Structurally, defining a macro is a very similar to defining a subprogram. In both cases, the program is assigned a program number. In its body, the repetitive data are stored and accessible under that number from the control system memory. In this respect, all rules of a subprogram definition have to be followed in a macro definition.

What is different in the macro program development, are the variable definitions, functions and logical conditions. Variable definitions use variables to store various data. Variables are temporary storage areas of the control system memory - in the macro body they are defined with a special symbol - the # sign. Even at their simplest level, macros will use variables, therefore they will use the # symbol. The upcoming chapters offer a lot more information and details.

Variables are the single most important key to macro programming

Macro Call

Visually, the major difference between calling a subprogram and calling a macro is defined by the programming format. Logically, both calls are the same and serve the same general purpose. In both cases, a previously stored program (a subprogram or a macro) is retrieved from the control storage area by a specific program code:

M98 P----	Calls a subprogram P----	(additional data <i>are not</i> normally required)
G65 P----	Calls a macro P----	(additional data <i>are</i> normally required)

Fanuc control system provides a G-code (preparatory command) to call a previously defined macro rather than a miscellaneous function M used for subprograms. This command is **G65**, and represents the call of a macro program by its stored number, supplemented by additional data. The following structure examples illustrate the differences:

➤ Example 1 - Main program and a SUBPROGRAM :

```
O0004 (MAIN PROGRAM)
N1 G21                               Startup block
N2 ...
...
N15 M98 P8001                         Call stored subprogram O8001
N16 ...
...
N52 M30                               End of main program
%
```

```
O8001 (SUBPROGRAM)
N1 ...
N2 ...
...
N14 M99                               End of subprogram
%
```

➤ Example 2 - Main program and a MACRO :

```
O0005 (MAIN PROGRAM)
N1 G21                               Startup block
N2 ...
...
N15 G65 P8002 F150.0                 Macro call of O8002 with the F argument (= variable #9)
N16 ...
...
N52 M30                               End of main program
%
```

```
O8002 (MACRO)
N1 ...
N2 ...
...
N8 G01 X150.0 Y200.0 F#9            Variable #9 applied to feedrate
...
N14 M99                               End of macro program
%
```

The two examples are included only to show the differences in structure. Note, that the macro example contains two *new types* of data, data that has no equivalent in a subprogram - one called *variables*, the other called *arguments*.

Arguments

The data defined with the macro call, that is with the **G65 P-** command, are called *arguments*. Arguments contain the actual program values required for a particular macro application only. They are always *passed to* the macro itself. Variable data in the macro are replaced with the supplied arguments and the toolpath or other activity is based on the current definitions (arguments) passed to the macro.

A typical program sample of a **G65** macro using three arguments will have the following schematic format:

```
G65 P- L- <ARGUMENTS>
```

☞ where ...

G65	Macro call command
P-	Program number containing the macro (stored as <i>O----</i>)
L-	Number of macro repetitions (<i>L1</i> is assumed as a default)
ARGUMENTS	Definition of local variables to be passed to the macro

An actual sample program macro call may be defined as:

```
G65 P8003 H6 A30.0 F150.0
```

☞ where ...

G65	Macro call command
P1234	Program number containing the macro (stored as <i>O8003</i>)
H6	Assignment of local variable <i>H</i> (#11) argument to be passed to the macro <i>O8003</i>
A30.0	Assignment of local variable <i>A</i> (#1) argument to be passed to the macro <i>O8003</i>
F150.0	Assignment of local variable <i>F</i> (#9) argument to be passed to the macro <i>O8003</i>

Assignments of variables is a separate subject covered in a separate chapter. An assignment simply means *giving the variable a value required at the time of call*. From the example, it is evident that custom macro call **G65** is only *similar to*, but definitely not the same as, the subprogram call **M98**. When two different calls (**M98** and **G65**) of a previously stored repetitive program are compared, there are several very important differences:

- ◆ In the **G65** command, argument is passed to the macro in the form of variable data. In **M98** only the subprogram can be called. No data passing is possible
- ◆ In a subprogram call **M98**, the block may include another data (*i.e.*, a motion to a tool location). In this case, the processing can be stopped in a single block mode. This is not possible in the **G65** mode
- ◆ In a subprogram call **M98**, the block may include another data (*i.e.*, a motion to a tool location). In this case, the processing of the macro starts only after the 'other data' is completed. The **G65** command calls a macro unconditionally
- ◆ Local variables are not changed with **M98** but they are changed with **G65**

Visual Representation

Figure 16 shows a schematic representation of a macro *definition* and a macro *call*. Note that the general structure is identical with the one shown earlier (Figure 4) - a single level subprogram nesting structure.

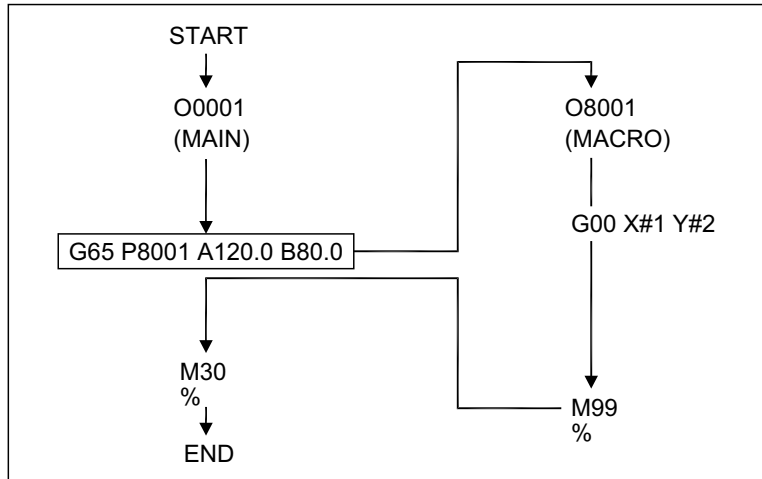


Figure 16
Macro definition and
macro call
Basic structure

In the main CNC program, the macro call command **G65 P8001** retrieves previously stored macro O8001 and passes two arguments to the macro - argument A and argument B. Argument A passes the current value of 120.0 to the macro O8001, argument B passes the current value of 80.0 to the same macro.

Arguments A and B have fixed variable numbers assigned to them (see *Chapter 8* for details). By definition, variable **#1** is assigned to argument A, variable **#2** is assigned to argument B. When either variable is called in the macro body, it will be replaced by the value assigned in that argument. In the case illustrated, the macro block **G00 X#1 Y#2** will be interpreted as **G00 X120.0 Y80.0**. In this case, the arguments represent tool motion as a location or distance, but could have hundreds of other meanings. The 'secret' of macros is that while the arguments in the macro call will change from job to job, the macro remains the same. For example, if the argument is changed to **G65 P8001 A200.0 B150.0**, the rapid motion block in the macro will be interpreted as **G00 X200.0 Y150.0**.

This short illustration does not explain all details, but should serve as the first step to full and complete understanding of macro concepts and their development.

In regards to the macro definition, one may ask where the program number O8001 came from. Is it a mandatory number? Why this number and not other? These, and many questions need some explanation, which is provided in the next section. It is important to keep in mind that all CNC programs (subprograms and macros included) may use *any* number within the provided range (O0001 to O9999 or O00001 to O99999). So the first question can be answered negatively. No, this number is not mandatory. To answer the second question additional knowledge is required. In short, Fanuc controls provide a selected range of program numbers that can have important attributes attached to them, for example, whether they can be edited or deleted. Selecting macro program number O8001 selects a program number that belong to such a range.

Macro Program Numbers

Although any 1 to 4 digit number within a range of O0001 to O9999 can be assigned as a macro program number, certain ranges can be manipulated to provide useful benefits. By definition, Fanuc programs can be separated into the following program number groups:

Program Number Range	Description
O0001 to O7999	Standard program numbers (typically for the main programs)
O8000 to O8999	Macro program number <i>Group 1</i> Can be locked by a <i>setting</i>
O9000 to O9049	Macro program numbers for special applications Can be locked by a <i>parameter</i> (used with <i>G</i> , <i>M</i> , <i>S</i> , and <i>T</i> functions)
O9000 to O9999	Macro program number <i>Group 2</i> Can be locked by a <i>parameter</i>

There are several good reasons why macros deserve special consideration and order when macro program system is considered. By selecting a macro program number from within a particular range, certain benefits become evident, as shown in the table and explained in more detail.

Macro Program Protection

The majority of standard CNC programs do not need protection of any kind. When the word *protection* is applied to these programs, it means one or both of the following attributes can be associated with the program number:

- Visibility of the program on the control screen (program directory display)
- Editing of the program contents (also including program deletion)

Macro programs need protection more than subprograms, and subprograms need more protection than the standard programs. When planning a macro program, it is important to understand the difference between the program number selection, particularly in the available range of O8000 to O9999.

Setting Definitions

In order to select any level of protection, the actual setting of an appropriate parameter must be known and controlled. Since all settings relating to protection of programs are of the *bit* type, they can only have a set value of 0 or 1. In this respect, some users may experience difficulty in the actual interpretation as provided in a Fanuc *Parameter Manual*.

Parameter settings of the bit type always define *one of two* possible states - never more and never less. When we apply these states to the editing and displaying program numbers, only two possibilities exist for each settings:

- Program editing is ALLOWED ... or ...
- Program editing is NOT ALLOWED

- Program display during execution is ALLOWED ... or ...
- Program display during execution is NOT ALLOWED

In their manuals, Fanuc used expressions such as *permitted, prohibited, performed, inhibited, protected*, all along with their opposites. There is nothing wrong with these expressions, except when used at random, they hardly provide convenience, easy interpretation or consistency - and they fail to provide user's confidence. The description of each program number range that follows will use consistent and easy to understand expressions.

Program Numbers - Range 00001 to 07999

Standard programs (even subprograms) can be stored in the control system under any legitimate program number, within the program number range of 00001 to 07999. These programs can be displayed and viewed at will, they can be registered into the system memory without restrictions, and they can be edited at will at any time, also without any restrictions.

If using macros, restrict standard program numbers within the range of 00001 to 07999

Program Numbers - Range 08000 to 08999

Two groups of program numbers are restricted by a parameter setting. The first group (Group 1) is in the range of program numbers 08000 to 08999. It covers programs within the range of 08000 to 08999 only. Programs using numbers from Group 1 cannot be edited, registered, or deleted, without a parameter setting. The parameter access number depends on the control system:

Parameters related to EDITING - 08000-08999 program range				
Control System	Parameter	Bit	Bit ID	Setting
Fanuc 0	#0389	#2	PRG8	0 = Program editing is ALLOWED
				1 = Program editing is NOT ALLOWED
Fanuc 10/11/15	#0011	#0	NE8	0 = Program editing is ALLOWED
				1 = Program editing is NOT ALLOWED
Fanuc 16/18/21	#3202	#0	NE8	0 = Program editing and display is ALLOWED*
				1 = Program editing and display is NOT ALLOWED*

Parameters related to DISPLAY - O8000-O8999 program range				
Control System	Parameter	Bit	Bit ID	Setting
Fanuc 0	n/a	n/a	n/a	0 = n/a 1 = n/a (not available)
Fanuc 10/11/15	#0011	#1	ND8	0 = Program display during execution is ALLOWED
				1 = Program display during execution is NOT ALLOWED
Fanuc 16/18/21	#3202	#0	NE8	0 = Program editing and display is ALLOWED*
				1 = Program editing and display is NOT ALLOWED*

Program Numbers - Range O9000 to O9999

The second group is named *Group 2*. It covers the range of program numbers O9000 to O9999 only. Programs using numbers from Group 2 cannot be edited, registered, or deleted, without a parameter setting. Again, the parameter access number depends on the control system:

Parameters related to EDITING - O9000-O9999 program range				
Control System	Parameter	Bit	Bit ID	Setting
Fanuc 0	#0010	#4	PRG9	0 = Program editing is ALLOWED
				1 = Program editing is NOT ALLOWED
Fanuc 10/11/15	#2201	#0	NE9	0 = Program editing is ALLOWED
				1 = Program editing is NOT ALLOWED
Fanuc 16/18/21	#3202	#4	NE9	0 = Program editing and display is ALLOWED**
				1 = Program editing and display is NOT ALLOWED**

Parameters related to DISPLAY - O9000-O9999 program range				
Control System	Parameter	Bit	Bit ID	Setting
Fanuc 0	n/a	n/a	n/a	0 = n/a 1 = n/a (not available)
Fanuc 10/11/15	#2201	#1	ND9	0 = Program display during execution is ALLOWED
				1 = Program display during execution is NOT ALLOWED
Fanuc 16/18/21	#3202	#4	NE9	0 = Program editing and display is ALLOWED**
				1 = Program editing and display is NOT ALLOWED**

NOTE: *Display* = Display during execution, * and ** identify the same settings for editing *and* display

Program Numbers - Range 09000 to 09049

Within the *Group 2* is a small subgroup, identifying a small selection of program numbers between 09000 and 09049. This range is used for special type of macros - those that are designed to define a new G-code, M-code, S-code or T-code.

As a subject, the creation of new G-codes, M-codes, S-codes or T-codes, is rather advanced at this point, and even a seasoned macro programmer does not always need these advanced programming methods. However, it is very important to establish a certain method in assigning macro program numbers right from the beginning, even if it is for cataloguing purposes only.

It is always a good practice to assign all macros the 8000 or even the 9000 series of numbers, so they can be locked and protected against accidental editing and deletion.

Difference Between the 08000 and 09000 Program Numbers

Looking at the definitions of program numbers carefully, it is easy to notice that both *Group 1* and *Group 2* have the same restrictions. In either group, programs using numbers from that particular group cannot be edited, registered, or deleted, without a parameter setting. So what are the unique differences between them?

The most significant difference is in the *method of how* the restrictions are activated - which parameters are used. Fanuc system 15 is a higher level control than Fanuc 16/18/21 or Fanuc 0. In most cases, the difference between the various systems is the flexibility and convenience of the parameter settings, rather than particular features or functionality.

Often, the main difference in the ease of setting. Fanuc distinguishes two ways of setting a system parameter (not applicable to all controls). One is through the *SETTING* key on the operation panel. This is also called *Handy Setting*, or *Setting (Handy)* or something similar. In order to activate a system parameter in this environment, the programmer will normally use the ON (1) or OFF (0) setting. This is available only on Fanuc 15 and 16/18/21 models. On Fanuc 15, parameter #8000, bit #0 (*PWE*), allows changes to parameters that cannot be set through the *Setting* screen. On Fanuc 16/18/21, the *Setting* screen allows all parameters to be changed. When parameters are enabled (on any control), an alarm (error condition) occurs naturally.

Understanding the machine specifications and control system parameters is extremely important in macro development. Even in its detailed approach, this handbook only offers some insights and explanations, the most important and common ones - it cannot cover all details for all occasions.

No macro programmer can work without the various machine and control manuals - they are the sources of concrete information and precise data source about the equipment used. Each CNC machine tool in the shop will have to be evaluated individually.

For specific details, always consult machine and control manuals supplied by the vendor

7

CONCEPT OF VARIABLES

In the previous chapter, several concepts of macro structure were covered and the purpose of variables has been introduced and identified, including their basic usage in macros. Variables in a custom macro have been designed with several considerations, so looking at them in more detail is essential to their full understanding.

The starting point - and the most important one - to understanding variables, is the understanding of their differences. In Fanuc custom macros, there are *four* different categories of variables, called the *variable types*.

Types of Macro Variables

All Fanuc control systems, regardless of their model number, support macro variables by type. They are classified into four types of variables:

Variable number range		Variable type	Description
From	To		
#0		NULL variable	A NULL variable has no value. It is defined as #0 variable, it is an empty variable, often called a <i>vacant</i> variable. This variable can be read by the macro program, but it cannot be assigned a value, which means data cannot be assigned to it
#1	#33	LOCAL variables	LOCAL variables are only temporary - they are used in a macro body and hold certain data. When the macro is called, the local variables are set to their assigned values. When the user macro is completed and exits (using the miscellaneous function M99), or the control power is turned off, all local variables are set to null values - they cease to exist
#100 #500	#149 #531	COMMON or Global variables	COMMON (also called <i>Global</i>) variables are still valid when a macro is completed. These variables are maintained by the system and they can be shared by several other macro programs. The higher level variables are normally cleared by a specially design macro program
#1000	...and up	SYSTEM variables	SYSTEM variables are used for setting and/or changing default conditions and can read and write different CNC data, for example, a current status of a G-code mode, the current work offset, etc. Their numbers are assigned by the Fanuc control

Note that some reference manuals may only refer to the last three types, leaving the 'vacant' variable (numbered as #0) alone - not considering it as a separate group type.

In a summary, variables are used in macros instead of actual data. The macro programmer assigns values to the variables on the basis of the current application. Variables add flexibility to the macro program but also benefit from other features, such as input data integrity, allowable range checking, etc.

Variables in Macros

Variables are the most noticeable feature of custom macros, either in their initial assignment, or in their use within the macro body. Custom macros depend on variables, so it is imperative to have a look at what variables are, from the ground up.

Definition of Variables

The word or expression '*variable*' can be defined in mathematical terms:

A variable is a mathematical quantity that can assume any value within its allowed range and format

Calculator Analogy

The concept of variables can be illustrated with a common scientific pocket calculator. Even the most inexpensive calculators have at least one memory feature. This memory is a temporary storage area for data values that can be stored now and used later. The data values in the storage will most likely be different every time the calculator is used for the same calculation, so such data is called the *variable data*, the storage area is the *variable* (calculators call it *memory storage* or just *memory*). The word *variable* means *change* or *changeable*. More advanced calculators have more than one memory storage area and they also offer storage of formulas and common calculations. If more than one memory is available on a calculator, identification numbers or letters for each memory are provided on the keypad, to distinguish one from another. Recalling the previously stored variable value by a letter or a number will retrieve it from memory and place it into the current calculation. In macros, many memories (variables) containing different data can be defined and available for calculations, depending on the control model.

Variable Data

In macros, the concept is the same as for calculators. Variables have a generic, rather than a specific, character. They serve as storage areas, and they contain values that can - and do - change with each macro use. For example, in standard programming, a macro may be used for repeating the same toolpath for different materials of the part. Although the toolpath itself may not change, the spindle speeds and the feedrates will be different for each material. For three materials, for example, three individual and separate programs with many repetitions would have to be written.

Quite likely, the only difference between the three programs will be the S-address for spindle speed in rev/min and the F-address for feedrate value in mm/min (in/min). With a macro, both addresses S and F can be defined as variables (because they will change for each of the three materials), then supply the suitable speed and feedrate values for different materials, as needed. By changing only those two values, the program can be used for many more different materials, not just three. The main programming benefit is that the body of the macro program does not change at all, once it is verified.

Variable Declaration

Before they can be used, variables have to be *defined* - macro expression refers to this activity as *declaration* of variables - variables have to be *declared*. Just like the data entry into the memory of a calculator, the basic rules governing the declaration of variables is that a variable *must be defined first*, and only then it can be used in a program or a macro. In the program that uses the variable, the form of definition is represented by the # symbol (commonly called the *pound sign* or the *sharp sign* or the *number sign*). This number sign will be used in all macros. The definition of a variable can take several forms, the first of them is the variable *value*:

#i = assigned current value

☞ ... where the letter 'i' represents the variable number - for example:

#19 = 1200

*Value of 1200 is assigned to variable number 19
it can be spindle speed (rev/min)*

#9 = 150.0

*Value of 150.0 is assigned to variable number 9
it can be feedrate (mm/min, m/min, ft/min, in/min, etc.)*

These two macro statements store values - the value of 1200 is stored into the variable **#19** and the value of 150.0 into the variable **#9**. Both values shown in the example are *numbers*, but they are two different *types* of a number.

Real Numbers and Integers

There are two basic types of numerical values used in macros - a number can be either:

- ◆ **REAL number** ... *real* number always requires a decimal point
- ◆ **INTEGER number** ... *integer* numbers cannot use decimal point

When performing mathematical calculations, the *type* of every numerical value is important. In simple terms, real numbers are typically used for calculations, whereby integer numbers are used for counting and other applications that do not require a decimal point. When a variable number is used in the macro program, its value can be changed as required at any time, two or more variables may be used for mathematical calculations, etc.

Variable as an Expression

Variables can also be defined by using an *expression*, where the expression is typically a mathematical formula or a general calculation. The simplest expression is normally a *direct* value assignment within the macro body, for example:

```
#9 = 250.0
```

Variable **#9** in the example contains an assigned value of 250.0 mm. This actual value may be used to replace a variable value in the macro, for example, the cutting feedrate:

```
G01 X375.0 F#9
```

The **F#9** macro statement will be interpreted as F250.0 (mm/min) actual statement. Redefinition of the variable, for example **#9=300.0**, will pass on the new definition to the macro body, so **G01 X375.0 F#9** will mean **G01 X375.0 F300.0**.

Variables may also use *complex* expressions, for example:

#i = #j + 50, where **#j** is a previously defined variable, which should be interpreted as - add the value of 50 to the current value of variable contained in **#j**, and store the new result in variable **#i**.


The variable definition **#9 = 150.0** in one part of the program, can be used again later, usually as a substituted statement, for example, as a definition **#9 = #9 * 1.1** in another part of the macro program, such as a feedrate input **F#9**, with the actual meaning of F165.0.

In all applications, the rule for applying the variables is simple (the current example is used):

 Take the stored value of variable #9 and use it as the current value of the programmed feedrate

When expressions are used in a macro, they always *evaluate* a multiple mathematical or logical operation. Expressions must be enclosed in square brackets [*expression*]:

```
#i = #i * [#j + #k]
```

 ... where the brackets force calculation of #j+#k to be performed first, before being multiplied by #i

Any complex calculations can be nested within square brackets, always following the standard mathematical hierarchy relating to the order in which calculations will be processed.

Usage of Variables

Macro variables can only be used in a program if they are defined first. Once a variable is defined, it can be used by preceding it with the desired Fanuc program related address (character), which is a capital letter of the alphabet, such as F, S, G, M, etc.

For example, the two variables defined earlier can be used in the body of a program:

➤ Variables declared (defined) :

```
#19 = 1200           Spindle speed defined
#9 = 150.0          Cutting feedrate
```

Variables must be declared (defined) before they can be used, as the example shows.

➤ Variables applied (used) :

```
. . .
G00 G90 G54 X350.0 Y178.34 S#19 M03   (VARIABLE SPINDLE SPEED)
G43 Z25.0 H03 M08
G01 Z-15.0 F200.0                     (FIXED FEEDRATE)
X425.0 F#9                             (VARIABLE FEEDRATE)
. . .
```

Note the use of a fixed and variable feedrates in the same program. Also note that no macros have been used at all. Many programmers do not realize that they may use variables in the main program (standard program) only, without a macro, providing the macro option is supported by the control system. A complete example of such an application has already been shown earlier, in *Chapter 1 - Figure 1*, and still another example is also included in the next chapter.

Decimal Point Usage

A variable that is defined in the macro program body *must always* be entered with the decimal point for all dimensional values, such as position locations, distances, feedrates, or any other definitions that use metric or English units (*see next section*). If these values are entered without the decimal point, the interpretation by the control will use the default settings and could cause some very serious problems. For example,

```
#11 = 45
```

may be interpreted in numerous ways, and not all will yield the same result. The stored value of 45 may become 45.0, 0.045, 0.0045 - or remain just as declared - 45.

In programming, never count on default values !

If an input value accepts the decimal point, always declare it with the decimal point included. In daily applications, typical values that require decimal point are all values relating to dimensions - they are also called *dimensional words* or *dimensional values*. It is important to keep in mind that the default values may work for you, but against you as well. For example, if the X-axis coordinate location is defined as X20, for example, in metric system it will be interpreted as X0.020, in the English units system as X0.0020. *A significant difference!*

There is also a function **ADP** (*Add Decimal Point*) available and described elsewhere, but not recommended even by Fanuc as the best solution to solve decimal point woes.

Metric and English Units

For all dimensional words used in a CNC program (such as X, Y, Z, I, J, K, R, F, etc.), the declared variables may be referenced with the appropriate dimensional word, for example:

```
#1 = 11.6348          Variable is declared or defined
...
...
G00 X#1              Variable is used
...
```

If the selected units in the program are *English* units (programmed in the **G20** mode), the motion block **G00 X#1** will be interpreted as **G00 X11.6348**. If the programming units are *metric* (programmed in the **G21** mode), the **G00 X#1** will be interpreted as **G00 X11.635**. This is a very important difference. A variable that is called in the macro will be *automatically rounded* to the least increment (smallest unit) of the program address.

Least Increment

All CNC programs can use values within an allowed range - up to a certain maximum, and down to a certain minimum. Maximum values are seldom an issue, but every programmer should understand the minimum values. They are often called the *minimum increment* or the *least increment*. These fancy expressions can be translated to a much more practical statement - the smallest amount of motion the machine can provide. Regardless of how they are called, they are distinguished by the number of decimal places:

Units system		Number of decimal places		Least increment
Metric	G21	3	xxxxx.xxx	0.001 mm
English	G20	4	xxxx.xxxx	0.0001 inch

Bear in mind that as a rule, many Fanuc controls *do not* convert from one unit of measurement to another, only *shift* the decimal point. Some features in the control may be converted, but never count on such conversion in any program development. If the decimal point is only shifted during units change, an English dimension of 12.3456 will become 123.456 in metric units - definitely not correct. Providing the **G20** or **G21** units selection command into the program is always highly recommended. What is definitely *never* recommended is to use both types of units in one program, that is between the start of main program and the M30 function for program end.

ALWAYS - provide units selection in every program

NEVER - use both available units in any one program

Positive and Negative Variables

A variable definition that is not equal to zero is called a *non-zero* variable. Non-zero variables may be expressed as either positive or negative variable values. For example,

#24 = 13.7 ... *this is a positive value variable definition*
 #25 = -5.2 ... *this is a negative value variable definition*

Why is this very simple and common fact so important? The reason is that in a macro, the call of the variable may *also* be positive or negative, which means two signs are in effect. When the variable is referenced in the macro, the sign can be intentionally *reversed*, in order to achieve the *opposite effect* of the definition, for example:

G00 X-#24 ... *will be equivalent to G00 X-13.7*
 G00 Y-#25 ... *will be equivalent to G00 Y5.2*
 G00 X#24 ... *will be equivalent to G00 X13.7*
 G00 Y#25 ... *will be equivalent to G00 Y-5.2*

The sign in the declaration is always used together with the sign in the actual execution - the same declaration used as above. Look at one of the above examples:

G00 Y-#25 ... *will be equivalent to G00 Y5.2*

The reason is strictly mathematical and relates to the use of a double sign in a calculation. In many instances, a negative number will have to be added or subtracted, and so on.

The following examples show all four possibilities:

Calculation	Result	Format	Example
Positive + Positive	Positive	$a + (+b) = a + b$	$3 + (+5) = 3 + 5 = 8$
Positive + Negative	Negative	$a + (-b) = a - b$	$3 + (-5) = 3 - 5 = -2$
Negative - Positive	Negative	$a - (+b) = a - b$	$3 - (+5) = 3 - 5 = -2$
Negative - Negative	Positive	$a - (-b) = a + b$	$3 - (-5) = 3 + 5 = 8$

This simplified method may be even easier to understand:

$+$ $+$ $=$ $+$	$+$ $-$ $=$ $-$	$-$ $+$ $=$ $-$	$-$ $-$ $=$ $+$
-----------------	-----------------	-----------------	-----------------

Note that the actual order of the plus and minus symbols within a calculation, such as +- or -- makes no difference to the result. However, the standard mathematical hierarchy of calculating order is and must always be maintained.

Syntax Errors

Making errors in any manually developed program is not uncommon, even if it is undesirable. Macro programs are not immune to being written wrong, even by experienced programmers. At the same time, once a macro is verified and fully functioning, there will be no more errors.

In CNC programming, there are two categories of errors:

- ◆ **Syntax errors** ... *control system will warn the user (alarm issued)*
- ◆ **Logical errors** ... *control system will not warn the user (alarm not issued)*

It is important to eliminate both categories, but it is much harder to eliminate the logical errors than the syntax errors. Briefly, syntax errors are errors that are in conflict with the designed format the control system expects. Logical errors are those, where the programmer intended one activity and provided another activity. For example, -X100.0 is a syntax error, because a program word must always begin with a letter - corrected version is X-100.0. An example of a logical error is when the programmer intends to move to Y-position of 750 mm, but programs Y75.0, instead of the correct input of X750.0.

The section on restrictions is really a section that covers some syntax errors but also includes statements and preferences as to what is allowed and what is not, what is legitimate entry and what is not - with explanations.

Restrictions

Programming in any language has to adhere to a number of very strict rules and restrictions imposed by the language developers for many good reasons. Programmers must follow the rules, conditions and restrictions that apply to the language used. Although not strictly a language, Fanuc macros are no different in this respect. Incorrect use of the macro tools can cause a system error (alarm) - it can also cause an unwanted result, even dangerous situations.

The following conditions and restrictions apply to all Fanuc custom macros, with some typical examples shown for each item - the first column specifies the statement, error or restriction, the second column provides condition or example:

Colon character :	Colon character is not allowed
Semicolon character ;	Semicolon character is not allowed
Zero value is neutral (neither positive nor negative)	+0 or -0 cannot be identified
Leading zeros are ignored	#1 = 003 is the same as #1 = 3

Program number as an EIA identification address O cannot be used with a variable	O#1 is not allowed
Program number as an ISO identification address : cannot be used with a variable	: #1 is not allowed
Block number identification address N cannot be used with a variable	N#1 is not allowed
Block skip identification address / (slash symbol) cannot be used with a variable	/#1 is not allowed
Maximum value of an address cannot be exceeded	If #1=1000 then G#1 is not allowed
Brackets for a single variable will be ignored	# [7] is the same as #7
One variable cannot replace another variable directly - <i>example 1</i> - incorrect	# #7 is not allowed
One variable cannot replace another variable directly - <i>example 2</i> - incorrect	##7 is not allowed
One variable cannot replace another variable directly - <i>example 3</i> - correct	# [#7] is allowed
Overflow or underflow situation when 0 and 90 degrees are used in trigonometric calculation	SIN [0] = negative underflow COS [90] = positive underflow TAN [0] = negative underflow TAN [90] = positive overflow
Nesting in calculations is allowed	If variables #7 and #9 have been previously defined, the following nesting is correct: #101=FIX [[#9*1000] / [3.1416*#7]]

More examples could be added, but the listing covers the most common considerations.

Custom Machine Features

One of the most common reasons for a CNC machine tool having the macro features available is - the *machine tool builder*. Machine manufacturers often incorporate many unique features into their machine tools, for example, advanced physical equipment, such as broken cutter detectors, gantry lines, programmable guard controls, etc. These 'hardware' features must be controlled by the 'software'. The software in such cases programmed in a PLC (*Programmable Logic Control*) but is frequently supplemented by a *special macro*, usually built into the control at the time of purchase. This macro may or may not be hidden from display, most likely will be protected, but it is always very important to be aware of its existence.

Restrictions and minimum or maximum values specified by the machine tool builder must always be honored in macro development or modifications. Programmers should always know the machine built-in ranges, for example, the work area, the minimum and maximum spindle speeds, feedrate ranges, travel limits, tool sizes, and so on.

In their macros, machine tool builders also use special G-codes or special M-codes for their equipment, and they will use many variables in the macros. In the next chapter, we look at variables in more depth.

8

ASSIGNING VARIABLES

In the general introduction to variables earlier, four groups of variables were identified that are used in macro programs:

- ◆ **Local variables**
- ◆ **Common variables**
- ◆ **System variables**
- ◆ **Null variables (same as *empty* or *vacant* variables)**

It is very important to understand these variables well, particularly their differences. This chapter explains how to specify a value of a variable - how to *assign* a value to a variable. The first two of the groups listed - the *local* variables and the *common* variables are covered by this topic.

Local Variables

Local variables transfer the user supplied data to the macro body. Up to 33 variables can be defined as local. Naming this group of variables local means their stored values are only applicable to the macro they have been defined in, they are not transferable between macros. In macro programs, each local variable is associated with an assigned letter of the English alphabet. There are two options available for the so called assignment lists, *Assignment List 1*, which has 21 local variables available, and *Assignment List 2*, which has 33 local variables available. Both assignment lists are described here in detail.

Defining Variables

Variables that are defined in the **G65** macro call, can be within the range of **#1** to **#33**. They are called the *local variables*, or *arguments*. They are available only to the macro that calls them and processes them. Once the processing of the macro is completed, each local variable is reset to a null value, which means it becomes empty and has no value - it becomes *vacant*.

In practical terms, the local variables are used to pass data definitions from the source program (such as a main program) to a macro. Once transferred, they have served their purpose and are no longer needed. These variables were local to the program that called them. We use local variables to assign values to macro program arguments. Local variables are also used for a temporary storage within the macro body, during calculations of formulas and other expressions.

In addition to the **G65** command, there are also preparatory commands **G66**, **G66.1**, and **G67**, all related to macros. The **G65** command is most significant of them and is covered here in depth.

Clearing Local Variables

Local variables are normally cleared (made vacant) by an intervention from the control panel (usually done by the CNC operator), or a program code (usually done by the CNC programmer). Each of the following actions will clear the local variables and sets them to null:

- Pressing the control *RESET* key will set all local variables to null
- Pressing the external *RESET* key will set all local variables to null
- Pressing the *EMERGENCY* switch will set all local variables to null
- Programming code *M30* (program end) will set all local variables to null
- Programming code *M99* (subprogram end) will set all local variables to null

Any local variable can be cleared by these means, but it can also be cleared in a macro program, if required. In the macro statement, it *must* be assigned the value of **#0**. Some manuals refer to the process of clearing variables in a program as a process of *purging* variables, with the same meaning. This example illustrates the clearing (purging) process of local variables in a program:

```
#1 = 135.0           Sets a value of variable #1 to 135.0
...
G00 X#1             Uses variable #1 in the macro (X will be equal to X135.0)
...
#1 = #0             Sets #1 variable to #0 (null) - it holds no value - it is called
                    an empty or null or vacant variable - with the same meaning
```

A null variable is always identified as **#0**, never as a **0** only !

Assigning Local Variables

Fanuc offers *two separate lists* for the assignment of local variables. They are called the *Assignment List 1* and the *Assignment List 2*. In both lists, a letter of the English alphabet is *arbitrarily* assigned a variable number, built into the control software. For example, in both assignment lists, the letter A is associated with a local variable **#1**, letter B is associated with a local variable **#2**, and the letter C is associated with a local variable **#3**.

The assignments vary greatly between the *List 1* and the *List 2*, and the order of numbers *does not* always follow the order of the letters as it may appear from A=#1, B=#2, C=#3 example.

Assignment List 1 - Method 1

The vast majority of macro applications use the variables from the *Assignment List 1*. It only contains 21 assignments for local variables, but that is a number more than sufficient for the majority of macros. The 21 letters of the English alphabet are assigned local variables, as arguments, defined in the **G65** macro call, and passed to the macro body.

The *Assignment List 1* is defined by Fanuc in the following table:

Argument List 1 Address	Local Variable in a Macro
A	#1
B	#2
C	#3
D	#7
E	#8
F	#9
H	#11
I	#4
J	#5
K	#6
M	#13
Q	#17
R	#18
S	#19
T	#20
U	#21
V	#22
W	#23
X	#24
Y	#25
Z	#26

Assignment List 2 - Method 2

Only a very few macro applications use the *Assignment List 2*. It contains 33 assignments of local variables, in case there is a need for more than the 21 local assignments from the *Assignment List 1*. The first three assignments, A, B, and C are the same, but that is where the similarity ends. These assignments are supplemented by a set of 10 argument groups, identified as $I_1 J_1 K_1$ to $I_{10} J_{10} K_{10}$. This method may be somewhat harder to implement, particularly by beginners.

A multiple definition with the same address is based on the *specified order*. Each I-J-K argument has a corresponding 1-2-3 suffix. The suffix of each set specifies the assignment order for the argument set defined in **G65** macro call.

Argument List 2 Address	Local Variable in a Macro
A	#1
B	#2
C	#3
I ₁	#4
J ₁	#5
K ₁	#6
I ₂	#7
J ₂	#8
K ₂	#9
I ₃	#10
J ₃	#11
K ₃	#12
I ₄	#13
J ₄	#14
K ₄	#15
I ₅	#16
J ₅	#17
K ₅	#18
I ₆	#19
J ₆	#20
K ₆	#21
I ₇	#22
J ₇	#23
K ₇	#24
I ₈	#25
J ₈	#26
K ₈	#27
I ₉	#28
J ₉	#29
K ₉	#30
I ₁₀	#31
J ₁₀	#32
K ₁₀	#33

In the macro call **G65**, the assignment of an I-J-K set can be made without difficulty, but be careful to *follow the proper order*:

```
G65 A10.0 B20.0 I30.0 J40.0 K50.0 I60.0 I70.0
```

 where ...

```
A = #1 = 10.0
B = #2 = 20.0
I = #4 = 30.0
J = #5 = 40.0
K = #6 = 50.0
I = #7 = 60.0
I = #10 = 70.0
```

Since there are *three I* definitions (of which two I definitions are consecutive), the first *I* is *I₁*, second *I* is *I₂*, and the third *I* is *I₃*. Because the *order of variable definition* is the key, the ‘missing’ J and K have to be accounted for. Since they are not used, that means they have to be skipped, including their number assignment. Use this method only if there is need for it.

Missing Addresses

In the much more commonly used *Assignment List 1* (Method 1), there are only 21 letters available as arguments (definitions) to the **G65** macro call. Although there 26 letters in the English alphabet, five of them are out of bounds - never to be used. See the next section for details on variable addresses that are disallowed.

In reality, there are *always* 33 variables available, even when the *Assignment List 1* is used. This mysterious statement needs an explanation - where are the remaining 12 variables? Why are they missing? Is there a connection between the missing numbers and the missing five letters?

In the *Assignment List 1*, there are only 21 variables (letter) that can be defined in the macro call **G65**, but the remaining 12 can only be defined *within the body of a macro*. Look carefully on the numbers of those variables missing in the *Assignment List 1*. The following numbers are only those *not* available in the *List 1*:

Variables

#10, #12, #14, #15, #16, #27, #28, #29, #30, #31, #32, and #33

are *not* part of the *Assignment List 1*. Yet, they *can* be defined internally, *within the macro body only*, also as local variables. They can be redefined and used again, but they are not tied up to a letter address like the ‘normal’ 21 variables.

A good example to illustrate the concept of using these variables in the macro body is variable **#33**, although the example applies equally to other ‘missing’ variables. Since **#33** is the last available variable number, many macro programmers often use it as a *counter* definition for macro loops (any other variable can be used for the same purpose with the same result). A counter for loops is often required within the macro body, but there is no need to define it in the **G65** macro call statement, where it would have been defined by one of the assignable 21 variables.

Disallowed Addresses

The next 'mystery' to solve is the mystery of the five missing letters - *addresses*. Why there are only 21 of the 26 English alphabet letters that can be used? The five remaining letters are missing for a good reason. Just by looking at the missing numbers in the *Assignment List 1* may provide a clue. These numbers are missing in the *Assignment List 1*:

#10, #12, #14, #15, and #16

Note - these variables can only be used inside of macro, as already established.

Although **#10** indicates a missing letter G, **#14** a missing letter N, **#15** a missing letter O, and **#16** a missing letter P, **#12** is out of order. Try to think of the disallowed letters, *not* the numbers. The letters that *cannot* be used in a variable assignment (**G65** block) are:

- G** address **Preparatory command**
- L** address **Number of repetitions (for macros, subprograms, and fixed cycles)**
- N** address **Block number (sequence number)**
- O** address **Program number designation**
- P** address **Program number call**

These are restricted letters (try the word **GNOPL** to remember them) and cannot be assigned any value for any purpose. Of the five, only the letter G can be used for a special purpose, such as a definition of a *new* G-code. Custom G-codes can actually called a macro, for example, a specially developed unique cycle. rather than using the **G65** macro call, the new G-code type macro call looks like a normal G-code, and is often easier to work with. *Chapter 21* covers this subject.

Simple and Modal Macro Calls

The **G65** command is defined as a *macro call*. That is correct, but it should really be defined as a *simple* or *single macro call*. The word 'simple' in this case means 'called once', or 'non-modal'. In a program, **G65** can only be used once at a time - as it is not a modal command. It may be called anytime when needed, but all variables must be always redefined. This may prove impractical, when a macro should retain the arguments for more than a single call. To satisfy this need, Fanuc does also offers a *modal* macro call command, in fact, it offers two of them:

G66	Macro is called with an axis movement command only
G66.1	Macro is called with any command (not available on all controls)

Like other modal commands, the modal macro call has to be canceled, when it is no longer required. The modal macro cancel command is another G-code:

G67	Modal macro call is canceled (G66 or G66.1)
------------	---

The **G66** is much more practical, therefore more often used, than the **G66.1**. Compare the typical formats for both, the **G65** and the **G66** commands, using the following example:

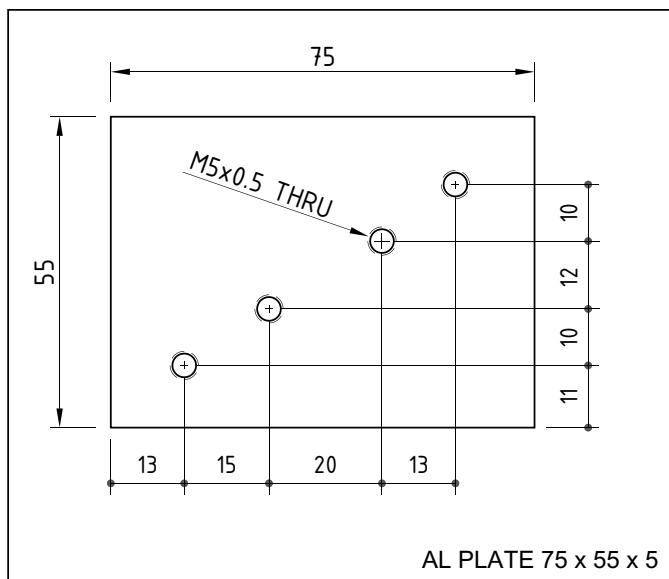


Figure 17

Drawing example for a modal macro call

X0Y0 is at the lower left corner,
Z0 is at the top of the 5mm plate

The simple example uses a part drawing in *Figure 17*, where four holes have to be tapped (drilling operation is omitted in the example). The macro will be designed for a special tapping operation only and **G84** tapping cycles *cannot* be used. This is also a good example of summing up the subjects covered so far.

The main objective of the macro is to program a lower feedrate when the tap moves into the material and a higher feedrate when the tap moves out. This tapping technique is useful for very fine threads in soft materials, to prevent thread stripping. These are the programming objectives:

- Spindle speed 850 r/min
- Nominal feedrate 425 mm/min (850 r/min x 0.5 pitch)
- Feedrate in 80% of the nominal feedrate cutting in
- Feedrate out 120% of the nominal feedrate cutting out
- Retract clearance 3 mm
- Cutting depth 6.5 mm (1.5 mm below the bottom of part)

Selection of Variables

Any assignment address can be used in the **G65** macro call, providing it meets the criteria of macros. Since letters will be used as assignments, the macro programmer has 21 of these letters to choose from. It makes sense to select letters that provide some relationship to their meaning in the macro. From the list above, selecting argument F for feedrate, S for spindle speed, Z for tapping depth, R for the initial and retract clearance, etc., makes it easier to fill in the assignments. This is only a teaching macro that does not have all the 'bells and whistles' incorporated into it. In this handbook, there are several versions listed.

For the example at this stage (using modal macro call), only the following assignments will be provided - the clearance R-value as 3 mm (**#18**), Z-depth as -6.5 (**#26**), and the feedrate as 425.0 (**#9**). Development of the macro O8004 is quite simple:

```

O8004
(SPECIAL TAPPING MACRO)
G90 G00 Z#18
G01 Z-[ABS[#26]] F[#9*0.8] M05      (FEED-IN AT 80 PERCENT OF FEEDRATE)
Z#18 F[#9*1.2] M04                  (FEED-OUT AT 120 PERCENT OF FEEDRATE)
M05
M03
M99
%
```

The macro call in the main program will use **G65** first (tapping only):

```

N81 M06
N82 T07
N83 G90 G00 G54 X13.0 Y11.0 S850 M03 T08      (MOVE TO HOLE 1)
N84 G43 Z25.0 H07 M08                        (INITIAL LEVEL)
N85 G65 P8004 R3.0 Z6.5 F425.0              (HOLE 1)
N86 G91 X15.0 Y10.0                          (MOVE TO HOLE 2)
N87 G65 P8004 R3.0 Z6.5 F425.0              (HOLE 2)
N88 G91 X20.0 Y12.0                          (MOVE TO HOLE 3)
N89 G65 P8004 R3.0 Z6.5 F425.0              (HOLE 3)
N90 G91 X13.0 Y10.0                          (MOVE TO HOLE 4)
N91 G65 P8004 R3.0 Z6.5 F425.0              (HOLE 4)
N92 G90 G00 Z25.0 M09                        (END OF TAPPING)
N93 G28 Z25.0 M05
N94 M01
```

Note that the O8004 macro call had to be repeated with *all the data definitions* for each hole location. Even a simple change to the given definitions would have to be made several times.

The CNC program above can be shortened - quite significantly - and made more flexible, with the *modal macro call* **G66**, and only one call of the macro definitions. **G67** command must be used to cancel the modal call:

```

N81 M06
N82 T07
N83 G90 G00 G54 X13.0 Y11.0 S850 M03 T08      (MOVE TO HOLE 1)
N84 G43 Z25.0 H07 M08                        (INITIAL LEVEL)
N85 G66 P8004 R3.0 Z6.5 F425.0              (TAP HOLE 1 - MODAL)
N86 G91 X15.0 Y10.0                          (MOVE AND TAP HOLE 2)
N87 G91 X20.0 Y12.0                          (MOVE AND TAP HOLE 3)
N88 G91 X13.0 Y10.0                          (MOVE AND TAP HOLE 4)
N89 G67                                       (CANCEL MACRO CALL)
N90 G90 G00 Z25.0 M09                        (END OF TAPPING)
N91 G28 Z25.0 M05
N21 M01
```

Additional improvements (those not listed) will most likely include the cancellation of feedhold, feedrate override and the single block mode, all for a more reliable execution of the program blocks. All of them can be controlled by a macro, using system variables and other features, described elsewhere in this handbook. Not all control models can accept the **G66.1** command.

Main Program and Local Variables

Any program that does *not* call subprograms or macros is called the *main program* - the only program there is. Normally, we do not associate variables with a main program, only with macro programs. Yet, there are many applications, where this programming technique can be very useful and very simple to implement for all controls that have the macro option installed. For those learning macros from the beginning, this may even be a very good way to start the training. The best start is a practical example, enlarging on the basic concepts described in *Chapter 7*.

For the purposes of training, one of the simplest examples of variables in a main program is peck drilling in different materials. Take two materials that are supposed to be the same, such as forgings or castings from two different suppliers. Chances are, the materials will not only be somewhat different in size and shape, they will most likely have a noticeably different hardness. Although the drawing is the same for the *finished* part from either source, the machining procedure is not. The forgings from one supplier will most likely use higher cutting speeds and feeds than forgings from the other supplier, perhaps even different pecking depth. In basic programming terms, we will need *two programs* to satisfy the given conditions.

The programming techniques for such a situation are illustrated in the sample program. A typical peck-drilling operation (deep hole drilling) is done on three holes. Drawing for the program is not necessary, it is a very simple example, and only the peck drilling operation is listed:

```
O0006
(PROGRAM FOR SOFTER MATERIAL)
...
(T05 - 6.5 MM DRILL)
N61 T05
N62 M06
N63 G90 G00 G54 X100.0 Y125.0 S1500 M03 T06      (HOLE 1 LOCATION)
N64 G43 Z25.0 H05 M08
N65 G99 G83 R2.5 Z-75.0 Q15.0 F225.0           (HOLE 1 DRILLED)
N66 X125.0                                       (HOLE 2 DRILLED)
N67 Y150.0                                       (HOLE 3 DRILLED)
N68 G80 G00 Z25.0 M09
N69 G28 Z25.0 M05
N70 M01
...
```

If the above example represents a good program for peck drilling of three holes in a material of lower hardness (softer material), what program data do have to be changed to make the program suitable for the higher hardness (harder material)?

Three items relating to actual machining data should be considered for a change in the program and most likely necessary to be applied in the program:

- | | | |
|---|-----------------------------|-----------------------------|
| <input type="checkbox"/> Spindle speed | Softer material: 1500 r/min | Harder material: 1100 r/min |
| <input type="checkbox"/> Cutting feedrate | Softer material: 225 mm/min | Harder material: 175 mm/min |
| <input type="checkbox"/> Depth of each peck | Softer material: 15 mm | Harder material: 12 mm |

A small table listing the same setting may be easier to read:

Material	Spindle speed (S)	Feedrate (F)	Peck depth (Q)
Softer	1500	225.0	15.0
Harder	1100	175.0	12.0
...

Based on the machining decisions - and without a macro feature available - *another* program has to be written, one that reflects the changes made, because of the material hardness. Here it is, with the proper changes:

```
O0007
(PROGRAM FOR HARDER MATERIAL)
...
(T05 - 6.5 MM DRILL)
N61 T05
N62 M06
N63 G90 G00 G54 X100.0 Y125.0 S1100 M03 T06      (HOLE 1 LOCATION)
N64 G43 Z25.0 H05 M08
N65 G99 G83 R2.5 Z-75.0 Q12.0 F175.0           (HOLE 1 DRILLED)
N66 X125.0                                       (HOLE 2 DRILLED)
N67 Y150.0                                       (HOLE 3 DRILLED)
N68 G80 G00 Z25.0 M09
N69 G28 Z25.0 M05
N70 M01
...
```

Although the application is simplified (only one tool is used), it is clear that only *three* numbers, *three values*, have changed in the whole program. Needless to say, more tools used or more complexity in machining may bring more changes to the program - yet the basic approach does not change at all. The majority of the program data remains identical in both instances. The obvious disadvantage is that if a change is necessary in one program, it will also be necessary in the other program. This could lead to administration problems and possible errors.

With variable data, with the basic features macros offer, only *one master* program is needed. In this master program, the three variable machining data will be defined as - variables. By changing the definitions of the three variable data, the machining will proceed as intended, whether the soft material or the hard material is used.

For convenience and the ability to change the variable data quickly, it is usually positioned at the top of the program (at its beginning). Here are the definitions for the *softer* material, in its first macro version:

```
O0008
#1 = 1500      Soft material
#2 = 225.0    Spindle speed
#3 = 15.0     Feedrate
              Peck drill depth
...
```

Once the cutting conditions are defined as variables, they can be used anywhere in the program:

```

(T05 - 6.5 MM DRILL)
N61 T05
N62 M06
N63 G90 G00 G54 X100.0 Y125.0 S#1 M03 T06      Spindle speed variable applied
N64 G43 Z25.0 H05 M08
N65 G99 G83 R2.5 Z-75.0 Q#3 F#2              Peck-depth and feedrate variables applied
N66 X125.0
N67 Y150.0
N68 G80 G00 Z25.0 M09
N69 G28 Z25.0 M05
N70 M01
...
N145 M30
%
```

Once the program O0008 is completed, all three local variables will be cleared automatically. Note the use of variable numbers. #1, #2, #3 were used arbitrarily. There is nothing wrong with that, except when a true macro call G65 or G66 is used, the variable values have to be assigned to the corresponding argument letter. Would it not make sense to get used to the idea right from the beginning and program letters that look like they 'mean' something? It would make the program much easier to read and interpret.

In the case presented, it is more practical to use variable #19 (assignment S) for the spindle speed, #9 (assignment F) for the feedrate, and #17 (assignment Q) for the peck depth. Here is the above program (still for soft material) modified:

```

O0009                                     Soft material
#19 = 1500                                 Spindle speed
#9 = 225.0                                 Feedrate
#17 = 15.0                                 Peck drill depth
...
(T05 - 6.5 MM DRILL)
N61 T05
N62 M06
N63 G90 G00 G54 X100.0 Y125.0 S#19 M03 T06  Spindle speed variable applied
N64 G43 Z25.0 H05 M08
N65 G99 G83 R2.5 Z-75.0 Q#17 F#9          Peck-depth and feedrate variables applied
N66 X125.0
N67 Y150.0
N68 G80 G00 Z25.0 M09
N69 G28 Z25.0 M05
N70 M01
...
N145 M30
%
```

Again, when the program O0008 is completed, all local variables will be cleared automatically.

Program for the harder material only replaces the three variable definitions (#19, #9, and #17) listed at the program beginning - the rest of the program (for T05 in the example) *does not* change at all - all blocks between N61 and N145 are identical:

```

O0010                                Hard material
#19 = 1100                            Spindle speed
#9 = 175.0                            Feedrate
#17 = 12.0                             Peck drill depth
...
(T05 - 6.5 MM DRILL)
N61 T05
N62 M06
N63 G90 G00 G54 X100.0 Y125.0 S#19 M03 T06  Spindle speed variable applied
N64 G43 Z25.0 H05 M08
N65 G99 G83 R2.5 Z-75.0 Q#17 F#9          Peck-depth and feedrate variables applied
N66 X125.0
N67 Y150.0
N68 G80 G00 Z25.0 M09
N69 G28 Z25.0 M05
N70 M01
...
N145 M30                                All three local variables are cleared
%
```

This method of using variables in the main program without actually developing macros can be a very powerful way to make many jobs more flexible and economical to run.

So far, the common variables (#100+) have not yet been discussed. Can they be used with some additional benefits? Once you understand their purpose, the decision will be yours.

In short, yes, the common variables (#100+) could also be used, but with a little benefit. Take the above example for what it is - it only *attempts* to demonstrate the principles of local variables, not necessarily present their most efficient usage. As a matter of fact, there are several *other* ways to improve on this program, all a little advanced at this point. One method includes the use of optional stop, providing function can be used in the middle of a command (not all controls support this feature). As a rule, keep away from using the block skip function (the slash function /) from programs using variables and particularly from using it in macros. There are times when the block skip function can be used very effectively, and times when it should not be used at all, for example, with variables. Since a block skip function is a very primitive branching method, to provide some minimum 'branching' flexibility to standard programs, it is actually not needed in macro programs at all. Macros offer a number of very sophisticated programming features - especially designed for branching - with much superior control than block skip function can provide.

At this level of macro program development, there is a possible small improvement that can be used to make the programming effort more efficient that has nothing to do with macros. The program for tool T05 can be stored as a *subprogram*, which can include all local variables. As long as the variables are defined in the main program *before* the subprogram call, this improvement can be very useful. When one set of parts is done, the variables will be changed for different cutting conditions, and the same subprogram can be called again. However, this brings very close the subject of 'real' macros - a subject needed to learn in more depth.

Local Variables and Nesting Levels

Subprograms and macros can both be nested within the program structure. Nesting, as a programming feature, means that one subprogram or one macro can call another subprogram or a macro, which can call another subprogram or a macro, and so on, up to *four levels* deep. Having a four-level depth of nesting offers some real programming power, but it is rather rare to program more than two levels of nesting depth. Regardless of how many levels the macro (or subprogram) is nested, it is important to understand the relationship between local variables and each macro level. *Figure 18* is a schematic representation of the macro nesting, showing all four levels:

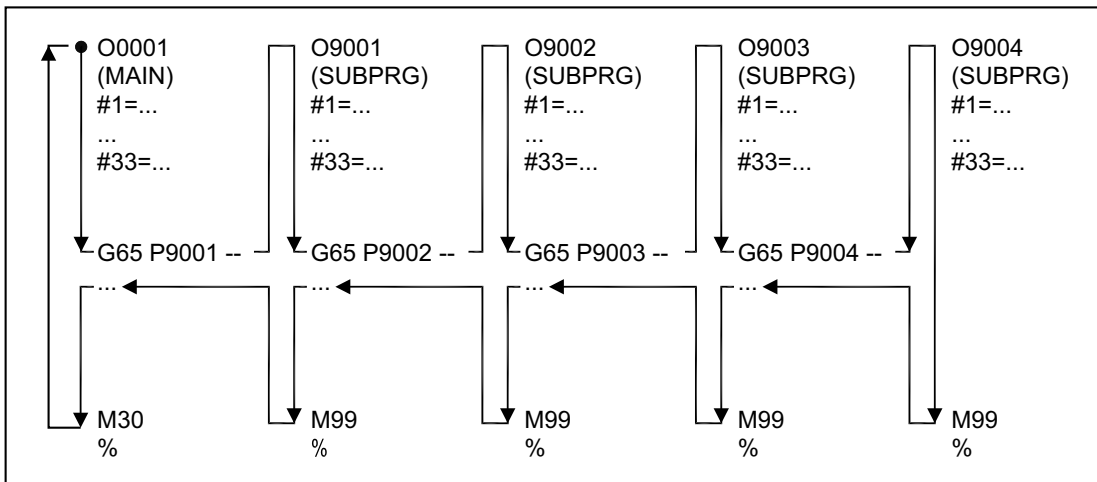


Figure18

Local variables definitions and levels of macro nesting

As the illustration shows, each set of local variables #1 to #33 can be defined up to *five times* - once in the main program, and once for each macro level (up to four more times).

Every time a new nesting level is processed, the new set of variables takes over for that macro, but the old set is still in memory, and will be recalled when the macro returns to the level it originated from. The variables will still retain their values. Remember that it is the miscellaneous function **M99** that clears all local variables, not a jump to another program. Until the program flow encounters the **M99** function (end of subprogram or a macro), no local variable is cleared. When the whole program is completed, program end function **M30** will clear any and all local variables defined in the main program. Local variables cannot be passed from one macro to another - that is why they are called *local* variables.

In many macro development cases, there is a strong need to pass a defined variable from one macro to another macro. For nesting, this requirement presents no problem, as just shown. The problem becomes apparent, when the local variables have been cleared and there still is the need to pass one or more variable values to another macro. To achieve this goal, Fanuc offers another set of variables - another range - called the *common variables* - variables that can be passed from one macro to another, without being cleared before the transfer.

Common Variables

The sole purpose of *common* variables is that they remain active when the macro they were defined in is completed. Make sure to understand how the common variables work (there are some differences between control models), and also when and how the common variables are cleared.

Common variables are never assigned as arguments in the **G65** macro call. They can only be defined in the macro body, and they start with the first common variable - **#100**. There is another range of common variables, one that starts with the common variable **#500**. The difference between the two is very significant:

Variables **#100 to #199** are cleared when the power of the control system is turned off

Variables of the **#500 to #999** range remain in effect
even when the power to the control system has been turned off

For the common variables in the ranges **#100 to #199** and **#500 to #999**, Fanuc offers four available options (actual number of available variables) on its various controls:

- | | |
|---|-----------------|
| <input type="checkbox"/> #100 to #149 and #500 to #549 | <i>Option A</i> |
| <input type="checkbox"/> #100 to #199 and #500 to #599 | <i>Option B</i> |
| <input type="checkbox"/> #100 to #199 and #500 to #699 | <i>Option C</i> |
| <input type="checkbox"/> #100 to #199 and #500 to #999 | <i>Option D</i> |

If the optional range of common variables is available (**#100 to #199** and **#500 to #999**), some loss of storage space is normal and should be expected. For the first three options (A, B, and C), the available memory will decrease by about 1000 characters, the memory storage capacity for the D option will decrease by about 3000 characters. Usually, this loss does not present a major hardship, but is a loss that should be considered when using the various options.

Volatile and Nonvolatile Memory Groups

In computing terms, the words *volatile* and *nonvolatile* are associated with the available RAM (*Random Access Memory*) of the computer. Important data has to be saved to files, in order to guarantee that the data is permanently stored. However, during the data development itself (for example, during word processing), the unsaved data is temporarily kept in the computer *random access memory* until it is saved. If there is a power interruption or a software failure before the data can be saved to a file, this data is lost, because RAM is a *volatile* type of memory.

The common variables **#100 to #149** or the optional set of **#100 to #199** are automatically set to the null value (empty and vacant), when the power of the CNC system is turned off. This group of variables is called the *volatile* group.

Variables in the range of **#500 to #999** will hold the stored data, even after the CNC system is turned off (power-off). This group of variables is called the *nonvolatile* group.

Input Range of Variables

All local and common variables (but *not* the system variables) can be programmed with a large range of input values. Even when seldom needed, it is important to know that there are maximum and minimum limits within the range and what the limits actually are. The limit ranges for local and common variables used in macros are:

Input of zero	0
Negative input	-10 ₄₇ to -10 ₂₉
Positive input	10 ₂₉ to 10 ₄₇

An out-of-range input, or an invalid out-of-range result of a calculation, will always result in an alarm condition. For example, alarm *No.111* is generated on Fanuc 16/18/21 controls, in case of out-of-range value.

Out-of-Range Values

If the row of asterisks, for example, *********, appears in the variable data display on the control screen, it indicates an *out-of-range* value, either as an *overflow* or an *underflow* value of the input data or calculation. This unwanted result is usually caused by an incorrect formula, typing error or some other calculation input.

Definitions of both *overflow* and *underflow* conditions can be easily defined:

OVERFLOW value is defined
when the absolute value of the variable
is greater than 9999999.0

UNDERFLOW value is defined
when the absolute value of the variable
is less than 0.0000001

Calculator Analogy

Overflow and underflow situations occurring in macro execution can be compared to errors generated by most scientific calculators. For example, when attempting to calculate the tangent value of a ninety degree angle - entering *TAN.90* or *90.TAN* (tangent of ninety degrees) will result in error (correct keyboarding is assumed).

Set Variable Name Function SETVN

On the some controls, for example the Fanuc 15, common variables of the 500+ range can be set to a common *name*, up to eight characters long. This is a very convenient reminder that these are special variables, usually permanent, and should not be tampered with.

The function available for this purpose is called **SETVN** (*Set Variable Name*), and can be used with a single variable or a range of variables:

```
...
SETVN500 [PROBEDIA]           Variable name defined for #500
#500 = 6                       Value of defined variable assigned
...
```

A string of definitions will define a range of a sequential variable, when the starting variable is specified:

```
...
SETVN500 [PROBEDIA, HOLEDIA, XPOS, YPOS] Variable names defined, starting with 500
#500 = 6                               Value of the starting variable PROBEDIA defined
#501 = 78.0                             Value of the next variable HOLEDIA defined
#502 = 300.0                             Value of the next variable XPOS defined
#503 = 250.0                             Value of the next variable YPOS defined
...
```

This section is only included here to provide additional information, not necessarily as a tool for everyday macro work. Usage of the **SETVN** function in macros is not common.

Protection of Common Variables

Common variables #500 to #627, for Fanuc 10/11/15 only, can be protected from any data input. Typically, the set value to be protected is input first, while the variable is not protected - then the variable or variables can be protected by setting two system parameters:

Parameter 7031	<i>The first variable to protect</i>	<i>(input is 0 to 127)</i>
Parameter 7032	<i>The number of variables to protect</i>	<i>(input is 0 to 127)</i>

➤ Example:

Fanuc 15 control parameter 7031 is set to 11, while parameter 7032 is set to 5 - *then ...*:

Variables #511, #512, #513, #514, and #515 will be protected from data input, such as copying, editing, deleting, etc.

9

MACRO FUNCTIONS

Up to now, the macro related subjects covered macro structure, local and common variables (system variables are still to come), and assigning variables. *Fanuc Custom Macros* support a variety of special *functions* that can be used in the body of a macro and some even in the body of the main program or subprogram. These functions are typically associated with mathematical calculations, logical operations, conversions, and various formulas. All together, they form a very strong group of macro programming tools.

Function Groups

Several macro examples have been presented in the previous chapters. In this chapter, the subject of *functions* will be covered in great detail, including examples of their usage.

Fanuc CNC system (in the macro mode) can perform many arithmetic, algebraic, trigonometric, miscellaneous, and logical calculations on existing variables, using various formula formats and conversions. For macros, the CNC programmer has a complete control. In the definition format of variables, the expression at the right side of the equal sign (=) may contain constants and various combined operations. There are some restrictions and limitations mentioned throughout the handbook, but overall, the subject of functions adds a very powerful and desired tool to macro programming.

Macro functions can be separated into groups, to make their understanding and usage easier to learn. When viewing the various settings of active variables on the control display screen, it is not unusual to see data with many leading and trailing zeros. This appearance is all part of the display only - the zeros are often not written in the macro program.

In all examples, the leading and trailing zeros are ignored, unless they are specifically required.

The available macro functions can be divided into six groups:

- ◆ **ARITHMETIC functions**
- ◆ **TRIGONOMETRIC functions**
- ◆ **ROUNDING functions**
- ◆ **MISCELLANEOUS functions**
- ◆ **LOGICAL functions and operations**
- ◆ **CONVERSION functions**

Definition of Variables Revisited

The local variables can be defined in the macro call **G65** or **G66**, or in the main program itself. The common variables can only be defined in the body of the program, either in the main program, or in the macro. Typically, the variable is defined first, and used later, once or many times. This process is called '*referencing*' a variable.

Referencing Variables

Referencing a variable means replacing the variable number with the previously stored data value. For example, in one of the earlier examples, the cutting feedrate value was stored into the variable **#9**:

```
#9 = 225.0           Assigns the value of 225.0 to variable #9
```

In the program, the feedrate was called as a normal CNC word, by referring the variable to the F-address but writing it as **F#9**. A local variable *not* available as a letter in the *Assignment List* (1 or 2) can only be referenced in the program body:

```
#33 = 1             Assigns the value of 1 to variable #33
```

In the program, the new variable can be used by itself or in an expression:

```
WHILE [#33 LE 6] DO1      Repeat a loop as long as condition [#33 LE 6] is true
```

The variable **#33** is not associated with any letter; in the example, it is used as a counter. It is used for evaluating the program flow, in a conditional statement - **[#33 LE 6]** is the condition. This subject will be covered in *Chapter 13*.

Not all values have to be positive. Negative definitions and references are very important in macros, and many errors are often caused by an incorrect referencing. Watch the difference in the following examples - the objective is to program the Z-depth as Z-12.75, using variable data:

```
#26 = 12.75         Assigns the positive value (12.75) to variable #26
```

In the program, the Z-depth must be programmed as **Z-#26**. The called variable must be *negative!* Can the definition itself be negative? Yes, it can:

```
#26 = -12.75       Assigns the negative value (-12.75) to variable #26
```

In the program, the Z-depth must be programmed as **Z#26**. The called variable must be *positive!*

There is a way to *always* guarantee that the required value will be negative, *regardless* whether the input value is positive or negative. It uses the macro function **ABS** - *absolute* value of a number, explained in the *miscellaneous function* section of this chapter.

Vacant or Empty Variables

In many cases, a variable may also be *undefined*. In this case, the variable is 'defined' as #0, which identifies a *null* variable (null state). A null variable has no value, it is *vacant*. It can be read, but it cannot be written to - these types of variables are often called the read-only variables. For example,

```
#500 = #0           Assigns a null value to variable #500
#33 = #0           Assigns a null value to variable #33
#1 = #33 + #500    #1 will add 0 to a 0 and return 0
```

The section describing various arithmetic functions in this chapter also describes handling of vacant variables in calculations. A null variable is also called a *vacant* or an *empty* variable. In a macro, certain rules of vacancy apply. It is important to know the return value of a variable, when vacancy is in effect.

Do not confuse a **vacant** variable with a variable that has a **zero** value !

```
#101 = 0           Variable #101 has a zero value - stored value is 0 !
#102 = #0         Variable #102 is vacant (empty) - no value !
```

Using these two variables as stored, look at the stored values of variables that use them:

```
#1 = #101         Variable #1 has a zero value - stored value is 0 !
#2 = #102         Variable #2 is vacant (empty) - no value !
```

More complex applications include axis motion commands, math functions and conditional expressions. These are the three conditions to watch for in terms of vacancy rules - they relate to the:

- Addresses for axis motion command**
- Mathematical operations**
- Conditional expressions**

Only the first two conditions are explained in this chapter, the last one is described in the chapter on branching and looping.

Axis Motion Commands and Null Variables

If an undefined variable is referenced, the variable is *ignored* during processing of tool motion, for example:

```
G90
#24 = #0           Used for X-axis: X-value is null = no X-value
#25 = #0           Used for Y-axis: Y-value is null = no Y-value
...
G00 X#24 Y#25     Same as G00 only (with no X or Y axis)
```

The previous example will result in the equivalent of **G00** rapid command only. The X-value and the Y-value in the example will be ignored! Fanuc does not 'assume' any value, and there is no default option.

If only *one* of the variables is undefined, only that variable will be ignored, the other one will be processed as intended:

```
G90
#24 = #0           Used for X-axis: X-value is null
#25 = 0           Used for Y-axis: Y-value is 0
...
G00 X#24 Y#25     Same as G00 Y0 (with no X axis)
```

The previous example will result in the equivalent of **G00 Y0** motion command. The X-value in the example will be ignored!

On the other hand, if the value of a variable is 0 (zero), it becomes the value of the specified axis motion address:

```
G90
#24 = 0           Used for X-axis: X-value is 0
#25 = 0           Used for Y-axis: Y-value is 0
...
G00 X#24 Y#25     Same as G00 X0 Y0 (both axes are active)
```

Errors caused by the wrong referencing of variables used for axis motion commands are easy to make, but often difficult to find.

Terminology

Just like any other field, CNC programming has its own special words, its own jargon - its own terminology. Macros, being a part of CNC programming, share the terminology and add some more of its own. In macros, there are several expressions related to variables and functions that may not be familiar. Some of these terms may be self-explanatory, others could be misunderstood and should be qualified.

Here is the list of the most common terms and expressions related to macros:

<input type="checkbox"/> Evaluate ...	Process - or act upon - the given variable or function
<input type="checkbox"/> Current value	The value stored in a variable at a given time
<input type="checkbox"/> Contents of ...	Same as <i>Current value</i>
<input type="checkbox"/> Referencing	Previously defined variable is called by its number
<input type="checkbox"/> Returned value	The new value that is the result of a calculation
<input type="checkbox"/> Result of ...	Same as <i>Returned value</i>
<input type="checkbox"/> Substitution	Storing <i>new data</i> in a previously defined variable, also known as <i>redefinition</i>
<input type="checkbox"/> Redefinition	Same as <i>Substitution</i>

Many of these terms are used throughout the handbook, and in many other publications.

Arithmetic Functions

There are several macro functions relating to mathematical calculations. The simplest of them are the four basic arithmetic functions, used in variables and also available to the macros. The arithmetic functions use the following symbols:

+ - * /

Function	Known as ...	Symbol
Sum	Addition	+
Difference	Subtraction	-
Product	Multiplication	*
Quotient	Division	/

In order to understand any function, it is important to evaluate a few examples. In the following examples, the function is on the left, its returned value - *the result* - on the right:

#1 = 3.5	3.5	<i>Returned value of variable #1 is 3.5</i>
#2 = 4.25	4.25	<i>Returned value of variable #2 is 4.25</i>
#3 = 2.0 + 5.0	7.0	<i>Returned value of variable #3 is a sum of 2+5</i>
#4 = #3 + 1	8.0	<i>Add 1 to the result of variable #3</i>
#5 = #2 - 0.8	3.45	<i>Subtract 0.8 from the current value of variable #2</i>
#6 = #1 - #3	-3.5	<i>Subtract contents of #3 from the contents of #1</i>
#7 = #2 * 6	25.5	<i>Multiply contents of #2 by 6</i>
#8 = 7.0 / 8.0	0.875	<i>Divide 7 by 8 as real numbers</i>
#9 = 7 / 8	0.875	<i>Divide 7 by 8 as integer numbers</i>

Nesting

Nesting means the contents of brackets (not parentheses) will be processed first - compare:

#10 = 9.0 - 3.0 / 2.0	7.5	<i>Division first, then multiplication</i>
#11 = [9.0 - 3.0] / 2.0	3.0	<i>Multiplication first, then division</i>

Arithmetic Operations and Vacant Variables

So far, the focus was at vacant variables as they were applied to the axis motion command. Vacant variables can also be used as a part of the various mathematical operations, and it is very important to understand how they behave in that environment. Mathematical operations include redefinition (substitution) of variables, as well as arithmetic, algebraic, trigonometric, and other types of calculations. On the basic level, the rules are slightly different for *addition* and *subtraction* than for *multiplication* and *division*. The following examples should clarify the most common possibilities encountered in macros:

➤ SUBSTITUTION

◆ Vacant variable substituted will remain vacant:

#1 = #0 *#1 defined as vacant*
 #2 = #1 *#2 also defined as vacant*

◆ Zero value variable substituted will remain zero:

#1 = 0 *#1 defined as a zero value*
 #2 = #1 *#2 also defined as a zero value*

➤ ADDITION

◆ Vacant variable added to a value is the same as an increase by zero:

#1 = #0 *#1 defined as vacant*
 #2 = 15.7 + #1 *#2 will add 0 and return 15.7*
 #3 = #1 + #1 *#3 will add 0 to a 0 and return 0*

◆ Zero value variable added to a value is the same as an increase by zero:

#1 = 0 *#1 defined as zero*
 #2 = 15.7 + #1 *#2 defined as 15.7*
 #3 = #1 + #1 *#3 will add 0 to a 0 and return 0*

➤ SUBTRACTION

◆ Vacant variable subtracted from a value is the same as a decrease by zero:

#1 = #0 *#1 defined as vacant*
 #2 = 15.7 - #1 *#2 will subtract 0 and return 15.7*

◆ Zero value variable subtracted from a value is the same as a decrease by zero:

#1 = 0 *#1 defined as zero (0)*
 #2 = 15.7 - #1 *#2 will subtract 0 and return 15.7*

➤ MULTIPLICATION

◆ Multiplication by a vacant variable is the same as a multiplication by zero:

```
#1 = #0           #1 defined as vacant
#2 = 15.7 * #1   #2 will multiply by 0 and return 0.0
```

◆ Multiplication by a zero value variable is the same as a multiplication by zero:

```
#1 = 0           #1 defined as zero
#2 = 15.7 * #1   #2 will multiply by 0 and return 0.0
```

➤ DIVISION

◆ Division by a vacant variable is the same as a division by zero:

```
#1 = #0           #1 defined as vacant
#2 = 15.7/#1     #2 will divide by 0 and return 0.0 (Error condition)
```

◆ Division by a zero value variable is the same as a division by zero:

```
#1 = 0           #1 defined as zero (0)
#2 = 15.7/#1     #2 will divide by 0 and return 0.0 (Error condition)
```

Division by Zero

Even the least expensive pocket calculator returns an *Error* message, if the calculation attempts to divide any value *by zero*. CNC system and macro calculations are no different. The two division examples above illustrate the point. Although the returned value may be a *displayed zero*, this value cannot be used, because of the error (alarm) condition that has been caused. To eliminate the error condition, the control system has to be reset first, then the cause of the error eliminated.

◆ Division BY zero is not permitted

```
#1 = 5/0           Returns an error condition
```

◆ Division OF zero is permitted (although seldom used)

```
#1 = 0/5           Returns zero (0)
```

It is unlikely that the programmer would divide by zero in the program directly, unless in error. What is more likely reason for such an error is a result of a calculation:

```
#1 = 5           Value of 5 stored in #1
#2 = #1 - 5     Value of 5 subtracted from #1, returning 0
#3 = 10/#2     Value of 10 divided by #2, which is zero - result is ERROR
```

Trigonometric Functions

Trigonometric variables available in macros are used to calculate *angles* or data related to angles. Examples include calculation of rectangular coordinates, solution of right angle triangles, angle values, etc. All trigonometric functions can be applied in macros, although not all are available on all Fanuc control models. The most common entry of an angle will be in the decimal format. For many part drawings that still indicate angles in the *Degrees-Minutes-Seconds* format (D-M-S), translation to decimal degrees is necessary.

Conversion to Decimal Degrees

An angle can be expressed either as a whole number, for example 38, or as a decimal degree representation, for example 12.86. The *Degrees-Minutes-Seconds* format (D-M-S) is not allowed, is generally considered obsolete in modern CAD/CAM, and must be converted to decimal degrees first, if necessary.

The conversion is quite simple:

D_d	D	$\frac{M}{60}$	$\frac{S}{3600}$
-------	-----	----------------	------------------

where ...

D_d = Decimal degrees
 D = Degrees (sometimes indicated as *H* or *HR* on calculators)
 M = Minutes (there is 60 minutes in an hour)
 S = Seconds (there is 3600 seconds in an hour)

Example:

$$10^\circ 36' 27'' = 10 + 36/60 + 27/3600 = 10.6075^\circ$$

Of course a calculation using variables can be used for the same purpose:

#1 = 10.0	<i>Value of degrees</i>
#2 = 36.0	<i>Value of minutes</i>
#3 = 27.0	<i>Value of seconds</i>
#101 = #1 + #2/60 + #3/3600	<i>Result is decimal degrees of the given D-M-S angle</i>

Available Functions

The following trigonometric functions are generally available for macros:

SIN	COS	TAN	ATAN	ASIN	ACOS
-----	-----	-----	------	------	------

All input for **SIN**, **COS**, and **TAN** is in degrees and the output of the inverse functions **ATAN**, **ASIN** and **ACOS** is also in degrees.

Inverse functions are usually marked as \tan^{-1} , \sin^{-1} , and \cos^{-1} on the calculator.

#1 = SIN [38]	0.6156615	(actual value must be in brackets)
#2 = 23.7	23.7	
#3 = COS [#2]	0.9156626	(reference to a variable must be in brackets)
#4 = TAN [12.86]	0.2282959	

The inverse trigonometric functions accept the length of two sides of a triangle, both enclosed in brackets, and separated by the slash symbol standing alone between them. The acceptable range is within $0 \leq \text{RESULT} < 360$:

#5 = ATAN [0.25]/[0.5]	26.5650512	(note the position of the slash !!!)
-------------------------------	------------	--------------------------------------

ASIN and **ACOS** functions are *not* available on 0/16/18/21 model controls !

Rounding Functions

Calculations often result in a value with too many decimal places. In CNC work, only three decimal places can be used for metric units, or four decimal places for English units in the program or a macro. Some rounding is necessary and should be expected. There are three functions available in macros that control the rounded value of a given number - they are similar to each other, but definitely not the same:

ROUND FIX FUP

The **ROUND** function is designed to round off the supplied value to a whole number (round off fractions under 1.0). The function disregards fractions that are *less than 0.5*. For fractions that are *equal to or greater than 0.5*, the next whole number is the rounded value:

ROUND [0.00001]	Returns 0.0
ROUND [0.5]	Returns 1.0
ROUND [0.99999]	Returns 1.0
ROUND [1.0]	Returns 1.0

A value that had been stored previously into a variable, can be rounded in the same way as a directly input value:

#1 = 1.3	Returns 1.3
#2 = 1.6	Returns 1.6
ROUND [#1]	Returns 1.0
ROUND [#2]	Returns 2.0

There are small differences in usage of the **ROUND** function, depending on application. If the **ROUND** function is used in a *definition* of a variable, the rounding effect will always be to the *nearest integer value*. For example:

```
#101 = 19/64           Returns 0.296875
#102 = ROUND[#101]    Returns 0.0
```

The **ROUND** function may also be used in a CNC statement. First, store a value:

```
#101 = 19/64           Returns 0.296875
```

In the English system of units, the smallest unit increment is 0.0001 of an inch, so the rounding will be to the *four decimal place* accuracy (also known as rounding to the least increment). In a CNC statement, the rounding will be accurate to 4-decimal places:

```
G20                     English mode
#101 = 19/64           Returns 0.296875 (English units)
G91 G01 X[ROUND[#101]] F10.0    Uses X0.2969 motion
```

In the metric system of units, the smallest unit increment is 0.001 of a millimeter (one micron), so the rounding will be to the *three decimal place* accuracy (to the least increment). In a CNC statement, the rounding will be accurate to 3-decimal places:

```
G21                     Metric mode
#101 = 19/64           Returns 0.296875 (Metric units)
G91 G01 X[ROUND[#101]] F250.0    Uses X0.297 motion
```

As a simplified example of a possible application, several simulated motions will be programmed using a predefined fractional dimensions. The tool will move in three stages - rapid out from the start position, feed out a little further, and rapid back in to the start position (only one axis is used in the demonstration). In order to test the usage of the **ROUND** function on the machine control, first enter the following program into the control system, then register the current XY coordinate of the tool position at the start of processing:

```
N1 G20                     English units input
N2 #100 = 3 + 19/64        Input value of 3.296875 (motion A)
N3 #101 = 2 + 5/64        Input value of 2.078125 (motion B)
N4 G91 G00 X-#100        Incremental motion A to the left X-3.2969
N5 G01 X-#101 F20.0      Incremental motion B to the left X-2.0781
N6 G00 X[#100+#101]      Incremental motion A+B to the right may not be rounded
N7 M00                   End of example
```

Compare the tool position at the beginning of the test, before running the program, and compare it with the tool position after the program has been executed. The tool position coordinates XY may or may not be the same. The start position and the end position of the tool may be off, depending on the type of value to be rounded. This problem is compounded by the fact that the error is accumulative - the more parts are machined, the more severe the deviation error will be. This is due to the rounding effect of the control system.

To correct the accumulative error, or if you want to be absolutely certain, you have to round the motion in one direction to equal to the motion in the opposite direction:

```

N1 G20                                     English units input
N2 #100 = 3 + 19/64                       Input value of 3.296875 (motion A)
N3 #101 = 2 + 5/64                         Input value of 2.078125 (motion B)
N4 G91 G00 X-#100                         Incremental motion A to the left X-3.2969
N5 G01 X-#101 F20.0                      Incremental motion B to the left X-2.0781
N6 G00 X[ROUND[#100]+ROUND[#101]]        Incremental motion A+B to the right will be rounded
N7 M00                                     End of example

```

Rounding to a Fixed Number of Decimal Places

There are times when a fractional value has to be rounded to a *specific* (fixed) number of decimal places. Typically, *three* decimal places are required for the metric system, *four* decimal places are required for the English system, and perhaps *one* decimal place is required for cutting feedrate, regardless of the units selected.

In the following two examples, two given values will use a few techniques, providing the results of different rounding methods:

➤ Example 1 - Given fractional value is **over 0.5** :

```
#1 = 1.638719                               Value to be rounded to a specific number of decimal places
```

If the **ROUND** function is applied to this defined value, it will return the *next whole* number:

```
ROUND[#1]                                   Returns 2.0
```

In order to round the given value to a certain number of decimal places, the total of *three* steps will be necessary.

STEP 1 - The first step requires the given value to be *multiplied by the factor* of:

10	... to round off to one decimal place	
100	... to round off to two decimal places	
1000	... to round off to three decimal places	<i>typical for metric system</i>
10000	... to round off to four decimal places	<i>typical for English system</i>
	... and so on	

For example:

```
#2 = #1 * 1000                               Returns 1638.719 (Metric example)
#3 = #1 * 10000                              Returns 16387.19 (English example)
```

STEP 2 - The second step requires using the **ROUND** function on the *returned* value:

```
#2 = ROUND [#2]           Returns 1639.0      (based on the result of Step 1)
#3 = ROUND [#3]           Returns 16387.0     (based on the result of Step 1)
```

STEP 3 - The third step will divide the rounded value by the *same multiplying factor* as before:

```
#2 = #2/1000              Returns 1.639       (based on the result of Step 2)
#3 = #3/10000             Returns 1.6387      (based on the result of Step 2)
```

In the macro program, the three steps can be used as described, but a more common method is to process all three functions in a *single nested* statement:

```
#1 = 1.638719             Value to be rounded to a specific number of decimal places
#2 = ROUND [#1*1000]/1000 Returns 1.639
#3 = ROUND [#1*10000]/10000 Returns 1.6387
```

Improper rounding may cause a cumulative error in calculations !

➤ **Example 2** - Given fractional value is *under 0.5* :

```
#4 = 1.397528             Value to be rounded to a specific number of decimal places
```

If the **ROUND** function is applied to this given value, it will return the *last whole* number:

```
ROUND [#4]                Returns 1.0
```

In order to round the given value to a certain number of decimal places, the total of *three* steps will be necessary.

STEP 1 - The first step requires the given value to be *multiplied by the factor* of:

```
10      ... to round off to one decimal place
100     ... to round off to two decimal places
1000    ... to round off to three decimal places   typical for metric system
10000   ... to round off to four decimal places   typical for English system
... and so on
```

For example:

```
#5 = #4 * 1000            Returns 1397.528 (Metric example)
#6 = #4 * 10000           Returns 13975.28 (English example)
```

STEP 2 - The second step requires using the **ROUND** function on the *returned* value:

```
#5 = ROUND[#5]           Returns 1398.0      (based on the result of Step 1)
#6 = ROUND[#6]           Returns 13975.0     (based on the result of Step 1)
```

STEP 3 - The third step will divide the rounded value by the *same multiplying factor* as before:

```
#5 = #5/1000             Returns 1.398       (based on the result of Step 2)
#6 = #6/10000            Returns 1.3975      (based on the result of Step 2)
```

In the macro program, the three steps can be used as described, but a more common method is to process all three functions in *one nested* statement:

```
#4 = 1.397528           Value to be rounded to a specific number of decimal places
#5 = ROUND[#4*1000]/1000 Returns 1.398
#6 = ROUND[#4*10000]/10000 Returns 1.3975
```

Accuracy in rounding is extremely important, not only for the final dimensions of the machined part, but also for tracking errors in the macro program. Inaccuracies caused by cumulative rounding error are not always easy to find.

Always use care in programming rounded values

FUP and FIX Functions

The remaining two rounding functions are used to round a given value *up* or *down* only, regardless of whether the decimal portion is *over* or *under* 0.5.

The **FUP** function is designed to *round up* the given value (raise fractions less than 1.0).

```
FUP[0.00001]           Returns 1.0
FUP[0.5]                Returns 1.0
FUP[0.99999]           Returns 1.0
FUP[1.0]                Returns 1.0
```

The **FIX** function is designed to *round down* the given value (discard fractions less than 1.0) - *i.e.*, strip all values after decimal point.

```
FIX[0.00001]           Returns 0.0
FIX[0.5]                Returns 0.0
FIX[0.99999]           Returns 0.0
FIX[1.0]                Returns 1.0
```

The **FUP** and **FIX** function are commonly used in establishing the number of iterations (counting loops) and for other counting (rather than calculating) purposes.

Miscellaneous Functions

Five macro oriented miscellaneous functions are available for programming macro expressions. The following five functions are available:

SQRT	ABS	LN	EXP	ADP
-------------	------------	-----------	------------	------------

Do not confuse these macro functions with CNC miscellaneous functions, such as M01.

SQRT and ABS Functions

Only the first two functions listed are used frequently in macros.

The **SQRT** function calculates the *square root* of a number supplied between brackets:

SQRT [16]	Returns 4.0
SQRT [16.0]	Returns 4.0
#1 = 16.0	Returns 16.0
SQRT [#1]	Returns 4.0

The **ABS** function (absolute function) always returns a *positive* value of a given number:

ABS [-23.6]	Returns 23.6
ABS [23.6]	Returns 23.6

Using the **ABS** function is very useful when integrity of the supplied or calculated value is important in terms of a mathematical sign. The **ABS** function will always return a positive equivalent of the supplied numerical value and guarantees a returned positive number.

The next example uses the **ABS** function, to guarantee the required sign of a given number.

➤ Example - using the **ABS** function :

Programming a depth of a tool (end mill, drill, tap, etc.) as an assigned value in the macro call **G65**, chances are that the assignment representing the Z-depth will be entered as negative value, especially for jobs where the Z0 is set at the top of the part - for example:

```
G65 P8999 R2.0 Z-15.6 F175.0
```

Somewhere within the body of macro program O8999, the program section that calls the variable **#26** (Z-depth), *must* be entered *without* the minus sign:

```
...
G99 G81 R#18 Z#26 F#9
...
```


This is a correct application, but not a safe application. What happens, if the programmer accidentally enters the Z-depth argument as a positive value? The tool will move *above* the work, instead of *into* the work. It may not present a big problem but an irritation it is. Of course, the signs of the argument and macro data could be *reversed* - the G65 Z-depth argument will be a positive number, for example Z15.6, the variable call in the macro will be negative, for example Z-#26. Again - good, but still not a safe application. What if the argument is also defined as negative?

Using the **ABS** function - and a little ingenuity - the G65 Z-depth argument can be programmed with either a positive value *or* a negative value, and still get a negative cutting direction (Z-minus direction). Impossible? Study the next examples - whether the Z-depth argument is defined as positive or negative, only one macro is used with a guaranteed direction of the toolpath into the material:

➤ Positive Z-depth argument:

G65 P8999 R2.0 Z15.6 F175.0 *Z-depth argument is positive*

➤ Negative Z-depth argument:

G65 P8999 R2.0 Z-15.6 F175.0 *Z-depth argument is negative*

The key to success is the macro call. It must use the **ABS** function that will convert the argument into a positive value. Then using a negative Z-value, the tool motion will always be *into* the part:

G99 G81 R#18 Z-[ABS[#26]] F#9 *Negative Z guarantees negative tool motion*

Note the Z-depth entry - *it must negative within the macro!* Once the macro is proven and saved, it can be protected by a parameter setting, so there is no danger of an accidental change. What exactly will happen when the above macro statement is processed?

If the specified argument is *positive*, **ABS [#26]** will leave it as positive, so **ABS [#26]** will be equal to 15.6. Since the Z-value in the macro is fixed as negative, the negative sign will precede the stored value and the result will be Z-15.6, which is the desired and correct entry.

If the specified argument is *negative*, **ABS [#26]** will change it into a positive value, so the **ABS [#26]** will be equal to 15.6. Since the Z-value in the macro is fixed as negative, the negative sign will precede the stored value and the result will also be Z-15.6, which is the correct entry.

NOTE: This example is simple and perhaps even clever. Although it illustrates a relatively small macro function, it is included here not only for the purpose of defining variables, but also for the purpose of a professional approach to programming. In macros, one of the biggest assets the programmer may have is the ability *to predict what can go wrong, before it goes wrong*. What kind of input error is possible, or even likely? Is there a way to protect the macro flow from such an error? If there is, write the appropriate program code. If there is not, at least try to find a way.

Not all errors in calculations can be expected and prevented. Some errors may be possible, but virtually impossible to prevent. Perhaps a message or a comment to the CNC operator may do some good in minimizing the possibility.

LN, EXP and ADP Functions

The remaining three miscellaneous macro functions are used for special purposes only:

LN	Natural logarithm function
EXP	Exponent with the base 'e' function
ADP	Add a decimal point function

These functions are *not* available on 0/16/18/21 model controls.

The **LN** function, the **EXP** function, and the **ADP** functions are rarely used. On the controls that accept these functions, the **ADP** function may be the one most likely to offer some benefit.

The **ADP** is the *Add Decimal Point* function. It accepts a local variable (**#1** to **#33**) as an argument, and adds a decimal point to a value in the macro body, that was passed by the **G65** argument *without* a decimal point. Parameter **#7000** (bit *CVA*) *must* be set *0*. For example:

```
G65 P8999 Z25           No decimal point in the Z-assignment
```

During macro execution, the value of the Z-variable (**#26**) will be 25.0, if **ADP[#26]** is programmed. This is a function that even Fanuc recommends to avoid and program the decimal point *in the argument, if it is required*.

Logical Functions

For a powerful macro development, powerful programming tools are needed. Logical functions are some of these power tools and they can be divided into two groups:

- Logical functions used for creating a *CONDITION* or a *COMPARISON*
- Logical operators performed on *BINARY NUMBERS*

Boolean Functions

To the first group belong the six standard comparison operators (often called *Boolean operators*, or *Boolean functions*):

EQ	NE	GT	LT	GE	LE
-----------	-----------	-----------	-----------	-----------	-----------

Boolean functions compare two values and return a *true* or *false* condition:

EQ	=	Equal to
NE	=	Not equal to
GT	=	Greater than
LT	=	Less than
GE	=	Greater than or equal to
LE	=	Less than or equal to

Binary Numbers Functions

To the second group belong the three logical operators, used to perform logical operation on a *binary number*, bit by bit:

AND	OR	XOR
------------	-----------	------------

These three macro functions are used for logical comparisons in various programming applications. The two most commonly used functions are the **AND** and the **OR** functions; the **XOR** (*Exclusive OR*) is used very seldom. All three are used at every bit of 32 bits.

The **AND** and **OR** functions compare two given conditions. The compared conditions are evaluated, and return either the *TRUE* value or the *FALSE* value. True value means 'True', and False value means 'Not True'. In plain English, it is easy to understand the difference between the **AND** and the **OR** functions, because they follow the basic logic of the English language.

For example, the sentence "*Jack and Jill will go shopping*", means that *both of them* will go shopping. The sentence "*Jack or Jill will go shopping*", says something different - that only *one of them* will go shopping. These functions have their equivalent in all high level languages - they are called the bit values of *TRUE* and *FALSE*, and have one of the two possible values - 1 or 0.

For example, if a given *<Value 1>* is compared with a given *<Value 2>*, and used with the **AND** function, *both* values must be true for the whole statement to be true. On the other hand, if a given *<Value 1>* is compared with a given *<Value 2>*, and used with the **OR** function, only *one* value of the statement must be true for the whole statement to be true. In either case, a *TRUE* value returns 1, and a *FALSE* value returns 0.

Boolean and Binary Examples

As an exercise, evaluate the following macro data entries. The first group is the *given data*, the second group is the *evaluated data*, and the final third group is the *compared data*.

➤ Given data :

#1 = 100.0	Stored value is 100.0
#2 = #0	No data - variable is VACANT (empty, null)
#3 = 100.0	Stored value is 100.0
#4 = 150.0	Stored value is 150.0

➤ Evaluated data :

#5 = [#1 EQ #2]	Returns 0 = FALSE
#6 = [#2 EQ #3]	Returns 0 = FALSE
#7 = [#2 EQ #0]	Returns 1 = TRUE
#8 = [#1 EQ #3]	Returns 1 = TRUE
#9 = [#4 GT #3]	Returns 1 = TRUE

➤ Compared data :

```
#10 = [[#1 EQ #3] AND [#2 EQ #0]]    TRUE    because both values are true
#11 = [[#1 EQ #3] OR [#2 EQ #0]]    TRUE    because both values are true
#12 = [[#1 NE #4] OR [#4 LT #3]]    TRUE    because at least one value is true
#13 = [[#2 EQ #1] AND [#3 GT #4]]    FALSE   because neither value is true
#14 = [[#3 NE #0] OR [#1 EQ #2]]    FALSE   because only one value is true
```

In all previous examples, the correct use of the brackets [] in the macro is very important. If the evaluated conditions are more complex, for example in a multi-depth nesting application, the brackets will be nested as well, up to so many levels that the macro program will eventually become difficult to interpret. The solution to this problem is to avoid excessive nesting, and use multi-block definitions instead.

Conversion Functions

Special conversions in a macro program can be used for signal exchange to PMC and from PMC. (PMC is the abbreviation of *Programmable Machine Control*, and is not available on all Fanuc control models). PMC is Fanuc version of PLC - *Programmable Logic Controller*. The two functions associated with the conversion are:

BCD	BIN
-----	-----

BCD function converts *Binary Coded Decimal* format into a *Binary format*, and the **BIN** function converts *Binary format* into a *Binary Coded Decimal format*. These are *not* common functions in a typical macro application, but if used, the knowledge of binary numbers is essential.

Evaluation of Functions - Special Test

The knowledge of how Fanuc control system actually evaluates macro functions is a critical element of any macro programming. The next page contains a very comprehensive test that covers as many functions as possible, several of them dependent on each other. All answers are provided as *returned values* next to the macro statement, except the last one, which is a special challenge (see comments following the test). The test is based on the following **G65** block:

```
G65 P8888 B42.0 C1.427 H30.0 X0.003    Four arguments defined for the test
```

The above block calls a macro program O8888 and passes four defined arguments to variables **B=#2=42.0**, **C=#3=1.427**, **H=#11=30.0** and **X=#24=0.003**. The macro O8888 has been specifically designed for training purposes and applies various functions as examples of usage. The order of data entry must be followed as presented. Place a sheet of paper to hide the return values from view, take a calculator, and try to identify the return values before looking at the results (all leading zeros that may appear on the control screen are omitted in the returned values):

```

O8888 (EVALUATION OF FUNCTIONS - SPECIAL TEST)
#100 = #11          30.0000
#101 = #2           42.0000
#102 = #8           Vacant
#103 = 0            0.0000
#104 = #3           1.4270
#105 = #104+4.125   5.5520
#106 = #101-15.0    27.0000
#107 = #104*6.7     9.5609
#108 = #101/2       21.0000
#109 = #101/#104    29.432376
#110 = SIN[0]       *****          negative underflow
#111 = SIN[90]      1.0000
#112 = SIN[#101]    0.6691306
#113 = COS[0]       1.0000
#114 = COS[90]     *****          positive underflow
#115 = COS[#101]    0.7431448
#116 = TAN[0]       *****          negative underflow
#117 = TAN[90]     *****          positive overflow
#118 = TAN[#101]    0.9004041
#119 = ATAN[0.75]/[1.625] 24.77514
#120 = SQRT[16]     4.0000
#121 = SQRT[#100+5] 5.9160798
#122 = -13.125162   -13.125162
#123 = ABS[#122]    13.125162
#124 = 0.327187     0.327187
#125 = ROUND[#124] 0.0000
#126 = FIX[#124]    0.0000
#127 = FUP[#124]    1.0000
#128 = 0.8235       0.8235
#129 = ROUND[#128] 1.0000
#130 = FIX[#128]    0.0000
#131 = FUP[#128]    1.0000
#132 = 0.5          0.5000
#133 = ROUND[#132] 1.0000
#134 = FIX[#132]    0.0000
#135 = FUP[#132]    1.0000
#136 = 3.0          3.0000
#137 = ROUND[#136] 3.0000
#138 = FIX[#136]    3.0000
#139 = FUP[#136]    3.0000
#140 = #3-#120      -2.5730
#141 = #[#24]       Vacant
#142 = #105*#105     30.824704
#143 = [#120+1]/TAN[8.6] 33.060961
#144 = 6.2+14/3-2*8.32 -5.7733333
#145 = [6.2+14]/3-2*8.32 -9.9066667
#146 = SQRT[3.6]    1.8973666
#147 = [[#136+#128]*#104+#124] 5.7833215
#147 = #147*12      69.399858
#148 = SIN[#147]+#146 2.8334253
#149 = #[FUP#[ROUND[#148]]] ?????????

```

As established a number of times already, variables that are *vacant* have no value. Note that zero - 0 - is an actual real value, so a variable containing a zero value *is* occupied - *it is never vacant*. Variables that are displayed with ********* are usually the result of calculations involving *infinite* values (usually caused by the division by zero).

In the example, probably the most elaborate of all the entries is the last one - stored in variable **#149** - hence the challenge. Let's evaluate this variable - *and its return value* - in sufficient detail.

Order of Function Evaluation

Complex functions (such as nested functions) are always evaluated from the *inside out*. That means the innermost function of the supplied data is processed first, then the next function, then the one after and so on and on. In the test example, the innermost function for **#149** variable definition is **ROUND [#148]**.

Since the previously stored value of variable **#148** is 2.8334253, the function is evaluated as **ROUND [2.8334253]**, and its return value is 3. At this point, the function looks like this:

```
#149 = #[FUP[# [3]]]
```

... where the return value of the **ROUND** function has been substituted. The new innermost function now is **# [3]**. That is a reference to a variable **#3**. From the earlier definition, in the **G65** statement, **#3** is designated by the *C* letter and had been assigned value of 1.427. The definition at this point looks like this:

```
#149 = #[FUP[1.427]]
```

... where the previously stored value of the **#3** has replaced the inner calculation. The new current innermost calculation is **FUP [1.427]**. The **FUP** function returns the next higher integer, which is 2 in this case. So the next version of the evaluation is:

```
#149 = #[2]
```

This is a simple form to evaluate:

```
#149 = #[2]
```

is the same as

```
#149 = #2
```

Since **#2** is defined by the *B* letter in the **G65** macro call and *B* is equal to 42.0, therefore, the final meaning - *the ever important return value* - of variable **#149** is:

```
#149 = 42.0
```

Approach to Practical Applications

This section includes several samples of standard programs and macros working together. The following examples illustrate the macro format and the practical use of variables in different ways. As in a standard CNC program, when writing a macro, there has to be a purpose - *an objective*. In the examples, the single objective is to use macro features *to calculate the Z-depth for a drill cutting through a plate of a given thickness*. Several versions of the program will show the continuously improved progress in development and use of the many available programming tools. Compare the differences between individual programs.

The first few examples will use local variables, later examples will use common variables.

Using Local Variables

A reminder - all local variables used by the program will be discarded when **M30** or **M99** is executed, or when the *RESET* key at the control panel is pressed.

➤ Example 1

In the initial version, evaluation of the job finds that four items have to be dealt with:

- Drill diameter
- Plate thickness
- Tool point length
- Clearance for the drill penetration

In order to make the program flexible, choose the drill and the plate thickness as variable data, then select defaults for the tool point length and the penetration clearance. To store the drill diameter, variable **#1** will be used, to store the plate thickness, variable **#2** will be used. Since only 118° point angle drills will be used, the standard default constant of 0.3 is used to calculate the tool point length (formula determining the constant is listed later in this section). The other default in the program will be the clearance *below* the plate for the drill penetration. Selection of 1.5 mm is reasonable. The first program with macro features uses a very simple and basic approach:

```
O0011 (MAIN PROGRAM 1)
#1 = 15.0                (DRILL DIAMETER)
#2 = 13.0                (PLATE THICKNESS)
(-----)
#3 = #1*0.3+#2+1.5      (Z-DEPTH CALCULATION - POSITIVE VALUE OF 19.0)
(-----)
N1 G21
N2 G90 G00 G54 X100.0 Y50.0 S800 M03
N3 G43 Z5.0 H01 M08
N4 G99 G81 R2.5 Z-#3 F150.0 (DISTANCE-TO-GO IS Z-21.5)
N5 G80 Z5.0 M09
N6 G28 X100.0 Y50.0 Z5.0 M05
N7 M30
%
```

Look at the program only as the objective states - was the objective achieved? The example shows yes, it was. The total Z-axis tool travel will be Z-21.5 (this is indicated at the control as *Distance-To-Go*). Remember, the 'distance to go' is measured from the start point (R-level) to the Z-depth. What is the returned value of variable #3? Once the program is completed (M30 function), or the reset button is pressed, all local variables are cleared. The screen display of macro variables will show no values. There are no variable data shown, because there are no variable data stored - all were flushed out of the memory, because they were defined as *local* variables.

There is no need to keep a value already used in the memory, once the job is done. If the same program is applied for a different drill diameter and/or plate thickness, just change the #1 and/or #2 variables, and the new values of the Z-depth will be calculated automatically and correctly for any 118° tool point angle (within the macro framework)!

There are other parts of the program that could benefit from variable data, but the focus of the examples is on the Z-depth calculation only. As an improvement, for example, the spindle speed and the feedrate would have to change with each drill diameter and a few other data as well.

➤ Example 2

In the second version of the example, only a minor change will be made. Look at the calculation of the Z-#3 in block N4 above? The calculation of variable #3 always produces a *positive* result, but in the program, the Z-depth must be negative. By making two changes in the program, the result of the #3 calculation will be negative and a *positive entry* of Z#3 will be used in block N4:

```
O0012 (MAIN PROGRAM 2)
#1 = 15.0 (DRILL DIAMETER)
#2 = 13.0 (PLATE THICKNESS)
(-----)
#3 = -[#1*0.3+#2+1.5] (Z-DEPTH CALCULATION - NEGATIVE VALUE OF -19.0)
(-----)
N1 G21
N2 G90 G00 G54 X100.0 Y50.0 S800 M03
N3 G43 Z5.0 H01 M08
N4 G99 G81 R2.5 Z#3 F150.0 (DISTANCE-TO-GO IS Z-21.5)
N5 G80 Z5.0 M09
N6 G28 X100.0 Y50.0 Z5.0 M05
N7 M30
%
```

Which version is better? In this case, it is not a question of better or worse - it is a question of personal preference. Many programmers would probably prefer the *Example 1* (program O0011), because it shows the negative value in block N4, where it logically belongs (at least in my opinion). Based on the method of developing standard manual programs, it seems a better choice.

➤ Example 3

There is yet another way to get the negative result of the depth calculation - just redefine the original definition of the #3 variable. Substitution (redefinition) will take the old value of a variable and replaces it with the new value.


```

O0013 (MAIN PROGRAM 3)
#1 = 15.0                               (DRILL DIAMETER)
#2 = 13.0                               (PLATE THICKNESS)
(-----)
#3 = #1*0.3+#2+1.5   (Z-DEPTH CALCULATION - POSITIVE VALUE OF 19.0)
#3 = -#3             (POSITIVE #3 BECOMES NEGATIVE #3 OF -19.0)
(-----)
N1 G21
N2 G90 G00 G54 X100.0 Y50.0 S800 M03
N3 G43 Z5.0 H01 M08
N4 G99 G81 R2.5 Z#3 F150.0             (DISTANCE-TO-GO IS Z-21.5)
N5 G80 Z5.0 M09
N6 G28 X100.0 Y50.0 Z5.0 M05
N7 M30
%
```

Even better, these examples could benefit from the **ABS** function and guarantee a negative Z-depth in block N4, regardless of previous positive or negative input:

```
N4 G99 G81 R2.5 Z-[ABS[#3]] F150.0     (DISTANCE-TO-GO IS Z-21.5)
```

➤ Example 4

In the fourth version, a bit more flexibility will be added to the macro program. So far, the tool point length was based only on a 118° angle, and the penetration clearance was arbitrarily assigned the value of 1.5 mm. What if the tool angle is 135° or some other angle? What if the clearance of 1.5 mm is too small or too large? These values can be defined as variables - they will add flexibility to the program, with only a few more variables to fill.

```

O0014 (MAIN PROGRAM 4)
#1 = 15.0                               Drill diameter
#2 = 13.0                               Plate thickness
#3 = 118.0                             Drill point angle
#4 = 1.5                                Penetration clearance added
```

Once the variables have been defined, they have to be used. In the example, the main goal is to calculate the Z-depth, based on the input values (assignments). Since there is a constant mathematical relationship between the drill diameter and the drill point angle, it can be expressed in a well known and standard machine shop formula:

$$P = \frac{D}{2} \tan 90 \frac{A}{2}$$

☞ where ...

P = Drill point length
D = Drill diameter
A = Drill point angle

The formula may be nested in the macro, but nesting may not be easy to view, interpret, or change, if necessary. Because it may be somewhat difficult to include the whole formula into the macro right away, we choose a simpler method that segments (or splits) the variable input into smaller portions, and makes it more manageable (and definitely much easier to read). (The *Example 5* shows nested version of the formula). The next version of the example shows you this better way. We use variable #5 for the Z-depth calculation, and redefine it several times - all you need is *one calculation at a time*.

```
O0015 (MAIN PROGRAM 4)
#1 = 15.0          (DRILL DIAMETER - D IN THE FORMULA)
#2 = 13.0          (PLATE THICKNESS)
#3 = 118.0         (DRILL POINT ANGLE - A IN THE FORMULA)
#4 = 1.5           (PENETRATION CLEARANCE ADDED)
(-----)
#5 = #3/2          (ONE HALF OF TOTAL DRILL POINT ANGLE)
#5 = 90-#5        (ANGLE TO BE CALCULATED)
#5 = TAN[#5]      (TANGENT OF THE ANGLE TO BE CALCULATED)
#5 = #1/2*#5      (DRILL POINT LENGTH P FOR A GIVEN DRILL DIA)
#5 = #5+#2+#4     (PLATE THICKNESS AND CLEARANCE ADDED)
(-----)
N1 G21
N2 G90 G00 G54 X100.0 Y50.0 S800 M03
N3 G43 Z5.0 H01 M08
N4 G99 G81 R2.5 Z-[ABS[#5]] F150.0      (DISTANCE-TO-GO IS Z-21.5)
N5 G80 Z5.0 M09
N6 G28 X100.0 Y50.0 Z5.0 M05
N7 M30
%
```

This programming method takes four arguments (inputs) rather than the original two, but can be used with *any* drill diameter and with *any* penetration clearance amount. By redefining the variable #5, the computer memory is managed in a more efficient way. There is no need for each statement (calculation) to have its own variable, because there is no real benefit.

One interesting comment - the program comment in block is not true anymore - at least not exactly. The expected distance-to-go because instead of using a rounded 0.3 constant, the calculation uses the complete formula with trigonometric function:

```
#5 = #3/2          Returns 59.0
#5 = 90-#5        Returns 31.0
#5 = TAN[#5]      Returns 0.600861
#5 = #1/2*#5      Returns 4.506455
#5 = #5+#2+#4     Returns 19.006455
```

The comment in block N4 should be changed as well (if necessary):

```
N4 G99 G81 R2.5 Z-[ABS[#5]] F150.0      Actual distance-to-go will be 21.506455
```

Practically, this is of little consequence, but macro programmer should pay attention to details.

➤ Example 5

The next improvement will improve the previous example even more and eliminate the multiple definition of variable #5. Individual blocks of intermediate calculations are very useful to step through the program and perhaps make it easier for general understanding and even debugging. However, in most cases, a shorter program is preferred. Writing only a single definition of variable #5 means combining - *nesting* - all individual calculations. This example shows how:

```
O0016 (MAIN PROGRAM 5)
#1 = 15.0          (DRILL DIAMETER - D IN THE FORMULA)
#2 = 13.0          (PLATE THICKNESS)
#3 = 118.0         (DRILL POINT ANGLE - A IN THE FORMULA)
#4 = 1.5           (PENETRATION CLEARANCE ADDED)
(-----)
#5 = #1/2*TAN[90-#3/2]+#4+#2      (SINGLE FORMULA NESTED IN A BLOCK)
(-----)
N1 G21
N2 G90 G00 G54 X100.0 Y50.0 S800 M03
N3 G43 Z5.0 H01 M08
N4 G99 G81 R2.5 Z-[ABS[#5]] F150.0 (DISTANCE-TO-GO IS Z-21.5)
N5 G80 Z5.0 M09
N6 G28 X100.0 Y50.0 Z5.0 M05
N7 M30
%
```

➤ Example 6

So far, all five examples used variable assignments internally, in the main program. This final example will advance ahead another significant step, and defines the Z-depth as a true *Fanuc Custom Macro*. Macro has been defined as special a program that contains common but variable data. In this case, the common data is the calculation of the Z-depth itself. Both, the main program and the macro program will be needed. The main program, O0017 will be call macro O8005, containing the depth calculation. Because the result - the returned value - calculated in the macro O8005 has to be transferred to the main program, where it is actually used, some stronger programming tools will be needed. These tools involve the *Common Variables* (rather than just *Local Variables*), and the example is completed in the next section.

Using Common Variables

The basic concept of common variables has been already introduced. Based on a dictionary definition, the word 'common' means 'shared'. Common variables in a macro are *shared* by at least one other program, and usually more than one. Common variables create a shared bond between the main program, subprograms, and all macros that are to be connected by another program.

Variables designed as common are used to store program data. Within the group of common variables, and depending on the type of the stored data, there are two mini-groups. One covers the *System Variables*, the other covers the *I/O (Input/Output) Interface*.

Variables that are allowed in the 'common' group, belong to the variables range of #100 to #149 or, optionally, #100 to #199 - this group is called the 'non-holding' group, and #500 to #599 or, optionally, #500 to #999 - this group is called the 'holding' group. The terms *non-holding* and *holding* refer to the ability of the control system to keep the stored variable data in its memory. The *non-holding* group is retained until the system is restarted, the *holding* group is retained until removed by a program. Common variables are not cleared by M99 or M30 functions. Study the same exercise - this version uses macro call and common variables.

➤ Example 5 Revisited

The main program O0017, is the one that calls the macro program O8005, and passes the returned values of any define variable to the macro body. That way, the contents of macro program will never change, only the variable data supplied to it by the main program (G65 macro call).

```
O0017 (MAIN PROGRAM)
N1 G21
N2 G90 G00 G54 X100.0 Y50.0 S800 M03
N3 G43 Z5.0 H01 M08
N4 G65 P8005 D15.0 T13.0 A118.0 C1.5      Macro call block with arguments
N5 G99 G81 R2.5 Z-[ABS[#100]] F150.0      Distance-to-go is Z-21.5
N6 G80 Z5.0 M09
N7 G28 X100.0 Y50.0 Z5.0 M05
N8 M30
%
```

The associated macro O8005 is short and simple - it only contains the formula itself, this time using the variable assignments matching those that are called by the G65 macro call:

```
O8005 (MACRO FOR EXAMPLE O0017)
#100 = #7/2*TAN[90-#1/2]+#20+#3
M99
%
```

The formula built into the macro is identical to the formula used in previous examples. The only items that change are the calling parameters - *the input values* - in the G65 block:

$D = \#7 = 15.0$, $A = \#1 = 118.0$, $T = \#20 = 13.0$, and $C = \#3 = 1.5$

The advantage of this method is that macro program O8005 can be used for any job, providing the definition of arguments is defined consistently. In the examples, a single objective has been achieved by several methods, some very similar.

Speeds and Feeds Calculation

Another example of using common variables is for calculations of the spindle speed and the cutting feedrate, using formulas. Formulas are used in macros quite often, because their input can easily be replaced with variables. Based on standard machine shop formulas, many related values can be calculated, using the macro programming tools.

- ◆ The spindle speed (r/min = rpm) formula - Metric:

$$r/\text{min} = \frac{\text{m/min} \cdot 1000}{D}$$

- ◆ The spindle speed (r/min) formula - English:

$$r/\text{min} = \frac{\text{ft/min} \cdot 12}{D}$$

- ◆ The metric feedrate (mm/min) formula:

$$\text{mm/min} = r/\text{min} \cdot \text{mm/tooth} \cdot N$$

- ◆ The English feedrate (IPM - in/min - inches per minute) formula:

$$\text{in/min} = r/\text{min} \cdot \text{in/tooth} \cdot N$$

☞ where ...

r/min	=	Revolutions per minute (spindle speed) - also 'rpm'
m/min or ft/min	=	Peripheral speed in meters or feet per minute
	=	Constant pi (3.14159265359...)
D	=	Drill diameter (Metric or English)
mm/min	=	Millimeters per minute feedrate (Metric only)
mm/tooth or in/tooth	=	Chipload per cutting edge rating in mm/tooth or inches/tooth
N	=	Number of teeth in a cutter (number of cutting flutes)

☞ Example 7

```
O0017 (MAIN PROGRAM)
N1 G21
N2 G65 P8006 D12.0 F50.0 C0.15 T2 (MACRO CALL WITH DEFINITIONS)
N3 G90 G00 G54 X100.0 Y50.0 S#101 M03 (SPINDLE SPEED CALCULATED BY MACRO)
N4 G43 Z5.0 H01 M08
N5 G99 G81 R2.5 Z-19.0 F#102 (FEEDRATE CALCULATED BY MACRO)
N6 G80 Z5.0 M09
N7 G28 Z5.0 M05
N8 M01

O8006 (MACRO FOR EXAMPLE O0017)
#101 = FIX[ [#9*1000]/[3.141593*#7]] (SPINDLE SPEED CALCULATION)
#102 = #101*#3*#20 (FEEDRATE CALCULATION)
M99
%
```

In the **G65** macro call, the values that are relevant to the current programming job have to be supplied to the macro. In the example, D12.0 definition means that the variable **#7** will store the value of 12.0 mm drill diameter, F50.0 means that the variable **#9** will store 50 m/min peripheral speed value, C0.15 means that the variable **#3** will store 0.15 mm/rev chip load value, and T2 means that the variable **#20** will store 2 cutting edges (flutes).

Notice the **FIX** function used for the *r/min* definition. If the formula is taken exactly, it will be

$$[\#9*1000]/[3.141593*\#7]$$

which is the same in a macro as

$$(50.0*1000)/(3.141593*12.0)$$

is on a calculator, and returns exactly

1326.291 r/min - revolutions per minute

The basic rules of CNC programming do not allow a decimal point in the spindle speed specification. The integer of 1326 (used as S1326 in the program) is allowed, but a real number of 1326.291 is not. The macro **FIX** function will discard all the decimal places of the calculated value, leaving only the integer. There will be no rounding, just the isolated integer value - the **FIX** function strips the decimal values of a real number, leaving only the integer. In the case of the spindle speed, the *r/min* will be accurate within one revolution.

10

SYSTEM VARIABLES

The last group of variables is called the *System Variables*. The word ‘system’ in the description of *System Variables* means the *Control System variables*. This group of variables is rather a special group and cannot be compared to the variable types already discussed (local and common). It is equally important in macros, but stands on its own.

In a macro program, this group is used to address the *registers* of the control memory (also called *addressable memory locations*). In certain situations (not normally), some system variables can also be used to change some *internal* data (also called *system* data) stored within the CNC system. For example, a work coordinate system (work offset) can be changed by manipulating the system variables (changing one or more system variables). In a similar way, items like the tool length compensation, macro alarms, parameter settings, parts count, modal values of the G-codes (plus several additional codes), and many others, can be changed as well. System variables are extremely important for automated environment, such as probing, unmanned and agile manufacturing, transfer systems, etc. There are many system variables available for each control system, and there significant differences between various control systems (even within the various Fanuc models available). It is very unlikely that any programmer will ever need them all. The control reference manual will come very handy as a reference resource.

Identifying System Variables

When working with system variables, there are two very important features to be aware of right from the beginning. Both relate to the way the system variables are identified by the control:

- ◆ **System variables are numbered from #1000 and up (four or five digit numbers)**
- ◆ **System variables are *not* displayed on the control display screen**

The numbering is fixed by Fanuc and cannot be changed. In this arbitrarily numbered system, a reference book or manual is required for each control model in the shop. Fanuc provides such a manual with the purchase of a particular control system. A great number of system variables are identified in this handbook as well.

Since the system variables cannot be directly displayed on the screen (applies to a large number of controls), there must be another way of finding what their current values are. The method used is called a *value transfer*. In the program, or in MDI mode, the value of a system variable can be *transferred* into a local or common variable. This chapter deals with this subject as well.

By organizing work, a tremendous step forward can be made. In the case of system variables, the first significant step to their better organization is by *grouping* them.

System Variables Groups

System variables are solely dependent on the CNC system. This is a very important and accurate statement. It means that different Fanuc controls may take on different meanings of the control system variables. Programmers and service technicians have to know which model of the control supports which features and the assignment of variables. The macro program you develop may only be used for the selected control unit and, most likely, for the selected machine as well.

Over the years, Fanuc has brought many different control models to the industry. Only the most common and the current ones are discussed here. They are listed with the FS (*FS* stand for *Fanuc System* or *Fanuc Series*) abbreviation:

◆ FS-0, FS-10, FS-11, FS-15, FS-16, FS-18, FS-21 (plus variations)

Older controls are easy to figure out as well, but the control reference manual will be needed. For example, Fanuc 3 is relatively similar to Fanuc 0. Fanuc 6 is the predecessor of Fanuc 10/11. Keep in mind that the model numbers do not indicate the higher or lower level of the control features. From the list, it is the Fanuc 15 that is classified as the top of the line, although its number is smaller than some others. All controls are available in the *milling* version (FS-xxM or FS-xxMB), for example FS-15M, and the *turning* version (FS-xxT), for example Fanuc 16T or 16TB. They are also available for wire EDM, grinding, and several other machine types, but only milling and turning are of an interest in this handbook.

Read and Write Variables

Variables contain data - data that change, or *variable data*. There are two types of variables in terms of how the data is acquired. Some variables can be written to, meaning they can be changed via a program or by *MDI*. This type of variables can also be read by the system, and stored values processed by the system. System variables in this group are called *Read and Write* variables.

The other group type covers variables that can be processed, displayed on the screen using local or common variables, but they *cannot* be changed by the user (CNC programmer, operator or service technician). These are so called *Read Only* systems variables. These are the most common system variables for typical work.

In the subsequent listings, the *Read and Write* variables are marked with an asterisk [*].

Displaying System Variables

Since not all Fanuc systems can display system variables directly, they must be displayed through the local or common variables. This is called variable transfer, or variable redefinition, or variable substitution. For example (Fanuc 15M):

```
#101 = #5221           X-value of G54 work offset stored in #101 from #5221
#102 = #5222           Y-value of G54 work offset stored in #102 from #5222
```

The local or the common variable can be displayed on the control screen.

System Variables for Fanuc Series 0

#1000 through #1015, #1032 . . .	Data In . . . (DI)
#1100 through #1115, #1132 . . .	Data Out . . . (DO)
#2000 through #2200	Tool compensation values (tool offsets)
#2500	External work offset value along the X-axis
#2600	External work offset value along the Y-axis
#2700	External work offset value along the Z-axis
#2800	External work offset value along the 4th axis
#2501	G54 work offset value along the X-axis
#2601	G54 work offset value along the Y-axis
#2701	G54 work offset value along the Z-axis
#2801	G54 work offset value along the 4th axis
#2502	G55 work offset value along the X-axis
#2602	G55 work offset value along the Y-axis
#2702	G55 work offset value along the Z-axis
#2802	G55 work offset value along the 4th axis
#2503	G56 work offset value along the X-axis
#2603	G56 work offset value along the Y-axis
#2703	G56 work offset value along the Z-axis
#2803	G56 work offset value along the 4th axis
#2504	G57 work offset value along the X-axis
#2604	G57 work offset value along the Y-axis
#2704	G57 work offset value along the Z-axis
#2804	G57 work offset value along the 4th axis
#2505	G58 work offset value along the X-axis
#2605	G58 work offset value along the Y-axis
#2705	G58 work offset value along the Z-axis
#2805	G58 work offset value along the 4th axis
#2506	G59 work offset value along the X-axis
#2606	G59 work offset value along the Y-axis
#2706	G59 work offset value along the Z-axis
#2806	G59 work offset value along the 4th axis
#3000	User macro generated alarm
#3001	Clock 1 - unit 1 ms
#3002	Clock 2 - unit 1 hour
#3003, #3004	Cycle Operation Control
#3005	Setting
#3011	Clock information - Year, Month, Day
#3012	Clock Information - Hour, Minute, Second
#3901	Number of parts machined
#3902	Number of parts required
#4001 to #4022	Modal Information <i>Pre-reading block - G-code groups</i>
#4102 to #4130	Modal Information <i>Pre-reading block - B, D, F, H, M, N, O, S, T codes</i>

#5001 to #5004	Block end position
#5021 to #5024	Machine coordinates position
#5041 to #5046	Work coordinates position
#5061 to #5064	Skip signal position
#5081 to #5086	Tool length compensation value
#5101 to #5104	Servo deviation

Fanuc Model 0 Compared to Other Models

Tables in this chapter describe the various groupings of system variables for several Fanuc control models. Fanuc Model 0 (FS-0) is one of them, described above. In relation to the other controls, FS-0 is the most modest control, and supports much smaller set of variables than the higher level controls. This is particularly noticeable in the use of various system variables relative to tool offsets, described in detail in the next chapter.

Always understand system variables for a particular control and machine unit
There is no guarantee of compatibility between different control or machine models

Most of the examples in this handbook are for the FS-15M and FS-16/18/21M controls and their B-version, if available, as well as their lathe equivalents. These two control groups are the most widely used models in machine shops using macros. In many ways, the models FS-10 and FS-11 are very close, but expect many different reference numbers.

System Variables for Fanuc Series 10/11/15

#1000 through #1035.	Data In (DI - from PMC)	
#1100 through #1135.	Data Out (DO - to PMC)	[*]
#2000 through #2999.	Tool compensation values (tool offsets)	[*]
#10001 through #13999.	Additional tool offsets	[*]
#3000, #3006	User macro generated alarm or message	[*]
#3001, #3002	Clock	[*]
#3003, #3004	Cycle Operation Control	[*]
#3007	Mirror Image	
#3011, #3012	Clock Information (time variables)	
#3901, #3902	Number of parts (parts count variables)	[*]
#4001 to #4130	Modal Information	<i>Pre-reading block</i>
#4201 to #4330	Modal Information	<i>Executing block</i>
#5001 to #5006	Block end position	
#5021 to #5026	Machine coordinates position	
#5041 to #5046	Work coordinates position	
#5061 to #5066	Skip signal position	
#5081 to #5086	Tool length compensation value	
#5101 to #5106	Servo deviation	

#5201 to #5206	Work offset value (shift or common) or up to #5215 [*]
#5221 to #5226	Work offset value G54 or up to #5235 . . . [*]
#5241 to #5246	Work offset value G55 or up to #5255 . . . [*]
#5261 to #5266	Work offset value G56 or up to #5275 . . . [*]
#5281 to #5286	Work offset value G57 or up to #5295 . . . [*]
#5301 to #5306	Work offset value G58 or up to #5315 . . . [*]
#5321 to #5326	Work offset value G59 or up to #5335 . . . [*]

[*] marks system variables of the *Read and Write* type

System Variables for Fanuc Series 16/18/21

#1000 through #1015.	Data In DI Sending 16-bit signal from PMC to macro (reading bit by bit)
#1032	Used for reading all 16-bits of a signal at one time
#1100 through #1115.	Data Out DO Sending 16-bit signal from macro to PMC (writing bit by bit)
#1132	Used for writing all 16-bits of a signal at one time to the PMC
#1133	Used for writing all 32-bits of a signal at one time to the PMC - Values of -99999999 to +99999999 may be used for #1133
#2001 through #2200.	Tool compensation values . (offsets 1-200) Memory Type A - Milling
#10001 through #10999.	Tool compensation values . (offsets 1-999) Memory Type A - Milling
#2001 through #2200.	Wear offset values (offsets 1-200) Memory Type B - Milling
#2201 through #2400.	Geometry offset values . . (offsets 1-200) Memory Type B - Milling
#10001 through #10999.	Wear offset values (offsets 1-999) Memory Type B - Milling
#11001 through #11999.	Geometry offset values . . (offsets 1-999) Memory Type B - Milling
#2001 through #2200.	Wear offset values of H-code (offsets 1-200) Memory Type C - Milling
#2201 through #2400.	Geometry offset values of H-code . (offsets 1-200) Memory Type C - Milling
#10001 through #10999.	Wear offset values of H-code (offsets 1-999) Memory Type C - Milling

#11001 through #11999.	Geometry offset values of H-code . (offsets 1-999) Memory Type C - Milling
#12001 through #12999.	Wear offset values of D-code (offsets 1-999) Memory Type C - Milling
#13001 through #13999.	Geometry offset values of D-code . (offsets 1-999) Memory Type C - Milling
#2500	External work offset value along the X-axis
#2600	External work offset value along the Y-axis
#2700	External work offset value along the Z-axis
#2800	External work offset value along the 4th axis
#2501	G54 work offset value along the X-axis
#2601	G54 work offset value along the Y-axis
#2701	G54 work offset value along the Z-axis
#2801	G54 work offset value along the 4th axis
#2502	G55 work offset value along the X-axis
#2602	G55 work offset value along the Y-axis
#2702	G55 work offset value along the Z-axis
#2802	G55 work offset value along the 4th axis
#2503	G56 work offset value along the X-axis
#2603	G56 work offset value along the Y-axis
#2703	G56 work offset value along the Z-axis
#2803	G56 work offset value along the 4th axis
#2504	G57 work offset value along the X-axis
#2604	G57 work offset value along the Y-axis
#2704	G57 work offset value along the Z-axis
#2804	G57 work offset value along the 4th axis
#2505	G58 work offset value along the X-axis
#2605	G58 work offset value along the Y-axis
#2705	G58 work offset value along the Z-axis
#2805	G58 work offset value along the 4th axis
#2506	G59 work offset value along the X-axis
#2606	G59 work offset value along the Y-axis
#2706	G59 work offset value along the Z-axis
#2806	G59 work offset value along the 4th axis
#3000	User macro generated alarm
#3001	Clock 1 - unit 1 ms
#3002	Clock 2 - unit 1 hour
#3003	Control of single block, wait signal for FIN
#3004	Control of feedhold, feedrate override, exact stop check
#3005	Setting
#3011	Clock information - Year, Month, Day
#3012	Clock Information - Hour, Minute, Second
#3901	Number of parts machined
#3902	Number of parts required

#4001 to #4022	Modal Information <i>Pre-reading block - G-code groups</i>
#4102 to #4130	Modal Information <i>Pre-reading block - B, D, F, H, M, N, O, S, T codes</i>
#5001 to #5008	Block end position
#5021 to #5028	Machine coordinates position
#5041 to #5048	Work coordinates position (Absolute position)
#5061 to #5068	Skip signal position
#5081 to #5088	Tool length compensation value
#5101 to #5108	Servo deviation
#5201 to #5208	External work offset value (1st to 8th axis)
#5221 to #5228	G54 work offset value (1st to 8th axis)
#5241 to #5248	G55 work offset value (1st to 8th axis)
#5261 to #5268	G56 work offset value (1st to 8th axis)
#5281 to #5288	G57 work offset value (1st to 8th axis)
#5301 to #5308	G58 work offset value (1st to 8th axis)
#5321 to #5328	G59 work offset value (1st to 8th axis)

Variables in the range of #5201 and #5328 are optional variables for work offsets.

#7001 to #7008	G54.1 P1 additional work offset value (1st to 8th axis)
#7021 to #7028	G54.1 P2 additional work offset value (1st to 8th axis)
#7041 to #7048	G54.1 P3 additional work offset value (1st to 8th axis)
#7061 to #7068	G54.1 P4 additional work offset value (1st to 8th axis)
#7081 to #7088	G54.1 P5 additional work offset value (1st to 8th axis)
#7101 to #7108	G54.1 P6 additional work offset value (1st to 8th axis)
#7121 to #7128	G54.1 P7 additional work offset value (1st to 8th axis)
#7141 to #7148	G54.1 P8 additional work offset value (1st to 8th axis)
#7161 to #7168	G54.1 P9 additional work offset value (1st to 8th axis)
#7181 to #7188	G54.1 P10 additional work offset value (1st to 8th axis)
#7201 to #7208	G54.1 P11 additional work offset value (1st to 8th axis)
#7221 to #7228	G54.1 P12 additional work offset value (1st to 8th axis)
#7241 to #7248	G54.1 P13 additional work offset value (1st to 8th axis)
#7261 to #7268	G54.1 P14 additional work offset value (1st to 8th axis)
#7281 to #7288	G54.1 P15 additional work offset value (1st to 8th axis)
#7301 to #7308	G54.1 P16 additional work offset value (1st to 8th axis)
#7321 to #7328	G54.1 P17 additional work offset value (1st to 8th axis)
#7341 to #7348	G54.1 P18 additional work offset value (1st to 8th axis)
#7361 to #7368	G54.1 P19 additional work offset value (1st to 8th axis)
#7381 to #7388	G54.1 P20 additional work offset value (1st to 8th axis)
#7401 to #7408	G54.1 P21 additional work offset value (1st to 8th axis)
#7421 to #7428	G54.1 P22 additional work offset value (1st to 8th axis)
#7441 to #7448	G54.1 P23 additional work offset value (1st to 8th axis)
#7461 to #7468	G54.1 P24 additional work offset value (1st to 8th axis)
#7481 to #7488	G54.1 P25 additional work offset value (1st to 8th axis)
#7501 to #7508	G54.1 P26 additional work offset value (1st to 8th axis)
#7521 to #7528	G54.1 P27 additional work offset value (1st to 8th axis)
#7541 to #7548	G54.1 P28 additional work offset value (1st to 8th axis)
#7561 to #7568	G54.1 P29 additional work offset value (1st to 8th axis)
#7581 to #7588	G54.1 P30 additional work offset value (1st to 8th axis)
#7601 to #7608	G54.1 P31 additional work offset value (1st to 8th axis)
#7621 to #7628	G54.1 P32 additional work offset value (1st to 8th axis)
#7641 to #7648	G54.1 P33 additional work offset value (1st to 8th axis)

#7661 to #7668	G54.1 P34 additional work offset value (1st to 8th axis)
#7681 to #7688	G54.1 P35 additional work offset value (1st to 8th axis)
#7701 to #7708	G54.1 P36 additional work offset value (1st to 8th axis)
#7721 to #7728	G54.1 P37 additional work offset value (1st to 8th axis)
#7741 to #7748	G54.1 P38 additional work offset value (1st to 8th axis)
#7761 to #7768	G54.1 P39 additional work offset value (1st to 8th axis)
#7781 to #7788	G54.1 P40 additional work offset value (1st to 8th axis)
#7801 to #7808	G54.1 P41 additional work offset value (1st to 8th axis)
#7821 to #7828	G54.1 P42 additional work offset value (1st to 8th axis)
#7841 to #7848	G54.1 P43 additional work offset value (1st to 8th axis)
#7861 to #7868	G54.1 P44 additional work offset value (1st to 8th axis)
#7881 to #7888	G54.1 P45 additional work offset value (1st to 8th axis)
#7901 to #7908	G54.1 P46 additional work offset value (1st to 8th axis)
#7921 to #7928	G54.1 P47 additional work offset value (1st to 8th axis)
#7941 to #7948	G54.1 P48 additional work offset value (1st to 8th axis)

Variables in the range of #7001 and #7948 are optional and are only available if *additional* work offset system G54.1 P1 to G54.1 P48 is available (or G54 P1 to G54 P48).

Organization of System Variables

The preceding lists of many system variables have been provided for reference only. They look boring - they are 'just' numbers. Yet, upon a more careful look, a certain *pattern* can be detected in the method the variable are numbered (at least in most places). Many system variables are numbered logically within groups, even if the numbers are different for each control model.

For example, there is a noticeable numbering pattern in the section relating to the work offsets, listed in the preceding definition. Here is the repeated listing:

#5201 to #5208	External work offset value (1st to 8th axis)
#5221 to #5228	G54 work offset value (1st to 8th axis)
#5241 to #5248	G55 work offset value (1st to 8th axis)
#5261 to #5268	G56 work offset value (1st to 8th axis)
#5281 to #5288	G57 work offset value (1st to 8th axis)
#5301 to #5308	G58 work offset value (1st to 8th axis)
#5321 to #5328	G59 work offset value (1st to 8th axis)

What is the pattern and - even more important - why is it significant? Each set of variables (seven sets listed above) contains variable numbers that differ by the amount of *twenty*. First set starts with system variable #5201, the second set with system variable #5221, the third set with system variable #5241, and so on. The seven sets cover one external work offset and six standard work offsets, G54–G59. Additional sets (G54.1 series), if available, use the same numbering logic, but the variables numbering starts at #7001.

In macros, it is often important to address these system variables in an organized way, with some logical and efficient approach. Taking advantage of the increments like 1 or 20, macro development can use loops with counters, if required.

The following table conveys the same information as the seven lines above, perhaps even better, but is without descriptions:

Axis	External	G54	G55	G56	G57	G58	G59
1st = X	#5201	#5221	#5241	#5261	#5281	#5301	#5321
2nd = Y	#5202	#5222	#5242	#5262	#5282	#5302	#5322
3rd = Z	#5203	#5223	#5243	#5263	#5283	#5303	#5323
4th	#5204	#5224	#5244	#5264	#5284	#5304	#5324
5th	#5205	#5225	#5245	#5265	#5285	#5305	#5325
6th	#5206	#5226	#5246	#5266	#5286	#5306	#5326
7th	#5207	#5227	#5247	#5267	#5287	#5307	#5327
8th	#5208	#5228	#5248	#5268	#5288	#5308	#5328

No doubt, the table looks better organized than a plain list; it also is longer and does not contain any descriptions. It does not matter which representation is better, this methodical numbering system offers numerous benefits. It is not the cosmetics of the numbering system, it is a practically oriented numbering system that just happens to look appealing as well. This numbering system is suitable to use formulas in the macros, with variables, and allows calculation of the required address number based on the number of another address.

Take, for example, the following situation. If the calculations are based on the system variable #5201, all that is needed is a simple multiplication to get another coordinate system:

- Through the macro, add 20 times 1 to get the X-value for G54
- Through the macro, add 20 times 2 to get the X-value for G55
- Through the macro, add 20 times 3 to get the X-value for G56
- Through the macro, add 20 times 4 to get the X-value for G57
- Through the macro, add 20 times 5 to get the X-value for G58
- Through the macro, add 20 times 6 to get the X-value for G59

The value of 20 in this case is called the *shift* value. Of course, *any other* variable can be used as the base variable for the calculations. The logic of this approach can be used with many calculations, using the built-in numbering method. Going a little step further, think about how to handle the jump from one axis to another. Take it as a small challenge, but the next section will reveal the process and explanation.

Resetting Program Zero

At least a small but quite practical application example is in place here. Its purpose is to illustrate the application of system variables for a desirable, yet simple process. The system variable will be used in an actual macro program. Macro program development will be covered later, so a small preview now may be useful then. The project is quite simple, and the small macro program code can be very useful in everyday CNC machining. The macro example will only do one thing - it will reset the *current work offset setting to program zero at the current tool location*, using the topics discussed previously. This is known as *zero shift* or *datum shift*.

A typical application of this macro could be a situation, where the work offset is set to the corner of a part for the convenience during setup, then changed by the macro to the center of a circle (bolt circle, for instance) for the convenience of programming. There are several other ways to do it, for example use **G52**. Our job is to create a macro for the same purpose. In O8007, the added comments describe what each block does. Check the system variables for Fanuc 15M earlier in the chapter, so you know what they mean.

```
O8007
(MACRO TO RESET PROGRAM ZERO AT CURRENT TOOL POSITION - VERSION 1)
N101 #1 = #4214           Store the current coordinate system number (54 to 59)
N102 #1 = #1-53          Store the current coordinate system group (1 to 6)
N103 #1 = 20*#1          Calculate the shift value for the current group (based on 20)
N104 #1 = #1+5201        Identify the applicable variable number
N105 #[#1] = #5021       Store the current X-axis machine coordinate in new variable
N106 #[#1+1] = #5022    Store the current Y-axis machine coordinate in new variable
N107 M99                 Exit macro
%
```

The block numbering in macro is used for reference only, and is not necessary. Local variable #1 was used, but any other local variable could have been used instead, for example, #33. This is just one version of the macro, and several others may be used. Improvements could include nested definitions and perhaps a totally different approach. This example will be revisited later, as one of the practical projects.

Special 'secret' of macro O8007 is the block N102. Notice that an *arbitrary* value of 53 is subtracted from the current value of variable #1. Since #1 stores the current work offset number (defined in block N101), subtracting 53 from it will return a value of 1 for **G54**, 2 for **G55**, 3 for **G56**, 4 for **G57**, 5 for **G58**, and 6 for **G59**. Block N103 will take this new value, and multiply it by 20 - remember that 20 is the arbitrary shift amount for work offset system variables. In this case, the shift of 20 is used for **G54**, 40 for **G55**, 60 for **G56**, 80 for **G57**, 100 for **G58**, and 120 for **G59**. Block N104 will add the *number* of 5201 to the shifted value, and *becomes* 5221 for **G54**, 5241 for **G55**, 5261 for **G56**, 5281 for **G57**, 5301 for **G58**, and 5321 for **G59**. Block N105 uses the *current number* and changes it into a genuine system variable number, for example 5221 will now be #5221, and so on. Since the system variable is on the left, it will *written-to*, using the current machine coordinate for X (still in block N105). Block N106 adds the value of one, and does the same think for the Y-axis.

Note what #[#1] means: take the returned value of variable #1 and convert it to a legitimate number of a variable. For example,

```
#1 = 100           Value of 100 is stored in variable 1
#100 = 1200.0     Value of 1200.0 is stored in variable 100
#2 = #[#1]        Definition is equivalent to #2=#100, so value of #2 is also 1200.0
```


11

TOOL OFFSET VARIABLES

The last chapter listed the system variables available for common - but different - Fanuc control models. It focused to some extent on handling the work offsets, mainly the macro applications of the standard set of **G54** to **G59** preparatory commands. In addition to the work offsets, there are also offsets relating to the cutting tool and many system variables related to tool offsets - as an *addition* to the work offsets. In fact, there are so many of them, that a separate chapter is necessary. The subject of tool offsets and the system variables that relate to them, continues the subject of system variables discussed in the previous chapter, but in a different specific area.

System Variables and Tool Offsets

In macro programming generally, and in on-machine probing (in-process gauging) particularly, the current values of various offsets change frequently, and have to be controlled automatically, for the most reliable and repeatable results. This is done through various custom made macro programs and routines. Into this category belong two special groups of offsets (also called compensations) that relate to certain measurement values of cutting tools:

- ◆ **Tool length offset** ... and the related applicable G-codes:
 G43, G44, and G49
- ◆ **Cutter radius offset** ... and the related applicable G-codes:
 G40, G41, and G42

Values and settings of either group of offsets can be read directly by a macro program, or written to by a macro, using the system variables of the Fanuc control system. Depending on the Fanuc model, the usage of these variables may be somewhat complicated. In order to organize the process, Fanuc distinguishes the tool offset application in a macro by three special groups, known as the *Tool Offset Memory Groups*. Even if a CNC machine does not have a macro option installed or active, it is a good idea to know what type of offset memory the machine has. This knowledge is very important for standard CNC programming as well, and it is surprising how many CNC programmers and operators do not have a clue of what type of the tool offset memory a particular CNC machine actually has. *Chapter 5* covered the subject of tool offset memory types in sufficient depth. Focus of this chapter will be the relationship of these offsets to system variables.

The tool offset memory groups are related to the particular control model, and can be established quite easily by looking at the control screen, and pressing the **OFFSET** button key on the keyboard. The number of columns and the column headings (columns contents) will provide the basic information. Do not expect to find the group itself listed or otherwise identified, however. You have to actually *know* the exact differences between the three groups, and that is the subject briefly revisited in this chapter.

Tool Offset Memory Groups

The memory registers that store the tool offsets depend on the model of the control, and its type (*milling or turning*). Programmers should always know which memory type is available on each Fanuc control in the machine shop. There are three groups for the milling controls, and are identified by the capital letters A, B, and C. There are two groups for the turning controls, and are identified by the capital letters A, and B. An earlier *Chapter 5* dealt with the subject of *Data Setting* - it described the *appearance* of each offset group on the control display screen (CRT). Reviewing the three main types used for the milling controls (usually only one type applies to lathes), will help to consider them in the context of the system variables used in macros.

Tool Offset Memory - Type A

This lowest level group is also known as the *shared offset group*. It can be recognized it by its simplicity. There is only a single column available in the control system to enter both the tool length offset values *and* the cutter radius offset values. That means the tool offset for the length is stored in the *same registry area* of the control as the tool offset for the radius. If a particular tool requires both offsets in the same program, a distinction between them must include different offset numbers. The same registry area *shares* both types of offsets. For example:

```
(TOOL 04 ACTIVE)
...
G43 Z2.0 H04           Uses tool length offset 04   (H04)
...
G01 G41 X50.0 D34     Uses cutter radius offset 34   (D34)
...
```

On some machines, the D-offset cannot be used, and the H-offset must be used also for the cutter radius:

```
(TOOL 04 ACTIVE)
...
G43 Z2.0 H04           Uses tool length offset 04   (H04)
...
G01 G41 X50 H34       Uses cutter radius offset 34   (H34)
...
```

The difference by 30 offset numbers is strictly optional, some programmers prefer 50. It only suggests a possible choice that is both suitable and practical. Any other number that is convenient is acceptable, as long as it is within the range of the available offset numbers. The more offsets are available, the higher increment can be used. On the practical side, determine the increment not only by the number of available offsets but also by their practical nature. For example, either the tool length offset or the cutter radius offset may have several values for any single tool, in some applications. Which one of them is more likely to have a multiple value in a practical application? Certainly the cutter radius offset. That means allocating a larger range of offset numbers for the radius offset than for the tool length offset.

Tool Offset Memory - Type B

The next type of the tool offset memory type is the *Memory Type B*. It is very similar to the *Memory Type A*, but on the control screen appears as two columns, not one. There is a separation between the *Geometry Offset* and the *Wear Offset*. As in *Memory Type A*, there is no distinction between the tool length offset number, and the tool radius offset number. The benefit of this memory type is that a nominal offset value (called the *geometry* offset) can be input and any adjustments and fine tuning are done in a separate column, called the *wear* offset. Since the ‘fine-tuning’ of the offset values takes place in two separate offset registers, the nominal offset value (geometry) is not normally changed. The usage of the offsets in the CNC program is *exactly* the same as in examples for *Memory Type A*:

(TOOL 04 ACTIVE)

```

...
G43 Z2.0 H04           Uses tool length offset 04   (H04)
...
G01 G41 X50.0 D34     Uses cutter radius offset 34   (D34)
...

```

On some machines, the D-offset cannot be used, and an H-offset must also be used for the cutter radius:

(TOOL 04 ACTIVE)

```

...
G43 Z2.0 H04           Uses tool length offset 04   (H04)
...
G01 G41 X50.0 H34     Uses cutter radius offset 34   (H34)
...

```

Note that the last program example is *identical* to the one before (for *Type A*). What is different is the methods of inputting values to the offsets, and that happens on the machine, during the setup process, not in the program.

Tool Offset Memory - Type C

The tool offset *Memory Type C* is the latest and the most flexible of the three. Like *Memory Type B*, it distinguishes between the *Geometry Offset* and the *Wear Offset*. In addition, it separates the *Tool Length Offset* and the *Cutter Radius Offset*, each with its own *Geometry* and *Wear* offsets. Because each offset has its own registry area, the *same* offset number may be used for the H and the D offsets:

(TOOL 04 ACTIVE)

```

...
G43 Z2.0 H04           Uses tool length offset 04   (H04)
...
G01 G41 X50.0 D04     Uses cutter radius offset 04   (D04)
...

```

In terms of convenience, the *Type C* is equally generous to the CNC programmer (same offsets numbers can be used in the program for H and D addresses), and the CNC operators (distinct control over geometry and wear for both length and radius offsets).

Tool Offset Variables - Fanuc 0 Controls

Since Fanuc 0 (FS-0) is the most modest control in terms of features, therefore not the most suitable control for serious and complex macro work, the details relating to system variables will be short. On the CNC machining centers (or mills), Fanuc 0 uses only two columns for the tool offset system variables - the *offset number* and the *variable number*.

Milling Control FS-0M

Typical number of available tool offsets is up to 200, and the input of tool offset related system variables reflect that.

Tool Offset Number	Variable Number
1	#2001
2	#2002
3	#2003
4	#2004
5	#2005
6	#2006
7	#2007
8	#2008
9	#2009
10	#2010
11	#2011
12	#2012
...	...
...	...
199	#2199
200	#2200

Only one column of system variables is available on Fanuc 0 control models.

Turning Control - FS-0T

Typical number of available tool offsets is 32, and the input of tool offset related system variables reflect that number.

Offset Registry	Tool Offset Number	Tool Wear Offset Value	Tool Geometry Offset Value
X-axis	1	#2001	#2701
	2	#2002	#2702
	3	#2003	#2703

	32	#2032	#2732
Z-axis	1	#2101	#2801
	2	#2102	#2802
	3	#2103	#2803

	32	#2132	#2832
Radius	1	#2201	#2901
	2	#2202	#2902
	3	#2203	#2903

	32	#2232	#2932
Tool Tip	1	#2301	#2301
	2	#2302	#2302
	3	#2303	#2303

	32	#2332	#2332

In the common *Type B* offset memory, four columns of system variables are required, 32 variables available for each column. Note that the *Geometry* and the *Wear* related variables are the same for the *Tool Tip* setting value, because they cannot be different for each mode. If a tool tip number 3, for example, is set to the *Geometry* offset, it will also appear as 3 in the *Wear* offset. Change of one will force the change of the other.

Tool Offset Variables - FS 10/11/15/16/18/21 for Milling

Fanuc controls 10/11/15/16/18 for milling share quite a few system variables related to tool offsets and are listed under the same heading. Regardless of this general statement, always double check for any latest changes in the Fanuc reference manual. Milling controls will be covered first.

The tool offset system variables are not only distinguished by the *Memory Type* (A, B, or C), but also by the number of offsets actually available to the particular control system. Depending on the options purchased, the distinction is between the number of offsets 200 or less, or over 200. The following tables list the appropriate system variables available.

Assignments for 200 Offsets or Less - Memory Type A

The following table lists the system variables for 200 or fewer offsets, in memory *Type A*:

Offset Number	Variable Number
1	#2001
2	#2002
3	#2003
4	#2004
5	#2005
6	#2006
7	#2007
8	#2008
9	#2009
10	#2010
11	#2011
12	#2012
...	...
...	...
...	...
198	#2198
199	#2199
200	#2200

Assignments for 200 Offsets or Less - Memory Type B

The following table lists the system variables for 200 or fewer offsets, in memory *Type B*:

Offset Number	Geometry Offset Variable Number	Wear Offset Variable Number
1	#2001	#2201
2	#2002	#2202
3	#2003	#2203
4	#2004	#2204
5	#2005	#2205
6	#2006	#2206
7	#2007	#2207
8	#2008	#2208
9	#2009	#2209
10	#2010	#2210
11	#2011	#2211
12	#2012	#2212
...
...
...
198	#2198	#2398
199	#2199	#2399
200	#2200	#2400

In the *Type B* tool offset memory, there still exists the sharing of the offsets between the tool length and the tool radius entries, but the separation of *Geometry* offset and the *Wear* offset may be very useful in many macro applications, mainly those that relate to the program controlled changes of *preset offset values*. In such situations, the preset value will be stored in the *Geometry* offset column (and will *not* change), whereby the adjustments to that value (the required changes) will be made into the *Wear* offset column. The preset offset values are typically measured off-machine, using a special presetting equipment, and are quite common in large volume manufacturing, agile manufacturing, and in machine shops that use a large number of similar CNC machine tools or special features.

Assignments for 200 Offsets or Less - Memory Type C

The following table lists the system variables for 200 or fewer offsets, in offset memory *Type C* - note the four columns of variables:

Offset Number	H-OFFSET		D-OFFSET	
	Geometry Offset Variable Number	Wear Offset Variable Number	Geometry Offset Variable Number	Wear Offset Variable Number
1	#2001	#2201	#2401	#2601
2	#2002	#2202	#2402	#2602
3	#2003	#2203	#2403	#2603
4	#2004	#2204	#2404	#2604
5	#2005	#2205	#2405	#2605
6	#2006	#2206	#2406	#2606
7	#2007	#2207	#2407	#2607
8	#2008	#2208	#2408	#2608
9	#2009	#2209	#2409	#2609
10	#2010	#2210	#2410	#2610
11	#2011	#2211	#2411	#2611
12	#2012	#2212	#2412	#2612
...
...
...
198	#2198	#2398	#2598	#2798
199	#2199	#2399	#2599	#2799
200	#2200	#2400	#2600	#2800

The benefits of the *Type C* tool offset memory are the same as those for the *Type B*, but the additional control of the tool length and the tool radius listed separately, brings a lot of convenience and flexibility to the CNC and macro programming.

As was the case for work offset related system variables, note the same logical identification and numbering of variables for tool length and radius.

Assignments for More than 200 Offsets - Memory Type A

The following table lists the system variables for controls that have more than 200 tool offsets available, in memory *Type A*:

Offset Number	Variable Number
1	#10001
2	#10002
3	#10003
4	#10004
5	#10005
6	#10006
7	#10007
8	#10008
9	#10009
10	#10010
11	#10011
12	#10012
...	...
...	...
...	...
997	#10997
998	#10998
999	#10999

The offsets listed in the above table are the same in principle as those used in applications for the *Assignments for 200 Offsets or Less - Memory Type A*, listed earlier in this chapter. The only difference is the much greater number of offsets is available to the programmer and operator. The large number of offsets usually appears on large CNC machines, and/or CNC machines that have an unusually large number of tools that can be stored and registered.

By the minimum number of 200, it is safe to assume that machines having over 200 tools would also use this group of tool offset system variables.

Assignments for More Than 200 Offsets - Memory Type B

The following table lists system variables for controls that have more than 200 tool offsets available, in memory *Type B*:

Offset Number	Geometry Offset Variable Number	Wear Offset Variable Number
1	#10001	#11001
2	#10002	#11002
3	#10003	#11003
4	#10004	#11004
5	#10005	#11005
6	#10006	#11006
7	#10007	#11007
8	#10008	#11008
9	#10009	#11009
10	#10010	#11010
11	#10011	#11011
12	#10012	#11012
...
...
...
997	#10997	#11997
998	#10998	#11998
999	#10999	#11999

The tool offsets listed in the table above are used for the same applications as those listed in *Assignments for 200 Offsets or Less - Memory Type B*, listed earlier in this chapter - only a greater number of offsets is available.

Assignments for More than 200 Offsets - Memory Type C

The following table lists system variables for controls that have more than 200 tool offsets available, in memory *Type C*:

Offset Number	H-OFFSET		D-OFFSET	
	Geometry Offset Variable Number	Wear Offset Variable Number	Geometry Offset Variable Number	Wear Offset Variable Number
1	#10001	#11001	#12001	#13001
2	#10002	#11002	#12002	#13002
3	#10003	#11003	#12003	#13003
4	#10004	#11004	#12004	#13004
5	#10005	#11005	#12005	#13005
6	#10006	#11006	#12006	#13006
7	#10007	#11007	#12007	#13007
8	#10008	#11008	#12008	#13008
9	#10009	#11009	#12009	#13009
10	#10010	#11010	#12010	#13010
11	#10011	#11011	#12011	#13011
12	#10012	#11012	#12012	#13012
...
...
...
997	#10997	#11997	#12997	#13997
998	#10998	#11998	#12998	#13998
999	#10999	#11999	#12999	#13999

The offsets are used the same as applications for *Assignments for 200 Offsets or Less - Memory Type C*, listed earlier in this chapter - only a greater number of offsets is available.

This and all previous tables for milling provide a good resource of system variables for several models of Fanuc controls for milling. The tables that follow will provide similar resources for turning controls.

Tool Offset Variables - FS 10/11/15/16/18/21 for Turning

Fanuc controls FS-10, FS-11, FS-15, FS-16, FS-18, and FS-21 for turning also share quite a few system variables related to tool offsets, and are listed under the same heading. Regardless of this, always double check for any latest changes in the Fanuc reference manual. The turning controls will be described next.

The tool offset system variables for lathe controls are not only distinguished by the offset *Memory Type* (A or B), but also by the number of offsets actually available to the particular control system. Depending on the options purchased, the distinction is between the total number of offsets 64 or less, or over 64. The tables in this section list the appropriate system variables available.

Tool Setting

This particular section does not really belong to the chapter dealing with tool offsets, but it is included here as a reminder of the tool offsets used in programming of a typical CNC lathe.

In a typical program for a CNC lathe, the cutting tool is called with the T-address, followed by a four digit number, for example T0101. There is no **M06** function for an automatic tool change on the lathe, the T-address does the tool change (tool indexing) as well as setting the offset values. The four digit number is, in fact, *two pairs of two digit* numbers. The first pair always calls the tool number station on the turret, the second pair is the offset number. So T0101 means tool 1, offset 1. With memory *Type A*, there is only a single set of variables, and they refer to the *Wear* offset only. There is no *Geometry* offset in *Type A*. In the program, it could be the **G50** command that sets the geometry offset (**G50** on lathes is equivalent to **G92** on mills). Since this method of setting is today considered obsolete, it is the *Type B* that is more common and popular, using the *Geometry and the Wear* offsets. In this case, the first pair of digits in T0101 means selection of the tool station number 1 on the turret (including the tool change) and the *Geometry* offset number, in this example, also 1. The second pair is strictly assigned to the *Wear* offset and does *not* have to be the same as the *Geometry* offset, although often is.

Although there are CNC lathe operators who totally ignore the *Geometry/Wear* offset distinction in the control and use only the *Geometry* offset, it is definitely *not* the recommended approach, definitely for macro programs. For example, if the measured *Geometry* offset entry for T0101 is Z-375.0 and the *Wear* offset for the same tool is Z1.5, it is exactly the same as having the *Geometry* offset set to Z-373.5, and the *Wear* offset set to Z0.0. Not the right approach, not the recommended approach, but it does work anyway.

The reason it is preferable to keep the *Geometry* offset unchanged and manipulate the *Wear* offset only, is because the *Geometry* offset represents a nominal, or original, given, or preset offset measurement. It may come from the operator's on-the-machine measurement, or from an external tool presetter. This approach also allows for an easier control in situations where more than one *Wear* offset is required for the same tool, which is a very powerful technique for maintaining tight tolerances. Managing both offset types properly will help in preparation of quality macro programs for CNC lathes.

Assignments for 64 Offsets or Less - Memory Type A

The offset memory *Type A* is not found very often in machine shops anymore. The following reference table lists system variables for 64 or fewer offsets, in memory *Type A*.

The listing is equivalent to the *Wear* offset listing only:

Offset Registry	Tool Offset Number	Tool Offset Value
X-axis	1	#2001
	2	#2002
	3	#2003

	64	#2064
Z-axis	1	#2101
	2	#2102
	3	#2103

	64	#2164
Radius	1	#2201
	2	#2202
	3	#2203

	64	#2264
Tool Tip	1	#2301
	2	#2302
	3	#2303

	64	#2364

Assignments for 64 Offsets or Less - Memory Type B

The offset memory *Type B* is quite common, and is found on many Fanuc lathe controls. It supports up to 64 tool offsets, more than enough for the majority of lathe applications.

The following reference table lists system variables for 64 or fewer offsets, in memory *Type B*:

Offset Registry	Tool Offset Number	Tool Wear Offset Value	Tool Geometry Offset Value
X-axis	1	#2001	#2701
	2	#2002	#2702
	3	#2003	#2703

	64	#2064	#2764
Z-axis	1	#2101	#2801
	2	#2102	#2802
	3	#2103	#2803

	64	#2164	#2864
Radius	1	#2201	#2901
	2	#2202	#2902
	3	#2203	#2903

	64	#2264	#2964
Tool Tip	1	#2301	#2301
	2	#2302	#2302
	3	#2303	#2303

	64	#2364	#2364

Assignments for More than 64 Offsets - Memory Type A

The offset memory *Type A* is not found very often in machine shops anymore, and with more than 64 offsets it is even more rare. The following reference table lists system variables for more than 64 offsets (160 listed - other number is also possible), in memory *Type A*.

The listing is equivalent to the *Wear* offset listing only:

Offset Registry	Tool Offset Number	Tool Offset Value
X-axis	1	#10001
	2	#10002
	3	#10003

	160	#10160
Z-axis	1	#11001
	2	#11002
	3	#11003

	160	#11160
Radius	1	#12001
	2	#12002
	3	#12003

	160	#12160
Tool Tip	1	#13001
	2	#13002
	3	#13003

	160	#13160

Assignments for More than 64 Offsets - Memory Type B

The offset memory *Type B* for more than 64 offsets represents quite a modern application, but it is not too common. It supports over 64 tool offsets (160 listed - other number is also possible), more than enough for some very complex lathe applications.

The following reference table lists system variables for more than 64 offsets in memory *Type B*:

Offset Registry	Tool Offset Number	Tool Wear Offset Value	Tool Geometry Offset Value
X-axis	1	#15001	#10001
	2	#15002	#10002
	3	#15003	#10003

	160	#15160	#10160
Z-axis	1	#16001	#11001
	2	#16002	#11002
	3	#16003	#11003

	160	#16160	#11160
Radius	1	#17001	#12001
	2	#17002	#12002
	3	#17003	#12003

	160	#17160	#12160
Tool Tip	1	#13001	#13001
	2	#13002	#13002
	3	#13003	#13003

	160	#13160	#13160

12

MODAL DATA

One of the most important practical examples of using system variables in a macro program deals with the subject of *modal data*. All basic CNC programming courses teach that the majority of data in a CNC program is modal. The word *modal* is a word based on the Latin word *modus*, which means *manner*. In English, we often use the words *mode*, *style*, *form*, *etc.*, to describe such a condition. When the same meaning is applied to the CNC modal word, for example a feedrate word F250.0, it means the specified feedrate has the same form, same style, same mode - it means that it does not change, or that it is *modal*, until another feedrate word replaces it. The same logic applies to many other CNC program statements (words), such as the spindle speed S, offsets H and D, and many others, including most of the G-codes and the M-codes. Of course, all axis data is modal as well (XYZ positions).

In this chapter, the emphasis will be placed on the importance of program modal values that exist *before* the custom macro is called, either from the main program, or from another subprogram. The emphasis will also be at how a macro can store the existing modal values, change them temporarily, and restore the original ones later, when required.

Working with modal commands in macros is not difficult, but care is needed to avoid problems.

System Variables for Modal Commands

The 4000 series of system variables (applicable to FS-0/10/11/15/16/18/21) covers the utilization of modal commands within a macro. In this 4000 series, there are two groups of system variables, based on the control model:

Fanuc 0/16/18/21 Modal Information

These control models use a set of two 4000 series variables:

#4001 to #4022	<i>Modal Information</i>	<i>(G-code groups)</i>
#4102 to #4130	<i>Modal Information</i>	<i>(B, D, F, H, M, N, O, S, and T codes)</i>

Fanuc 10/11/15 Modal Information

These control models also use a set of two 4000 series variables, but with a wider range:

#4001 to #4130	<i>Modal Information</i>	<i>(Preceding Block)</i>
#4201 to #4320	<i>Modal Information</i>	<i>(Executing Block)</i>

Preceding and Executing Blocks

The purpose of the 4000 series of system variables is to provide the CNC macro programmer with modal information that is *current at any given time*. There are two groups of information available, the *preceding block*, and the *executing block*.

◆ Preceding Block

In this group is the modal information that is already active
This block is also called the *pre-reading block*

◆ Executing Block

In this group is the modal information that will become active
when the *current* block is being executed

Note that the Executing Block is not available
 on Fanuc models FS-0, FS-16, FS-18 and FS-21

Modal G-codes

Apart from axis commands, of all the remaining modal commands, the G-codes are the most prominent and most commonly used in macros. For all Fanuc controls, the first system variable is **#4001**, where the last digit (1) means modal G-code *Group 01*, **#4002** refers to the G-code *Group 2*, and so on. *Group 00* is not supported, because the 4000 series of system variables serves the *modal* information only, and G-codes in the *Group 00* are *non-modal*. For the Fanuc 0/16/18/21 controls, the status of various *modal* G-codes is always stored in system variables within the **#4001–#4022** range, and the other codes within the range of **#4102–#4130**. All these variables are modal information of the *preceding block*. For the Fanuc 10/11/15 controls, the status of modal system variables is divided between the *preceding block* (system variables within the **#4001–#4130** range), and the *executing block* (system variables within the **#4201–#4330** range).

Within either range of variables, the current value of the G-codes of all modal groups can be saved into a local or a common macro variable, typically *before* the G-code is changed in the macro. The main purpose of the saving the current modal G-code(-s) is the safety being built into the macro program, but also the effort of maintaining professional programming environment. For example, if the work offset **G56** is used in the macro, and no action is taken, the **G56** will become the current coordinate system after the macro is completed, for any program that is loaded after. Such a situation may be very destructive, if - for instance - the flow of the main program execution depends on the **G54** work offset. A professional programmer always saves the current modal G-values *within* the macro, then changes the values that need to be used in the macro body. The new values can be used freely, as needed, as many times as needed, within the macro, while the macro is active. Finally, before the macro exits, the original values used in the main program or another macro are restored and applied for the subsequent program flow.

Fanuc 0/16/18/21

Typical listing of G-codes (preparatory commands) modal information for the *lower* level CNC controls (preceding block only - executing block is *not* available):

System Variable Number	G-code Group	G-code Commands
#4001	01	G00 G01 G02 G03 G33 Note: G31 belongs to Group 00
#4002	02	G17 G18 G19
#4003	03	G90 G91
#4004	04	G22 G23
#4005	05	G93 G94 G95
#4006	06	G20 G21
#4007	07	G40 G41 G42
#4008	08	G43 G44 G45
#4009	09	G73 G74 G76 G80 G81 G82 G83 G84 G85 G86 G87 G88 G89
#4010	10	G98 G99
#4011	11	G50 G51
#4012	12	G65 G66 G67
#4013	13	G96 G97
#4014	14	G54 G55 G56 G57 G58 G59
#4015	15	G61 G62 G63 G64
#4016	16	G68 G69
#4017	17	G15 G16
#4018	18	N/A
#4019	19	G40.1 G41.1 G42.1
#4020	20	N/A to FS-M and FS-T controls
#4021	21	N/A
#4022	22	G50.1 G51.1

For example, when the macro program contains expression **#1=#4001**, and the variable is processed, the returned value stored in **#1** may be 0, 1, 2, 3, or 33, depending on the active G-code in *Group 01*.

Fanuc 10/11/15

Typical listing of G-codes (preparatory commands) modal information for the *higher* level CNC control systems:

System Variable Number		G-code Group	G-code Commands
Preceding Block	Executing Block		
#4001	#4201	01	G00 G01 G02 G03 G33 Note: G31 belongs to <i>Group 00</i>
#4002	#4202	02	G17 G18 G19
#4003	#4203	03	G90 G91
#4004	#4204	04	G22 G23
#4005	#4205	05	G93 G94 G95
#4006	#4206	06	G20 G21
#4007	#4207	07	G40 G41 G42
#4008	#4208	08	G43 G44 G45
#4009	#4209	09	G73 G74 G76 G80 G81 G82 G83 G84 G85 G86 G87 G88 G89
#4010	#4210	10	G98 G99
#4011	#4211	11	G50 G51
#4012	#4212	12	G65 G66 G67
#4013	#4213	13	G96 G97
#4014	#4214	14	G54 G55 G56 G57 G58 G59
#4015	#4215	15	G61 G62 G63 G64
#4016	#4216	16	G68 G69
#4017	#4217	17	G15 G16
#4018	#4218	18	G50.1 G51.1
#4019	#4219	19	G40.1 G41.1 G42.1
#4020	#4220	20	N/A to FS-M and FS-T controls
#4021	#4221	21	N/A
#4022	#4222	22	N/A
...

Saving and Restoring Data

The two of the most elementary programming rules are *logical approach* and *programming neatness*. This handbook tries to follow these rule diligently, because they help in making a high quality macro program. The goal is a CNC program or a macro that is written in a logical manner, is well organized, follows operational steps in a methodical way, does not take anything for granted, and, yes - is neat and elegant. The result is a program that is much easier to interpret, document, and change, if necessary, even by a relative beginner in macro development. There are two methods applied in macro programming that belong to this category.

Saving Modal Data

To save (store) the current value of a G-code (or other modal codes), is to *preserve* them for later use - or rather re-use. The current value is stored into a variable and retrieved to restore the original setting. In a typical macro, there will be many G-codes used, most of them modal. This programming convenience also presents a potential problems. When a macro exits, the modal G-codes used by the macro will still be in effect. That creates a very disorganized way of macro program development, and can literally be a cause of many serious and hard to find problems. Although any modal G-code groups can be saved (and eventually restored), only two or three groups are commonly saved and replaced in most macros (add others, if needed):

- G-code Group 01** *Motion Commands*
 Rapid, Linear, Circular *G00, G01, G02, G03, G33*

- G-code Group 03** *Dimensioning Mode*
 Absolute or Incremental mode *G90 or G91*

- G-code Group 06** *Measuring Units*
 Metric or English *G21 or G20*

Typical method of saving the current G-code mode is to assign the selected system variable into a local variable. A common variable may also be used, in some very special applications. Here is an example that stores the current mode of *Group 01* (motion commands), and the current mode of dimensioning from *Group 03*:

- #31 = #4201** *Store the current motion command mode* *Group 01 (G00, G01, G02, G03 or G33)*

- #32 = #4203** *Store the current dimensioning mode* *Group 03 (G90 or G91)*

- #33 = #4206** *Store the current units mode* *Group 06 (G20 or G21)*

Note that the last two digits of the system variable match the modal G-code *Group* number. This is no coincidence. Such logical numbering offers an easy way of remembering. It can also be applied within a macro program in some ingenious way, to take advantage of it. When a macro is called, any current modal command should *always* be registered at the *beginning* of the macro, if that is not the case, then definitely before any changes are made within the macro.

Restoring Modal Data

Since the original G-code(-s) have been stored for the single reason of restoring them later, they have to be restored *before* the macro ends, typically at the end, just before the **M99** function. Using two system variables introduced in the previous example, here is a schematic layout of a macro program structure, showing both the storage and the restoration of two modal values:

```
O0018 (MACRO MODAL VALUES)
#31 = #4201           Store the current motion command
#32 = #4203           Store the current dimensioning mode
...
G90 G00 G54 X150.0 Y75.0
...
< ... macro processing ... >
...
G#31 G#32           Restore both previously saved modes
M99
%
```

In the example segment, the variables **#31** and **#32** store the current values of the motion and dimensioning modes, at the very beginning of the macro. The macro then proceeds with its own definitions, G-code changes, and so on, and before the macro end (before **M99**), the original values, *the previously stored values*, are retrieved to become modal after the macro exits. Since both of the previously stored values represent modal commands, the programming returns from the macro to the same environment that existed before the macro was called. Logical method of numbering system variables will be also applied to the other modal codes.

Other Modal Functions

In addition to the modal G-codes, there are additional eleven modal codes used in typical macro programming. Just like the G-codes, in a macro calculation (or a formula), these program codes cannot be programmed to the left side of the equal sign, which means they cannot be assigned value through the program. This is similar to the concept of 'read-only' and the 'read-and-write' type of variables in many general commercial programming languages, covered in the last chapter. The listing of the 'other' eleven modal addresses that can be used in macro programming, is presented here:

B D E F H M N O S T P

These modal codes are in *addition* to the modal G-codes. On the next two pages are the listings of the system variables relating to the 'other' modal addresses, for the two common types of Fanuc controls. Observe the method of *how* these variables are numbered, again, there is a logical method to it, and a little different one from the one used for modal G-codes. Also note the last two digits of each system variable number. They correspond to the *Local Variables Assignment List 1*. For example, the letter B is assigned to the local variable **#2**, hence the **#4102** system variable, the letter D is assigned to the local variable **#7**, hence the **#4107** system variable, and so on. *Very valuable observation that can come handy.*

Fanuc 0/16/18/21

As in the table for the modal G-codes listed earlier, the *lower* level controls use only system variables applying to the *preceding* block. The system variables related to the *executing* block are *not* available for this group of Fanuc controls (FS-0/16/18/21).

The following table shows the other modal information (eleven common addresses) frequently used in a macro program with their corresponding system variables.

System Variable Number	Program Address (Code Letter)
#4102	B-code - indexing axis position
#4107	D-code - cutter radius offset number
#4108	E-code - feedrate value (if available)
#4109	F-code - feedrate value
#4111	H-code - tool length offset number
#4113	M-code - miscellaneous function
#4114	N-code - sequence number
#4115	O-code - program number
#4119	S-code - spindle speed value
#4120	T-code - tool number
#4130	P-code - additional work offset number

As the table illustrates, the only exception in the table is the #4130 variable - it has no corresponding value in the *Assignment List 1* for local variables. It was added by Fanuc later, when the CNC technology advanced, to accommodate the extended work offset set, also known as the *additional work offsets* - G54.1 P1 to G54.1 P48. Variables that would 'fit naturally' in the table, but are missing, for example #4118, are quite legitimate to use, providing they are supported by the control system for the particular CNC machine tool (not exactly a likely scenario).

There are two system variable numbers that may seem out of order - they are #4114 (the letter N, the address for the sequence numbers), and #4115 (the letter O, the address for the program number). In the *Assignment List 1*, there are no #14 and #15 local variables listed. These assignments are physically *excluded* in the list, but they are *implied* by their 'non-presence'.

Fanuc 10/11/15

The listing of system variables for the *higher* group of Fanuc controls uses both sets of variables - for the preceding block, *and* for the executing block.

System Variable Number		Program Address (Code Letter)
Preceding Block	Executing Block	
#4102	#4302	B-code - indexing axis position
#4107	#4307	D-code - cutter radius offset number
#4108	#4308	E-code - feedrate value (if available)
#4109	#4309	F-code - feedrate value
#4111	#4311	H-code - tool length offset number
#4113	#4313	M-code - miscellaneous function
#4114	#4314	N-code - sequence number
#4115	#4315	O-code - program number
#4119	#4319	S-code - spindle speed value
#4120	#4320	T-code - tool number
#4130	#4330	P-code - Additional work offset number

One observation that may be valuable from this chapter relates to the list. It is not what *is* included, but what is *missing*. Modal commands used in the macro programming have been discussed extensively, along with the related system variables. Another subject discussed was the major group of the modal G-code addresses, and eleven other modal addresses. What is missing?

What is missing are all system variables that have something to do with a *tool position*, such as endpoint coordinates in a block, work coordinates, machine coordinates, various tool offset positions, coordinates related to the skip function, and so on - even deviation amount of the servo position, etc.

In the chapters that follow this one, more system variables will be covered, most listed earlier in a descriptive form. They include those related to alarms, timers, and various axis position information. The next chapter covers another part of the most important macro programming tools - *conditional testing, branching and looping*.

13

BRANCHES AND LOOPS

About half way through the handbook, now comes a real powerhouse of macros. At this level of macro development, strong basic knowledge of macro command structure, offsets, memories, as well as understanding variables, etc., is essential. A variable is assigned a value, macro is processed with that value in effect, macro exits. This straightforward approach is convenient, and very common, but it cannot and does not stand alone. It needs some additional forms of data manipulation, forms based on some kind of *decision making process*.

Decision Making in Macros

The structure of a typical Fanuc macro program is based on the oldest and simplest of all computer languages - Basic™. The *Basic* language proved to be simple, yet powerful, language for its time. Although the *Basic* language in its original form is now a history, many of its rules and structural forms still do exist. *Basic* language has developed into the current *Visual Basic*, very modern, and structured, high level language. One of the remnants of the old *Basic* is the function **GOTO**n, and **GOSUB**, both considered today a very poor way of language based program structure. However, the other branching functions (**IF**, **IF-THEN**, and **WHILE**) are available to control the flow of the macro program.

In either form, the decision is always based on the *result* of a given condition, or given situation. Depending on this result, at least two other options must be available for further considerations. As an example, the everyday English equivalent of - “*If I have money, I will buy a car.*” has two parts. The condition here is '*if I have money*' - '*to have money*'. There are two logical outcomes - they are '*I will buy a car*' and '*I will not buy a car*', one or the other, but never both. “*If I have money, I will buy a car, but I do not have money, so I will not buy a car.*” Simple logic, and very powerful when applied to macros.

The simplicity of the above example is very strong when transferred into the realm of a macro programming. Of course, the macro conditions and macro options will always be different, but the logic, the evaluation, the thinking process, and the decision will not.

For example, one objective of a macro could be to check if the cutting tool travels within the limits of the machine. For each axis to be tested, a very specific condition will be created - “*If the length of travel is greater than the given distance, then ...*”, and the macro will have to contain the decision in its own format. If the condition specified is *true*, the CNC operator may be notified with an alarm message or at least a program comment. If the condition is *false* (not true), the macro will proceed with the program flow without interruption. The operator may not even be aware of the evaluation and decision taking place. This chapter covers various aspects of conditional testing, branching, and looping, depending on the complexity of the given condition and the purpose of such evaluation.

The first and the single major function in all the above examples is the **IF** function.

IF Function

IF

The **IF** function has several names - it is called the *decision* function, the *divergence* function, or most commonly, the *conditional* function. The format of the **IF** function is:

```
IF [ CONDITION IS TRUE ] GOTOn
```

where *n* is the block number to branch to, but *only if* the evaluated condition (the returned value) is *TRUE*. If the condition is true, all statements between the *IF*-block and *GOTO**n*-block will be bypassed. If the evaluated condition is not *TRUE*, it is *FALSE*, and the program will continue processing the next block following the block containing the **IF** function. We can schematically represent the last example in a simple flow chart in *Figure 19*.

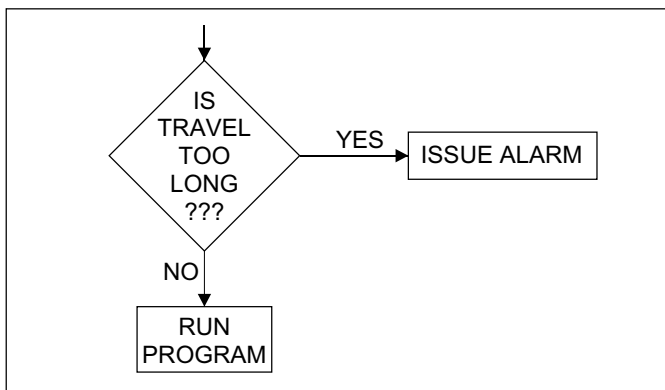


Figure19

Schematic flowchart representation of the **IF** conditional branching

The flowchart only shows the decision making and the results, *not* any complete program. The diagonal box identifies the condition to be evaluated (for example, machine travel limits), and the two rectangular boxes identify two - *and only two* - possible outcomes (YES or NO). Each outcome results in an action to be taken:

- If the travel *IS* too long, generate alarm condition and stop processing
- If the travel *IS NOT* too long, run the rest of the program normally

The **IF** function is one of the macro statements that control the order of program processing.

Conditional Branching

Branching from one block of the program to another block of the same program is unique to macros - it always means bypassing one or more program blocks. The bypass has to be done in a selective and controlled way, otherwise all kinds of problems will take over. The conditional function **IF** serves as a decision maker between at two options. The main statement in a macro is:

IF [condition is true] GOTOn

for example,

```

IF [#7 LT 0] GOTO65           If the value of variable #7 is less than 0, branch to block N65
...
...                               If the above condition is true, bypass this section up to N65
...
N65 ...                          Target block of the IF conditional statement

```

The branching will *only* occur if the specified condition is *true* (that means the condition is satisfied). If not, the block immediately *following* the **IF** statement will be executed and no branching will take place.

It is very important that the target block of the branching, the one called in the **GOTOn** statement does exist in the same program, and is not duplicated. If necessary, it is possible to replace the *n* in the **GOTOn** with a variable number or the result of a previous calculation. For example, the following example is perfectly legitimate:

```

#33 = 65
...
IF [#7 LT 0] GOTO#33          If the value of variable #7 is less than 0, branch to block N65
...
...                               If the above condition is true, bypass this section up to N65
...
N65 ...                          Target block of the IF function conditional statement

```

The N-address block cannot be used as a variable statement, for example **N#31** is illegal.

Unconditional Branching

GOTOn

The **GOTOn** statement can be programmed on its own, without using the **IF** function. In this case, the macro program will branch *unconditionally*, to the *n* block number specified at the **GOTOn**. Without the **IF** function, the **GOTOn** statement has no condition (in this case, it is called an *unconditional statement* or *unconditional branch* or *unconditional jump*). The range of the **GOTOn** statement (*conditional* or *unconditional*) is only limited by the maximum of sequence numbers available for the given control model:

- ◆ **Four-digit sequence number** **Range of *n* is 1-9999**
- ◆ **Five-digit sequence number** **Range of *n* is 1-99999**

Do NOT change the block numbers if the macro contains GOTOn branch !

Fanuc system will generate an alarm, if the sequence number range is exceeded (*Sequence No. out of range* error). Block designation N0 is not allowed and will also result in an alarm condition. Duplicate N-addresses in the same program are possible, but they are not only strongly discouraged, they may as well be outright forbidden in macros especially. Definitely *do not* duplicate the sequence numbers (N-numbers) in a macro program.

A macro expression (a variable number) may also be used for the block number specified by the unconditional **GOTO**n statement, similar to the conditional statement, for example:

```
#33 = 65           Variable #33 stores the target block number
...
GOTO#33           Unconditional branching to the block N65
...
...              Arbitrarily bypass this section up to N65
...
N65 ...           Target block of the GOTO statement
```

Whether used conditionally or unconditionally, the **GOTO**n function should only be used when necessary. There are perfectly legitimate reasons why to use this function in macros for branching, but try to limit its usage for branching and looping. There are better functions to establish this goal, namely the **WHILE** function.

IF-THEN Option

IF [condition is true] THEN [argument]

Only the Fanuc control models 10/11/15 support the **IF-THEN** structure of conditional testing, 0/16/18 do not (Fanuc model 21 does). The main concept of the conditional **IF-THEN** structure is *simplicity*. **IF-THEN** is a shortcut when only two choices are available. Rather than invoking a formal structure of the **IF** statement alone, combined with the **GOTO**n statement, the **IF-THEN** option offers an immediate and short solution. Compare the two following examples - both statements will yield *identical* results. The two examples define the Z-axis clearance in the current system of units (either *English* or *Metric*):

➤ Example 1 - Macro control without the **IF-THEN** structure

```
#100 = #4006      Check the current units (English G20 or Metric G21)
IF[#100 EQ 20.0] GOTO20  If the units are English, branch to block number N20
IF[#100 EQ 21.0] GOTO21  If the units are Metric, branch to block number N21
N20 #100 = 0.1       Set 0.1 of an inch as the current clearance (English)
GOTO999            Bypass Metric setting
N21 #100 = 2.0       Set 2.0 mm as the current clearance (Metric)
N999              Start section common to both English and Metric
...
<... Macro continues normally ... >
```

➤ Example 2 - Macro control *with* the **IF-THEN** structure

```
#100 = #4006           Check current units (English G20 or Metric G21)
IF[#100 EQ 20.0] THEN #100 = 0.1  Clearance above work is 0.1 inch for G20
IF[#100 EQ 21.0] THEN #100 = 2.0  Clearance above work is 2 mm for G21
...
< ... Macro program continues normally ... >
```

Using the **IF-THEN** method makes the program shorter by one half and is easier to interpret.

Single Conditional Expressions

Macros support all six available conditional expressions, also known as the *Boolean Operators*. They compare two sides of an expression:

Math Symbol	Expression	Macro Function	Format
	Equal To	EQ	#1 EQ #j
	Not Equal To	NE	#1 NE #j
	Less Than	LT	#1 LT #j
	Less Than Or Equal To	LE	#1 LE #j
	Greater Than	GT	#1 GT #j
	Greater Than Or Equal To	GE	#1 GE #j

For example, the macro expression

```
IF [#1 EQ #2] GOTO99
```

will be true, only if the current value of variable **#1** is the same as the current value of variable **#2**. In such a case, the branching will take place. If the current values of the two variables are different, the condition is *false*, and the macro continues with processing of the next block, and no branching takes place.

Calculation formulas can be nested, providing the square brackets are used correctly:

```
IF [#1 EQ [#2+#3]] GOTO99
```

In this case, the current values of variables **#2** and **#3** are *summed-up* first, and the result of the sum is compared with the current value of variable **#1**. If they match, the specified condition is true, and branching *will* take place, otherwise the macro continues in the next block.

Combined Conditional Expressions

In the more complex calculations, two or more conditions have to be evaluated (compared). The result is often dependent on the returned value of *several* combined conditions. For example, in English, you may say “*If I have money **and** time, I will take a vacation.*”. In this statement, a single *true* condition is not enough - *both* conditions must be true for the whole expression to be true. No exceptions - even if I have the money, but I have no time, I cannot take a vacation. On the other hand, you may say “*If I save enough money **or** win a lottery, I will take a vacation.*”. This is a different statement. In this case, only *one* expression has to be true, for me to take a vacation. If I save enough money, I don't need to win a lottery, and I still can take a vacation. If I do win a lottery, I don't need to save money at all, and I can also take a vacation. These statements and expressions are found in everyday language.

In macro programming, there two *other* functions available (actually three) that can be combined to evaluate a given condition on binary numbers, bit by bit. They are:

AND OR

For any given condition, these macro functions can be used:

- ◆ **AND** **All given conditions must be true,
for the whole condition to be true**
- ◆ **OR** **Only one given condition must be true,
for the whole condition to be true**

XOR

Also, there is the third function - the **XOR** (*exclusive OR*) - function, but that is quite difficult to understand at the moment, and is very seldom used in a normal macro work.

When you evaluate a combined conditional statement, always ask the question ‘*Do the conditions have to be true ALL at the same time?*’. If the answer is *Yes*, use the **AND** function, otherwise use the **OR** function. The format of input of these functions into a macro has already been described earlier. The **AND** and the **OR** functions are typical *Binary Functions*, because their returned value can only have two states - either *TRUE* or *FALSE*.

Although not always necessary, understanding the *Binary Number System* may be helpful in certain situations. A brief overview of binary numbers is included in this handbook as well - see *Chapter 4 - System Parameters*.

Concept of Loops

Looping is another method of making a decision in the macro program flow, also based upon a specified condition. Just like the **IF** function requires a true/false evaluation, looping condition can also return only *True* or *False* state.

The greatest difference between the single condition testing (**IF**) and looping (**WHILE**) is how many processes are involved - *one* or *many*.

Single Process

Single process has already been covered - the **IF** function represents a single process. In order to understand the concept of looping, it is critical to understand the simple top-down general program process, without a loop, for any operation. This is the typical process used in standard programs, with no conditions and no decisions. It can be represented as a generic step-by-step orderly procedure for any operation or activity (only a very generic model is presented here):

- | | |
|-------------------------|---|
| 1. Start program | <i>Initialization - defaults - cancellations - units, ...</i> |
| 2. Input data | <i>Tool - spindle - location - clearances, ...</i> |
| 3. Process data | <i>Cycles - macros - feedrates - offsets, ...</i> |
| 4. Output result | <i>Actual machining - this is the main program objective</i> |
| 5. Stop program | <i>Clear conditions, ...</i> |

This method will work well on a single data input (single process), every time. For example, the preceding five generic steps can represent drilling of a single hole, at any desired location.

Now replace the generic terms with specific drilling operations (all five step numbers match):

- | | | |
|------------------------------------|----------------|--------------------------|
| 1. Start program | <i>same as</i> | 1. Start program |
| 2. Input XYZ hole location | <i>same as</i> | 2. Input data |
| 3. Move to the new location | <i>same as</i> | 3. Process data |
| 4. Drill the hole | <i>same as</i> | 4. Output results |
| 5. Stop program | <i>same as</i> | 5. Stop program |

Although only the basic drilling process illustrated, the orderly flow shows a clear concept of what is happening. What is the actual result of this process? Using the hole drilling as an example, one hole - *and only one hole* - has been drilled. The single process does not need the **IF** function, most standard programs use it daily, however in macros, the **IF** function represents the single process, such as a single hole. If more than one hole is needed, the single process is not sufficient.

Multiple Process

In the next stage, the same example will be explored, this time to drill more than a single hole. The above *top-down process* is not quite sufficient and a different technique has to be used. Before thinking of a technique, think of a process. *What has to change?* We do *not* want to always start and stop after drilling the first hole, do we? We want to input the next hole location (item 2), move to that location (item 3), drill the hole at that location (item 4), and - we want to repeat these three steps until *all* holes are done. Hopefully, the description was clear enough, but a graphical representation - using a simple flowchart - shows the comparison of both methods (*Figure 20*).

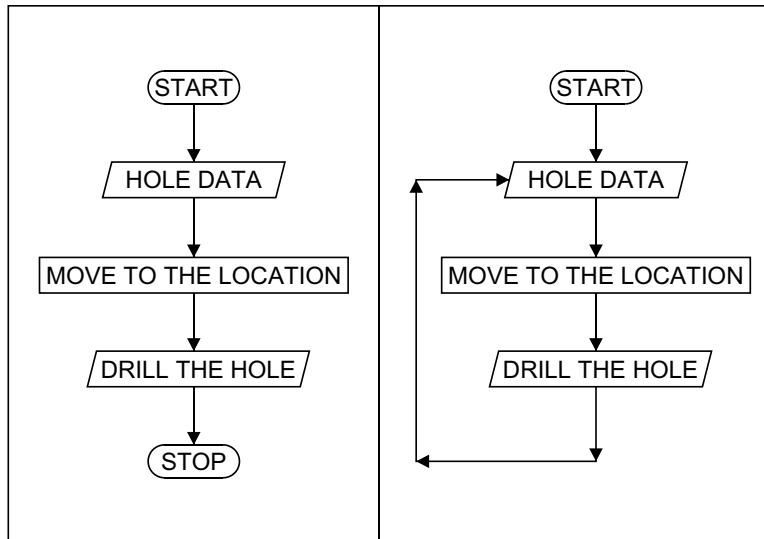


Figure 20

Flowchart comparison of a single structure process, shown at left, and a repeated structure process at right

Note that the example as shown at right represents an endless or infinite loop - no STOP

The two comparable flowcharts indicate the drilling process in principle. However, there is a major and a very serious problem with the repeated structure. *There is no way out!* The repetition shown at the right side of *Figure 20* has no provision to stop the processing - *the loop is not stopped - it is not controlled* - logically, it runs forever!

It is extremely important to provide an *exit* from a loop, when certain conditions are satisfied, for example, when the last hole is drilled. Failure to provide an exist from a loop will cause an *endless or infinite loop*. Infinite loops are the most common causes of problems in macro loops. Terminating a loop is always determined by a specific condition. This condition has to be part of the loop, based on the job requirements, with the provision to branch out of the loop, when the condition becomes *false*. The repetitive flowchart shown in *Figure 20* has to be modified.

Figure 21 shows the final flowchart for the drilling process, including the conditional statement and only two possible outcomes - *drill more holes or stop the program processing*.

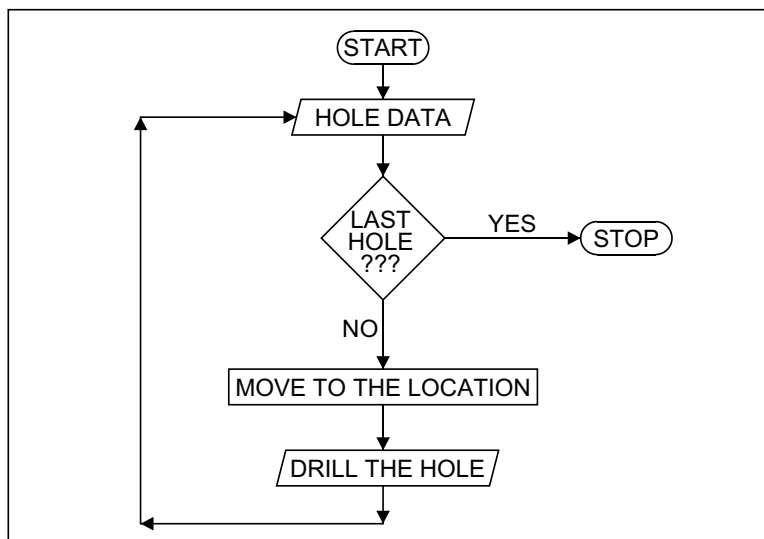


Figure 21

Flowchart showing a logical flow of the loop, based on the result of a conditional decision

The STOP branch provides exit from the loop

YES and NO represent TRUE and FALSE conditions respectively

WHILE Loop Structure

In Fanuc type macro programming, the **WHILE** function is used to program loops. The format of the looping function **WHILE** consists of the function, condition and action:

WHILE [condition] DOn

In a plain language terms, think of the **WHILE** function as the '*as-long-as*' function. The looping function **WHILE [condition] DOn** in a Fanuc macro language means '*process the body of the loop as long as the specified condition is true*'. The **DOn** action establishes the connection with the end of the loop, where the **n** is replaced with a number of the matching **ENDn** statement. The loop is programmed with the **ENDn** function that corresponds to the **DOn** call, for example, **DO1** with **END1**, **DO2** with **END2**, and **DO3** with **END3**. Only *three* loop depths can be programmed.

The three allowed loop depths - often known in programming as the *levels of nesting* - have three similar forms:

- Single level nesting**
- Double level nesting**
- Triple level nesting**

As the number of nesting levels increases, so does the programming complexity. The majority of loops for most macro applications are single level, double levels are not too unusual either. Triple level has a lot of power, but it does need a suitable application to employ it.

Single Level Nesting Loop

Programming only a single **WHILE** loop function between the **WHILE-DOn** loop call and its matching **ENDn**, defines the single level loop. This is the simplest and most commonly used looping function used in macro programs. The single level loop processes and controls one event at the time - *Figure 22*:

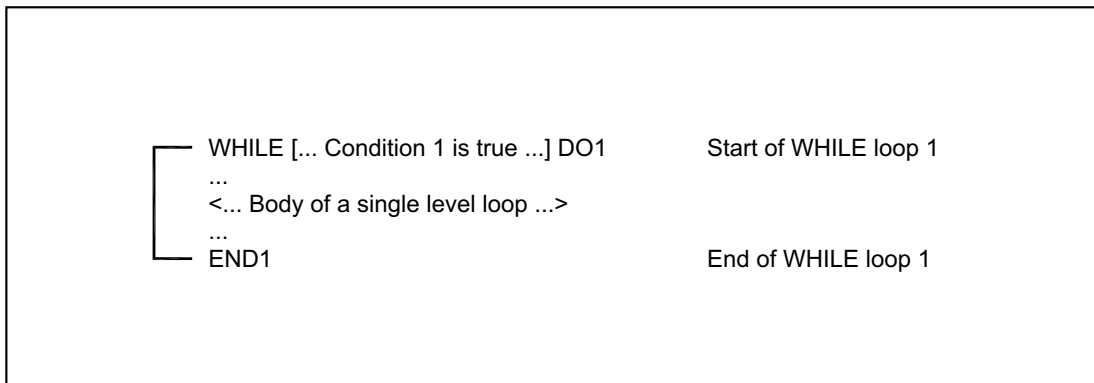


Figure 22

Single level of macro looping - controls one event at the time

Double Level Loop

Programming two levels of **WHILE** between the **WHILE-DO**n loop and the **END**n defines the double level loop - indents show the programming structure - *Figure 23*:

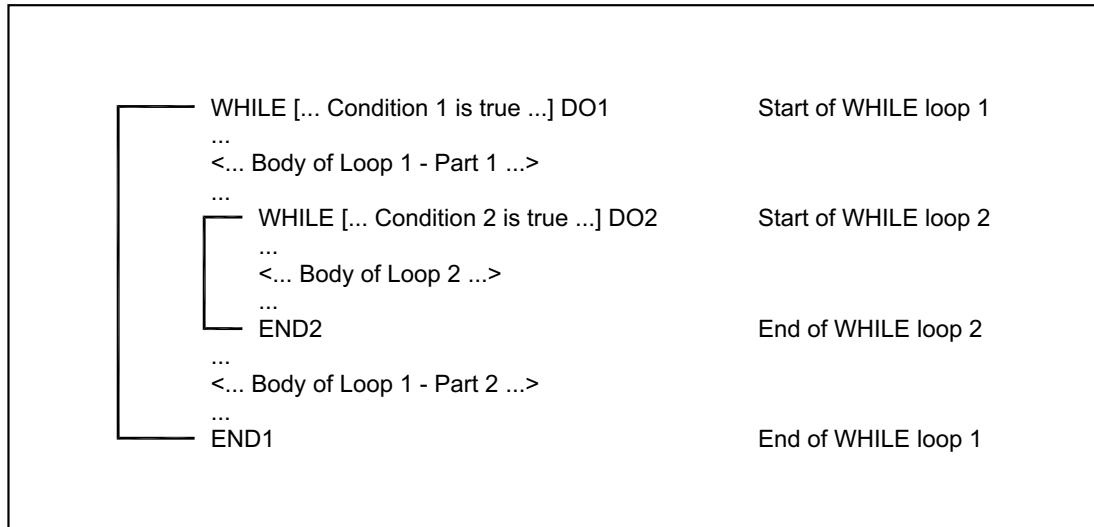


Figure 23

Double level of macro looping - controls two events at the time

The double level loop is also quite common, because it adds more decision making power to the macro. If properly structured, it should not present any difficulties at all. Remember that two events are controlled simultaneously in a double level loop.

A *single level of looping* should be easy to understand. Continuing with the example of a drilled hole, the single level macro is suitable to be used when the hole is to be drilled at different - *but equally spaced* - locations. A bolt circle (explained in *Chapter 20*) is an excellent example.

Understanding the *double level of looping* is a bit more difficult. The double level of looping is defined as controlling two events at the same time. For example, each hole of an equally spaced pattern has two internal grooves that share the same XY location. Hole location macro would be the first level, and machining the two grooves would be the second level.

Triple Level Loop

Programming all three levels of **WHILE** within the **WHILE-DO**n and the **END**n defines the triple level loop. The triple level loop is much less common than the other two, but it does bring even more decision making power to the macros. It is very important that a proper structure is used. With more levels, the possibility of a structural or logical error increases. Also keep in mind that the control software is intentionally designed to cover more than necessary in everyday work.

This last type of a loop structure controls three events simultaneously - again, indents show the programming structure - *Figure 24*:

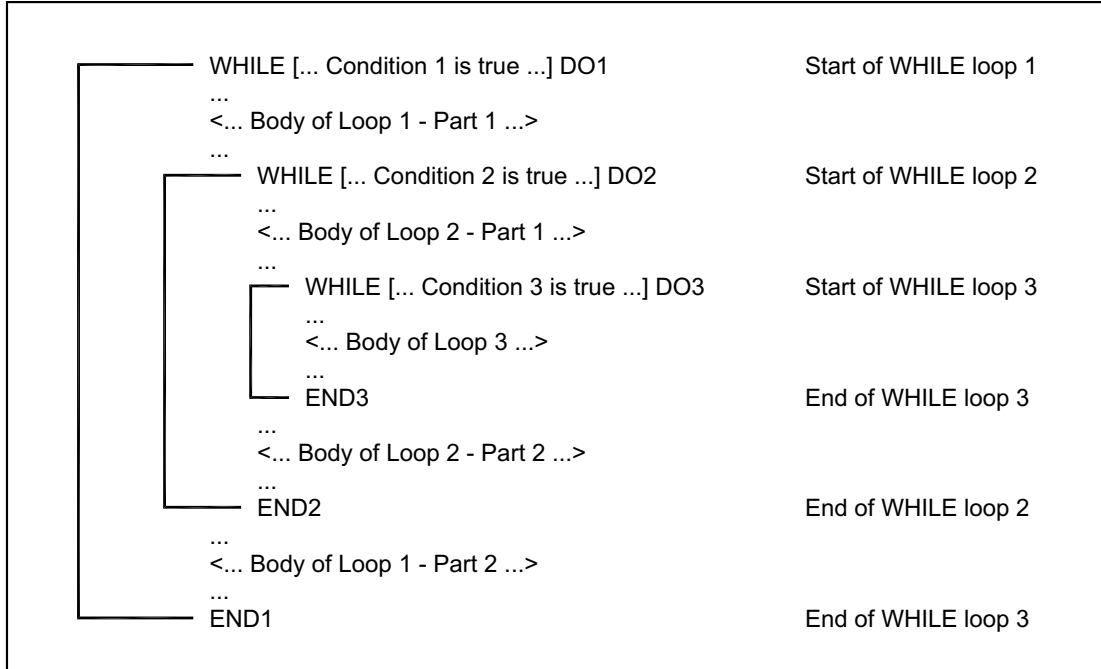


Figure 24

Triple level of macro looping - controls three events at the time

General Considerations

From the several examples of the **WHILE** function structure, its proper usage should become much clearer. When the *<condition>* in the **WHILE** statement is satisfied (that means it is *true*), the blocks between the **DO***n* and the corresponding **END***n* are executed repeatedly, in the order in which they are programmed. Each new pass through the loop *always* evaluates the given condition again and again. When the condition fails, that means it is *not true* anymore - it is *false*, then the macro flow of the loop is transferred to the block immediately following the **END***n* statement. In some rare cases, the **DO***n* and **END***n* can also be used *without* the **WHILE** statement, but this is definitely *not* a recommended practice or professional approach to programming.

Restrictions of the WHILE Loop

From the previous examples of a macro loop structure, a very definitive pattern emerges in the structural nesting of the **WHILE** function. The **DO***n* and the **END***n* must always be programmed in pairs, working from the innermost loop outwards. Depending on the nesting level (1, 2, or 3), the correct macro program must follow the pattern order. The order for each nesting level is shown here in a simplified form (*S**n* is the start level number, *E**n* is the corresponding end level number):

S1 .. E1	Single level
S1 .. S2 .. E2 .. E1	Double level
S1 .. S2 .. S3 .. E3 .. E2 .. E1	Triple level

Each listed digit indicates the current nesting level, and the order of digits represents the order of program flow from one nesting level to another and then back. This is the nesting structure as it should be. Unfortunately, mistakes do happen, and in macro looping, the most common mistake is *crossing* the levels when more than one level is programmed.

Crossing the **WHILE** loop between nesting levels is *not allowed*, for example, the following **WHILE** structure (and structures that are similar) is *wrong* - Figure 25:

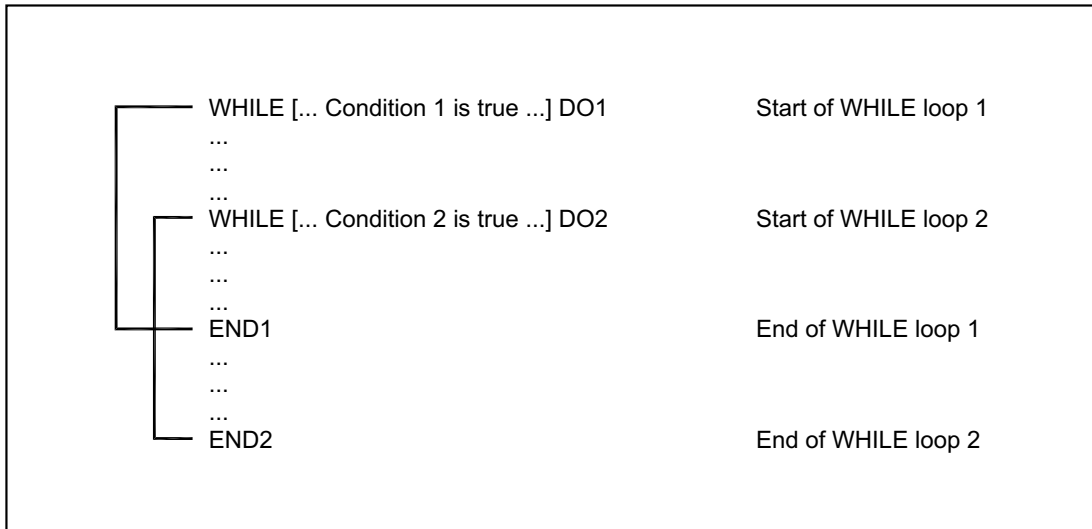


Figure 25

Common macro looping error - major structure problem (compare with previous formats)

Errors of looping, in any number of nested levels, may not always be the easiest ones to find, particularly in long or complex macros. That is the main reason why always maintaining order and consistency in macro program development is so important. For some macro programmers, a flowchart is a mandatory tool, for others, they can develop a macro very well without a flowchart. However, a well designed flowchart is important for the beginner, as well as for the seasoned programmer - it helps to design the macro logic during the development stages and retrace the macro flow at a later date.

Conditional Expressions and Null Variables

Earlier in this handbook, the important subject of return values of various expressions and calculations was introduced - one of the elements was a *null variable* (an *empty* variable or a *vacant* variable). In this section, the same null variable and its relationship to conditional expressions will be evaluated. Make sure to understand this subject well, it can help in many troubleshooting situations. Conditional expressions used with the **IF** and **WHILE** functions (explained earlier) always compare two values, using comparison operators such as **EQ**, **NE**, **GT**, **LT**, **GE**, and **LE**. If a null variable is compared with another value, the return value may be either *TRUE* or *FALSE*, depending on the exact situation. Comparing to a zero value is also shown, for more in-depth reference.

A null variable is different than a variable with a value of zero! As both the **IF** and the **WHILE** functions share the same logic, only the **IF** function is shown in the following examples, because the **WHILE** function uses the same format:

➤ Comparing a null variable to a null variable:

```
#1 = #0           #1 is defined as null (that means #1 is vacant)
IF[#1 EQ #0]     Returns TRUE
IF[#1 NE #0]     Returns FALSE
IF[#1 GT #0]     Returns FALSE
IF[#1 GE #0]     Returns TRUE
IF[#1 LT #0]     Returns FALSE
IF[#1 LE #0]     Returns TRUE
```

➤ Comparing a zero to a null variable:

```
#1 = 0           #1 is defined as zero (that means #1 is equal to 0)
IF[#1 EQ #0]     Returns FALSE
IF[#1 NE #0]     Returns TRUE
IF[#1 GT #0]     Returns FALSE
IF[#1 GE #0]     Returns TRUE
IF[#1 LT #0]     Returns FALSE
IF[#1 LE #0]     Returns TRUE
```

➤ Comparing a null variable to a zero:

```
#1 = #0           #1 is defined as null (that means #1 is vacant)
IF[#1 EQ 0]      Returns FALSE
IF[#1 NE 0]      Returns TRUE
IF[#1 GT 0]      Returns FALSE
IF[#1 GE 0]      Returns TRUE
IF[#1 LT 0]      Returns FALSE
IF[#1 LE 0]      Returns TRUE
```

➤ Comparing a zero to a zero:

```
#1 = 0           #1 is defined as zero (that means #1 is equal to 0)
IF[#1 EQ 0]      Returns TRUE
IF[#1 NE 0]      Returns FALSE
IF[#1 GT 0]      Returns FALSE
IF[#1 GE 0]      Returns TRUE
IF[#1 LT 0]      Returns FALSE
IF[#1 LE 0]      Returns TRUE
```

Formula Based Macro - Sine Curve

When it comes to machining unique contours based on specific machining definitions (typically mathematical formulas), most CNC controls do not offer direct support. Developing a contour cutting program for a parabola, hyperbola, ellipse, sine curve, cycloid, and many other curves, may be not possible in standard CNC programs, but presents no problem in macros. This section illustrates the development of a sine curve as the actual cutting toolpath macro example. Since the control system does not directly support sine curve interpolation (or parabolic or hyperbolic interpolation, etc.), the toolpath will be simulated by many small linear motions, in **G01** mode.

Sine curve is one of several mathematical curves that may come handy in certain applications. Since it is based on a formula, it becomes a very fitting subject for macro development, the main reason for using this example. It is a simple formula that will be adapted to generate cutting tool motion. The *Figure 26* illustrates a typical sine curve along with the related terminology. The sine curve is, in effect, a flat representation of a full circle, from 0 to 360°. The distance between the start and end angles is called *Period*. The height of the curve is called *Amplitude* and is always the same above and below the *X*-axis:

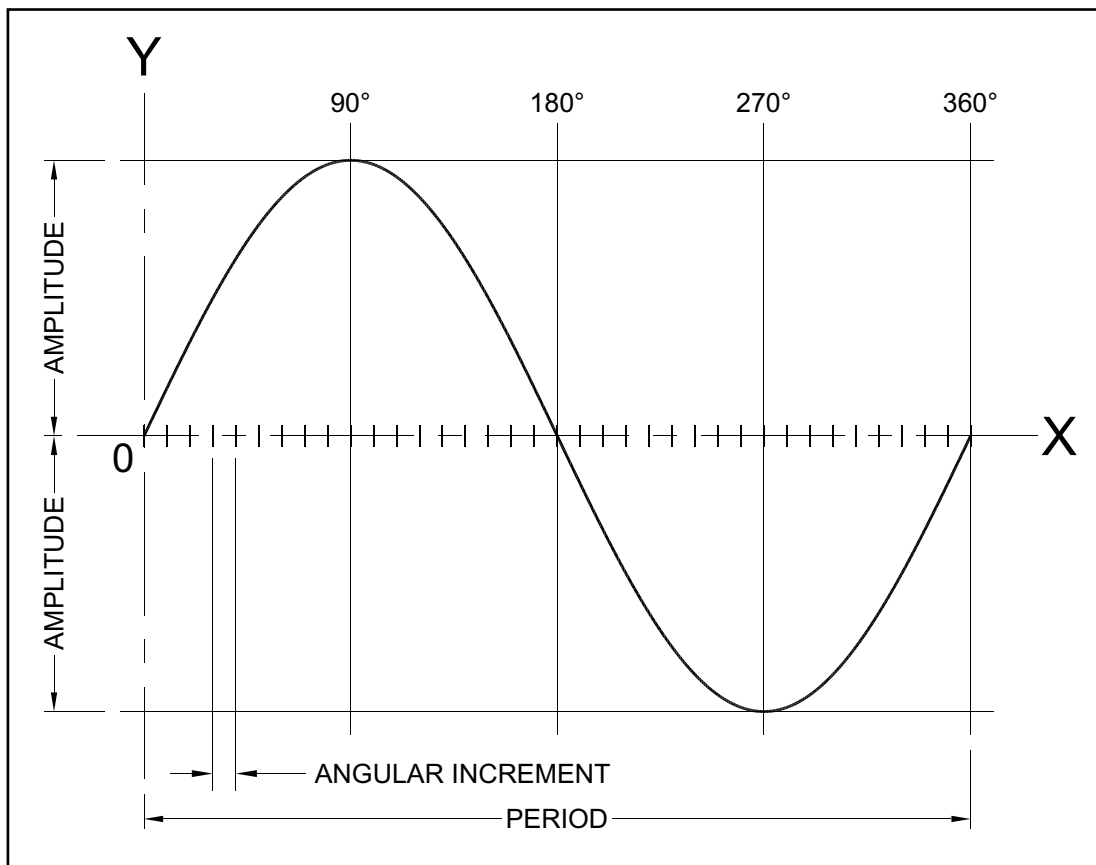


Figure 26


Sine curve - graphical layout

In terms of machining programs, only lines or arcs are allowed, not special curves. Any need to machine a special curve - such as the sine curve shown in the illustration - technique called *approximation* has to be used to *simulate* the curve by a series of very short lines. The shorter each line is, the more accurate the simulation, but at the price of a longer standard program. Length of the program is not a concern in macro, since the looping function will always have the same size.

The first step to take is to define the formula mathematically. Since it is a trigonometric sine curve, the **SIN** function will be used. Mathematically, the formula to calculate the *Y* is:

Y	Amplitude	sinX
----------	------------------	-------------

In many math books, the sine curve formula is listed as $Y = \sin X$. This designation is the same as $Y = 1 \sin X$, considering the unspecified amplitude as having the value of 1. Otherwise, the amplitude must always be specified. The defined period of the curve has to be segmented into small angle increments for best fit. The first angle increment will be used to create the first linear motion, the second increment will be used to create the second linear motion, and so on, until all 360° have been calculated in equal increments. The machined result is the required sine curve.

 Assignment of variables in the sine curve macro call O8009 is short and simple:

Amplitude	... assigned letter <i>A</i> (variable #1)
Angular increment	... assigned letter <i>I</i> (variable #4)
Cutting feedrate	... assigned letter <i>F</i> (variable #9)

The macro call will contain only three variables:

```
G65 P8009 A120.0 I5.0 F250.0
```

Note that the current units must be used (metric shown) and the angular increment must conform to the minimum programmable input of 0.001° . In the example, the sine curve will be machined as a linear motion - a series of straight lines in increments of 5° . *Decrease* the increment for *more* accuracy, *increase* the increment for *less* accuracy. Z-axis motions must be applied in the main program. The macro will be a single level loop, using a counter of the current incremental degrees compared with the final angle, such as the 360° used in the example:

```
O8009 (SINE CURVE MACRO)
#25 = 0                               Set initial counter for degrees increment
WHILE [#25 LE 360.0]                 Loop for each linear segment until 360 degrees are machined
#26 = #1 * SIN[#25]                  Calculate current Y-location
G90 G01 X#25 Y#26 F#9                Make a linear motion to the calculated XY location
#25 = #25+#4                          Increase the counter by specified increment
END1                                  End of loop
M99                                   End of macro
%
```

If required, the start and final angles can be also input as variables, if only a portion of the sine curve is needed. The sine curve macro can be very easily changed into a cosine curve macro, by shifting the sine curve 90° to the left, or moving the Y-axis 90° to the right.

Clearing Common Variables

One very useful and practical example of a simple **WHILE** loop is the development of a macro that can be stored in the control memory permanently, to be used by any program or from MDI. This macro clears the 500+ series of common variables, which can only be cleared by using macro functions. The CNC operator may clear all the variables at the control, in MDI mode, one by one. Much better solution is to have a program handy that covers all variables in the range, and sets them to the null (**#0**) state individually, by a macro loop:

```
O8010 (CLEAR 500+ VARIABLES - INDIVIDUALLY ONE BY ONE)
#500 = #0           Common variable #500 cleared (set to null)
#501 = #0           Common variable #501 cleared (set to null)
#502 = #0           Common variable #502 cleared (set to null)
#503 = #0           Common variable #503 cleared (set to null)
. . .
#999 = #0           Common variable #999 cleared (set to null)
```

Such a program can be quite long and will take unnecessary memory space. Using a loop, the program will be shortened significantly and be more professional as well:

```
O8011 (CLEAR 500+ VARIABLES - BY A MACRO LOOP)
#33 = 500           Initialize counter to the first variable (no # symbol !!!)
WHILE[#33 LE 999] DO1 Loop through variables - shown range is #500-#999
#[#33] = #0        Set the current variable number to null (clear current variable)
#33 = #33+1        Update variable number count by one
END1               End of loop - return to the WHILE block and evaluate again
M99               End of macro
%
```

Variable **#33** is a local variable and serves as a counter. Its initial setting is 500, the first variable of the range to be cleared. The maximum range is controlled by the **WHILE** loop, and the example shows **#999** as the last variable in the range. This number should be changed to match the control system. This is also a macro, where the basic **G65** statement needs *no arguments*:

```
O0019              Main program number
. . .
G65 P8011          Calls macro O8011 to clear all 500+ variables - no arguments
. . .
M30               End of main program O0019
%
```

The macro *O8011* can be very easily adapted to learning variables in the 100+ series as well. Just change the initial setting (**#33**) and the maximum range in the **WHILE** loop.

Many examples in the handbook use branching and looping functions that can be used in everyday work. The majority of them contain comments and explanations along with practical applications. Use them as a resource to create unique macros that can be used on a daily basis.

14

ALARMS AND TIMERS

The idea of actually *causing* an alarm during machining operations by a CNC program may be a bit uncomfortable or even peculiar to many users, regardless of experience. Yet, creating a specific control system alarm - *for a good reason, of course* - is nothing more than applying a very important tool in macro development. Even under normal operating conditions, all control systems automatically switch to the alarm mode, if a serious and *detectable* problem occurs. The keyword here is '*detectable*'. Creating macro alarms means just *adding* customized alarms to the ones already in the control system.

One basic rule applies to all custom generated alarms - they should be implemented by a macro program only under one condition - whenever an adverse situation is *predictable*. The purpose of all alarms is to terminate the current program activity and force a change in the current conditions, whatever they may be.

Alarms in Macros

Macro can include a programmed alarm (also known as an *intentional error condition*), using the system variable #3000. The variable #3000 must be followed by an alarm number, with an optional message.

Alarm Number

Depending on the control system, the alarm number can be within a range of:

- 0 to 200 and more *for FS-0/16/18/21 controls*
- 0 to 999 *for FS-10/11/15 controls*

The alarm number selection is at the programmer's discretion, subject to control specifications.

Alarm Message

Alarm *must have* a number, but the alarm message is optional. Programming a descriptive message will inform the CNC operator about the cause of the alarm. Alarm message must be in the same block as the alarm number, enclosed in parentheses, and it can be up to 26 characters long (31 characters on some controls), including spaces. Its contents should be clear, without ambiguous meaning. For example,

```
(TOOL ERROR)           ... is an ambiguous message
(TOOL RADIUS TOO LARGE) ... is a clear message
```

If the message is present, both the alarm number *and* the message will appear on the screen when the alarm is tripped. If the alarm message is not present, only the alarm number will appear.

Alarm Format

The macro O8012 illustrates the actual application of a macro alarm that checks the input of an assigned variable (*i.e.*, argument R, assignment #18). Macro will check if the input radius is greater than 2.5 mm:

```
G65 P9000 R2.5           Macro call with one argument (radius amount)

O8012                   Macro start
...
IF[#18 GT 0.25] GOTO1001  Check condition for alarm - true or false ?
...                     Process all blocks if condition is false
N1001 #3000 = 118 (RADIUS TOO LARGE) Force alarm if condition is true
```

The selected alarm number and message to the operator is displayed on the screen as either

```
118 RADIUS TOO LARGE           or           3118 RADIUS TOO LARGE
```

Slight variations may be expected. This is a typical application of a programmed alarm - a controlled generation of an alarm by a macro, for a predictable possibility of an error.

Embedding Alarm in a Macro

Regardless of which alarm conditions are used in the macro, the transfer between the processed and the unprocessed portions of the program must be smooth, regardless of the returned value (*true* or *false*). For example, a macro may contain the following three alarms (to 'go-to'):

```
N1001 #3000 = 101 (HOLE SPACING IS TOO SMALL)
N1002 #3000 = 102 (TWO HOLES MINIMUM REQUIRED)
N1003 #3000 = 103 (DECIMAL POINT NOT ALLOWED)
```

In the macro O8013, these alarms will most likely be located towards the macro end. However, the macro program that *precedes* the alarms, using **G65 P8013 H8 I12.0 X75.0 Y100.0** macro call, will have to be processed without interruption, if the conditions are *false* (that means running good program, with no alarms). For example, this macro structure is *NOT* correct:

```
O8013                   INCORRECT way to program alarms
IF[#4 LE 0] GOTO1001   I=#4 variable stores the hole spacing
IF[#11 LT 2] GOTO1002  H=#11 variable stores the number of holes
IF[#11 NE FUP[#11]] GOTO1003  Check if #11 contains the decimal point
G90 X#24 Y#25          Previously defined tool location XY
< ... macro body processing ... >
N1001 #3000 = 101 (HOLE SPACING IS TOO SMALL)
N1002 #3000 = 102 (TWO HOLES MINIMUM REQUIRED)
N1003 #3000 = 103 (DECIMAL POINT NOT ALLOWED)
M99
%
```

The **G65** block for the above macro contains all input correctly - 8 holes, 12 mm apart. It means all **IF** tests will be *false* and the macro will continue as intended. It will continue until it reaches the first alarm message. At this point the alarm takes over and macro processing stops. Ironically, if one argument is input incorrectly, the macro will issue the appropriate alarm. With all arguments being correct, one alarm *will always be issued!* That means a flawless macro will generate alarm 3101 or 101 (*HOLE SPACING IS TOO SMALL*), indicating wrong data input. Nothing is wrong with the input, so what is the reason? If a branch is based on a certain condition, the macro *true* and *false* sections have to be separated. In the O8013 example they were not. The alarm messages were not bypassed if all data input is good. To bypass them, the unconditional **GOTO**n function must be included by itself in a block. The **n** is the block number to branch to. In *unconditional* branching, there is no **IF**, no **WHILE** - just **GOTO**n. Program O8014 corrects the previous macro:

```
O8014
IF[#4 LE 0] GOTO1001
IF[#11 LT 2] GOTO1002
IF[#11 NE FUP[#11]] GOTO1003
G90 X#24 Y#25
  < ... macro body processing ... >
GOTO9999
N1001 #3000 = 101 (HOLE SPACING TOO SMALL)
N1002 #3000 = 102 (TWO HOLES MINIMUM REQUIRED)
N1003 #3000 = 103 (DECIMAL POINT NOT ALLOWED)
N9999 M99
%
```

CORRECT way to program alarms
I=#4 variable stores the hole spacing
H=#11 variable stores the number of holes
Check if #11 contains the decimal point
Previously defined tool location XY
Unconditional bypass added
Block number to branch to

Compare the two previous versions (changes are identified in the O8014 version). The only addition that can make the macro even better is the saving and subsequent restoring of the current modal values. Such improvement has nothing to do with alarms, and is included here to show the complete program segment using intentional alarms. The following example shows the saving of the current state of G-codes (*Group 3*) at the beginning of the macro, and restoring it at the end:

```
O8015
#10 = #4003
IF[#4 LE 0] GOTO1001
IF[#11 LT 2] GOTO1002
IF[#11 NE FUP[#11]] GOTO1003
G90 X#24 Y#25
  < ... macro body processing ... >
GOTO9999
N1001 #3000 = 101 (HOLE SPACING TOO SMALL)
N1002 #3000 = 102 (TWO HOLES MINIMUM REQUIRED)
N1003 #3000 = 103 (DECIMAL POINT NOT ALLOWED)
N9999 G#10
M99
%
```

Current G90 or G91 saved
Unconditional bypass of alarm list
Previously saved G90 or G91 restored

Many macro programmers do not use alarms at all or use them rather poorly. When writing a macro, write it first without the alarms. When everything works, try to *predict* what type of errors are possible later, when the macro is actually used. Then add all alarms covering these situations.

Resetting an Alarm

When a macro alarm is generated, it will have an accurate resemblance to an alarm generated by the control system in non-macro applications. Typically, this is the sequence of events:

1. **CYCLE START** light will be turned off
2. The word **ALARM** will flash on the screen
3. Alarm number and message (if available) will appear on the screen

At this moment, the control system has stopped all operations. To get rid of the alarm, press the **RESET** key. The source (the cause) of the alarm has to be removed, so make sure all tool positions are correct, then press the **CYCLE START** key to run the macro again, this time without an alarm.

Message Variable - Warning, Not an Alarm

The system variable **#3006** is only available on FS-10/11/15 controls - it allows the programmer to issue a message in the macro, *without* creating an alarm condition. Think of the message variable as means of issuing a warning, rather than an alarm. A message (warning) can be used instead of an alarm in situations that are not wrong, but when the operator should be warned or otherwise informed of an important issue.

For example, the following program uses argument D (**#7**) as clearance amount:

```
G65 P8016 D1.5           Macro call (D = #7 - is clearance amount)
```

In the macro body, the system variable **#3006** is programmed in a format similar to the alarm variable, but with a different number:

```
O8016
...
IF [#7 LT 2.0] GOTO101
...
...
N100 GOTO9999
N101 #3006 = 1 (2 MM MINIMUM CLEARANCE RECOMMENDED)
...
N9999 M99
%
```

When the program message is activated, the **CYCLE START** light of the control will be turned off and the message will appear on the screen. When the CNC operator presses the **CYCLE START** button again, the part program processing will continue. No reset is necessary in this case and there is no need to press the **RESET** button - it will actually be a counterproductive effort, as it would cancel program execution. Watch the flow of the macro here - the program may be constructed in such a way that the bypass block may not be needed, even if the condition is true.

Use the message variable sparingly. This is an example when the programmer is giving up certain control over the program and leaves it in the hands of the CNC machine operator.

Timers in Macros

In the last section of this chapter, the topic covers strictly *programmable* timers, not timers related to hardware settings in the service or maintenance sense. There are several system variables available on Fanuc controls relating to timers. Basically, these variables cover the information about the *date* and *time*, and several other options for timing various events.

Time Information

System variables #3001, #3002, #3011, and #3012 relate to the various time data. Time information may be read as well as written to (*Read & Write* or *R/W*).

Variable Number	Description
#3001	This is a millisecond timing variable, with the count of one millisecond at a time. The counting starts from zero when the power is turned on, and continues up to 65535.0 milliseconds, then starts from zero again. It counts all the time.
#3002	This is an hour timing variable, with the count of one hour at a time. The counting starts from zero when the cycle start is first pressed, and continues to the 114534.612 hours, then starts from zero again. Timer is updated only when the cycle start lamp is turned in (in cycle start mode only).
#3011	This variable contains the current date, in the form of year, month, day (format is YYYYMMDD). A given date, for example, December 7, 2005, will be displayed as 20051207.
#3012	This variable indicates current time, in the form of hours, minutes, seconds (format HHMMSS). A time, for example, 8:36:17 p.m., will be displayed in a 24-hour format as 203617.

Timing an Event

An event may be timed using either the #3001 or #3002 system variables. The following example does not do very much in practical terms, but it is designed in such a way that the expected result may be calculated.

Evaluate the enclosed comments or try at the control to see how the timer works exactly:

```

O8017 (TIMING AN EVENT)
(PART --ONE-- USING #3001)
#3001 = 0                               Reset to zero (start counting from zero)
G91 G01 X-100.0 F200.0                 Duration of this motion is 30 seconds
X100.0 F400.0                           Duration of this motion is 15 seconds
N999 (THIS MUST BE AN EMPTY BLOCK)     An empty block to prevent look-ahead !!!
#101 = #3001                             Returns calculation of 45632.000 (milliseconds)
#102 = #3001/1000                         Returns calculation of 00045.632 (seconds)
M00                                       Temporary stop to check variables

(PART --TWO-- USING #3002)
#103 = #3002                             Reset to zero (start counting from zero)
G91 G01 X-100.0 F200.0                 Duration of this motion is 30 seconds
X100.0 F400.0                           Duration of this motion is 15 seconds
N999 (THIS MUST BE AN EMPTY BLOCK)     An empty block to prevent look-ahead !!!
#104 = [#3002-#100]*3600                Returns calculation of 45.631993 (seconds)
M00                                       Temporary stop to check variables
M30                                       End of program
%

```

Note the blocks N999 and the attached comment. Since the control is in the look-ahead mode, it calculates the final value prematurely. The empty block guarantees accurate calculated value.

Dwell as a Macro

Although the dwell function **G04** can be used much more efficiently in the majority of programs, the dwell may also be programmed with a macro, using the system variable **#3001**. For example, **G04 P5000** (a five second dwell) is equivalent to the following macro (and its call):

◆ Macro call:

```

...
G65 P8018 T5000                         T=#20 can be any other local variable (T is in ms)
...

```

◆ Macro definition:

```

O8018 (TIMER AS A DWELL)
#3001 = 0                               Set system variable for timer to zero
WHILE[#3001 LE #20] DO1                 Loop until #3001 reaches the set delay
END1                                       Loop end
M99                                       Macro end
%

```

Note that even with the **WHILE** loop in effect, there is no need to program a counter, since the system variable **#3001** is always counting.

15

AXIS POSITION DATA

During machining, the cutting tool location changes constantly. Looking at the control *Position* display screen, the current tool location can easily be viewed at any time. There are several ways of looking at the displayed data, for example, the view may show the current absolute position of the tool (from program zero), or the machine position of the tool (from machine zero) - they are just two possible options. Control system keeps track of all position related tool data, called *the axis position information*. They are described in this chapter.

Axis Position Terms

Fanuc uses several abbreviations that appear in the reference manual. They should be familiar to any macro programmer who works with axis positions. They make look a bit intimidating at first, but are logical and easy to get used to. These are the four variables that relate to the axis position information:

➤	ABSIO	ABSMT	ABSOT	ABSKP
◆	ABSIO	Programmed endpoint coordinate of the previous block		
		#5001 - #5015	for the 1st to 15th axis respectively	
◆	ABSMT	Machine position - always current machine coordinates		
		#5021 - #5035	for the 1st to 15th axis respectively	
◆	ABSOT	Absolute position - always current absolute position		
		#5041 - #5055	for the 1st to 15th axis respectively	
◆	ABSKP	Position stored during a block skip motion in G31 block		
		#5061 - #5075	for the 1st to 15th axis respectively	

In addition, there are two sets of system variables relating to the tool length offset value, and servo system deviation error.

The stored *Machine* and *Absolute* coordinates are the same as during a regular operation of the CNC machine. In macros, it means we cannot register (store) the current axis position value, until the active block has been completed. This is very useful in the block skip mode using the **G31** command for probing, but may be undesirable in many other cases. The system variable range of **#5001** to **#5015** stores the programmed endpoint (XYZ...) of the last block before the macro statement, even if these coordinates have not been actually reached. That allows for execution and calculations to be done *before* the next block. Improved processing speed is the result.

The **G31** skip motion command is described at the end of *Chapter 23*.

Position Information

Variables #5001 to #5115 are *read-only* variables, and cannot be written to.

Variable Number (1)	Position Information	Coordinate System	Tool Offset Value	Read Operation During Motion
#5001 to #5015 ABSIO	Previous block endpoint	Workpiece offset (G54+)	Not included	Enabled
#5021 to #5035 ABSMT	Current axis position	Machine coordinate system	Included	Disabled
#5041 to #5055 ABSOT	Current axis position	Workpiece offset (G54+)		
#5061 to #5075 ABSKP	Skip signal position (2)			Enabled
#5081 to #5095	Current tool offset value (3)			Disabled
#5101 to #5115	Amount of deviated servo position			

- (1) Each range of variable numbers is for 1 to 15 axes. The first number is for the X-axis, the second number is for the Y-axis, the third number is for the Z-axis, the fourth number is for the fourth axis, and so on up to the 15th axis possible.
- (2) During the execution of the G31 skip function, the range of variables #5061 to #5075 holds the tool position where the skip function is turned on. If the skip function is not turned on, this range of variables holds the end point of the specified block.
- (3) Note that the tool offset value range of #5081 to #5095 represents the *current* tool offset value, rather than the last value.
- (4) Read operation during a tool movement can be *enabled* or *disabled*. In disabled mode, buffering takes place and the expected values cannot be read.

16

AUTO MODE OPERATIONS

During CNC machining in *AUTO* mode, the operator decides whether and when to use the various overrides available on the machine operation panel. Overrides include the *Feedrate Override*, *Feedhold*, *Spindle Speed Override* and *Single Block*. With the exception of the *Spindle Speed Override*, all other functions can be controlled by macro and made *effective* or *not effective*. In addition, the macro can also control the *Exact Stop Check* mode and some wait code signals.

Controlling Automatic Operations

System variables **#3003**, **#3004** and **#3005** are used to control the state of various automatic operations. All these variables use a binary format in different combinations (0 or 1 entries). Depending on the actual machining requirements, these settings may be changed by a series of system variables. These variables are:

#3003	<i>Control of single block, wait signal for end signal (FIN)</i>
#3004	<i>Control of feedhold, feedrate override, exact stop check</i>
#3005	<i>Settings (System Settings)</i>

The default setting for **#3003** and **#3004** is 0, meaning that no features are disabled.

Single Block Control

System variable **#3003** is used for the automatic operation control of the single block switch. Single block may not be desirable in certain machining operations, for example in threading or tapping and some special operations.

Variable **#3003** can have four settings, where the 0 (zero) setting is the default when the machine and control power is turned on (the same as *active* or *enabled*):

System Variable #3003	Single Block Mode	M-S-T Function Completion
0	Enabled	Waits for completion
1	Disabled	Waits for completion
2	Enabled	Does not wait for completion
3	Disabled	Does not wait for completion

Single block mode in may be enabled or disabled using this system variable in a macro. If the single block is disabled using the variable #3003, the machine control panel setting of the single block switch has no effect on the result. The single block operation of the machine will be disabled regardless whether the *Single Block* switch is turned ON or OFF (OFF position is recommended). Pressing the *RESET* button or *Power Off* switch also clears the system variables #3003-#3004.

M-S-T Functions Control

Another feature of the variable #3003 deals with the completion of an auxiliary function when used in a single block, particularly the M-codes (miscellaneous functions). There are three auxiliary functions available on Fanuc CNC systems, often called the M-S-T or MST functions:

M	MISCELLANEOUS FUNCTIONS - also known as M-codes (most common) ... programmed with the address M - for example: M08
S	SPINDLE FUNCTIONS - also known as S-codes ... programmed with the address S - for example: S1250
T	TOOL FUNCTIONS - also known as T-codes ... programmed with the address T - for example: T05 for milling or T0202 for turning

➤ If the *M-S-T* completion is set to 'Wait for completion':

The next block of the program will not be executed until the *M-S-T* function initiated is completed

➤ If the *M-S-T* completion is set to 'Do not wait for completion':

The next block of the program will be executed without the wait for the initiated *M-S-T* function.

For example, if the system variable #3003 is set to *one* (1), that means the single block mode is disabled, and the pending M-S-T functions have to be fully completed, before the next block is executed. This is a typical mode of the built-in **G81** drilling cycle. No additional M-S-T function should be programmed unless the previous block has been completed. The variable #3003 is designed for such situations. The macro entry is simple - note the required mode must always be programmed in pairs - *ON/OFF* or *OFF/ON*. The **G81** equivalent setting will be:

```

O8019
#3003 = 1                               Disable single block, wait for M-S-T functions completion
...
    < ... tool motions ... >
...
#3003 = 0                               Enable single block, wait for M-S-T functions completion
M99
%
```

Most CNC lathes may benefit from these functions a little more than machining centers.

Regardless of the application (milling or turning), an important reminder:

Be careful when activating the state of the M-S-T functions !

Feedhold, Feedrate, and Exact Check Control

System variable **#3004** is similar to the **#3003**, but is used for automatic operation control of the *feedhold* switch, the *feedrate override* switch, and the *exact stop check* control. This variable can have up to eight settings, with the 0 (zero) setting as the default for all three functions, when the machine and control power is turned on. Zero setting means the function is active. Pressing the *RESET* button or *Power Off* will clear both system variables **#3004** and **#3003**.

System Variable #3004	Feedhold	Feedrate Override	Exact Stop Check
0	Enabled	Enabled	Enabled
1	Disabled	Enabled	Enabled
2	Enabled	Disabled	Enabled
3	Disabled	Disabled	Enabled
4	Enabled	Enabled	Disabled
5	Disabled	Enabled	Disabled
6	Enabled	Disabled	Disabled
7	Disabled	Disabled	Disabled

The **#3004 variable** controls three *states* of operation:

- Feedhold
- Feedrate *override*
- Exact *stop check mode*

Make sure to understand how these operations work before attempting to use them in macros.

◆ Operation State 1 - FEEDHOLD

When the *feedhold* is disabled in a macro, using the variable **#3004**, and the feedhold button on the control operation panel is pressed, the machine stops in the single block mode. If the system variable **#3003** (described earlier) has disabled the single block mode, there will be no single block mode operation available at all.

◆ Operation State 2 - FEEDRATE OVERRIDE

When the *feedrate override* is disabled in a macro, using the variable **#3004**, all machining will be done at 100% feedrate, regardless of the setting of the feedrate override switch on the operation panel. 100% feedrate is defined as the feedrate value specified in the CNC program or macro, using the F-address and applies equally to feedrate per minute and feedrate per revolution. It works the same for English and metric units of feedrate data input.

◆ Operation State 3 - EXACT STOP CHECK

When the *exact stop check* is disabled in a macro, using the variable **#3004**, there will be no exact stop check performed, even in blocks without a cutting motion. Exact stop check provides special means to check a tool position with the program commands **G09** (non-modal blocks), or **G61** (modal blocks). This state is not available on Fanuc 0.

Example of Special Tapping Operation

In *Chapter 8* (macro O8004) introduced a special tapping macro (or a tapping macro as a cycle). Although correct in principle, it did not provide built-in safety features. The macro required that when processed by the control system all overrides must be set to 100%. This requirement is difficult to maintain and serious machining problems could happen. Using various modes of the variable settings will disable the feedhold, the feedrate override and the single block modes. If the exact stop check should not be disabled, the macro definitions of the two variables will be **#3003=1**, and **#3004=3**. If the exact stop check should be disabled, the macro definitions of the two variables will be **#3003=1**, and **#3004=7**. See the above table for details.

As system variables **#3003** and **#3004** are often used for special cycles, for example, a custom cycle for tapping (described later in the handbook). In this macro example, we want to simulate the effect of the **G84** fixed cycle, except the cutting feedrate on the way in will be 80% of the given feedrate and on the way out 120% of the given feedrate. The tapping macro will be called at the current X and Y tool location.

◆ Macro call (in English units):

First, the **G65** block has to be defined with all required arguments and their assignments:

```
G65 P8020 R0.35 Z1.15 S750 T36
```

where ...

R = (**#18**) R-level clearance equivalent
 Z = (**#26**) Z-depth
 S = (**#19**) Spindle speed (r/min)
 T = (**#20**) Number of threads per inch (TPI)

Other arguments could be added, for example, calculation of the tapping depth, automatic calculation of the top clearance (known as the R-level in **G84** cycle), etc. For the purposes of the subject presented, there is no benefit in complicating the macro. Z0 of the part is the top face - note the positive value of the Z-depth (forced as negative in the macro).

◆ **Special tapping macro design:**

```

O8020 (SPECIAL TAPPING MACRO)
#3003 = 1           Disable single block setting
G00 Z[ABS[#18]] S#19 M03   Rapid to positive R-level + spindle rotation
#3004 = 7           Disable feedhold, feedrate override, and exact stop check
G01 Z-[ABS[#26]] F[#19/#20*0.8] M05  Feed to depth at 80% of feedrate - stop spindle
Z#18 F[#19/#20*1.2] M04    Feed back to R-level at 120% feedrate - reverse spindle
#3004 = 0           Enable feedhold, feedrate override, and exact stop check
M05                 Stop spindle
M03                 Restore normal spindle rotation
#3003 = 0           Enable single block setting
M99                 End of macro
%
```

Note that both variables **#3003** and **#3004** are used twice. This is important - if a particular state of a setting is changed for the macro only, it should be changed back, when the macro exits. Although some machining centers do not require stopping the spindle between **M03** and **M04** or **M04** and **M03**, it is a safe practice to use the **M05** in the macro anyway - it saves the spindle.

Systems Settings

System settings - represented by the last variable **#3005** in this series - relate to the current values of certain basic system configurations. System variable **#3005** may not be available on Fanuc controls 10/11/15.

Typical system settings include:

- Compatibility between controls (for example FS-15 vs. FS-16/18/21)**
- Automatic insertion of block numbers (sequence numbers - using the N-address)**
- English or metric input of dimensional values (using G20 and G21 respectively)**
- EIA or ISO mode selection for the output code**
- TV (Test Vertical) check performed or not performed**
- the TV check is only applied to punched tape equipment

Binary values are automatically converted to decimal values.

Mirror Image Status Check

Mirror image is generally a basic feature of most CNC machining centers and even some CNC lathes. Its main purpose is to reverse the directional sign of the specified axis, either the X-axis, the Y-axis, or both the XY-axes on CNC machining centers, and the X-axis on CNC lathes. In addition, the axis reversal may cause not only the change in the axis-motion direction, but also the change in the arc direction (CW vs. CCW), and the cutter radius offset. It is the cutter radius offset that is most critical, not the other features. Machinability of the part (*climb* milling vs. *conventional* milling) may also be affected.

In the macros, the status of the mirror image can be monitored for each axis individually. This macro feature is called the *mirror image check signal*. At any time during the macro processing, the macro can inquire as to the status of the current setting of the mirror image. The result of the inquiry is a binary value received and converted into a decimal format.

On most Fanuc controls, the system variable that stores the mirror image related information is **#3007** (*bit* type) - note the details of bits evaluations:

8th axis	7th axis	6th axis	5th axis	4th axis	3rd axis	2nd axis	1st axis
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

This typical status shows up to eight axes. In practice, however, only the first two axes will probably be used most often - the X and the Y axis (commonly the 1st and the 2nd axis) on a CNC machining center. Some older controls may have fewer than eight axes available under the variable **#3007**. The first line of the table is the axis description, the second line is the axis identification - the third line contains the binary equivalent of each bit (remember - 'binary' means 'based on only two selections', and counting starts from *zero*, not from one, to the left).

In each available bit, the setting can be either 0 or 1, depending on whether the mirror image is currently *disabled* or *enabled*:

- ◆ **0 = Mirror image function for the selected axis is *disabled***
- ◆ **1 = Mirror image function for the selected axis is *enabled***

Variable **#3007** cannot be written to - it is a ***read-only*** variable

As usual in this type of variables, it is the *logical sum* of the current status (sum of bits) that determines the status of all axes, therefore the return value of the system variable **#3007**. The return value of variable **#3007** will be the sum of all bits; it is important to know how to interpret this returned value correctly. Incidentally, this '*sum of bits*' is quite common in programming, and requires at least the basic knowledge of the *binary number system* (see *Chapter 4*).

Interpreting System Variable **#3007**

For the example, the common mirror image setting in machine shop applications will be used. This setting applies to the first two axes only (typically the X-axis and the Y-axis) of a typical CNC machining center. To find out what the status of the current mirror image setting is, evaluate the following example - reading it twice may help:

➤ Example - #3007 reads 3 - Active mirror image is in the X and Y axes

Why? When the system variable #3007 is equal to 3 (returned value is #3007 = 3), the currently mirrored axes are the first two axes in the chart, the X-axis and the Y-axis. This outcome can only be known by the *interpretation* of the returned value, stored in system variable #3007. To interpret the returned value, step through a small step-by-step procedure. The *step one* is to subtract the largest bit value possible from the value stored in the #3007 variable. In this particular case, the largest bit value that can be subtracted from 3 is 2. The number 2 is defined as the Y-axis, so the Y-axis is currently mirrored. The *step two* is to subtract the new value (calculated as 2 in the example) from the value of variable #3007 (= 3 in the example):

$$3 - 2 = 1$$

Since the resulting number 1 is related to the X-axis, it means the X-axis is mirrored as well. *But* - there still is the *step three* to do - to see what all other axes are there to calculate. Take the resulting number that is 1, and subtract one from it, $1 - 1 = 0$, so there are no more axes to consider. In the final evaluation, if #3007=3, both axes are mirrored. The same method can be shown in three simplified steps:

- | | | | |
|--------------------------|---------|-----------|--|
| <input type="checkbox"/> | Given: | #3007 = 3 | Largest bit value that can be subtracted from 3 is 2 |
| <input type="checkbox"/> | Step 1 | | The number 2 is defined as the Y-axis, so Y-axis is mirrored |
| <input type="checkbox"/> | Step 2: | 3 - 2 = 1 | The number 1 is defined as the X-axis, so X-axis is mirrored |
| <input type="checkbox"/> | Step 3: | 1 - 1 = 0 | No other axis is mirrored |

➤ Example - #3007 reads 2 - Active mirror image is in the Y axis only

In this case, the system variable #3007 is equal to 2 (returned value is #3007 = 2), the currently mirrored axis is the Y-axis. The interpretation is the same as before:

- | | | | |
|--------------------------|---------|-----------|--|
| <input type="checkbox"/> | Given: | #3007 = 2 | Largest bit value that can be subtracted from 2 is 2 |
| <input type="checkbox"/> | Step 1 | | The number 2 is defined as the Y-axis, so Y-axis is mirrored |
| <input type="checkbox"/> | Step 2: | 2 - 2 = 0 | No other axis is mirrored |

➤ Example - #3007 reads 1 - Active mirror image is in the X axis

In this last case, the system variable #3007 is equal to 1 (returned value is #3007 = 1), the currently mirrored axis is the X-axis. The interpretation is the same as before:

- | | | | |
|--------------------------|--------|-----------|--|
| <input type="checkbox"/> | Given: | #3007 = 1 | Largest bit value that can be subtracted from 1 is 1 |
| <input type="checkbox"/> | Step 1 | | The number 1 is defined as the X-axis, so X-axis is mirrored |
| <input type="checkbox"/> | Step 2 | 1 - 1 = 0 | No other axis is mirrored |

If #3007 = 0, there are no axes that are mirrored. From these examples and with a little knowledge of binary numbers, there should be no problem to 'read' - to interpret - other returned values of the system variable #3007.

Controlling the Number of Machined Parts

There are two more system variables that relate to auto mode operation. Two system variables, **#3901** and **#3902** control the counting of machined parts during an automatic operation.

They are:

#3901 *Number of parts completed (machined)*

#3902 *Number of parts required*

System variable **#3901** is used for the number of parts *machined*. The value of this variable stores the number of completed parts.

System variable **#3902** is used for the number of parts *required*. The value of this variable stores the number of required parts (the target number).

Both variables can be used to write to or to read from (read/write type). A negative value should not be used with these variables in macros (consider using the **ABS** function).

17

EDITING MACROS

Custom macro files are often loaded from an external computer to the CNC system memory. Long or complex macros are better edited in the computer file, then reloaded to the control system later. There are also many cases where this method may not be convenient, for example when the macros are very short or when editing existing macros, already loaded in the system. In such cases, macro statements can also be input through the control panel keyboard, right at the CNC machine. There is not much of a difference in editing macros than editing conventional CNC programs - they are edited by words; for example, to change the word X-123.456 to X123.456, the whole word has to be altered (editing of individual characters is not normally possible on most controls). Macro programs are entered and/or edited by special editing units.

Editing Units

A macro that has been stored in the control memory can be edited by moving the cursor to any editing unit that starts with the following character or symbol:

- ◆ **Address (alpha character of a word, for example X, Y, Z)**
- ◆ **Number symbol # at the left side of the = sign**
- ◆ **First character of IF, WHILE, GOTO, END, DO, POPEN, BPRNT, DPRNT and PCLOS**
- ◆ **Symbols /, (, =, and ;**

Program Comments

In a program, comments, messages and alarm indicators aimed at the machine operator, are placed within the body of the program, using parentheses (not brackets). For example, a message to the operator used with the miscellaneous function **M00** may read:

```
N34 M00 (CHECK THE DEPTH OF POCKET)
```

An example of a user generated alarm could be

```
N1001 #3000 = 118 (RADIUS TOO LARGE)
```

When entering a comment or a message from the control keyboard, look for two characters. The *control-out* character '(' and the *control-in* character ')' must be available on the keyboard.

Not all Fanuc models have these characters on the keyboard. For example, some earlier types of Fanuc 16 do not have the two characters on the keyboard, but do have it available as a softkey selection. If the control systems allows messages of any kind to be used, it may be convenient to type them (with the rest of the program) on an external computer, and load them via a cable from the computer to the CNC unit.

Abbreviations of Macro Functions

Many macro functions are special words (called functions) of more than two or three characters. During a keyboard input of these characters, extra time is needed to enter the whole multi-character words, often with the frequent use of the *Shift* key. Fanuc offers a two-character shortcuts for most of the available functions to speed up the manual keyboard input. This shortcut can be used for *inserting* new words or *altering* existing words:

Macro Function	Editing Shortcut	Macro Function	Editing Shortcut	Macro Function	Editing Shortcut
ABS	AB	BPRNT	BP	PCLOS	PC
ACOS *	AC	COS	CO	POPEN	PO
ADP *	AD	DPRNT	DP	ROUND	RO
AND	AN	END	EN	SIN	SI
ASIN *	AS	EXP *	EX	SQRT	SQ
ATAN	AT	FIX	FI	WHILE	WH
BCD	BC	FUP	FU	TAN	TA
BIN	BI	GOTO	GO	XOR	XO

The functions identified with (*) are *not* available on Fanuc 0/16/18/21 model controls. An abbreviated shortcut of the macro function will be fully displayed on the control screen. The following examples compare the two methods of macro abbreviations:

- Abbreviated length input example:

WH[AB[#1] LE RO[#2]]

Abbreviated input format

- Full length input example - the abbreviated input above is the same as the full input below:

WHILE[ABS[#1] LE ROUND[#2]]

Full input format

18

PARAMETRIC PROGRAMMING

This chapter introduces the basic and the key part of the handbook - it introduces various Fanuc control features that relate to practical uses for macros in a typical machine shop environment. It also covers the benefits that can be expected. Previous chapters have already provided many necessary 'tools of the trade'. That does not mean there are no more 'tools' available - on the contrary - but enough 'tools' have already been presented to be able develop some actual macros that can be used on a daily basis. Before discussing the actual development some background should help.

What is a Parametric Programming ?

Since the days of language based NC and CNC programming, parametric approach to program development was promoted. The requirements were quite expensive, because the user had to have a powerful mainframe computer (usually leased on a monthly basis) and equally powerful software. Needless to say, the high cost of ownership, various line-time fees, and even the rental costs, were the deterrent. Today, the only computer that is needed is the CNC system at the machine tool, equipped with the relatively inexpensive *Fanuc Custom Macro B* option. Personal computer or a laptop do help as convenience, but are not absolutely mandatory.

Parametric Programming is also called the *Family of Parts Programming*. As the name suggests, a group of similar parts, those belonging to the same family, can be programmed by using variable - rather than specific - dimensional and machining data. In this type of programming, decisions are included in the program, based on the supplied data, and adhering to certain constraints. Of course, much stronger programming tools than those available for standard CNC programming are required. Macros provide those tools. Parametric program is always a macro, but a macro does not have to be a parametric program in the sense of family of similar parts.

The next two chapters will provide details about actual development of macros for a family of similar parts and a family of similar operations.

Variable Data

What data can be of the variable type? Just about any data in the program can be variable. Typically, machining conditions are changed by different materials (soft or hard), types of cutting tool material (HSS or carbide), machine tool used (heavy built or light built), dimensional data, surface finish requirements, tolerances, and so on. Depth of cut, width of cut, number of cuts, spindle speed, feedrate, etc., they also may change, while the fundamental features do not.

On a very simple level, take a rectangular shape that has to be machined to a certain length and width. These two dimensional features are *variable* features, if many rectangles have to be machined. Making a separate program for each rectangle drawing is the traditional way. Making one macro that will do *any* rectangle is the most efficient way, the macro way. By substituting the length and width variables, the 'new' program can be used. Benefits are fast becoming clear.

Benefits of Parametric Programming

Fast turnaround in production is the most significant benefit of family of parts macros. More time is often needed to develop a macro than a standard program, but this time is an excellent investment, especially if the macro will be used often. Knowing the benefits parametric programming offers, contributes to better decision when to develop a parametric program and when a standard program is more suitable. Parametric programming benefits in these improvements:

◆ Overall benefits

- Quick turnaround between parts
- Reduced time for program checking
- Product quality improvement
- Decrease of overall production costs

Individually, the benefits may be further identified in the *production* and *programming* areas:

◆ Benefits in the production area:

- Reduction of scrap parts
- Increased quality of the machined part
- Tooling cost down due to standardized tooling
- Increased productivity of the CNC machine
- Lower maintenance costs

◆ Benefits in the programming area:

- Drastic reduction in programming time
- Programming errors reduced or eliminated
- Consistency for all similar parts
- Easier workload transition

In order to benefit from the parametric approach to programming, the first step is to identify suitable parts. Not every programming job is suitable for the additional investment in time.

When to Program Parametrically

The several areas already mentioned are also very important in determination whether the parametric programming will bring benefits or not:

- Large number of parts that are same in shape but different in dimensions
- Large number of parts that are similar in shape
- Parts that repeat fairly frequently
- Parts that contain repetitive tool path
- Various machining patterns

Parametric of programming is never a replacement for other methods - it only enhances them. There could be a significant investment in time spent on parametric macro program development. The resulting benefits must be tangible and measurable, in order to be economically efficient.

Planned Approach to Macro Development

When it comes to actually writing a parametric program, or any other macro, there are many personal preferences programmers choose from. Macros are usually written by experienced programmers, who have developed a certain programming style already. However, some techniques have proven to work well for most programmers. The first consideration, and the one that is also the most important, is to have a *goal - a purpose*. What objectives the macro should achieve?

Following this chapter is a simple but quite comprehensive practical example of a planned approach to macro development for family of parts. The techniques and considerations from this section will be applied to the practical example that follows. It is important to understand them well. The presented list is a suggestion only - its purpose is to offer guidance through all steps of developing a successful macro. The practical example will use many of these suggestions.

1. FIRST ESTABLISH THE MAIN OBJECTIVE

Many programmers can get a little too ambitious and try to set the objective too high and want a single macro to do too much. That could be a very serious mistake. Decide what the macro must do, evaluate other possibilities, discard what is impractical and adhere to that objective. Often two small macros are better than one large macro.

2. PLAN WELL AHEAD

Good planning is the key to success. Start with the drawing first, and for parametric programs, study several similar drawings. Identify the features that never change, and features that may change. Do not forget the material of the part, the setup methods, the machine used, and the tools. Try to predict what features may exist on similar drawings in the future. Always think ahead, and evaluate as many options as possible. Ask the right people for an opinion. Even with a well established objective, poor planning will result in a poor macro - establish strict criteria.

3. MAKE A GENERIC DRAWING SKETCH

Seeing is believing - draw a schematic sketch that shows all features of the planned macro. Use details if necessary, and establish critical locations, such as the program zero, clearances, start point for the tool, offsets, tool change point (if required), etc. If the macro requires the use of a mathematical formula, include the generic formula in the sketch and test the formula on *all* typical features. Such a working sketch, with or without calculations, should always be kept up to date and filed for future reference.

4. DECIDE ON THE TOOLPATH METHOD

Decide on the method of how the tool is going to approach the material, cut the material, and depart from the material. Think of the current part as well as the future parts. Can one tool be used or more tools are necessary? Can the toolpath be uniform? Is the starting point in a safe location? How about calculating the depth, width, stepover amount, number of passes, penetration clearance, roughing and finishing, and dozens of other considerations? Collect all information that can be collected, including machining conditions such as spindle speeds and feedrates. Keep in mind that the more variable data is included, the more powerful the parametric program or the macro becomes. On the negative side, it will take longer to develop and verify such a program.

5. IDENTIFY AND ORGANIZE VARIABLE DATA

Once the information is collected, identify and organize the data into coherent units. Decide what local variables will have to be defined as arguments in the **G65** command block. *Do not include data that can be calculated.* Include data that can be read from the drawing, even if they are not needed directly. For example, macro may need a circle radius for calculation, but the drawing specifies a diameter. Rather than asking for the radius input as an argument, supply the diameter, then divide it by two in the macro body. Watch for entries that require decimal point and/or negative values. Use relevant and mnemonic variable assignments if possible, for example **A (#1)** for an *angle* input, **R (#18)** for a *radius* input, and so on. This is not always possible, but some are better than none. Always document the meaning of all variables - easy to forget later!

6. DESIGN THE PROGRAM FLOW

A flowchart definitely helps at this stage of macro development. Many programmers consider the flowchart development a mandatory step, even insist on it. All the programming aids available in a macro, such as looping, conditional testing, branching, decision making, etc., can be represented graphically, in a flowchart. Once the flowchart is designed, test it several times, using different input conditions and decisions. The macro should work in all instances. Do not be afraid to test seemingly impossible or improbable conditions. When the flowchart logic fails and the flowchart is correct, the macro needs to be redesigned and tested again, *always from scratch!* With more experience, another way to design a smooth program flow is to establish so called *pseudo-code*, which is a common method by many software programmers. Pseudo-code is a very tight and detailed procedure, written in normal language, that sequentially lists every step and all steps of what has to be done. It is not as convenient method as a flowchart, but it does work.

7. DON'T COUNT ON DEFAULTS

In standard CNC programming, many programmers count on the defaults of the control system and do not include many program codes, particularly the preparatory G-codes. For example, they count on the default units system and do not include the **G20** or **G21** commands in the program. The same may apply to the **G90** or **G91** commands, and a number of others. In macros, always keep in mind that all decisions have to be reflected in the macro - never take anything for granted, and never count on system defaults.

8. WRITE THE MACRO PROGRAM

This is the stage that places the macro code on paper, in the control, or in the computer file. Its purpose is to develop the actual program. Data in the flowchart or in the pseudo-code is used, in the same order, with the same logic, and converted into the Fanuc macro code. It is important to document every macro - good documentation is not enough - only a first class documentation will do. Documenting the macro is not just aimed at the CNC operator, it is a permanent document available to any programmer who may work with the macro. Procedure that is clear today will fade away in a very short time. Documentation can be internal, in the form of commenting each block, or external, with descriptions in plain language. Equally important, actually very imperative, is to preserve all current program settings before a macro is executed, change the settings within the macro as needed, and restore the original settings before the macro exists. This approach is a sign of professionalism and makes a perfect and practical sense as well.

19

FAMILY OF SIMILAR PARTS

The previous chapter set the foundations of parametric programming in general - this chapter presents a complete practical application - an actual macro development for a typical family of similar parts. The next chapter will build on the same concept and offer samples of macro development for a family of similar machining operations.

Macro Development in Depth - Location Pin

This section may well be one of the most important subjects to study, as it covers macro development in a comprehensive manner. The project in the example is called *Location Pin*, and is designed for a CNC lathe. The logic and procedures presented here apply to both types of machines equally, as do many macro features. As is standard in CNC programming, the provided engineering drawing forms the first element in the process. In this example, all parts in the series are listed in a single drawing, so the drawing itself is parametric. In other cases, the programmer has to get the information from several individual drawings.

Figure 27 shows the pin data as supplied to the programmer. The first step is to study and evaluate the drawing and information provided in it.

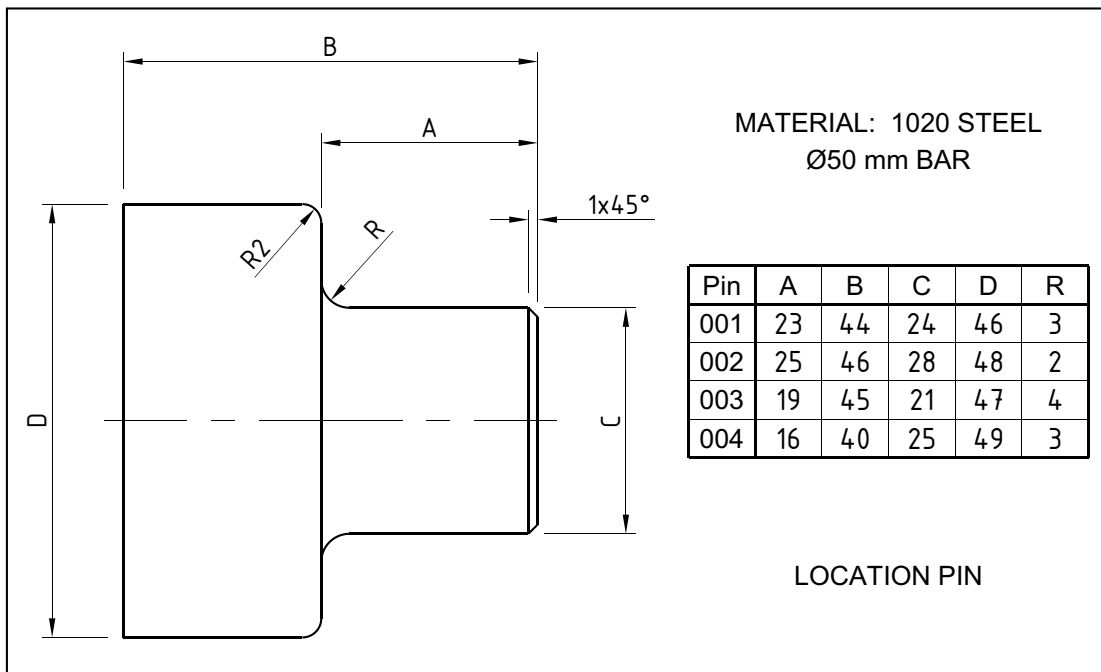


Figure 27

LOCATION PIN - drawing used to illustrate the development of a typical macro for family of similar parts

Drawing Evaluation

Even a casual look at the drawing reveals that this is not a single part drawing. There are four pins required to be machined (and programmed). All dimensions are given, so is the material. The designer of this part has chosen one generic drawing rather than four individual drawings. In a sense, the designer has approached the project as a family of parts - the same what the programmer will do. All four pins are similar - they share several characteristics. In all, there are seven dimensions specified. Two of these dimensions are fixed, the other five are variable. All dimensions provided are correct and will make the part as shown. For example, there will always be a flat shoulder face between the two radiuses. This and similar observations are very important to find out before the macro is even started.

Objective of the Macro

The single most important objective of the macro is to design it in such a way that all four pins in the series (and possibly others) can use a single part program - so they can be machined just by changing the assignments of the **G65** arguments (variables) in the main program. Before the objective can be met, a few technological decisions have to be made. The first of them is to decide on the part setup and the method of machining the part.

Part Setup, Tooling and Machining Method

The way the job is setup has to be considered together with the way it will be machined. Selection of one method often influences the other. Machining method influences the tooling selection. The drawing identifies the material as mild steel supplied as a bar of the same diameter for all parts (50 mm). For the purposes of macro development, these conditions have been established:

- | | |
|--|---|
| <input type="checkbox"/> Part zero at the front of the finished face | <i>X0 = mandatory center line</i> |
| <input type="checkbox"/> Minimum face cut (< 0.5 mm) | <i>single cut</i> |
| <input type="checkbox"/> Only one tool is used - T1 with wear offset 1 | <i>program can easily be changed to two tools</i> |
| <input type="checkbox"/> Spindle speed and feedrates do not change | <i>all parts are from the same material</i> |
| <input type="checkbox"/> G71 and G70 multiple repetitive cycles will be used | <i>two block format</i> |
| <input type="checkbox"/> No back operation will be included | <i>part-off and secondary operation</i> |
| <input type="checkbox"/> Coolant will be used | |

Every setup can be improved - this is only a suggestion and also illustrative method used for the example. The project focuses on macro development only.

The actual toolpath can also be defined in detail:

- Step 1 - Rapid towards the part for facing cut
- Step 2 - Face off the front just below center line
- Step 3 - Rapid to the start point of the G71 cycle
- Step 4 - Rough out the shape - leave suitable stock
- Step 5 - Finish the shape with G70

In five simple steps the machining is completed. Two tools can be used instead of one and other changes can be made as well.

Drawing Sketch

Even for simple parts, a good drawing sketch helps visualize the toolpath and data associated with it, such as clearances. *Figure 28* shows the part contour used for the **G71** and **G70** cycles.

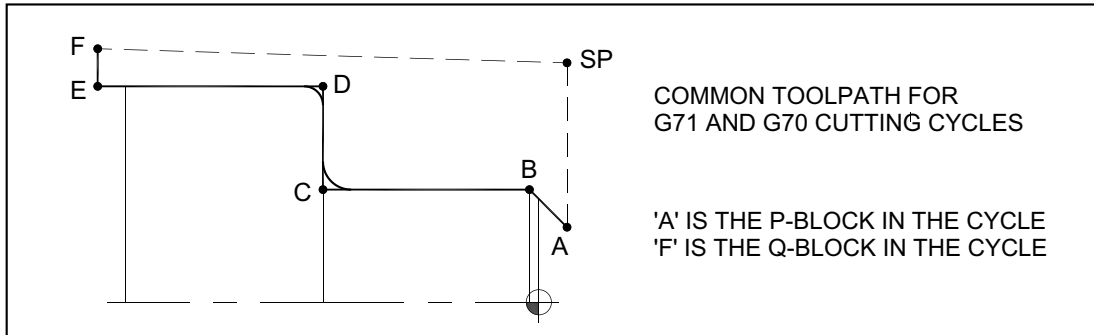


Figure 28

Common toolpath for all parts in the family - used by G71 and G70 cutting cycles

Standard Program

Select one part of the series (PIN-001 shown) and make a standard program for that part only, applying the previous selections. This is a step that can be avoided with growing experience, but going through it presents certain coherence to the development. Standard program for PIN-001 is listed with relevant comments:

(PIN-001 STANDARD PROGRAM)	<i>Standard program for PIN-001 only</i>
(X0Z0 - CENTERLINE AND FRONT FINISHED FACE)	
(BAR PROJECTION FROM CHUCK FACE = PART LG + 5 MM)	
N1 G21 T0100	<i>Metric units and Tool 1 - no wear offset</i>
N2 G96 S100 M03	<i>CSS at 100 m/min - CW spindle rotation</i>
N3 G00 X53.0 Z0 T0101 M08	<i>Start position for face cut + wear offset + coolant</i>
N4 G01 X-1.8 F0.1	<i>Face just below centerline at 0.1 mm/rev feedrate</i>
N5 G00 Z3.0	<i>Clear-off face - Z-axis only - by 3 mm 'A' for Z</i>
N6 G42 X51.0	<i>X-start for G71 cycle and tool radius offset 'A' for X</i>
N7 G71 U2.5 R1.0	<i>G71 - 2.5 mm cutting depth, 1.0 retract</i>
N8 G71 P9 Q14 U1.5 W0.125 F0.3	<i>G71 - N9 to N14 contour - XZ stock - 0.3 mm/rev</i>
N9 G00 X16.0	<i>Calculated X-diameter for chamfer - '1'</i>
N10 G01 X24.0 Z-1.0 F0.1	<i>Cut front chamfer at 0.1 mm/rev - '2'</i>
N11 Z-23.0 R3.0 F0.15	<i>Cut small dia + inner radius at 0.15 mm/rev - '3'</i>
N12 X46.0 R-2.0	<i>Cut face and outer radius - '4'</i>
N13 Z-47.0	<i>Cut large diameter 3 mm past part length - '5'</i>
N14 X54.0 F0.3	<i>Clear-off stock diameter - X-axis only by 2 mm - '6'</i>
N15 G70 P9 Q14 S125	<i>G70 finish contour at 125 m/min</i>
N16 G00 G40 X100.0 Z50.0 T0100 M09	<i>Rapid to tool change position + cancellations</i>
N17 M01	<i>Program stop (optionally - next tool expected)</i>

Once the standard program is established as correct, identify all values that have to be changed for any of the three remaining pins.

Identify Variable Data

The purpose of finding the data values that change from part to part means finding *variable data*. Data that changes will help establish variables for the macro, either as a direct input or for further calculations. In the next listing, the same standard program is presented, this time with all variable data underlined:

(PIN-001 STANDARD PROGRAM)	<i>VARIABLE DATA IS UNDERLINED</i>
(X0Z0 - CENTERLINE AND FRONT FINISHED FACE)	
(BAR PROJECTION FROM CHUCK FACE = PART LG + 5 MM)	
N1 G21 T0100	<i>Metric units and Tool 1 - no wear offset</i>
N2 G96 S100 M03	<i>CSS at 100 m/min - CW spindle rotation</i>
N3 G00 X53.0 Z0 T0101 M08	<i>Start position for face cut + wear offset + coolant</i>
N4 G01 X-1.8 F0.1	<i>Face just below centerline at 0.1 mm/rev feedrate</i>
N5 G00 Z3.0	<i>Clear-off face - Z-axis only - by 3 mm</i>
N6 G42 X51.0	<i>X-start for G71 cycle and tool radius offset</i>
N7 G71 U2.5 R1.0	<i>G71 - 2.5 mm cutting depth, 1.0 retract</i>
N8 G71 P9 Q14 U1.5 W0.125 F0.3	<i>G71 - N9 to N14 contour - XZ stock - 0.3 mm/rev</i>
N9 G00 <u>X16.0</u>	<i>Calculated X-diameter for chamfer - '1'</i>
N10 G01 <u>X24.0</u> Z-1.0 F0.1	<i>Cut front chamfer at 0.1 mm/rev - '2'</i>
N11 <u>Z-23.0</u> <u>R3.0</u> F0.15	<i>Cut small dia + inner radius at 0.15 mm/rev - '3'</i>
N12 <u>X46.0</u> R-2.0	<i>Cut face and outer radius - '4'</i>
N13 <u>Z-47.0</u>	<i>Cut large diameter 3 mm past part length - '5'</i>
N14 X54.0 F0.3	<i>Clear-off stock diameter - X-axis only by 2 mm - '6'</i>
N15 G70 P9 Q14 S125	<i>G70 finish contour at 125 m/min</i>
N16 G00 G40 X100.0 Z50.0 T0100 M09	<i>Rapid to tool change position + cancellations</i>
N17 M01	<i>Program stop (optionally - next tool expected)</i>

This is a simple example - six program entries have been identified (underlined). Study them individually and very carefully - these values will become variables in the macro. Block by block evaluation yields some insight into the selected data:

N9 G00 X16.0 *Calculated X-diameter for chamfer - '1'*

Block N9 represents the first point of the contour '1' (P9 in the cycle block). It is the X-position for the chamfer cutting that follows. This diameter is not on the drawing, it has to be calculated, based on the chamfer size (1 mm at 45°) at the small diameter and the current Z-clearance (3 mm as per block N5). Working with a 45° chamfer is always easy and no trigonometry is required. The small diameter is 24 mm, chamfer is 1 mm, and the Z-clearance is 3 mm.

To calculate the corresponding X-diameter is easy - just watch the process carefully and make sure all values are calculated for a diameter, not per side (radius):

$$X = 24 + 2 \cdot 1 + 2 \cdot 3 = 16 \text{ mm} = X16.0$$

This calculation will be part of the macro, using other variables, still to be defined.

N10 G01 X24.0 Z-1.0 F0.1

Cut front chamfer at 0.1 mm/rev - '2'

The smaller of the two diameters has two characteristics - it always starts at the end of the 1 mm chamfer (Z-1.0), and it is always defined directly in the drawing, although different for each part. As such, it automatically qualifies for a variable definition (assignment). The letter C in the drawing can also be used in the macro. Assignment C corresponds to the local variable #3 in the *Assignment List 1* (see Chapter 8).

Therefore, the first definition can be made, with a value assigned to each part:

Part number	#3 variable assignment
PIN-001	#3 = 24.0
PIN-002	#3 = 28.0
PIN-003	#3 = 21.0
PIN-004	#3 = 25.0

N11 Z-23.0 R3.0 F0.15

Cut small dia + inner radius at 0.15 mm/rev - '3'

In block N11, there are two variable data, again, both are directly defined in the drawing. The Z-position represents the length of the small diameter ('A' dimension in the drawing - between the front face and the shoulder), the R-value represents the inner fillet radius dimension in the drawing). Virtually all Fanuc lathe controls support *automatic cornerbreak* for chamfers or radiuses that are formed at 90°, between a face and a shoulder or a shoulder and a face, with enough travel for the break. If the automatic cornerbreak is not available on the control system, both starting and ending points of each arc will have to be calculated. In that case, a circular interpolation command G02 or G03 will be used.

Both letters from the drawing can also be used as assignments - the letter A corresponds to variable #1 and the letter R corresponds to variable #18, from the *Assignment List 1*.

Part number	#1 variable assignment	#18 variable assignment
PIN-001	#1 = 23.0	#18 = 3.0
PIN-002	#1 = 25.0	#18 = 2.0
PIN-003	#1 = 19.0	#18 = 4.0
PIN-004	#1 = 16.0	#18 = 3.0

Keep in mind that the tool will *not* cut all the way to the sharp corner. The control will detect the radius and start cutting at the proper location. The same cutting method applies to the outer radius.

As the program continues, the facing cut follows the fillet of the inner radius.

N12 X46.0 R-2.0

Cut face and outer radius - '4'

In block N11, the transition was between a diameter and a shoulder, using the automatic corner-break feature. Block N12 does exactly the same from a shoulder to a diameter. In this block, the outer radius of 2 mm is common to all parts in the series, and no variable is necessary. The target for all parts is the large diameter, identified in the drawing by the letter D.

Letter D corresponds to variable #7 in the *Assignment List 1*, and another table can be made:

Part number	#7 variable assignment
PIN-001	#7 = 46.0
PIN-002	#7 = 48.0
PIN-003	#7 = 47.0
PIN-004	#7 = 49.0

There is still one more cut left, that will also use a variable value - the part length B.

N13 Z-47.0

Cut large diameter 3 mm past part length - '5'

The table of the four parts specifications lists the letter B as the overall length of the part, that is the finished length when all operations are completed.

The letter B can also be used as a variable assignment (#2) and will be assigned the *defined* length of the pin, as per drawing:

Part number	#2 variable assignment
PIN-001	#2 = 44.0
PIN-002	#2 = 46.0
PIN-003	#2 = 45.0
PIN-004	#2 = 40.0

So far, all assignments matched the required drawing dimensions. In this case, that is not so. The PIN-001 example shows the Z-position in block N13 as Z-47.0, not as Z-44.0. The 3 mm difference is intentional - it provides a machined diameter for the subsequent part-off tool, to allow for a smooth entry of the part-off tool into the material. It may also include extra clearance for any secondary operation that will follow. This fixed amount of 3 mm has to be accounted for - somewhere in the program. That bring out a few questions:

Question 1 - should the variable #2 be changed by 3 mm? *Question 2* - should the 3 mm be a new variable? *Question 3* - should the 3 mm be part of the macro? Each of the three questions can be answered yes, but only one answer can be used, only one decision.

As part of the macro development, this question represents many possibilities that will arise. Each question is equally important and deserves to be evaluated on its own:

◆ **Question 1 - Should the variable #2 be *changed* by 3 mm?**

Definitely NOT! This is a very poor practice. Yes, it would work, but experienced programmers consider this type of input 'contaminated' or 'tarnished'. Any assignment that cannot be traced to some specific definition is not a good assignment. Seeing the #2 defined as B-47.0 or B47.0 (for example) does not provide the tie needed to the part drawing. In fact, it may be confusing, if the actual length of another part is true 47 mm. Stay away from this type of variable assignment.

◆ **Question 2 - Should the 3 mm be a *new* variable?**

Possibly. It depends whether the extra length provision for the part-off tool will change from one part to another. This can happen, for example, if the part-off tool that follows has a different insert width selected for each job or an additional facing clearance will be required for the secondary operation (not part of this example).

◆ **Question 3 - Should the 3 mm be *part* of the macro?**

If the answer to *Question 2* is positive, the answer to *Question 3* must be negative - and vice versa. There is no other option.

For the four pins example, there is no reason to change the 3 mm extended length from one part to another, so the answer is **NO** to *Question 2* and **YES** to *Question 3*. The 3 mm tool travel extension will be incorporated in the macro as a fixed value. Of course, it can be any other reasonable length of the extra motion, but always consider its impact on the part setup.

Creating Arguments

Once each variable has been assigned and related decisions have been made, it is a good time to put all information into one place - to create macro arguments for the G65 macro call. The following table sums up the arguments and variable assignments for all four parts - see *Figure 29*.

Part number	Dimension A	Dimension B	Dimension C	Dimension D	Dimension R
	A = #1	B = #2	C = #3	D = #7	R = #18
PIN-001	A23.0	B44.0	C24.0	D46.0	R3.0
PIN-002	A25.0	B46.0	C28.0	D48.0	R2.0
PIN-003	A19.0	B45.0	C21.0	D47.0	R4.0
PIN-004	A16.0	B40.0	C25.0	D49.0	R3.0

Note that the selected arguments do not always have to match the dimensions of the parametric drawing. Any legitimate arguments can be used, as long as it belongs to the *Assignment List 1*.

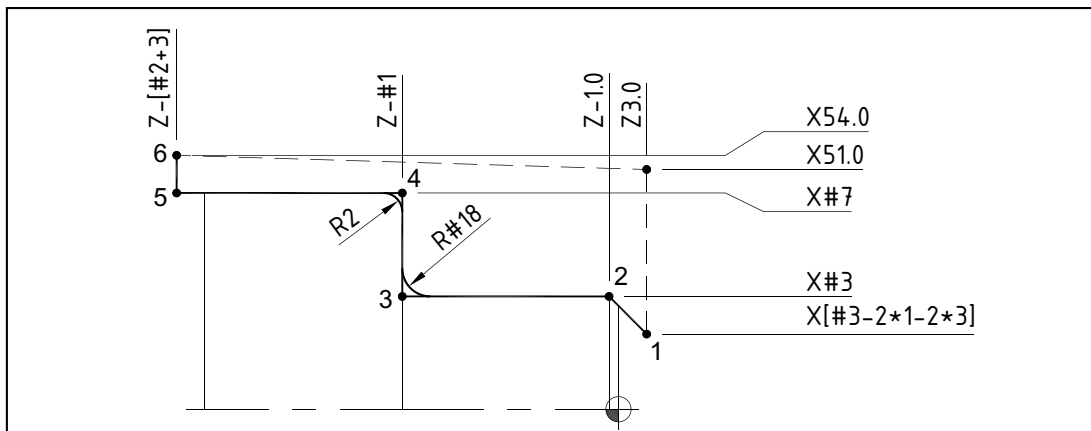


Figure 29

Common toolpath - fixed and variable assignments

Using Variables

Once the arguments have been defined, the macro can be developed, using the *Figure 29* as a reference. The structure of the macro must be such that it covers *each* part of the family. In its first version, the program will take on many macro characteristics. The underlined words that need no calculation will be changed into variables and those that *do* need calculation will create a composite variable entry.

All words affected by the change are still underlined for emphasis:

(PIN-XXX MACRO PROGRAM)	VARIABLE DATA IS UNDERLINED - PART 1
(X0Z0 - CENTERLINE AND FRONT FINISHED FACE)	
(BAR PROJECTION FROM CHUCK FACE = PART LG + 5 MM)	
N1 G21 T0100	<i>Metric units and Tool 1 - no wear offset</i>
N2 G96 S100 M03	<i>CSS at 100 m/min - CW spindle rotation</i>
N3 G00 X53.0 Z0 T0101 M08	<i>Start position for face cut + wear offset + coolant</i>
N4 G01 X-1.8 F0.1	<i>Face just below centerline at 0.1 mm/rev feedrate</i>
N5 G00 Z3.0	<i>Clear-off face - Z-axis only - by 3 mm</i>
N6 G42 X51.0	<i>X-start for G71 cycle and tool radius offset</i>
N7 G71 U2.5 R1.0	<i>G71 - 2.5 mm cutting depth, 1.0 retract</i>
N8 G71 P9 Q14 U1.5 W0.125 F0.3	<i>G71 - N9 to N14 contour - XZ stock - 0.3 mm/rev</i>
N9 G00 <u>X[#3-2*1-2*3]</u>	<i>Calculated X-diameter for chamfer - '1'</i>
N10 G01 <u>X#3</u> Z-1.0 F0.1	<i>Cut front chamfer at 0.1 mm/rev - '2'</i>
N11 <u>Z-#1</u> <u>R#18</u> F0.15	<i>Cut small dia + inner radius at 0.15 mm/rev - '3'</i>
N12 <u>X#7</u> R-2.0	<i>Cut face and outer radius - '4'</i>
N13 <u>Z-[#2+3.0]</u>	<i>Cut large diameter 3 mm past part length - '5'</i>
N14 X54.0 F0.3	<i>Clear-off stock dia - X-axis only - by 2 mm - '6'</i>
N15 G70 P9 Q14 S125	<i>G70 finish contour at 125 m/min</i>
N16 G00 G40 X100.0 Z50.0 T0100 M09	<i>Rapid to tool change position + cancellations</i>
N17 M01	<i>Program stop (optionally - next tool expected)</i>

Block N9 is a composite variable entry - it contains a calculation. A separate variable can be defined internally in the macro or the definition can be embedded into the tool motion address, creating a composite variable entry (as shown). Any variable defined as an assignment takes away memory space of the control system, whereby a composite calculation does not. An internal calculation will be provided in this case - PIN-001 shown:

X = #3 2 chamfer size 2 Z-clearance = #3 2 1 2 3 = 24 2 6 = 16 = X16.0

Figure 30 illustrates the calculation of the X-diameter in block N9 (P-address in the G71/G70 cycle) - already shown in the overall illustration (Figure 29) on the previous page.

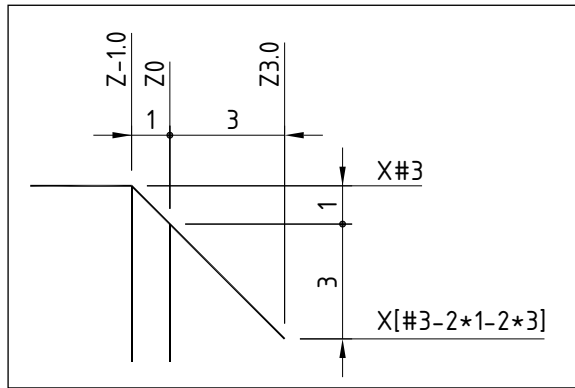


Figure 30

Detail view of the front chamfer calculation

Block N9 can now be written in a variable format (common calculation):

N9 G00 X[#3-2*1-2*3]

Calculated X-diameter for chamfer (variable)

Similar to N9 block calculation, but much simpler, block N13 (not shown) represents the extended tool motion by 3 mm along the Z-axis - note how the 3 mm distance has also been incorporated into the motion, along with the variable #2: **N13 Z- [#2+3.0]**. As in block N9, if the arithmetic or similar expression is enclosed in square brackets, the control system will calculate the combined return value first, then acts upon it - no separate variable definition is needed.

Writing the Macro

The final step is to take the standard program with macro features and turn it into a real macro. Macros should only include program blocks that change from one job to another (within the same family). In our case of the four pins, the roughing cycle (and its data) is the only area of the program that changes - the macro will only contain the G71 rough turning cycle, currently represented by blocks N7 to N14 and the G70 finish turning cycle, currently represented by the block N15 in the previous examples. Although only the actual contour changes, it is better to include both G71 and G70 cycles for easier orientation.

In order to write the macro, the last example presented has to be split into two sections:

- Section 1 - will include the main program with the G65 macro call and arguments
- Section 2 - will use the G71 and G70 machining cycles to cut the defined toolpath

Note the change in block numbers in the following programs:

```
(PIN-001 - MAIN PROGRAM)
(X0Z0 - CENTERLINE AND FRONT FINISHED FACE)
(BAR PROJECTION FROM CHUCK FACE = PART LG + 5 MM)
N1 G21 T0100                               Metric units and Tool 1 - no wear offset
N2 G96 S100 M03                             CSS at 100 m/min - CW spindle rotation
N3 G00 X53.0 Z0 T0101 M08                  Start position for face cut + wear offset + coolant
N4 G01 X-1.8 F0.1                          Face just below centerline at 0.1 mm/rev feedrate
N5 G00 Z3.0                                 Clear-off face - Z-axis only - by 3 mm
N6 G42 X51.0                               X-start for G71 cycle and tool radius offset
N7 G65 P8021 A23.0 B44.0 C24.0 D46.0 R3.0 (PIN-001 MACRO ARGUMENTS)
N8 G00 G40 X100.0 Z50.0 T0100 M09         Rapid to tool change position + cancellations
N9 M01                                       Program stop (optionally - next tool expected)
...                                         Other tool(-s) may follow

O8021 (PIN-XXX MACRO PROGRAM)              Four pins in the family are covered by this macro
N101 G71 U2.5 R1.0                         G71 - 2.5 mm cutting depth, 1.0 retract
N102 G71 P103 Q108 U1.5 W0.125 F0.3        G71 - N103 to N108 contour - XZ stock - 0.3 mm/rev
N103 G00 X[#3-2*1-2*3]                    Calculated X-diameter for chamfer - 'A'
N104 G01 X#3 Z-1.0 F0.1                   Cut front chamfer at 0.1 mm/rev - 'B'
N105 Z-#1 R#18 F0.15                      Cut small dia + inner radius at 0.15 mm/rev - 'C'
N106 X#7 R-2.0                             Cut face and outer radius - 'D'
N107 Z-[#2+3.0]                           Cut large diameter 3 mm past part length - 'E'
N108 X54.0 F0.3                            Clear-off stock dia - X-axis only - by 2 mm - 'F'
N109 G70 P103 Q108 S125                   G70 finish contour at 125 m/min
N110 M99                                   End of macro
%
```

Any legitimate block numbers are allowed, as long as there are no duplicates in the same program (subprograms and macros included). Keep in mind that the P and Q addresses in the cycles indicate block numbers actually used in the contour program.

Final Version

The purpose of this presentation was to develop a basic macro program for a family of similar parts. Without a doubt, many additions to the macro can easily be made, depending on the exact nature of the existing job. Tolerances and surface finish may play a great role in the program development, as may some specific requests by the customer. These are all easy to implement. The main objective was to introduce a skilled CNC programmer into the world of macros.

One significant change that can be made - and it can also show how flexible macros are - is to make it easier to change machining from one part to another.

In the macro O8021, the only way to change the assignments of variables for different pins is in the block N7 - the **G65** block. This is quite a common method, but not the best method. Much better method is to include all four definitions into a single main program, and change just one variable number (at the program top) to select the required part (pin type). This objective is easy to achieve by including the four definitions along with the **IF** function in the main program:


```

(PIN-001 TO PIN-004 SERIES - MAIN PROGRAM - MASTER)
(X0Z0 - CENTERLINE AND FRONT FINISHED FACE)
(BAR PROJECTION FROM CHUCK FACE = PART LG + 5 MM)
(-----)
N1 #33 = 1                                PART SELECT: 1=001 2=002 3=003 4=004
(-----)
N2 #30 = #4006                            Save current units of dimensioning (G20 or G21)
N3 IF [#33 GT 4] GOTO991                  ERROR (alarm) if part number is greater than 004
N4 IF [#33 LT 1] GOTO992                  ERROR (alarm) if part number is less than 001
N5 G21 T0100                              Metric units and Tool 1 - no wear offset
N6 G96 S100 M03                           CSS at 100 m/min - CW spindle rotation
N7 G00 X53.0 Z0 T0101 M08                 Start position for face cut + wear offset + coolant
N8 G01 X-1.8 F0.1                         Face just below centerline at 0.1 mm/rev feedrate
N9 G00 Z3.0                               Clear-off face - Z-axis only - by 3 mm
N10 G42 X51.0                             X-start for G71 cycle and tool radius offset
N11 IF [#33 EQ 1] GOTO15                  #33 = 1 ... selects PIN-001
N12 IF [#33 EQ 2] GOTO17                  #33 = 2 ... selects PIN-002
N13 IF [#33 EQ 3] GOTO19                  #33 = 3 ... selects PIN-003
N14 IF [#33 EQ 4] GOTO21                  #33 = 4 ... selects PIN-004
N15 G65 P8021 A23.0 B44.0 C24.0 D46.0 R3.0 (PIN-001 MACRO ARGUMENTS)
N16 GOTO22                                Bypass next three macro calls
N17 G65 P8021 A25.0 B46.0 C28.0 D48.0 R2.0 (PIN-002 MACRO ARGUMENTS)
N18 GOTO22                                Bypass next two macro calls
N19 G65 P8021 A19.0 B45.0 C21.0 D47.0 R4.0 (PIN-003 MACRO ARGUMENTS)
N20 GOTO22                                Bypass next macro calls
N21 G65 P8021 A16.0 B40.0 C25.0 D49.0 R3.0 (PIN-004 MACRO ARGUMENTS)
N22 G00 G40 X100.0 Z50.0 T0100 M09       Rapid to tool change position + cancellations
N23 GOTO998                               Bypass error messages if all OK
(-----)
N991 #3000 = 991 (PART NUMBER TOO LARGE)
N992 #3000 = 992 (PART NUMBER TOO SMALL)
N998 G#30                                 Restore previous units of dimensioning (G20 or G21)
N999 M01                                 Program stop (optionally - next tool expected)
...                                       Other tool(-s) may follow

O8021 (PIN-XXX MACRO PROGRAM)           Four pins in the family are covered by this macro
N101 G71 U2.5 R1.0                       G71 - 2.5 mm cutting depth, 1.0 retract
N102 G71 P103 Q108 U1.5 W0.125 F0.3     G71 - N103 to N108 contour - XZ stock - 0.3 mm/rev
N103 G00 X[#3-2*1-2*3]                  Calculated X-diameter for chamfer - 'A'
N104 G01 X#3 Z-1.0 F0.1                 Cut front chamfer at 0.1 mm/rev - 'B'
N105 Z-#1 R#18 F0.15                   Cut small dia + inner radius at 0.15 mm/rev - 'C'
N106 X#7 R-2.0                          Cut face and outer radius - 'D'
N107 Z-[#2+3.0]                         Cut large diameter 3 mm past part length - 'E'
N108 X54.0 F0.3                         Clear-off stock dia - X-axis only - by 2 mm - 'F'
N109 G70 P103 Q108 S125                 G70 finish contour at 125 m/min
N110 M99                                 End of macro
%
```

The definition of variable #33 has been visually enhanced to show the block where the selection of the active part will take place.

Macro Improvements

Every macro can always use some additional improvements and changes that make it stronger and more reliable. Some changes could be considered standard, others depend largely on the actual job. The macro development presented in this chapter is a thorough demonstration of macro development and presents a good overall look. It does not pretend to be the best example or even the only example possible. Feel free to change the macro to any unique conditions.

What possible improvements could be added to this or any macro? Not necessarily applicable to the macro just demonstrated, here is a summary of typical macro features that should help in the development:

1. **Safety considerations**
2. **Careful selection of variable assignments**
3. **Internal calculations rather than definitions**
4. **Included messages and alarms**
5. **Quality documentation**

One rule of computer programming for any application, is that the first and main objective is to develop the basic program core. Achieve the goal in as short code as possible. Forget the 'bells and whistles', forget the 'beautification' of the program. All that can - and should - be added only after the main objective has been met. What is the point of making a bad program look good?

20

MACROS FOR MACHINING

Macros can be used for many different purposes. The previous chapter covered the topic of macro development for a family of similar parts. The current chapter extends the subject and covers a development of various macros for general machining operations rather than a family of parts. Repetitive and predictable machining operations are one of the most common applications of a *parametric* program. All parametric macros provided in this chapter have many more additional features than the similar ones often found on various web sites or internet forums.

Study ALL macros provided in this chapter - each example presents a new technique that can be used in programming of other macros. Using a specific technique used in one macro and adapting it to another macro will increase the macro usefulness and flexibility.

There are virtually limitless possibilities for macro applications as parametric programs and only a small selection is offered here. The purpose of these examples is to show actual samples of a macro code for several very useful types of machining. Study the logic of each macro for the programming techniques used and follow the complete design. The explanations with each example will serve as a guide to a macro development, regardless of the final goal.

All macros are provided for training purposes only with no guarantees !

Angular Hole Pattern - Version 1

One of the most common applications - and one of the simplest - is a linear pattern of holes, with an equal distance between holes. The objective of this parametric macro is to create a toolpath for any drilling operations applied to a pattern of holes arranged in a line. The drawing in *Figure 31* is an example of this type of pattern. Program zero is at lower left corner and top of the part.

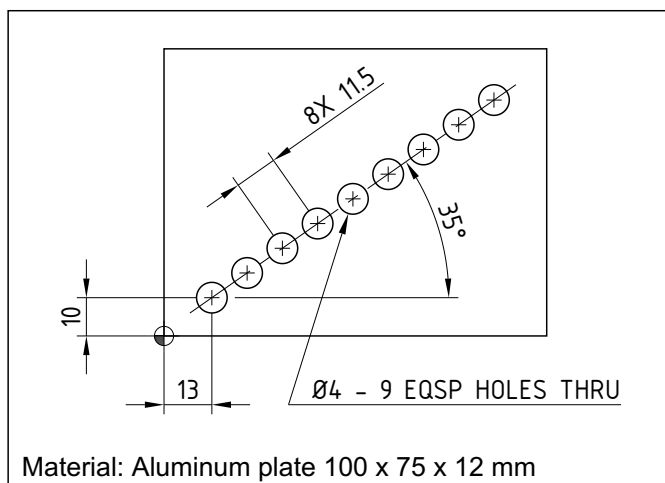


Figure 31

Drawing example of a typical angular hole pattern - version 1

In this application, the coordinates for the first hole are given, along with the pattern angle

When considered with similar drawings, this example provides all data needed to set up macro framework. It covers the conditions and restrictions, in which the macro application will be valid. For the educational purposes, this first application will be relatively simple (but definitely useful). Many programming techniques used in this macro will repeat in subsequent examples, with added features. First, evaluate the various self-imposed conditions and constrains towards the set goal:

- All holes are spaced equally within the pattern *EQSP holes*
- Any number of holes is acceptable - minimum of two holes *within machine capabilities*
- The distance between the holes must be known *pitch of holes*
- Any pitch between holes is acceptable *within machine capabilities*
- The location of the first hole must be known *as XY coordinates*
- Any angle between the first hole and the last hole must be known *establishes direction*

Once the conditions have been established and applied to an example, like the one in *Figure 31*, the most important first step has been completed. When evaluating a single drawing, always think of all other possibilities that may exist in similar drawings. For example, is the pattern of holes horizontal or vertical, is it rotated in the opposite direction, should the macro still be able to handle this pattern? Logically, there is no fundamental difference between one orientation and another. Always consider *all* features, including the angle, even if the angle is zero. Zero degree angle will define the horizontal orientation to the right of the first hole (east direction). A one-hundred-eighty degree angle defines the horizontal orientation to the *left* of the first hole (west direction). In the macro, the defined angle controls the orientation of the linear hole pattern. Based on all these considerations, including the self-imposed restrictions and other decisions, a common macro specific drawing will be necessary, applicable to *all* similar patterns - *Figure 32*:

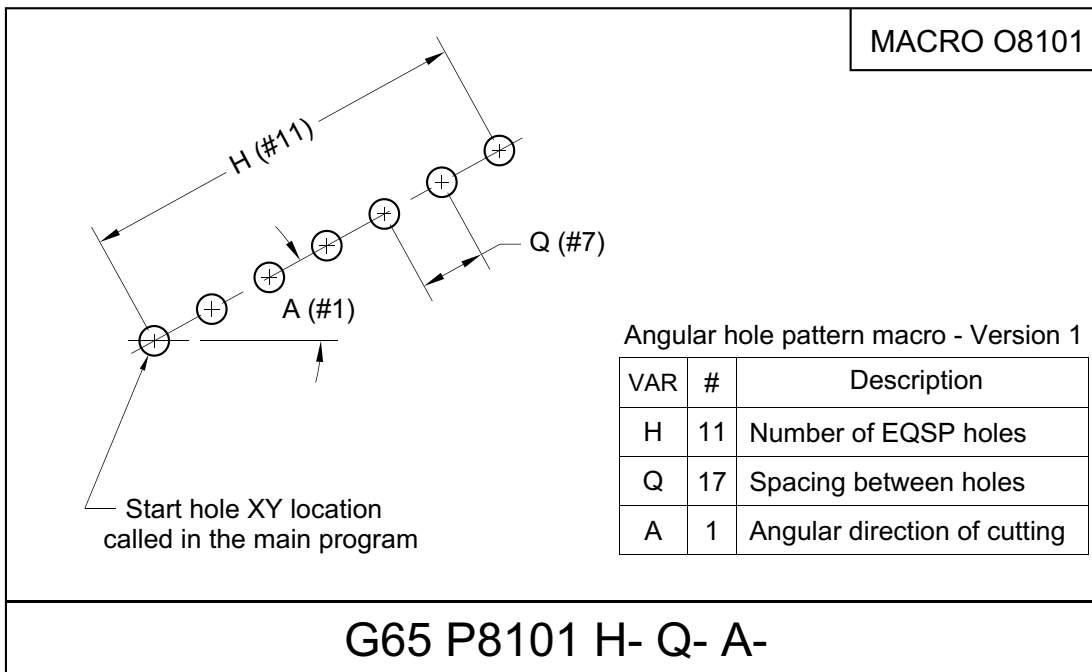


Figure 32

Variable data for angular pattern of holes - Version 1 - Macro O8101

Variable Data for Angular Hole Pattern

From the given conditions, it is easy to establish the required variable data and define the convenient argument assignments for them, as shown in *Figure 32*. For reasons of simplicity in this first macro, the absolute XY location of the start hole has been left out. In the examples that follow, such a location will also be part of the macro, if required.

- Number of EQSP holes** ... assigned letter H (variable assignment #11)
- Spacing between the holes (pitch)** ... assigned letter Q (variable assignment #17)
- Angular direction of cutting** ... assigned letter A (variable assignment #1)

In the main program, the process flow will be very similar to any standard program that calls a subprogram. A fixed cycle will be used to drill the first hole at its absolute location, change to incremental mode and repeat the cycle for all remaining holes.

```
O0021 (MAIN PROGRAM)
N1 G21 Metric mode
N2 G90 G00 G54 X13.0 Y10.0 S800 M03 First motion block with spindle speed
N3 G43 Z25.0 H01 M08 Tool length offset with clearance above
N4 G99 G81 R2.5 Z-14.7 F150.0 Hole #1 machined at current location
N5 G65 P8101 A35.0 H9 Q11.5 Macro call with assignments
N6 G90 G80 Z25.0 M09 Retract above work
N7 G28 Z25.0 M05 Return to machine zero
N8 M01 End of current tool

O8101 (ANGULAR HOLE PATTERN MACRO - VERSION 1)
#11 = #11-1 Change number of holes to number of spaces
#24 = #17*COS[#1] Calculation of the X-increment
#25 = #17*SIN[#1] Calculation of the Y-increment
G91 X#24 Y#25 L#11 Increment L-times (K-times for Fanuc 16/18/21)
M99 End of macro
%
```

The macro occupies two variables *undefined* in the macro call (**#24** and **#25**). Although correct, any storage of values into separate variables takes away memory resources. There is no need to define separate variables, because the calculation they provide will be used only once. As a better variation of the O8101 macro - without the two variables, consider this method:

```
O8101 (ANGULAR HOLE PATTERN MACRO - VERSION 1)
#11 = #11-1 Change number of holes to number of spaces
G91 X[#17*COS[#1]] Y[#17*SIN[#1]] L#11 New location and increment (L or K)
M99 End of macro
%
```

That concludes the development of a simple, yet quite versatile, first macro application for machining operations. With this macro, any row of equally spaced holes with a given angle can be programmed very easily into any direction, based on the given first hole location and the angle definition (measured from zero degrees). Many improvements can be added to this macro, as already shown before, and as also shown in the several macros that follow.

Angular Hole Pattern - Version 2

Just because the word '*angle*' appears in the definition, it does not mean the actual angle is always defined in the drawing. In fact, there is another common method of dimensioning an angular hole pattern. Rather than using the coordinates of the first hole and a specified angle, it uses coordinates for the first hole and the distance between the first and the last holes in the pattern, with no angle definition. Even if the drawing shows the absolute coordinates of the second hole, it is easy to find the distance between holes. Which drafting method is used depends on the engineering intent - what is the purpose of the design. Of course, any skilled CNC programmer can change one method to the other, but there is always a risk of some rounding error that could be significant. The solution? Another macro. The typical part example drawing is shown in *Figure 33*.

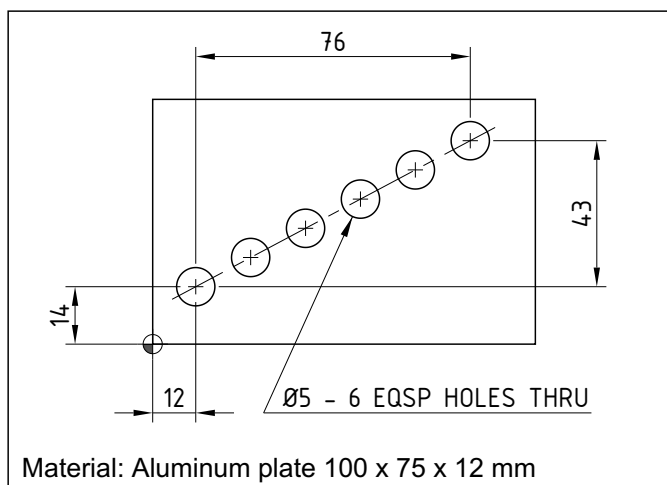


Figure 33

Drawing example of a typical angular hole pattern - version 2

In this application, the coordinates for the first hole are given, along with the X and Y distance between the end holes

When developing macros where the differences are small, as in these examples, it is a good idea to maintain as many variable assignments as possible. For example, the number of holes can still use the H assignment (#11).

Note the difference in drawing data between *Version 1* and *Version 2* - there are two distances but no angle or space between holes. The provided data must be part of the variable assignments, but the spacing between holes will be calculated inside the macro. In this version, the location of the first hole will be defined in the main program, but different two variables will be the X-length and the Y-length of the pattern:

- Length of the pattern along X-axis ... assigned letter U (variable assignment #21)
- Length of the pattern along Y-axis ... assigned letter V (variable assignment #22)
- Number of EQSP holes ... assigned letter H (variable assignment #11)

The macro can be improved in number of ways. As only one tool is shown in the example, it may not be convenient to repeat the macro exactly the same for two or even three tools. Even the drilling cycle can be built in the macro, depending on the exact conditions at the time of programming. This macro is very similar to the previous one, but can also include a few extra features that can easily be adapted to any macro. The variables are visually defined in *Figure 34*.

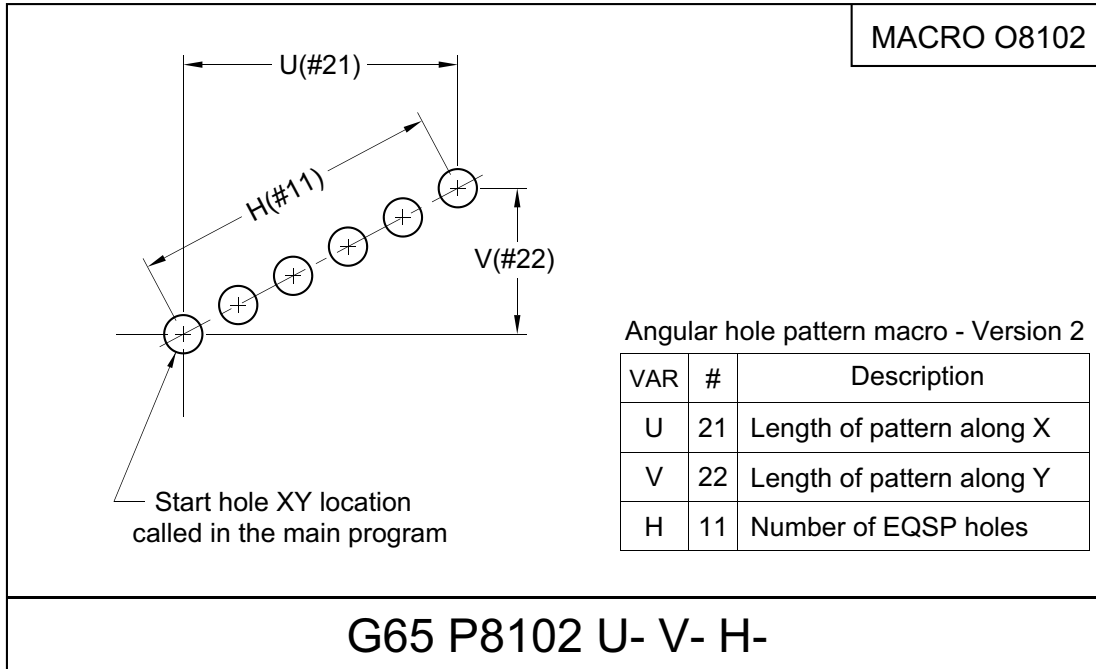


Figure 34

Variable data for angular pattern of holes - Version 2 - Macro O8102

O0022 (MAIN PROGRAM)

```

N1 G21
N2 G90 G00 G54 X12.0 Y14.0 S775 M03
N3 G43 Z25.0 H01 M08
N4 G99 G81 R2.5 Z-15.0 F150.0
N5 G65 P8102 U76.0 V43.0 H6
N6 G90 G80 Z25.0 M09
N7 G28 Z25.0 M05
N8 M01

```

*Metric mode**First motion block with spindle speed**Tool length offset with clearance above**Hole #1 machined at current location**Macro call with assignments (U V)**Retract above work**Return to machine zero**End of current tool***O8102 (ANGULAR HOLE PATTERN MACRO - VERSION 2)**

```

#11 = #11-1
#24 = #21/#11
#25 = #22/#11
G91 X#24 Y#25 L#11
M99
%

```

*Change number of holes to number of spaces**Calculation of the X-increment**Calculation of the Y-increment**Increment L-times (K-times for Fanuc 16/18/21)**End of macro*

In typical machining, the drilling tool would be preceded by a spot drilling or center drilling operation, but the definition of the macro and its call would remain the same.

These two macros have demonstrated that a job that could take a while to program manually, can be done literally in seconds when using an existing macro. The following examples will add some extra features, which can also be added to the first two macros.

Frame Hole Pattern

Frame hole pattern is quite common in many machine shops, and consists of a series of equally spaced holes, forming a rectangular pattern. In effect, this pattern consists of four sets of holes in an angular arrangement, so the first macro (O8101) can be used - four times. However, a frame pattern is more efficient and - if developed correctly - prevents double cutting of the corner holes, which can easily happen using other methods. The macro to develop is a macro that defines such a pattern of holes, starting at the lower left hole of the rectangle, then continuing around the frame in the CW or CCW direction. Again, certain decisions, conditions and restrictions have to be imposed first, based on the type of work.

Figure 35 represents a typical drawing for a frame hole pattern.

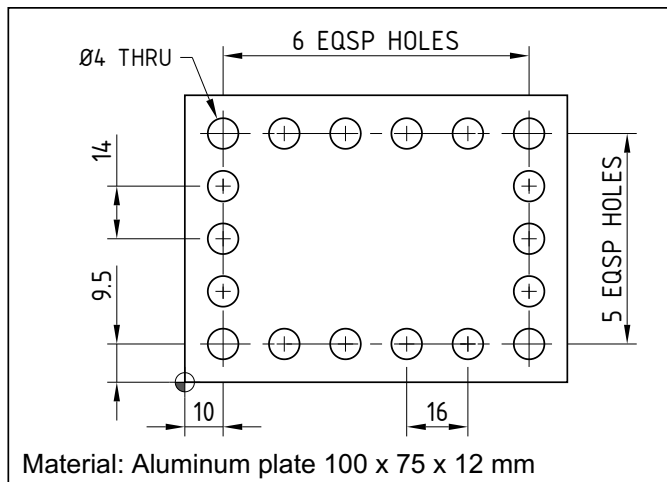


Figure 35

Frame hole pattern - definitions

Based on the typical pattern shown, study carefully what features have been considered. Note that the equal spacing between holes is (or could be) *different* for the X-axis and the Y-axis. This reality has to be taken into consideration. Based on the example drawing, as well as on the result of the necessary thinking process, the following features have been employed in the macro presented here:

- | | |
|---|--|
| <input type="checkbox"/> All holes are spaced equally within the pattern | <i>pitch in X can be different from the pitch in Y</i> |
| <input type="checkbox"/> Any number of holes is acceptable | <i>machine permitting - 2 min per row or column</i> |
| <input type="checkbox"/> Distance between holes must be known | <i>pitch along X and Y (both positive)</i> |
| <input type="checkbox"/> Any pitch between holes is acceptable | <i>within machine capabilities</i> |
| <input type="checkbox"/> Location of the first hole must be known | <i>XY coordinates</i> |
| <input type="checkbox"/> First hole is the lower left corner of the pattern | <i>must be known</i> |
| <input type="checkbox"/> Machining direction is CCW | <i>X+ Y+ X- Y-</i> |

In the macro, the key element will be to prevent cutting any corner hole twice. That can be achieved by programming L0 or K0 in the fixed cycle called. From these selected conditions, the assignments in the G65 macro call block can now be defined.

Figure 36 shows the visual definition of all required variables.

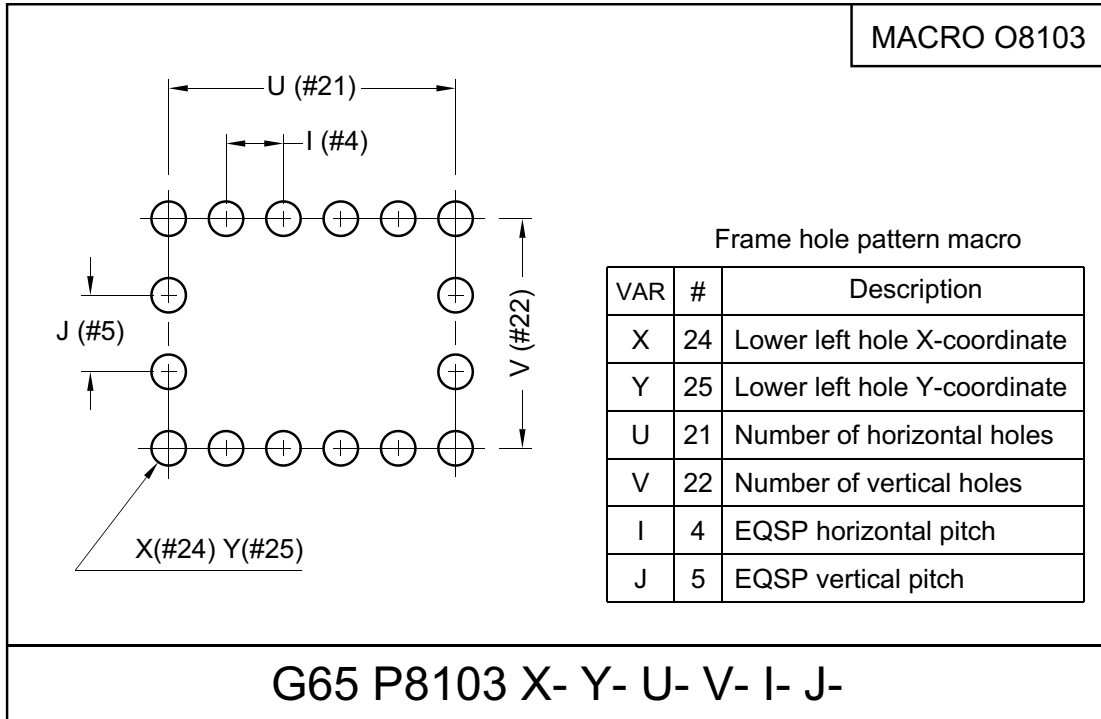


Figure 36

Variable data for frame pattern of holes - Macro O8103

In this example, the absolute location of the X and Y axes have been added, making the user input a little bit longer, but the resulting macro is much more flexible. Understanding how this initial location is used in the macro (any macro), presents an opportunity to upgrade the previous macro examples for the angular hole pattern (if it provides benefits). The direction of machining around the pattern periphery is strictly arbitrary and has no real impact on actual machining. At this point, the variables can be assigned - again, using some logical and convenient method.

Variable Data for Frame Hole Pattern

The following pattern features will be defined:

- Lower left hole absolute X-location ... assigned letter X (variable #24)
- Lower left hole absolute Y-location ... assigned letter Y (variable #25)
- Number of holes - along the X-axis ... assigned letter U (variable #21)
- Number of holes - along the Y-axis ... assigned letter V (variable #22)
- Spacing between holes (X-pitch) ... assigned letter I (variable #4)
- Spacing between holes (Y-pitch) ... assigned letter J (variable #5)

The implementation of the suitable fixed cycle must be called in the main program, with the necessary data (X and Y may be omitted), but it must be programmed with the L0 or K0 mode (depending on the control system). The programming is based on the program zero being at the lower left corner, and the top of part.

```

O0023 (MAIN PROGRAM)
N1 G21                                     Metric mode
N2 G90 G00 G54 X0 Y0 S800 M03             Any X and/or Y motion may be included
N3 G43 Z25.0 H01 M08                       Tool length offset + clearance above
N4 G99 G81 R2.5 Z-14.7 F150.0 L0           No machining but cycle data memorized
N5 G65 P8103 X10.0 Y9.5 U6 V5 I16.0 J14.0 Macro call with assignments
N6 G80 Z25.0 M09                           G90 omitted on purpose (see macro)
N7 G28 Z25.0 M05                           Return to machine zero
N8 M01                                     End of current tool

O8103 (FRAME HOLE PATTERN MACRO)
#10 = #4003                                Store current setting of G90 or G91
IF[#4 LE 0] GOTO9101                       Alarm if I not defined (spacing between holes in X)
IF[#5 LE 0] GOTO9101                       Alarm if J not defined (spacing between holes in Y)
IF[#21 LT 2] GOTO9102                      Alarm if U less than 2 (minimum of 2 horizontal holes)
IF[#21 NE FUP[#21]] GOTO9103              Alarm if U uses a decimal point (number of horizontal holes)
IF[#22 LT 2] GOTO9102                      Alarm if V less than 2 (minimum of 2 vertical holes)
IF[#22 NE FUP[#22]] GOTO9103              Alarm if V uses a decimal point (number of vertical holes)
G90 X#24 Y#25                              Lower left corner hole = the first hole of the pattern
#33 = #21-1                                Number of spaces horizontally (positive)
WHILE [#33 GT 0] DO1                       Start the loop for positive horizontal holes
G91 X#4                                     Incremental + complete bottom row (to the right in X+)
#33 = #33-1                                Update counter
END1                                       End of loop
#33 = #22-1                                Number of space vertically (positive)
WHILE [#33 GT 0] DO1                       Start the loop for positive vertical holes
Y#5                                        Complete right column (up in Y positive)
#33 = #33-1                                Update counter
END1                                       End of loop
#33 = #21-1                                Number of spaces horizontally (negative)
WHILE [#33 GT 0] DO1                       Start the loop for negative horizontal holes
X-#4                                       Complete bottom row (to the left in X negative)
#33 = #33-1                                Update counter
END1                                       End of loop
#33 = #22-1                                Number of spaces vertically (negative)
WHILE [#33 GT 1] DO1                       Loop for vert. neg. holes - see condition - no first hole!
Y-#5                                       Complete left column (down in Y negative)
#33 = #33-1                                Update counter
END1                                       End of loop
GOTO9999                                    Bypass all alarms if data input in G65 macro call is correct
N9101 #3000 = 101 (HOLE SPACING TOO SMALL) Generates alarm number 101 or 3101
N9102 #3000 = 102 (TWO HOLES MINIMUM REQUIRED) Generates alarm number 102 or 3102
N9103 #3000 = 103 (DECIMAL POINT NOT ALLOWED) Generates alarm number 103 or 3103
N9999 G#10                                  Original setting of G90 or G91 restored
M99                                         End of macro
%
```

Study the G65 macro statement - the input values are from the drawing in Figure 35:

```
N5 G65 P8103 X10.0 Y9.5 U6 V5 I16.0 J14.0
```

There are some variable entries in the macro call statement that use a decimal point, and there are others that do not. These are the two available options. Since the variables U6 and V5 are used for counting of holes only (this type of variable is called the counter in a macro), they can be programmed without a decimal point, using the integer format input. Variables X10.0 Y9.5 I16.0 and J14.0 are all dimensional variables and the decimal point must be programmed, using the real number input. Otherwise, X10 will be interpreted as X0.010, Y95 as Y0.095, I16 as I0.016, and J14 as J0.014 - this should be common knowledge from the basic CNC training.

➤ Reference notes:

An *integer input* means that the decimal point is *not available* and/or *not required*, such as for counting the repeats of certain activity within the macro. *Real number input* means that the variable value defined in the **G65** statement requires a decimal point, or at least can use one, if necessary. In this case, the decimal input is *critical*, for example, to represent a fractional number in a decimal format. Real number are not normally used as counters.

Bolt Hole Circle Pattern

Both previous examples have shown how the logical flow of a relatively simple macro is established. These basic concepts are extremely important, since they will be used many times in one form or another in almost every macro. In effect, they will become the foundation of a successful macro development for more complex applications. This current macro example of a bolt hole circle pattern is quite simple in one way, yet it could be very special in so many other ways and for only one reason - it offers a great amount of flexibility. A similar macro for an arc hole pattern follows. Programming a bolt hole circle pattern using the manual method is not a difficult work, but it could be time consuming and certainly subject to errors. Some programmers use a stand-alone utility, other may develop a spreadsheet program for bolt hole circle. In CNC, using a verified macro for this task is not only convenient, it is also economical in terms of productivity.

Figure 37 shows a typical drawing of a bolt hole pattern. There are six equally spaced holes within 360°, and the first hole is located at a given angle.

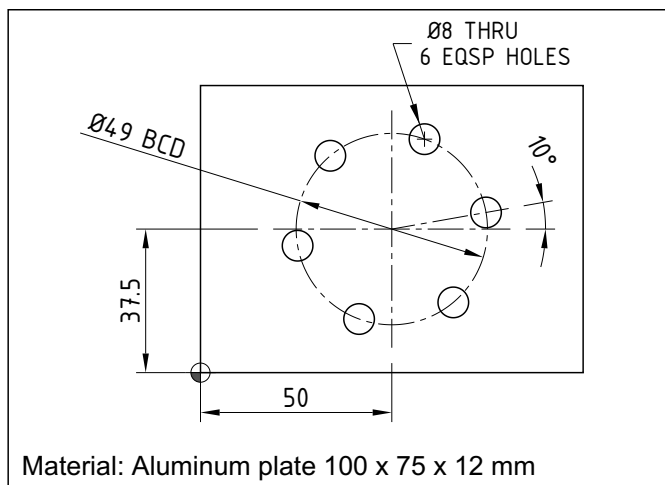


Figure 37

Example of a drawing for a typical bolt hole circle pattern

The bolt hole pattern is very likely the most common example used in various macro tutorials - and there is a very valid reason for it. Just about every CNC programmer, even a beginner, will encounter a bolt hole circle pattern sooner or later, while programming CNC mills and machining centers. Every CNC machinist assigned for milling applications has probably had the opportunity to utilize a bolt hole circle at least once. What makes the bolt hole circle pattern so special? Before reading further, evaluate the sample drawing of a bolt hole circle pattern, illustrated in *Figure 37*.

A true pattern of holes has to include several critical features that are present in every drawing in this pattern group. In case of a bolt hole pattern, there are at least three critical features that are always present but may vary from one drawing to another, depending on the engineering design:

- Location of the bolt hole center** ... *X50.0Y37.5 in the example*
- Bolt Circle Diameter = Pitch Circle Diameter (BCD or PCD)** ... *49 mm in the example*
- Number of equally spaced holes** ... *uses EQSP abbreviation*

Another important feature of the macro design is the *angular position of the first hole*. Most patterns of this kind have the *first* hole aligned at the 'three o'clock position', which is in itself an angular position - *at zero degrees*. A bolt hole pattern macro can be designed in such a way, that the first hole location is always assumed to be at zero degrees. This design makes the macro a bit easier to develop, but also provides rather limited applications. Including the angle of the first hole in the macro call - even if it is at zero degrees - greatly enhances the macro flexibility and its usefulness.

The basic concept of this design should be quite easy to understand - the actual development follows the initial logical thinking. With more features added to the macro, the programming process will inevitably become more involved and more complex and has to be strictly controlled. The most important benefit of such an approach is that the extra effort will produce a very useful macro that can be used for a variety of bolt circle designs. Such a macro can literally save hundreds of programming hours when the right applications are present.

From this detailed description of the approach to macro development, at least two methods of bolt hole circle macro development can be identified:

- A macro that will make a fixed bolt circle pattern only** ... *a simple approach*
- A macro that will add flexibility to the bolt circle pattern** ... *a more advanced approach*

In either case, there are the usual - *and always necessary* - decisions to be made, those that relate to all basic conditions, restrictions and other design requirements. The programmer always makes these decisions *before* writing a single block of the macro. For the bolt hole circle pattern illustrated, the corresponding macro will be based on these conditions, restrictions and decisions:

- Bolt hole diameter (pitch diameter) must always be known, along with its absolute center coordinates**
- All bolt holes are spaced equally within the bolt circle pattern (angular measurement)**
- Any number of holes is acceptable (within machine capabilities) - minimum of two**
- The angular location of the first hole can be anywhere from zero degrees (3 o'clock position)**
- Direction of machining is from the first hole into the CCW direction (arbitrary decision)**
- Macro must be available to any fixed cycle selection**

Thorough initial planning is essential - it may be very difficult, even impossible, to add a feature or two to an existing macro later on.

Additional advanced conditions can be applied to the macro design, if required:

- Any number of holes is acceptable (within machining capabilities) - including the minimum of one
- Initial holes can be skipped, resulting in an arc hole pattern - rather than a full circle hole pattern

➤ Reference note:

Arc pattern is defined as a *portion* of a circle pattern. Typically, it is defined by its first hole and the angular increment between equally spaced holes. To see the difference, compare the arc pattern capabilities of the *bolt hole circle pattern* (this section) and the *arc hole pattern* (next section).

Variable Data for Bolt Hole Circle Pattern

The following bolt hole circle pattern features will be defined - *Figure 38*:

- Diameter of the full bolt hole circle ... assigned letter W (variable #23)
- Absolute X-location of the center ... assigned letter X (variable #24)
- Absolute Y-location of the center ... assigned letter Y (variable #25)
- Number of EQSP holes ... assigned letter H (variable #11)
- Angle of the first hole ... assigned letter A (variable #1)
- Hole number to start with (default=1) ... assigned letter S (variable #19)

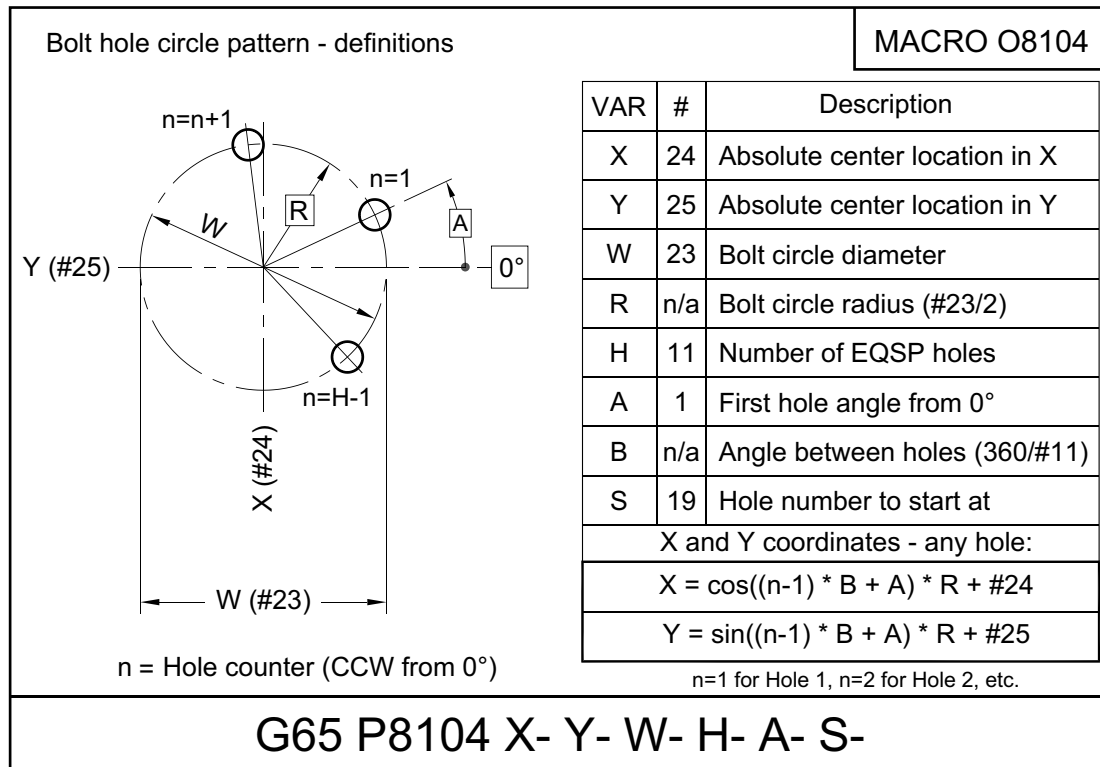


Figure 38

Assignment of variables for a typical bolt hole circle pattern - Macro O8104

Although the concept of this macro may need some effort to understand it thoroughly, a few points may help. First, note the input of the BCD - *the Bolt Circle Diameter* - internally, in the macro, the diameter is irrelevant - it is the *radius* that is needed within the macro for calculations. However, it is the diameter value that is the normal way of dimensioning the drawing, so it should be the *diameter value* that is input into the **G65** variable definitions (variable R in the above figure is actually not used - instead, the variable W is redefined, to save memory resources of the control system. Another variable that is necessary for calculations (but not defined as an assignment) is identified as B in the illustration only, and nested internally in the macro.

For final reference, finished top of the part is program zero for the Z-axis (Z0). X0Y0 is located at the lower left corner (but could be anywhere). The following main program reflects the bolt circle drawing illustrated in *Figure 37*:

```

O0024 (MAIN PROGRAM)
N1 G21                               Metric mode
N2 G90 G00 G54 X0 Y0 S1200 M03       First motion block + spindle speed
N3 G43 Z10.0 H01 M08                 Tool length offset + clearance above
N4 G99 G82 R1.0 Z-15.9 P300 F225.0 L0 Fixed cycle call data - no machining (or K0)
N5 G65 P8104 X50.0 Y37.5 W49.0 H6 A1.0 S1 Macro call with assignments - full circle
N6 G80 Z10.0 M09                     Retract above work
N7 G28 Z10.0 M05                     Return to machine zero
N8 M01                               End of current tool
...
%

O8104 (BOLT HOLE CIRCLE MACRO)       Macro number and description
#10 = #4003                           Store current setting of G90 or G91
IF[#23 LE 0] GOTO9101                 Bolt circle diameter to be greater than zero
IF[#11 NE FUP[#11]] GOTO9102          No fractions allowed for number of holes
IF[#11 LE 0] GOTO9103                 Minimum number of holes is one
IF[#19 EQ #0] THEN #19 = 1            Start hole number = 1 (one) if not specified
IF[#19 NE FUP[#19]] GOTO9102          No fractions allowed for start hole number
IF[#19 LT 1] GOTO9104                 Start hole number must be one or higher
IF[#19 GT #11] GOTO9105               Start hole number must be less than all holes
#23 = #23/2                           Change diameter of bolt circle to radius
WHILE[#19 LE #11] DO1                 Start loop for holes
#30 = [#19-1]*360/#11+#1              Calculate current hole angle
X[cos[#30]*#23+#24] Y[sin[#30]*#23+#25] Calculate current X and Y hole location
#19 = #19+1                            Update counter for the loop
END1                                   End of loop
GOTO9999                               Bypass alarm messages
N9101 #3000=101 (DIA MUST BE GT 0)    Alarm number 101 or 3101
N9102 #3000=102 (HOLES DATA MUST BE INTEGER) Alarm number 102 or 3102
N9103 #3000=103 (ONLY POSITIVE NUM OF HOLES) Alarm number 103 or 3103
N9104 #3000=104 (START HOLE MUST BE INTEGER) Alarm number 104 or 3104
N9105 #3000=105 (START HOLE NUMBER TOO HIGH) Alarm number 105 or 3105
N9999 G#10                             Restore modal G-code
M99                                    End of macro
%
```

One of the more interesting features of the macro is the use of a *default value* (6th macro block):

```
IF[#19 EQ #0] THEN #19 = 1           Start hole number = 1 (one) if not specified
```

In the **G65** variable assignment is the variable S, assigned value of 1, i.e., **S1**. This variable controls the hole number at which the macro starts. For a full circle hole pattern, the variable will always be 1, however, for an arc pattern only, it will be *greater than 1*. Since bolt circle patterns are more common than arc patterns, the macro provides a default - if the assignment of variable S is *not specified* in the **G65** macro call, the value of *one* will be automatically supplied by the above statement within the macro. Note that not all controls accept the *IF-THEN* argument, in which case the more cumbersome *IF* and *GOTOn* combination will have to be used.

To summarize this default feature, both following **G65** statements will achieve a full bolt circle pattern, starting at the first hole:

```
N5 G65 P8104 X50.0 Y37.5 W49.0 H6 A1.0 S1   Macro call with assignments - full circle
```

```
N5 G65 P8104 X50.0 Y37.5 W49.0 H6 A1.0     Macro call with assignments - full circle
```

The macro itself can be used in a very flexible way. With an innovative and creative use of the H, A, and S variables, any hole can be the first hole, any number of holes can be specified, providing they are located on a bolt circle. A bolt circle with an even number of equally spaced holes offers more flexibility than a bolt circle with an odd number of equally spaced holes.

Arc Hole Pattern

Certain hole machining applications do not require a full bolt hole circle pattern, just a part of it. This *arc hole pattern* of equally spaced holes along an arc is very similar to the bolt hole circle pattern, but does not cover the full 360° circle. The angular increment between holes is provided in the drawing and must always be part of the macro call assignments.

Figure 39 illustrates a typical arc hole pattern:

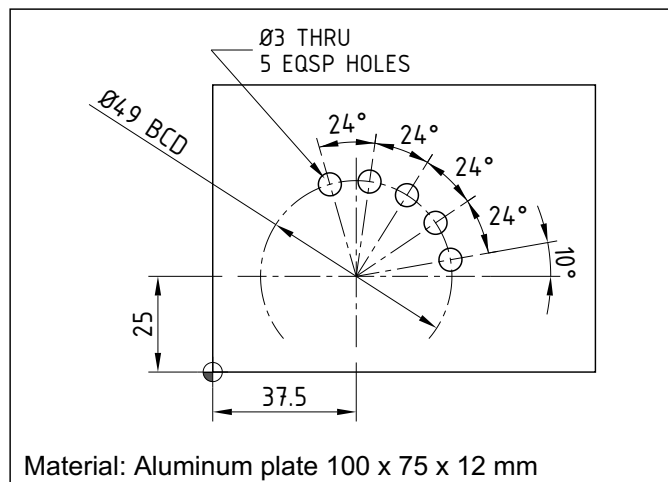


Figure 39

Example of a drawing for a typical arc hole pattern

In comparison, the bolt hole macro in the previous section can skip certain holes, based only on the 'hole number to start with' (assignment $S=\#19$). It will always skip holes beginning at the *first* hole of the pattern. While useful for many practical machining applications, the macro structure does not allow increased flexibility in controlling the actual hole pattern. The *arc hole pattern* macro has been designed specifically for machining patterns of holes with the given angular increment that forms a pattern of holes in the range that is less than 360 (the variable assignment S is not required in this case).

Variable Data for Arc Hole Pattern

The following arc pattern features will be defined:

- Diameter of the partial bolt hole circle ... assigned letter W (variable #23)
- Absolute X-location of the center ... assigned letter X (variable #24)
- Absolute Y-location of the center ... assigned letter Y (variable #25)
- Number of EQSP holes ... assigned letter H (variable #11)
- Angle of the first hole ... assigned letter A (variable #1)
- Increment angle between holes ... assigned letter I (variable #4)

SUGGESTION: Study both macros O8104 and O8105 to the last detail

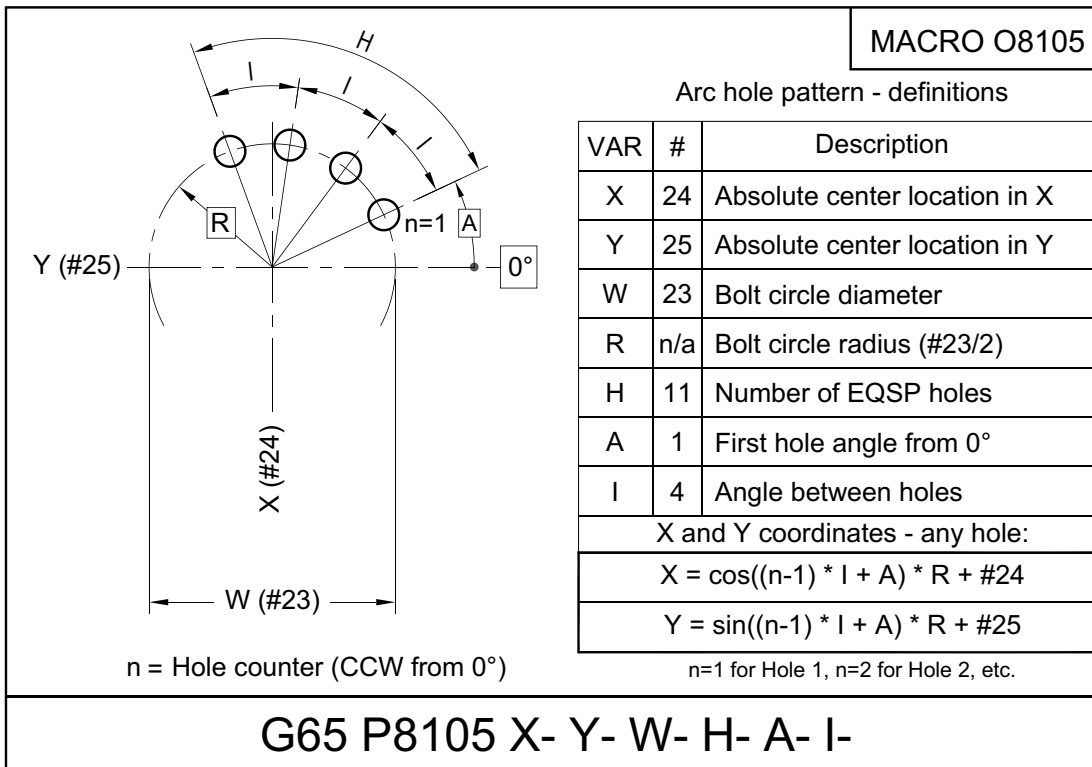


Figure 40

Assignment of variables for a typical arc hole pattern - Macro O8105


```

O0025 (MAIN PROGRAM)
N1 G21 Metric mode
N2 G90 G00 G54 X0 Y0 S1200 M03 First motion block + spindle speed
N3 G43 Z10.0 H01 M08 Tool length offset + clearance above
N4 G99 G82 R1.0 Z-14.4 P300 F225.0 L0 Fixed cycle call data - no machining (or K0)
N5 G65 P8105 X37.5 Y25.0 W49.0 H5 A10.0 I24.0 Macro call with assignments - ARC
N6 G80 Z10.0 M09 Retract above work
N7 G28 Z10.0 M05 Return to machine zero
N8 M01 End of current tool
...

O8105 (ARC HOLE PATTERN MACRO) Macro number and description
#10 = #4003 Store current setting of G90 or G91
IF[#24 EQ #0] THEN #24 = 0 If X-assignment is missing, X=0.0
IF[#25 EQ #0] THEN #25 = 0 If Y-assignment is missing, Y=0.0
IF[#23 EQ #0] GOTO9101 ERROR if DIAMETER is missing
IF[#11 EQ #0] GOTO9102 ERROR if NUMBER OF HOLES is missing
IF[#1 EQ #0] GOTO9103 ERROR if HOLE 1 ANGLE is missing
IF[#4 EQ #0] GOTO9104 ERROR if ANGLE INCREMENT is missing
IF[#23 LE 0] GOTO9105 Arc pattern diameter to be greater than zero
IF[#11 NE FUP[#11]] GOTO9106 No fractions allowed for number of holes
IF[#11 LE 0] GOTO9107 Minimum number of holes is one
#23 = #23/2 Change diameter of the arc pattern to radius
#19 = 1 Start counter of holes
WHILE[#19 LE #11] DO1 Start loop for holes
#30 = [#19-1]*#4+#1 Calculate current hole angle
X[ $\cos$ [#30]*#23+#24] Y[ $\sin$ [#30]*#23+#25] Calculate current X and Y hole location
#19 = #19+1 Update counter for the loop
END1 End of loop
GOTO9999 Bypass alarm messages
N9101 #3000=101 (NO DIAMETER) BCD not specified
N9102 #3000=102 (NO NUMBER OF HOLES) Number of holes not specified
N9103 #3000=103 (NO HOLE 1 ANGLE) First hole angle not specified
N9104 #3000=104 (NO ANGLE INCREMENT) Increment angle not specified
N9105 #3000=105 (DIA MUST BE GT 0) BCD must be greater than zero
N9106 #3000=106 (INTEGER INPUT REQUIRED) Number of holes cannot have decimal point
N9107 #3000=107 (ONLY POSITIVE NUMBER) Number of holes can be one or greater
N9999 G#10 Restore modal G-code
M99 End of macro
%
```

There is a new feature in the macro - a check against 'no input'. If one or more variables are missing in the macro call assignment, the macro cannot be executed correctly. In some cases, the macro will be executed, but with undesirable results. To prevent possibly dangerous situations, the presence of each assignment will be checked. In macros, any variable not defined is an *empty* - or null - variable (described earlier). A null variable is defined as #0. The checking method for a variable assignment shown here can be added to any macro. The above macro also addresses the issue of the arc pattern center being at X0Y0. If the X and/or Y assignment is not provided, it will be defined as X0 and/or Y0 through the macro.

Circular Pocket Roughing

A machining macro that can be used specifically for the removal of material from an internal circular area (circular pocket) can have many variations. The objective of this example is to cover the roughing operations, whereby the pocket final diameter is defined as a rough diameter. The drawing in *Figure 41* shows a sample pocket used for the macro call example.

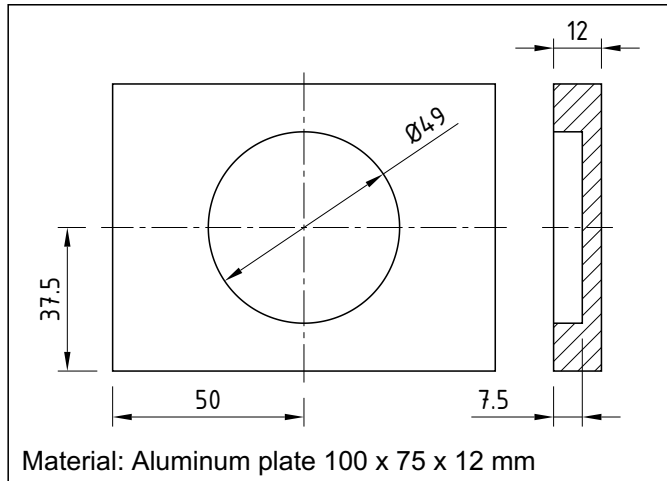


Figure 41

Example of a drawing for a typical circular pocket roughing

As in all previous examples, certain decisions, conditions and restrictions have to be imposed:

- | | |
|--|--|
| <input type="checkbox"/> The diameter of the pocket must be known | <i>required rough diameter specified in G65</i> |
| <input type="checkbox"/> Only a single Z-depth can be applied | <i>no steps at the bottom</i> |
| <input type="checkbox"/> The location of the pocket center must be known | <i>XY coordinates</i> |
| <input type="checkbox"/> Width of each cut must be selected | <i>so called stepover amount</i> |
| <input type="checkbox"/> Direction of machining | <i>center into the X+, then the full circle</i> |
| <input type="checkbox"/> Tool radius offset <i>number</i> must be entered in G65 | <i>not a radius value - only the offset number</i> |
| <input type="checkbox"/> Cutting feedrate must be entered in G65 statement | <i>assignment F (#9)</i> |

In order to accommodate more flexibility within the macro, additional initial settings will be required. The most important improvement to the macro will be the addition of *segmented depth of cut*. This addition will bring an extra power to the macro by allowing to cut a deep pocket in calculated depth segments, rather than at full depth. However, the macro will not require specification of the depth of each cut (variable assignment), in which case the full depth will be reached by the cutting tool at the macro beginning.

Another improvement to the macro is an added feature that allows the macro to be used for a drawing that uses metric dimensions, as well as for a drawing designed in English units (inches). This feature will only affect the clearance above the part, which is *not* specified in the macro assignment block G65. Other drawing dimensions have to be entered in their respective units within the G65 macro block call. Of course, any other macro can use the same technique and even enlarge on it. Other techniques, already presented, may be used as well. The programming is based on the program zero being at the lower left corner and the finished top of part.

After these considerations, the variables that will have to be defined can now be assigned.

Variable Data for Circular Pocket Roughing

Circular pocket roughing includes many settings that change from job to job. Good planning is important and when completed, the following features will be defined to develop a macro toolpath for a circular pocket roughing:

- Pocket center as absolute X-location ... assigned letter X (variable #24)
- Pocket center as absolute Y-location ... assigned letter Y (variable #25)
- Pocket final depth ... assigned letter Z (variable #26)
- Pocket final diameter ... assigned letter D (variable #7)
- Depth of each cut ... assigned letter K (variable #6)
- Width of each cut ... assigned letter W (variable #23)
- Tool offset number ... assigned letter T (variable #20)
- Cutting feedrate ... assigned letter F (variable #9)

If the K (#6) is omitted, macro will cut to the full depth, as specified in assignment Z (#26). This macro is also a good example of using two loops simultaneously - a loop within a loop.

The Figure 42 shows a graphical representation of the variables used for circular pocket roughing macro. Note that the common variable #120 is defined as a *calculated* value within the macro, based on the tool radius stored in the offset number identified by variable T (#20).

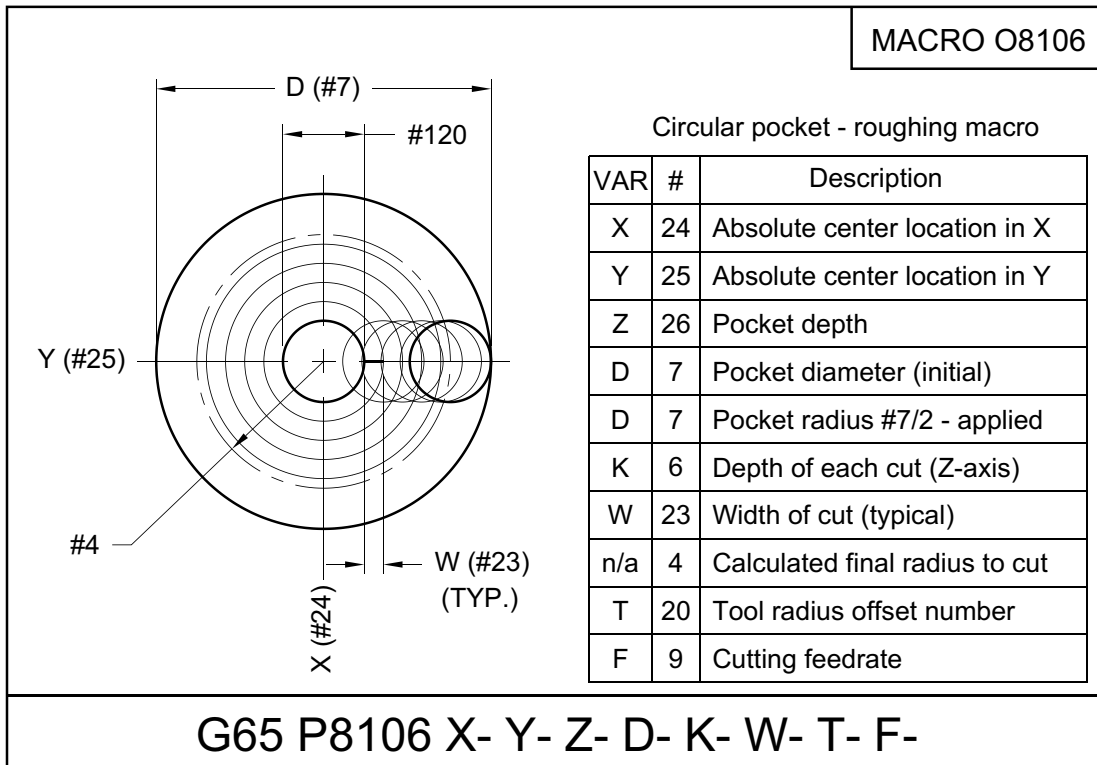


Figure 42

Assignment of variables for a typical circle pocket roughing - Macro O8106

```

O0026 (MAIN PROGRAM)
N1 G21 Metric mode
N2 G90 G00 G54 X0 Y0 S1200 M03 Any first motion location with spindle speed
N3 G43 Z25.0 H05 M08 Tool length offset + initial clearance above part
N4 G65 P8106 X50.0 Y3.75 Z7.5 D49.0 K2.5 W4.0 T5 F500.0 Macro call
N5 G80 Z25.0 M09 Retract to start position
N6 G28 Z25.0 M05 Return to machine zero
N7 M01 End of current tool
...
%

O8106 (ROUGHING CIRCULAR POCKET MACRO) Macro number and description
IF[#7 EQ #0] GOTO9101 Pocket diameter must be defined
IF[#20 EQ #0] GOTO9102 Tool offset number must be defined
IF[#23 EQ #0] GOTO9103 Width of cut must be defined
IF[#26 EQ #0] GOTO9104 Pocket depth must be defined
IF[#9 EQ #0] GOTO9105 Cutting feedrate must be defined
#10 = #4003 Store current setting of G90 or G91
#7 = ABS[#7/2] Change defined pocket dia. to pos. pocket radius
#120 = [ABS[#2400+#20]+#[2600+#20]] Retrieve stored abs. value of selected tool offset
IF[#120 GE #7] GOTO9106 Offset value cannot be GE the pocket radius
#26 = ABS[#26] Guarantee that the Z-depth is a positive value
#126 = #4006 Check current units (English G20 or Metric G21)
IF[#126 EQ 20.0] THEN #126 = 0.1 Clearance above work is 0.1 inch for G20
IF[#126 EQ 21.0] THEN #126 = 2.0 Clearance above work is 2 mm for G21
G90 G00 X#24 Y#25 Rapid to start position X and Y
Z#126 Rapid to start position Z (above pocket center)
G01 Z0 F[#9/2] Feed to Z0 absolute start position
#16 = #26 Store total depth in a separate register
IF[#6 EQ #0] GOTO1 Branch to default if depth of cut not defined
#6 = ABS[#6] Guarantee that the Z-depth is a positive value
GOTO2 Bypass full depth setting
N0001 #6 = #26 If depth of cut not defined, use full depth
N0002 #4 = #7-#120 Calculate the final cutting radius
WHILE [#16 GE #6] DO1 This loop controls the depth of cut
#33 = #23 Initial width of cut (= first circle radius)
G91 G01 Z-#6 F[#9/2] Feed to current depth at one half of the feedrate
WHILE [#33 LT #4] DO2 Cutting radius must be smaller than final radius
G90 G01 X[#24+#33] F#9 Stepover to the next cutting radius
G03 I-#33 Cutting the current circle radius
#33 = #33+#23 Increase cutting radius by the width of cut
END2 End of loop 2
G90 G01 X[#24+#4] F#9 Final cutting radius approach - current depth
G03 I-#4 Cutting last radius (final dia.) - current depth
G01 X#24 F[#9*3] Fast feed back to X-start location - current depth
#16 = #16-#6 New depth of cut
END1 End of loop 1
IF[#16 LE 0] GOTO9000 Branch to retract if pocket depth is 0 or negative
G91 G01 Z-#16 F[#9/2] Feed to current depth at one half of the feedrate

```

#33 = #23	<i>Initial width of cut (= first circle rad.) last depth</i>
WHILE[#33 LT #4] DO1	<i>Cutting radius must be smaller than final radius</i>
G90 G01 X[#24+#33] F#9	<i>Stepover to the next cutting radius</i>
G03 I-#33	<i>Cutting the current circle radius</i>
#33 = #33+#23	<i>Increase cutting radius by the width of cut</i>
END1	<i>End of loop 1</i>
G90 G01 X[#24+#4] F#9	<i>Final cutting radius approach - last depth</i>
G03 I-#4	<i>Cutting last radius (final dia.) - last depth</i>
G01 X#24 F[#9*3]	<i>Fast feed back to X-start location - last depth</i>
N9000 G00 Z#126	<i>Retract from the center above the pocket</i>
GOTO9999	<i>Bypass alarm messages</i>
N9101 #3000=101 (NO POCKET DIAMETER)	<i>Alarm number 101 or 3101</i>
N9102 #3000=102 (NO T-OFFSET)	<i>Alarm number 102 or 3102</i>
N9103 #3000=103 (NO CUTTING WIDTH)	<i>Alarm number 103 or 3103</i>
N9104 #3000=104 (NO POCKET DEPTH)	<i>Alarm number 104 or 3104</i>
N9105 #3000=105 (NO FEEDRATE)	<i>Alarm number 105 or 3105</i>
N9106 #3000=106 (TOOL RADIUS TOO LARGE)	<i>Alarm number 106 or 3106</i>
N9999 G#10	<i>Restore modal G-code</i>
M99	<i>End of macro</i>
%	

Center-cutting end mill or a similar tool must be used to match the program requirements, unless the center of the circular pocket is open (rather than solid). Also note the modification of the feedrate within the macro program, by either decreasing it or increasing it by a given factor. This approach can be applicable to other macro features, not just to the feedrate.

Amount of Stock Left

Nowhere in the macro is specified the amount of stock to be left for later finishing. This amount should be left to the CNC operator, for greater control at the machine. How is the stock handled? This is the first macro that uses a system variable - two system variables, in fact. It is important to understand that system variables depend on the control system - and its tool offset memory type, as in this example. Controls with *Memory Type C* and less than 200 offsets use the 2401-series system variables for the *Geometry D*-offset, and 2601-series for the *Wear D*-offset (*see Chapter 11 for details*). In the example, assignment $T=\#20=5$, therefore offset 5 will store the geometry and the wear settings for the cutter radius. For example, if the tool is 12 mm end mill, the stored geometry offset will be set to the cutter radius of 6 mm and the wear offset will store the amount left for finishing, for example, 0.5 mm. The variable #120 controls only the last circular cut, which makes the final pocket size. The control system will interpret the block

#120 = [ABS[# [2400+#20]+# [2600+#20]]] ... as ...

#120 = [ABS[# [2400+5]+# [2600+5]]] = [#2405+#2605] = [6.0+0.5] = 6.5

Since the actual cutter radius is 6 mm but the wear offset setting is additional 0.5 mm, the completed pocket diameter will be 1 mm smaller and should measure 48 mm. If the wear offset 5 at the control is set to zero, the pocket will be finished to size, as per drawing.

Circular Pocket Finishing

Finishing cut applied to a circular pocket (or a similar machining operation) is a follow up to the previous example and is the main subject of this macro. The macro can be used as it is on its own, or as a macro following the circular pocket roughing macro, described in the previous section. The example drawing in *Figure 43* is similar to the last example.

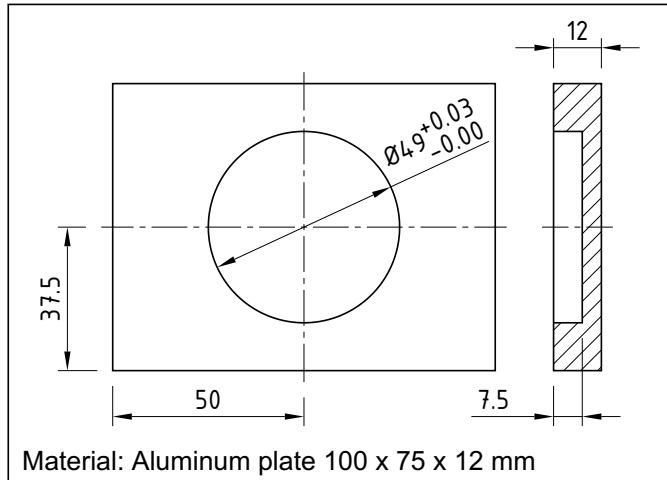


Figure 43

Example of a drawing for a typical circular pocket finishing

A typical internal circular toolpath is commonly programmed from the center of the pocket, through the roughing motions and ending again at the center. Here is a typical machining process:

1. Tool moves to the center of the pocket, above the work
2. Tool feeds-in to the Z-depth
3. Cutter radius offset is applied along the linear XY motion from center
4. Tool moves along the lead-in arc towards the pocket diameter
5. The full circle is machined in one block
6. Tool moves along the lead-out arc away from the pocket diameter
7. Cutter radius offset is canceled in the linear XY motion towards center
8. Tool retracts above the work

This macro uses an automatically calculated *lead-in* and *lead-out* tangential arc (blend radius). The same tangential toolpath can also be adapted to other finishing applications, particularly for the purpose of achieving a better surface finish.

NOTE - Several control systems support a G-code for milling a circular pocket, typical identified as **G12** (CW) and **G13** (CCW). Fanuc controls do not have this feature built-in for direct use, but Fanuc provides tools to develop the G-code using a macro. The complete development of this new command is described in the next chapter (*Chapter 21 - Custom Cycles*).

Variable Data for Circular Pocket Finishing

The following circular pocket finishing macro features will be defined:

- Circle pocket diameter ... assigned letter *W* (variable #23)
- Absolute X-location of the center ... assigned letter *X* (variable #24)
- Absolute Y-location of the center ... assigned letter *Y* (variable #25)
- Z-depth of the pocket ... assigned letter *Z* (variable #26)
- Tool offset number for radius ... assigned letter *T* (variable #20)
- Cutting feedrate for profile ... assigned letter *F* (variable #9)

The illustration in *Figure 44*, shows the pictorial representation of the variable assignments.

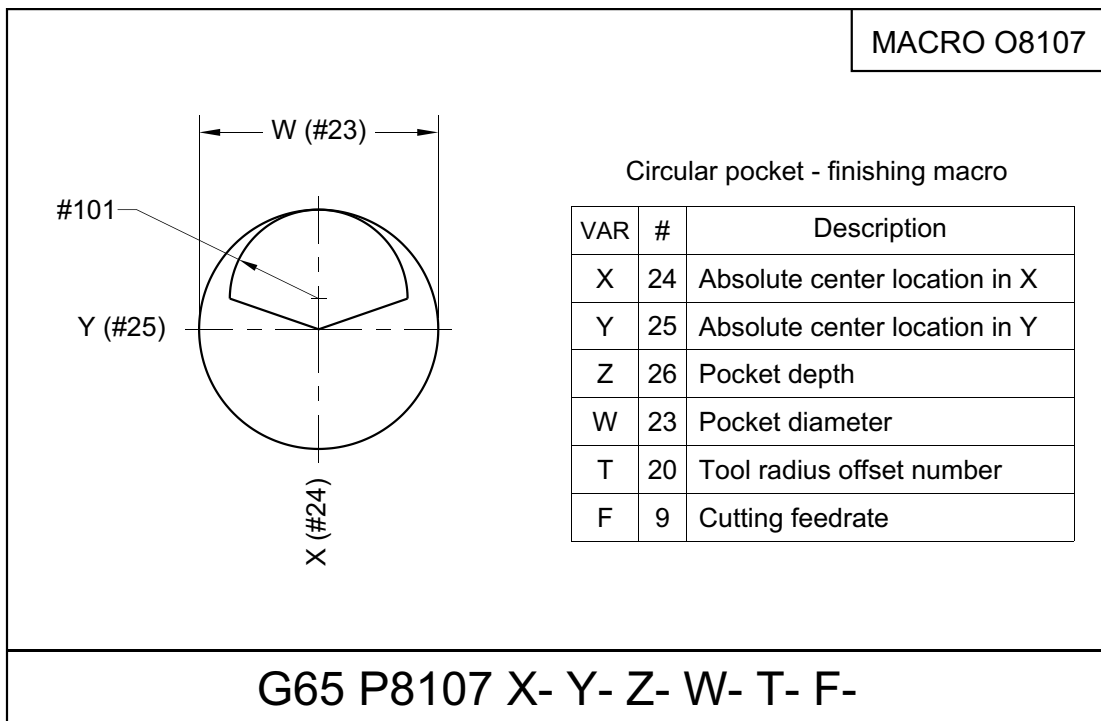


Figure 44

Assignment of variables for a typical circle pocket finishing - Macro O8107

All variables used in the macro has been carefully selected, based on machining procedures and several other factors. The selected machining procedure for the pocket finishing is applied within the macro body. As in any other macro, there are some conditions, restrictions and other requirements, this time applied to the circular pocket finishing toolpath:

- Circular pocket can have any positive diameter
- The center of the pocket must be known (including X0Y0)
- Z-depth may be positive or negative but not a zero
- The location of the contour start is arbitrarily set at 90° (12 o'clock)

- Z-clearance above the work is 2 mm or 0.1 inches (automatically selected)
- Feedrate for Z-axis infeed is one half of the programmed feedrate
- Direction of machining is from the center in climb milling mode (G41 D..) at M03
- The tool offset numbers must be within the range of 1-33
- Tool offsets used are for *Memory Offset Type C* - less than 200 offsets - as per Fanuc control designation
- Tool diameter must be greater than 0 and less than the pocket diameter
- LEAD-IN* and *LEAD-OUT* arcs are identical and calculated by the macro as:

$$(\text{POCKET DIA} - \text{TOOL DIA}) / 2$$

Additional conditions can be applied only for more advanced approach:

- If cleanup of the bottom is required during finishing, the tool diameter must be *POCKET DIA / 3 or greater*
- Stepped Z-depth may be added to the macro, if desired

O0027 (MAIN PROGRAM)

N1 G21	<i>Metric mode</i>
N2 G90 G00 G54 X0 Y0 S800 M03	<i>Any first motion location with spindle speed</i>
N3 G43 Z25.0 H04 M08	<i>Tool length offset with clearance above part</i>
N4 G65 P8107 X50.0 Y37.5 Z7.5 W49.0 T4 F150.0	<i>Macro call with assignments</i>
N5 Z25.0 M09	<i>Retract above work</i>
N6 G28 Z25.0 M05	<i>Return to machine zero</i>
N7 M01	<i>End of current tool</i>
%	

O8107 (CIRCULAR POCKET FINISHING MACRO)

IF[#26 EQ 0] GOTO9101	<i>Depth of pocket must not be a zero</i>
IF[#23 LE 0] GOTO9102	<i>Pocket diameter must be a positive value</i>
IF[#20 LE 0] GOTO9103	<i>Offset number required for cutter radius offset</i>
IF[#20 GT 33] GOTO9104	<i>Maximum number of offsets is 33</i>
IF[#9 EQ #0] GOTO9105	<i>Cutting feedrate must be defined</i>
#120 = #[2400+#20]+#[2600+#20]	<i>Retrieve the stored value of selected tool offset</i>
IF[#120 LE 0] GOTO9106	<i>Offset value radius must be positive</i>
#23 = #23/2	<i>Change diameter of pocket to radius</i>
IF[#23 LE #120] GOTO9107	<i>Pocket radius must be larger than offset radius</i>
#101 = [#23+#120]/2	<i>Calculate Lead-in/Lead-out arc radius</i>
#10 = #4003	<i>Store current setting of G90 or G91</i>
#26 = ABS[#26]	<i>Guarantee that the Z-depth is a positive value</i>
#126 = #4006	<i>Check current units (English G20 or Metric G21)</i>
IF[#126 EQ 20.0] THEN #126 = 0.1	<i>Clearance above work is 0.1 inch for G20</i>
IF[#126 EQ 21.0] THEN #126 = 2.0	<i>Clearance above work is 2 mm for G21</i>
G90 G00 X#24 Y#25	<i>Rapid to start position X and Y</i>
Z#126	<i>Rapid to start position Z (above pocket center)</i>
G01 Z-#26 F[#9/2]	<i>Feed to depth at one half of the feedrate</i>
G91 G41 X#101 Y[#23-#101] D#20 F#9	<i>Motion from center + cutter radius offset</i>
G03 X-#101 Y#101 I-#101	<i>Lead-in arc tool motion</i>
J-#23	<i>Full circle tool motion</i>
X-#101 Y-#101 J-#101	<i>Lead-out arc tool motion</i>
G01 G40 X#101 Y-[#23-#101]	<i>Return to the center + cancel cutter radius offset</i>
G90 G00 Z#126	<i>Retract above the finished pocket</i>


```

GOTO9999
N9101 #3000=101 (Z-DEPTH IS ZERO)
N9102 #3000=102 (POCKET DIA ERROR)
N9103 #3000=103 (OFFSET NUMBER ERROR)
N9104 #3000=104 (OFFSET NUMBER TOO BIG)
N9105 #3000=105 (FEEDRATE NOT DEFINED)
N9106 #3000=106 (OFFSET VALUE IS NOT POSITIVE)
N9107 #3000=107 (TOOL OFFSET TOO LARGE)
N9999 G#10
M99
%
```

Bypass alarm messages
Generates alarm number 101 or 3101
Generates alarm number 102 or 3102
Generates alarm number 103 or 3103
Generates alarm number 104 or 3104
Generates alarm number 105 or 3105
Generates alarm number 106 or 3106
Generates alarm number 107 or 3107
Restore modal G-code
End of macro

In this macro, two system variables are used to defined local variable #120. Compare the current definition of this variable with the similar definition in the previous example (O8106) - there is no **ABS** function used in the current macro. Instead, the variable definition is immediately followed by a conditional **IF** test - if the offset value is negative, the system will issue an error condition (alarm). This is not a better solution, it is shown here only as 'a solution' - the **ABS** function is a much better choice, one that will not require an error test.

In this and other macros included in the handbook, a test against missing feedrate or feedrate being zero can also be made, if desired. Make sure to write either statement correctly - the entries are similar, but will produce a very different result (assignment $F=#9$ is used for the example):

```

IF[#9 EQ #0] GOTO...
IF[#9 EQ 0] GOTO...
```

Cutting feedrate must be defined
Cutting feedrate must NOT be a zero

In Figure 45 are two drawings that can be used for testing the integrity of circle pocket macros.

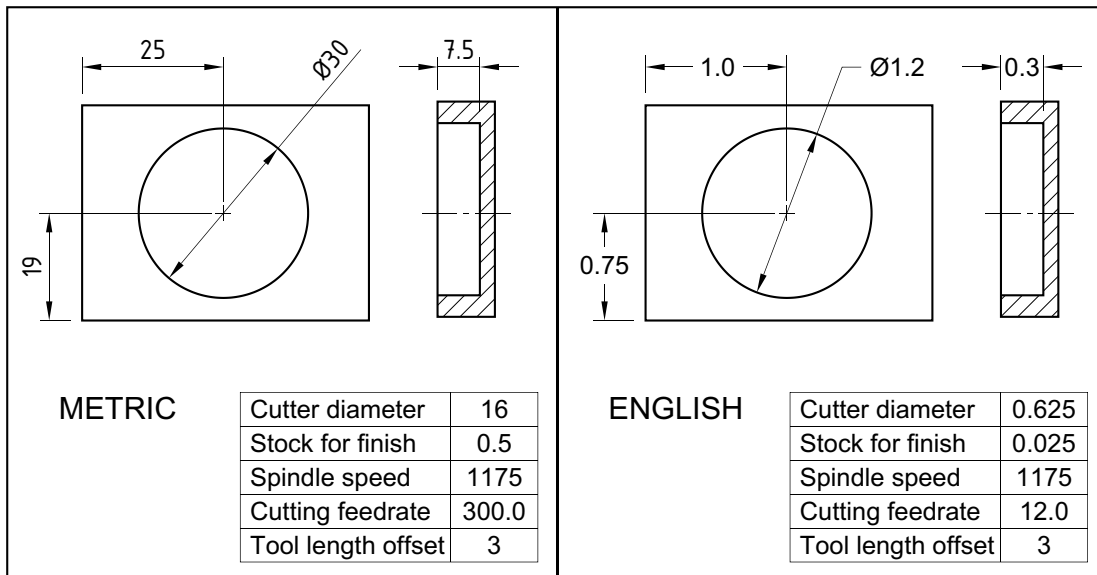


Figure 45

Metric and English drawings that can be used to test the circular pocket macros

Slot Machining Macro

Machining a closed slot is a common milling operation, one that requires a number of similar calculations. The basic shape of a slot is always the same, but its location, length, radius, angle and depth vary (in addition to other non-geometrical values). A closed slot machining is the perfect choice to be developed as a custom macro. *Figure 46* shows a drawing of a typical closed slot.

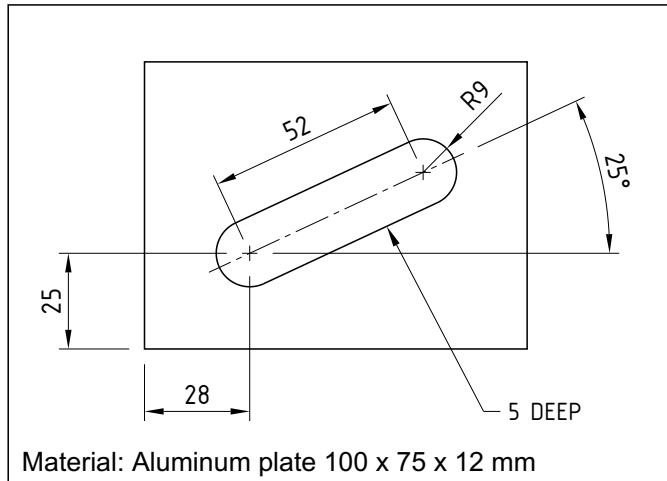


Figure 46

Example of a drawing for a typical slot machining macro

Typical machining process to cut a common closed slot is quite simple:

1. **Rapid to the first arc center and above work**
2. **Feed-in to slot depth**
3. **Feed to the second arc center**
4. **Lead-in towards contour - apply cutter radius offset**
5. **Contour slot**
6. **Lead-out towards arc center - cancel cutter radius offset**
7. **Retract above work**

As in all other macros, there are certain criteria to be established:

- Absolute center location of one end radius must be known
- Depth of the slot must be known
- Slot angular orientation from the defined center location must be known
- The length of slot between centers of the two radii must be known
- The width of the slot must be equal to the double slot radius
- Tool radius offset number must be defined
- Cutting feedrate must be specified
- Lead-in and lead-out arc will be calculated automatically

The *positive* or *negative* cutting direction is calculated by the macro, using the defined angle orientation. This is a more structured approach than forcing a minus sign in the program.

Variable Data for Slot Machining

Once the initial conditions have been established, the macro variables can be assigned.

- Absolute X-location of one radius center ... assigned letter X (variable #24)
- Absolute Y-location of one radius center ... assigned letter Y (variable #25)
- Z-depth of the slot ... assigned letter Z (variable #26)
- Angular orientation of the slot ... assigned letter A (variable #1)
- Distance between slot centers ... assigned letter D (variable #7)
- Slot radius (one of two same radiuses) ... assigned letter R (variable #18)
- Tool radius offset number ... assigned letter T (variable #20)
- Cutting feedrate slot profile ... assigned letter F (variable #9)

Figure 47 shows a graphical representation of the variable data assignments in macro call G65:

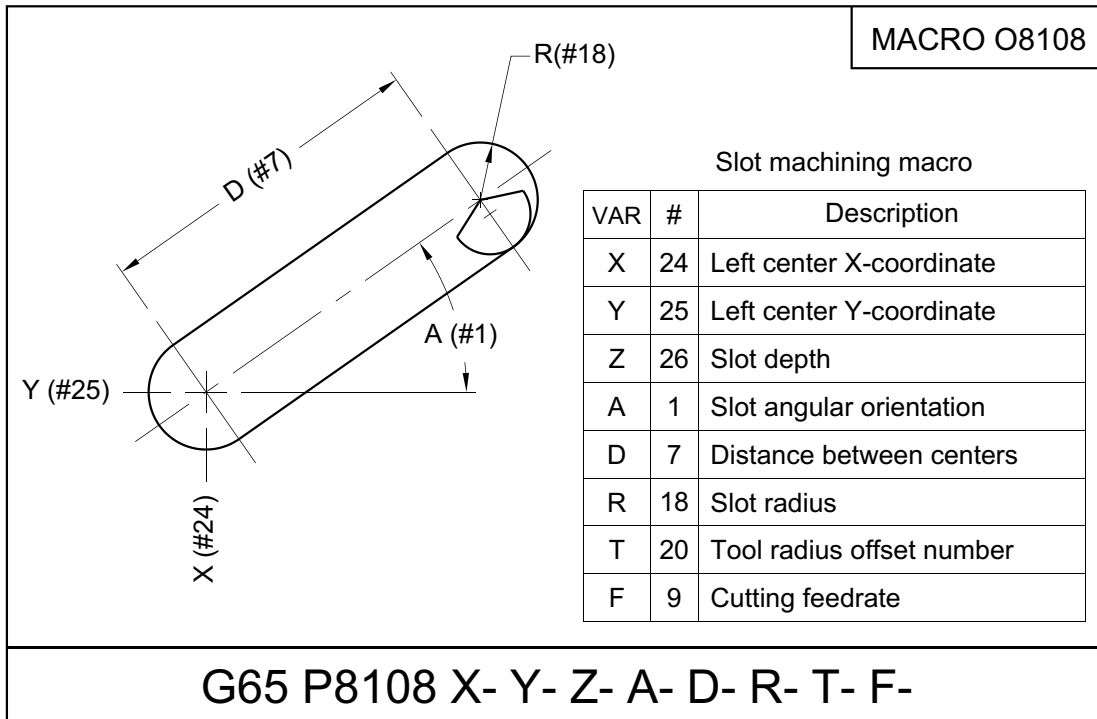


Figure 47

Assignment of variables for a typical closed slot macro - Macro O8108

In the following program example, some of the initial input checking blocks and their corresponding alarm definitions have been omitted to keep the macro clear for studying. However, several earlier examples show how the techniques work and once the macro is verified, the blocks testing proper input should be added. This macro is a very good example of programming various trigonometric functions (angular dimensions). Also note that several common variables are redefined, once their original meaning had been applied.

```

O0028 (MAIN PROGRAM)
N1 G21
N2 G90 G00 G54 X0 Y0 S800 M03
N3 G43 Z25.0 H05 M08
N4 G65 P8108 X28.0 Y25.0 Z5.0 R9.0 D52.0 A25.0 T5 F300.0
N5 G00 Z25.0 M09
N6 G28 Z25.0 M05
N7 M30
%

O8108 (SLOT MACRO)
#10 = #4003
    <... place error definitions here ... >
#126 = #4006
IF[#126 EQ 20.0] THEN #126 = 0.1
IF[#126 EQ 21.0] THEN #126 = 2.0
G90 G00 X#24 Y#25
Z#126
G01 Z-[ABS[#26]] F[#9/2]
#120 = [ABS[#2400+#20]+#[2600+#20]]
#101 = [#18+#120]/2
#102 = #18-#101
#124 = [COS[#1]*#7]
#125 = [SIN[#1]*#7]
G91 X#124 Y#125 F#9
#103 = ATAN[#102]/[#101]
#104 = SQRT[#102*#102+#101*#101]
#124 = [#104*COS[#1+#103+180]]
#125 = [#104*SIN[#1+#103+180]]
G41 X#124 Y#125 D#20
#105 = SQRT[#101*#101*2]
#114 = [COS[#1]*#101]
#115 = [SIN[#1]*#101]
#124 = [COS[#1-45]*#105]
#125 = [SIN[#1-45]*#105]
G03 X#124 Y#125 I#114 J#115
#114 = [COS[#1+90]*#18]
#115 = [SIN[#1+90]*#18]
#124 = [COS[#1+90]*[#18*2]]
#125 = [SIN[#1+90]*[#18*2]]
X#124 Y#125 I#114 J#115
#124 = [COS[#1+180]*#7]
#125 = [SIN[#1+180]*#7]
G01 X#124 Y#125
#114 = [COS[#1-90]*#18]
#115 = [SIN[#1-90]*#18]
#124 = [COS[#1-90]*[#18*2]]
#125 = [SIN[#1-90]*[#18*2]]
G03 X#124 Y#125 I#114 J#115
#124 = [COS[#1]*#7]

```

Macro number and description

Store current setting of G90 or G91

Check current units (English G20 or metric G21)

Clearance above work is 0.1 inch for G20

Clearance above work is 2 mm for G21

Rapid to the first center slot position in XY

Clear above work in Z

Feed to depth at half the feedrate

Retrieve the stored value of selected tool offset

Calculate Lead-in/Lead-out arc radius

Calculate difference between radiuses

Calculate the X-length of the center motion

Calculate the Y-length of the center motion

Motion from the center

Calculate motion angle for G01

Calculate motion length for G01

Calculate the X-motion from center to lead-in arc

Calculate the Y-motion from center to lead-in arc

Make the G01 motion + cutter compensation on

Calculate motion length for G03 (at 45 degrees)

Calculate the I-value of the lead-in arc

Calculate the J-value of the lead-in arc

Calculate the lead-in arc X-increment

Calculate the lead-in arc Y-increment

Lead-in arc motion

Calculate the I-value of the right slot arc

Calculate the J-value of the right slot arc

Calculate the right slot arc X-increment

Calculate the right slot arc Y-increment

Right slot arc of 180°

Calculate the X-length of linear motion 1

Calculate the Y-length of linear motion 1

Linear motion 1

Calculate the I-value of the left slot arc

Calculate the J-value of the left slot arc

Calculate the left slot arc X-increment

Calculate the left slot arc Y-increment

Left slot arc of 180°

Calculate the X-length of linear motion 2

#125 = [SIN[#1]*#7]	<i>Calculate the Y-length of linear motion 2</i>
G01 X#124 Y#125	<i>Linear motion 2</i>
#114 = [COS[#1+90]*#101]	<i>Calculate the I-value of the lead-out arc</i>
#115 = [SIN[#1+90]*#101]	<i>Calculate the J-value of the lead-out arc</i>
#124 = [COS[#1+45]*#105]	<i>Calculate the lead-out arc X-increment</i>
#125 = [SIN[#1+45]*#105]	<i>Calculate the lead-out arc Y-increment</i>
G03 X#124 Y#125 I#114 J#115	<i>Lead-out arc motion</i>
#124 = [SIN[#103-#1-90]*#104]	<i>Calculate X-motion from lead-out arc to center</i>
#125 = [COS[#103-#1-90]*#104]	<i>Calculate Y-motion from lead-out arc to center</i>
G01 G40 X#124 Y#125	<i>Back to start point motion + cutter offset off</i>
G90 G00 Z#126	<i>Retract above the finished slot</i>
GOTO9999	<i>Bypass alarm messages</i>
<... place error messages here ...>	
N9999 G#10	<i>Restore modal G-code</i>
M99	<i>End of macro</i>
%	

One method of a macro design is evident in the slot macro. Note that many variables are used over and over again, with different values. This approach results in fewer variables and makes the macro easier to interpret. There is no need for each value to have its own unique variable.

Circular Groove with Multiple Depth

The macro described in this section is a rather simple one, yet very practical in many machining applications - it also serves as a very good example to study custom macro design. Its purpose is to cut a circular groove, strictly a utility type with no tight tolerances, *but with multiple depth cuts*. This method of machining is very useful for groove roughing in tough materials. Before the macro will be developed, it will be preceded by a subprogram development. Subprograms are often excellent candidates for macros, and this example shows the comparison.

From the drawing information in *Figure 48*, the subprogram - *and macro* - can be developed:

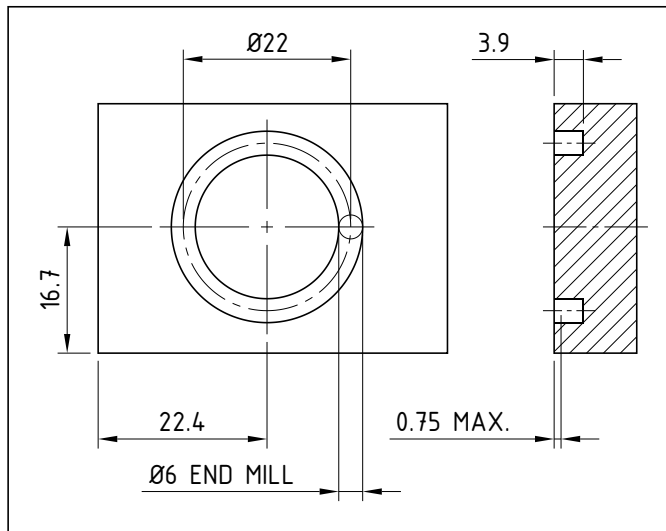


Figure 48

Example of a drawing for a typical circular groove with multiple depth macro

From Subprograms to Macros

Developing a subprogram rather than a macro is perfectly justified for a one-of-a-kind job that will not likely be repeated in the future. Subprograms are also the only 'automation' tools, when the macro option is not available for the existing control system. To program the rough groove illustrated in *Figure 48*, the main program will call a subprogram O8022:

```
(MAIN PROGRAM)
(XOY0 LOWER LEFT CORNER - Z0 TOP OF PART)
N1 G21                               Metric units mode
N2 G90 G00 G54 X22.4 Y16.7 S750 M03  Start XY location at zero degrees
N3 G43 Z5.0 H01 M08                  Clearance above the start point
N4 G01 Z0.6 F100.0                   Subprogram must start at Z0.6
N5 M98 P8022 L6                       Subprogram call - REPEAT six times (or K6)
N6 G90 G00 Z5.0 M09                  Rapid to clearance above the part
N7 G91 G28 Z0 M05                     Machine zero return for Z-axis
N8 G28 X0 Y0                           Machine zero return for XY-axes
N9 M30                                 End of main program
%

O8022 (SUBPROGRAM)
G01 G91 Z-0.75 F100.0                 Feed in the Z-axis by the depth increment
G03 I-11.0                             Cut the full circle
M99                                     Subprogram end
%
```

It looks like a simple subprogram - and it is. Yet, some critical calculations had to be done. The first calculation is the start Z-position - at Z0.6. Reason? The required absolute depth of Z-3.9 must *not* be exceeded. The travel distance (distance-to-go) from the start position to the full depth must be *equally divisible* by the cutting depth, with no leftovers. When the cutting depth is programmed at 0.75 mm, it takes six equal depth cuts to reach the Z-3.9 final depth:

$$\begin{array}{ll} 0.6 + 3.9 = 4.5 & \text{Total travel distance (distance-to-go) = 4.5} \\ 4.5 / 0.75 = 6 & \text{Total number of EQUAL depth cuts = 6} \end{array}$$

When the tool starts at Z0.6 and then moves down incrementally by the amount of Z-0.75, it will feed into the absolute tool position of Z-0.15 during the first cut. That means the first depth cut will physically be not as deep as all subsequent depth cuts, because part of the 'depth' is in the air (above the part). In subprograms, this kind of compromise is often necessary, as are many special calculations.

In well designed macros, compromise calculations are not required. Macros do need many calculations (as the previous examples have shown), but once developed and verified, macros offer much greater flexibility, streamlined approach to problem solving, and functions not available in subprograms (or standard programs).

Understanding macros usually starts with understanding subprograms. This example - and all previous examples - have shown that good macro design follows the same logical thinking that is needed for the subprogram development - macros just offer more tools to do the job.

Macro Version Development

Normally, custom macros are developed from scratch - there is no need to create a subprogram first and then 'translate' it. Based on the provided drawing in *Figure 48* (and all similar drawings), a general master template of all required variables can be made (*Figure 49*):

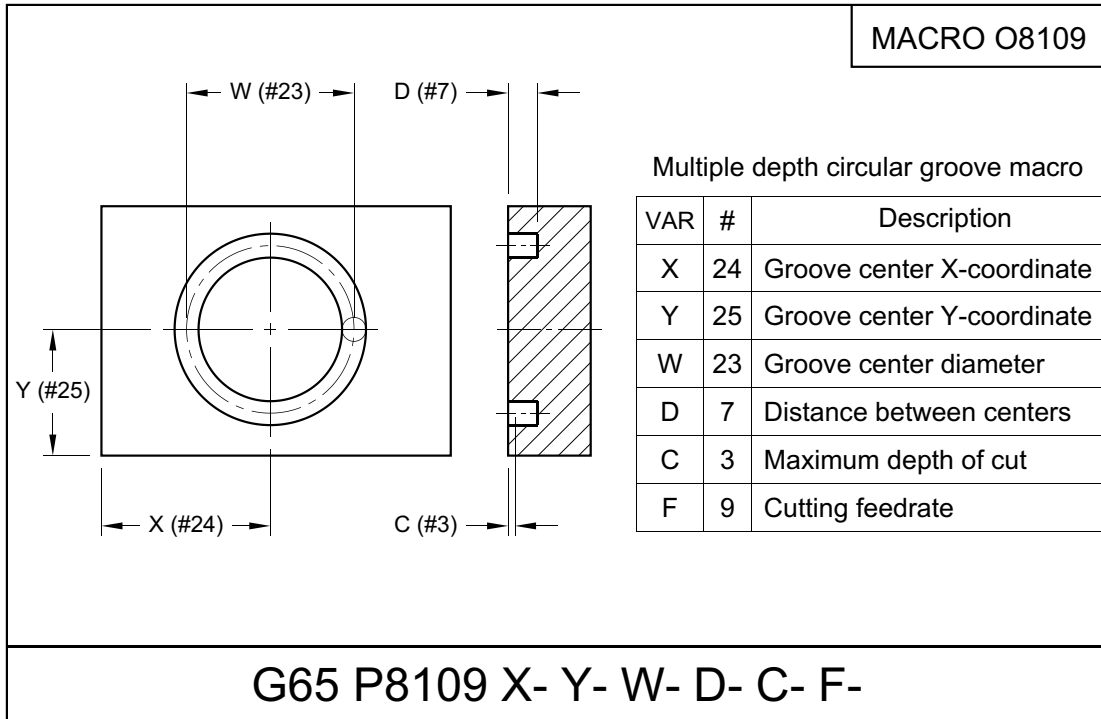


Figure 49

Assignment of variables for a typical circular groove cutting macro - Macro O8109

Based on the drawing specifications, the macro call will be developed as expected:

G65 P8109 X- Y- W- D- C- F-

☞ where ...

- X = (#24) Center location of the circular groove in X-axis
- Y = (#25) Center location of the circular groove in Y-axis
- W = (#23) Groove diameter on the centerline (groove pitch diameter)!
- D = (#7) Total depth of the groove
- C = (#3) Cutting depth of the groove (maximum depth of each cut)
- F = (#9) Cutting feedrate for the circle machining

Although this macro only covers the most important general concepts of controlling the groove depth, it should not be very difficult to develop another macro, one that completes both walls of the groove, once the final depth has been reached (finishing cuts). Regardless of the macro complexity (easy or difficult), thorough pre-planning is absolutely essential. Good planning can save many valuable hours of work and yield excellent results.

In a complete application, the dimensions from *Figure 48*, will be transferred to the macro call:

```

O0029
(X0Y0 LOWER LEFT CORNER - Z0 TOP OF PART)
N1 G21 Metric units mode
N2 G90 G00 G54 X22.4 Y16.7 S750 M03 Any reasonable XY location is OK
N3 G43 Z5.0 H01 M08 Clearance above the start point
N4 G01 Z0 F100.0 Macro must start at Z0
N5 G65 P8109 X22.4 Y16.7 W22.0 D3.9 C0.75 F100.0 Macro call and assignments
N6 G90 G00 Z5.0 M09 Rapid to clearance above the part
N7 G91 G28 Z0 M05 Machine zero return for Z-axis
N8 G28 X0 Y0 Machine zero return for XY-axes
N9 M30 End of main program
%

O8109 (MULTI DEPTH CIRCULAR GROOVE MACRO)
#11 = #4001 Store current G-code of Group 01
#13 = #4003 Store current G-code of Group 03
#23 = #23/2 Change groove pitch diameter to radius
G90 G00 X[#24+#23] Y#25 Rapid to XY start of groove at current Z-depth
#100 = #7/#3 Calculate exact number of depth passes
#101 = FIX[#100] Discard fractions of #100 for actual number of depth passes
#33 = 1 Reset counter of depth passes to the first one
WHILE[#33 LE #101] DO1 Check if more depth cuts are necessary
G01 G91 Z-#3 F#9 Feed in the Z-axis by the depth increment
G03 I-#23 Cut the full circle
#33 = #33+1 Increase depth pass counter by 1
END1 End the loop
IF[#100 EQ #101] GOTO999 Check if full depth reached
G90 G01 Z-#7 Cut to full depth of the groove
G03 I-#23 Cut the full circle
N999 G90 G00 Z0 Rapid to the top of part
G#11 G#13 Restore G-codes of Group 01 and Group 03
M99 End of macro
%
```

Just one short note relating to the **WHILE** loop. If variable **#100** (exact number of passes) is a whole number, that number will also become the number of depth passes. Otherwise, there will be one more circular cut at the full groove depth, which could be as small as the minimum increment of the selected dimensional units (0.001 mm or 0.0001 inches), but never equal to or greater than the full groove depth defined in variable assignment D (**#7**).

Macros in this chapter demonstrate parametric programming. Compare the macro development with the subprogram example - both types are valid in their own way, but there are significant differences.

Rectangular Pocket Finishing

Machining a rectangular pocket is very similar to machining a circular pocket. In this chapter, both roughing and finishing examples were shown for the circular pocket machining. As far as the general programming approach is concerned, there is not much difference in the macro design to machine a roughing toolpath for a rectangular pocket, but there are some differences when it comes to machining a finished rectangular pocket. Another name for this type of machining is *frame machining* or frame macro, because only the walls and corner radii are machined.

Figure 50 shows a typical drawing of a rectangular frame machining.

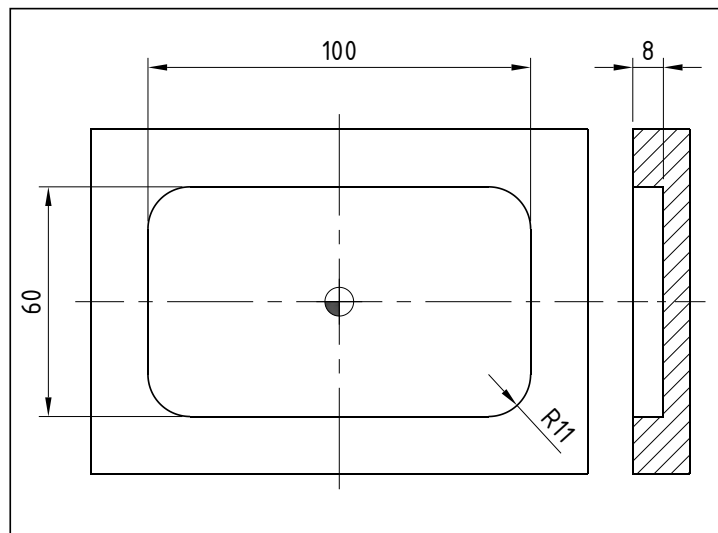


Figure 50

Example of a drawing for typical rectangular pocket (frame) finishing

As with all macro examples in this handbook, the objective is to develop a macro for learning purposes, not necessarily macro with all bells and whistles. Before developing the macro, decide on the method of machining, for example:

1. **Start at the pocket middle position (part zero in the example)**
2. **Plunge-in to depth at one-half the programmed feedrate**
3. **Apply cutter radius offset G41 D- along a lead-in linear motion**
4. **Approach the contour as a lead-in arc**
5. **Machine the contour back to the starting point**
6. **Leave the contour as a lead-out arc**
7. **Cancel cutter radius offset - G40 - along a lead-out linear motion**
8. **Retract above the part**

Items 1 and 8 will be included in the main program, items 2-7, as well as various checks and calculated values, will be part of the macro. Figure 51 illustrates the assignment of variables.

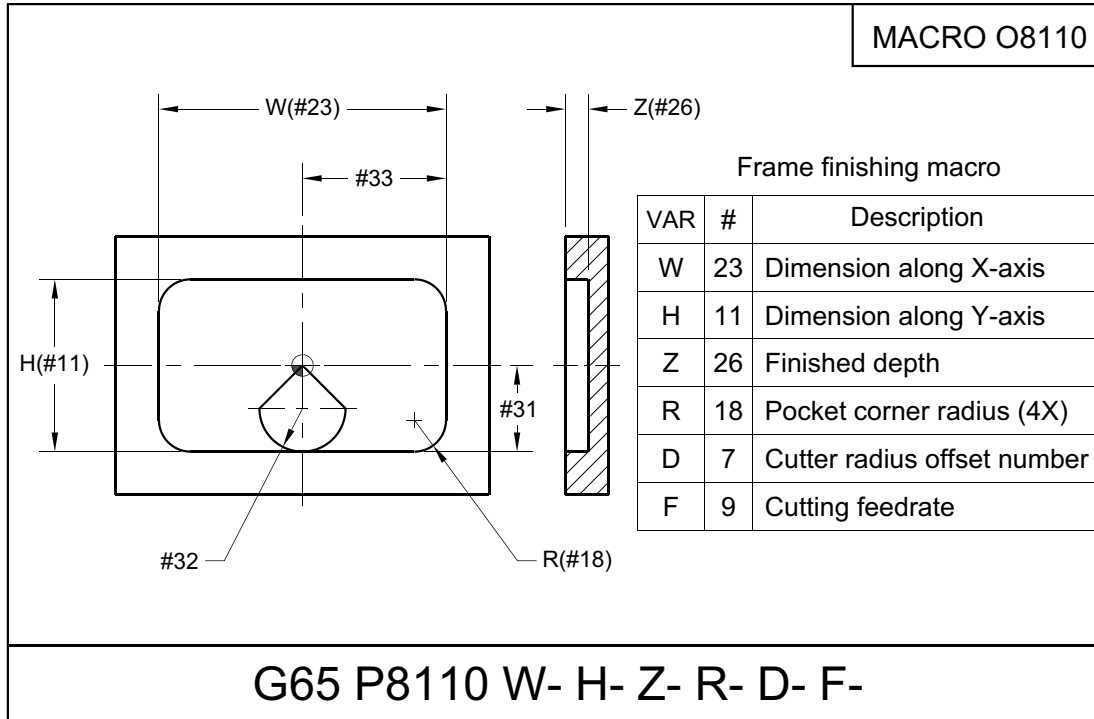


Figure 51

Assignment of variables for a typical rectangular pocket (frame) finishing macro - Macro O8110

The four supplied dimensions will become variable assignments, the center of the pocket is set at X0Y0. The macro can be modified for other XY coordinates. Based on the engineering specifications provided by the drawing, the frame finishing macro call can be developed:

G65 P8110 W- H- Z- R- D- F-

☞ where ...

- W = (#24) Pocket length along the X-axis
- H = (#25) Pocket width along the Y-axis
- Z = (#23) Pocket finished depth
- R = (#18) Pocket corner radius
- D = (#7) Cutter radius offset number
- F = (#9) Cutting feedrate for the pocket machining

Machining decisions and special requirements will influence the macro design. In this example, the tool starts and ends at the pocket middle position. That may be a suitable position for small and medium size pockets, but rather inefficient for large pockets. Other additions to the macro can include variable part zero, angular pocket orientation, finishing the bottom surface, changing feedrates in the corners, clearance above the part, and so on. As presented, the macro does include checking against missing variable assignments, but does not check against wrong input. It also checks if the stored cutter radius offset value is small enough to cut both the lead arcs and the corner radiuses. With a specific goal, the macros in this chapter can be greatly enhanced.

```

O0030
N1 G21                               Metric units
N2 G17 G40 G80 G49 T03               Status settings and tool call
N3 M06                               Tool change
N4 G90 G54 G00 X0 Y0 S800 M03 T04   Center of pocket location
N5 G43 Z10.0 H03 M08                Initial level clearance
N6 65 P8110 W100.0 H60.0 Z8.0 R11.0 D3 F175.0 Macro call
N7 G90 G00 Z2.0                     Clear above part
N8 G28 Z2.0 M09                     Machine zero Z-axis only
N9 M01                               Optional stop

O8110 (RECTANGULAR POCKET FINISHING)
IF[#23 EQ #0] GOTO9101              Length along X-axis must be defined
IF[#11 EQ #0] GOTO9102              Length along Y-axis must be defined
IF[#18 EQ #0] GOTO9103              Corner radius must be defined
IF[#7 EQ #0] GOTO9104               Cutter radius offset number must be defined
IF[#9 EQ #0] GOTO9105               Cutting feedrate must be defined
IF[#26 EQ #0] GOTO9106              Pocket (frame) depth must be defined
#120 = #[2400+#7]+#[2600+#7]        Retrieve the stored value of selected tool offset
IF[#120 GE #18] GOTO9107            Offset value radius must be less than corner radius
#31 = [ABS[#11/2]]                  One half of the length along the Y-axis (positive)
#32 = #31/2                          Lead-in and lead-out line and arc
IF[#120 GE #32] GOTO9107            Offset value radius must be less than lead radius
#33 = [ABS[#23/2]]                  One half of the length along the X-axis
G90 G00 Z2.0                         Arbitrary Z-clearance
G01 Z-[ABS[#26]] F[#9/2]             Half feedrate for plunge-in
G91 G01 G41 X-#32 Y-#32 D#7 F[#9*2] Linear lead-in with cutter radius offset ON
G03 X#32 Y-#32 I#32 J0 F#9           Arc lead-in
G01 X[#33-#18]                       Lower bottom wall
G03 X#18 Y#18 I0 J#18                Lower right corner
G01 Y[2*[#31-#18]]                   Right wall
G03 X-#18 Y#18 I-#18 J0              Upper right corner
G01 X-[2*[#33-#18]]                   Top wall
G03 X-#18 Y-#18 I0 J-#18             Upper left corner
G01 Y-[2*[#31-#18]]                   Left wall
G03 X#18 Y-#18 I#18 J0               Lower left corner
G01 X[#33-#18]                       Lower bottom wall
G03 X#32 Y#32 I0 J#32                Arc lead-out
G01 G40 X-#32 Y#32 F[#9*2] M09       Linear lead-out with cutter radius OFF
GOTO9999                              Bypass alarm messages
N9101 #3000 = 101 (LENGTH ALONG NOT DEFINED) Generates error
N9102 #3000 = 102 (LENGTH ACROSS NOT DEFINED) Generates error
N9103 #3000 = 103 (CORNER RADIUS NOT DEFINED) Generates error
N9104 #3000 = 104 (RADIUS OFFSET NUMBER NOT DEFINED) Generates error
N9105 #3000 = 105 (FEEDRATE MUST BE DEFINED) Generates error
N9106 #3000 = 106 (POCKET DEPTH MUST BE DEFINED) Generates error
N9107 #3000 = 107 (OFFSET VALUE TOO LARGE) Generates error
N9999 M99                             End of macro
%
```

Note the use of the **ABS** function - regardless of the user's input, the macro controls the output with the desired sign. The **ABS** (absolute value) function guarantees that a positive or a negative input value will always be translated into a positive. For example, the following input of the depth could be either positive or negative:

```
G65 P8110 ... Z8.0 ...           positive data input
```

```
G65 P8110 ... Z-8.0 ...         negative data input
```

Within the macro, the input of variable **#26 (Z)** is made positive and the required output is built in the macro - negative Z-value is guaranteed, regardless of the input:

```
G01 Z-[ABS[#26]] F[#9/2]       Z-depth is guaranteed to be negative
```

This is a very powerful programming technique - it anticipates the type of user's input and provides a method that will always yield the expected result.

Thousands more examples could be covered in this chapter - in fact, a whole thick book could deal with examples only - and that is exactly the beauty of custom macros in general and parametric programming in particular. All examples presented in this chapter only cover several varieties and only very few of the many possibilities that exist.

The majority of CNC machine tool manufacturers offer a variety of sophisticated options on their CNC machines - options that are basically hardware additions, additions that must be supported by a special software code, in order to make them useful in a CNC program. Typical options in this category cover special coolant functions, tool change functions, broken tool detectors, operations of pallets, various interfaces, and many others, too numerous to list. The machine tool reference manual that accompanies every CNC machine specifies that the majority of these options are controlled by calling a special (*i.e.*, non-standard) G-code or an M-code in the CNC part program (acceptable as an input to the control system).

Fanuc macros offer similar possibilities - any experienced macro programmer can create and subsequently use special G-codes or M-codes (and some other codes as well). It is not very likely that a production oriented CNC programmer you would develop a G-code macro or an M-code macro for some special hardware feature of the machine tool, but it can be very likely that a macro can be generated as a special purpose software, such as a unique repetitive machining. Some typical examples in this category belong to the *fixed cycles* or *canned cycles*. Fixed cycles shorten programming by eliminating repetitive data. In reality, they are special purpose macros that Fanuc provides as a standard programming feature (built-in). We call these macros by an established G-code, for example, by **G81** command for the standard drilling cycle.

Special Cycles

As useful as the standard fixed cycles are in their own purpose, sometimes there is a need for a cycle that is either slightly different from the existing cycle, or is a totally new cycle - a cycle that will have the same look and feel of the 'traditional' fixed cycles. Take, for example, a fixed cycle that requires a certain feedrate during the cutting motion, but a different feedrate during the retract motion. No standard fixed cycle can do it, but with a special macro, it can be done.

Normally, Fanuc controls allow storage of a macro as a special type of subprogram with variable data, using the **O-** address. Such a macro is called by the **G65** macro call command, followed by the macro program number **P-** and the required variable assignments. In order to define a new macro as a G-code cycle or a special M-function, there are only three considerations to follow:

- | | |
|---|---|
| <input type="checkbox"/> Select the G-code or the M-code to be used | <i>no duplication with existing codes</i> |
| <input type="checkbox"/> Select the macro program number from a given range | <i>depends on the control system</i> |
| <input type="checkbox"/> Set the system parameters of the machine control system | <i>depends on the control system</i> |

It is important that neither the new G-code nor the M-code is already available at the control. In other words, it requires very good knowledge of all G/M codes the control system uses, so any new cycle selection is unique. Knowing which system parameter is to be used to register the selection is also very important. The equivalent parameter numbers vary from control to control, so assuring the correct data entry is imperative.

Options Available

Although the most common programming codes that are used as special cycles are the G-codes, M-functions are often used to allow a hardware function call by the special M-code, selected by the CNC macro programmer (often at the manufacturer's level).

Typically, the following addresses can be used for either a macro call or a subprogram call:

- G-code macro call** ... *common*
- M-code macro call** ... *common*
- M-code subprogram call** ... *less common*
- S-code subprogram call** ... *less common*
- T-code subprogram call** ... *less common*
- B-code subprogram call** ... *less common*

G-code Macro Call

Total of 10 (that is ten) of G-codes can be defined as special custom macros that can be called by a G-code. Only the range between **G01** and **G255** is allowed, with the exception of **G65**, **G66**, and **G67** codes. Positive value is the same as **G65**, negative value is the same as **G66** (or **G66 . 1**).

Depending on the control system, the system parameters related to the *G-code Macro Call* are listed in the following tables (different control systems are shown):

FANUC SYSTEM 0	
G-code Macro Call - 10 options available - G65, G66 and G67 excluded	
Parameter Number	Description <Valid data 1 - 255>
220	G-code that calls the custom macro stored in program O9010
221	G-code that calls the custom macro stored in program O9011
222	G-code that calls the custom macro stored in program O9012
223	G-code that calls the custom macro stored in program O9013
224	G-code that calls the custom macro stored in program O9014
225	G-code that calls the custom macro stored in program O9015
226	G-code that calls the custom macro stored in program O9016
227	G-code that calls the custom macro stored in program O9017
228	G-code that calls the custom macro stored in program O9018
229	G-code that calls the custom macro stored in program O9019

FANUC SYSTEM 10 / 11 / 15	
G-code Macro Call - 10 options available - G65, G66 and G67 excluded	
Parameter Number	Description <Valid data 1 - 255>
7050	G-code that calls the custom macro stored in program O9010
7051	G-code that calls the custom macro stored in program O9011
7052	G-code that calls the custom macro stored in program O9012
7053	G-code that calls the custom macro stored in program O9013
7054	G-code that calls the custom macro stored in program O9014
7055	G-code that calls the custom macro stored in program O9015
7056	G-code that calls the custom macro stored in program O9016
7057	G-code that calls the custom macro stored in program O9017
7058	G-code that calls the custom macro stored in program O9018
7059	G-code that calls the custom macro stored in program O9019

FANUC SYSTEM 16 / 18 / 21	
G-code Macro Call - 10 options available - G65, G66 and G67 excluded	
Parameter Number	Description <Valid data 1 - 255>
6050	G-code that calls the custom macro stored in program O9010
6051	G-code that calls the custom macro stored in program O9011
6052	G-code that calls the custom macro stored in program O9012
6053	G-code that calls the custom macro stored in program O9013
6054	G-code that calls the custom macro stored in program O9014
6055	G-code that calls the custom macro stored in program O9015
6056	G-code that calls the custom macro stored in program O9016
6057	G-code that calls the custom macro stored in program O9017
6058	G-code that calls the custom macro stored in program O9018
6059	G-code that calls the custom macro stored in program O9019

M-functions Macro Call

Total of 10 (ten) of M-functions can be defined as custom macros called by an M-function. Only the range between **M01** and **M97** is allowed. M-functions are *NOT* passed to the PMC (*Programmable Machine Control*), unless they are programmed in the macro body.

Depending on the control system, the system parameters related to the *M-function Macro Call* are listed in the following tables:

FANUC SYSTEM 0	
M-code Macro Call - 10 options available	
Parameter Number	Description <Valid data 1 - 97>
230	M-code that calls the custom macro stored in program O9020
231	M-code that calls the custom macro stored in program O9021
232	M-code that calls the custom macro stored in program O9022
233	M-code that calls the custom macro stored in program O9023
234	M-code that calls the custom macro stored in program O9024
235	M-code that calls the custom macro stored in program O9025
236	M-code that calls the custom macro stored in program O9026
237	M-code that calls the custom macro stored in program O9027
238	M-code that calls the custom macro stored in program O9028
239	M-code that calls the custom macro stored in program O9029

FANUC SYSTEM 10 / 11 / 15	
M-code Macro Call - 10 options available	
Parameter Number	Description <Valid data 1 - 97>
7080	M-code that calls the custom macro stored in program O9020
7081	M-code that calls the custom macro stored in program O9021
7082	M-code that calls the custom macro stored in program O9022
7083	M-code that calls the custom macro stored in program O9023

7084	M-code that calls the custom macro stored in program O9024
7085	M-code that calls the custom macro stored in program O9025
7086	M-code that calls the custom macro stored in program O9026
7087	M-code that calls the custom macro stored in program O9027
7088	M-code that calls the custom macro stored in program O9028
7089	M-code that calls the custom macro stored in program O9029

FANUC SYSTEM 16 / 18 / 21	
M-code Macro Call - 10 options available	
Parameter Number	Description <Valid data 1 - 97>
6080	M-code that calls the custom macro stored in program O9020
6081	M-code that calls the custom macro stored in program O9021
6082	M-code that calls the custom macro stored in program O9022
6083	M-code that calls the custom macro stored in program O9023
6084	M-code that calls the custom macro stored in program O9024
6085	M-code that calls the custom macro stored in program O9025
6086	M-code that calls the custom macro stored in program O9026
6087	M-code that calls the custom macro stored in program O9027
6088	M-code that calls the custom macro stored in program O9028
6089	M-code that calls the custom macro stored in program O9029

Machine tool manufacturers often provide time saving procedures into their hardware. Creating new *M-functions* to activate such procedures is consistent with the use of 'normal' M-functions. In order not to conflict with any existing functions, it is not unusual to see M-functions that have three digits (for example, **M123**) or even belong to a certain group that shares certain characteristics (for example, **M201** to **M220**).

The various function codes specified as subprogram calls are used less frequently, and in many ways are similar to the described macro calls. Always consult the Fanuc reference manual for details relating to a particular control model features.

G13 Circle Cutting Cycle

In the previous chapter, a roughing and a finishing macro is described for circular cuts. It is quite common in CNC programming to machine a *circular pocket*. Whether it is a utility circular pocket with minimum tolerances (such as most counterboring operations), or a pocket where the diameter and the depth must be of very high accuracy, there is a way to define a macro or a special cycle (macro based) for this type of machining. Once developed, the macro should be very easy to use, minimizing the possibility of an input error. Some controls do have **G13** macro command, for example, many Yasnac controls. A Fanuc macro can be developed that will be fully consistent with the Yasnac input - such a program can become more portable between different controls, if designed properly. Yasnac's **G12** cycle is similar to **G13**, except that it is used for conventional milling, rather than climb milling.

In order to appreciate the effect of possibilities offered, consider the two most common ways of cutting a circular pocket - both methods are illustrated in *Figure 52*:

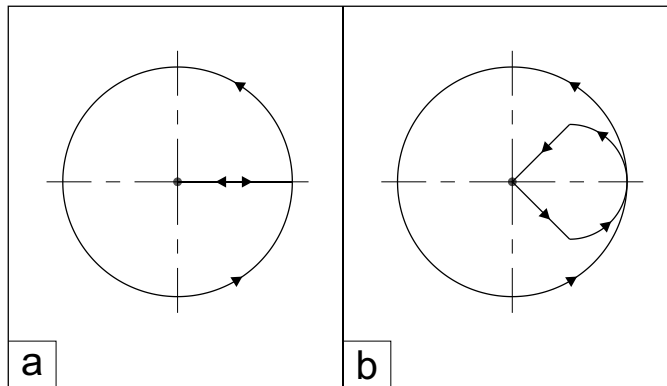


Figure 52

Typical toolpath applied to cutting a circle (circular pocket)

a/ Straight lead-in/lead-out

b/ Tangential lead-in/lead-out

In the left example (*a*), rather a crude but simple circular cutting takes place. Because of the tool mark that is typically left on finished wall, this method is only good for circular cuts when the surface finish and the overall dimensions are not too critical. The second method (*b*), shown on the right, is much more precision oriented, but it requires several extra calculations that may slow down the programming process. In both cases, the cutting tool plunges into the depth at the center of the circular pocket, then continues as a linear motion, before any arc can be profiled. The cutting motion also terminates at the center of the circle. There is a very important reason for this toolpath - to maintain precision dimensions and tolerances, cutter radius offset has to be in effect, and it must be applied only *during a linear motion!* The CNC operator stores the cutter radius in the appropriate offset register and the macro will do the rest - all without the **G41** cutter radius offset command.

The macro that will be stored as a cycle will eliminate the need for the straight linear lead-in and lead-out and will use only three major motions - see *Figure 53*:

- Lead-in arc - from the circular pocket center Step 1 in the illustration
- Full circle machining Step 2 in the illustration
- Lead-out arc - to the circular pocket center Step 3 in the illustration

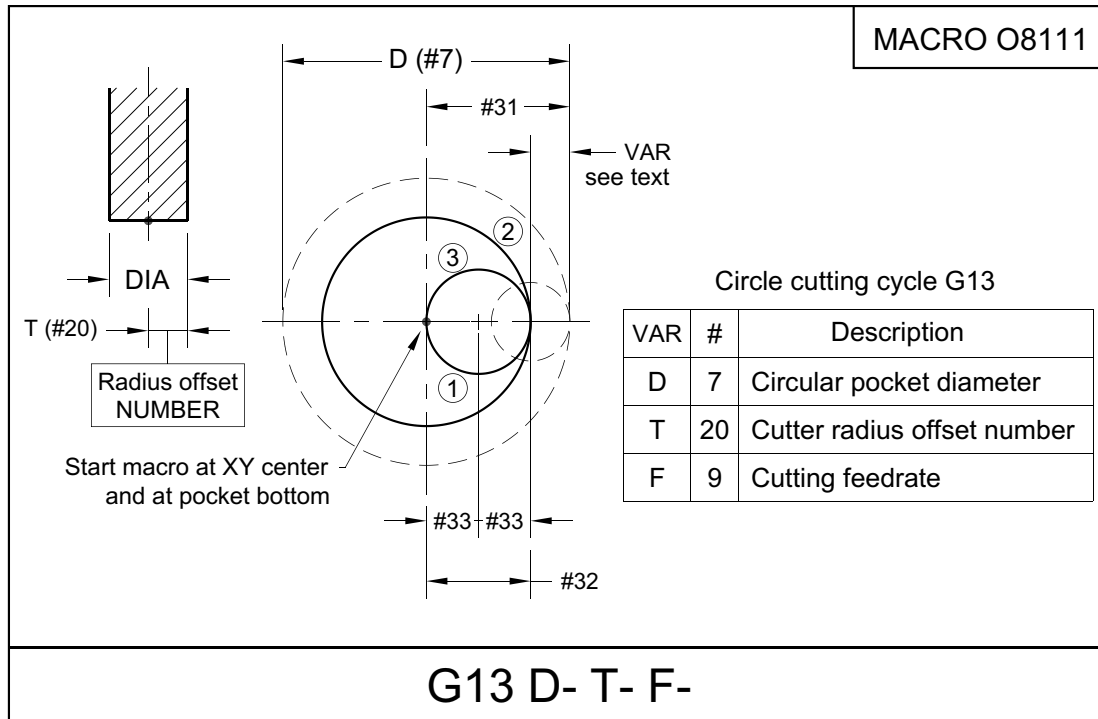


Figure 53

Illustration for the development of G13 circle cutting macro cycle (climb milling mode)

The cutter radius offset using the **G41** command will not be necessary, since the macro reads the radius offset value directly from the control register. In fact, it would be wrong to program **G41**. Without the **G41**, only arcs can be programmed, without linear lead-in and lead-out tool motions. The *Figure 53* also shows assignments of the three variables used in the macro.

The variable data is short for this macro - only three assignments are required:

- The pocket size - normally given on the drawing as a diameter Variable D (#7)
- The tool offset number where the cutter radius is stored Variable T (#20)
- The cutting feedrate Variable F (#9)

There are macros similar to this one that require the pocket radius input rather than its diameter. Selecting the input of a diameter is a better choice, since circular holes or pockets are dimensioned as diameters. For the internal calculations, when the radius is needed, a simple calculation will store the radius as one half of the given diameter.

For most machining applications, the climb milling mode built into the macro (cycle **G13**) is the desirable way of metal removal. However, the macro is not suitable to cut in conventional mode, should such need arise. For that purpose, another macro (cycle **G12**) will have to be developed. Essentially, both macros will be the same, except the machining order of 1-2-3 for the climb milling mode will be reversed to 3-2-1 for the conventional mode (**G03** will change to **G02**). Both macros (cycles) are listed in this chapter.

Macro Call - Normal

Since the previous macro examples, there has been some order established in their numbering. Including the last chapter, this is the 11th macro example, so its number should be O8111. If the stored macro is called the normal way, the way any other macro was called so far, the program number must be assigned first. For example, if the macro program is assigned number O8111, the macro will be called by a simple **G65** statement:

```
G65 P8111 D60.0 T56 F200.0
```

☞ where ...

D = (#7) Circular pocket diameter
 T = (#20) Cutter radius offset *number* **Do not enter the actual cutter radius with T !!!**
 F = (#9) Cutting feedrate for the circle machining

This is an entirely normal macro and the program number assigned is not too important, as long as it is unique and follows standard conventions.

Macro Call - as a Special Cycle

However, this type of machining has all the features expected in a fixed or canned cycle - after all - *it is* a genuine fixed cycle. In order to appear as a cycle to the CNC programmer and/or operator, it also has to have the 'look and feel' of a true cycle, which means it needs a G-code. So far, that has not been achieved with the **G65** macro call statement. In order to transfer the macro into a true fixed cycle, two changes must take place:

- Assign the macro program number from the fixed range provided by the control system
- Register the selected G-code (or the M-code) as a parameter setting

From the several tables listed earlier in this chapter, it is apparent that regardless of the control system, the number of macros that can be called as cycles using the G-code is only 10 (ten), and these macros *must* be stored within a range of program numbers beginning with O9010 and ending with O9019. This range is common to all control systems (a similar range is also available for the M-functions). For reasons of logical association and practical convenience, the new macro will be named O9013. Of course, this change will also change the **G65** macro call - but nothing else - there is no other benefit:

```
G65 P9013 D60.0 T56 F200.0
```

So far, the change was only superficial. One more step is critical and necessary - and that is to *register* the preferred G-code into a control system parameter. Since the **G13** G-code has been selected as the cycle command (macro call), the number 13 has to be registered in the parameter that corresponds to the calling program, which is O9013.

Here are the parameters for all three different control groups, that will be applied to the example presented (**G13** stored as O9013) - see the earlier tables for complete listing:

Fanuc control system	Parameter Number
Fanuc 0	0223
Fanuc 10/11/15	7053
Fanuc 16/18/21	6053

Only after all these settings have been done, the new macro can be called as a true cycle. In this case, it will be cycle **G13**, which looks like many other Fanuc cycles:

G13 D60.0 T56 F200.0

The variable assignments have not changed, just the method of calling them. The G-code could have been any number within 1 to 255 range, with the exception of **G65**, **G66** and **G67**. Before the actual macro can be evaluated, keep in mind that M-codes can be set in a very similar way, also using the above tables.

The circular pocket cycle macro is fairly simple to understand, particularly with the commented descriptions on the side. Yet, the definition of variable **#32** may be a bit unusual and may need some explanation. Study the macro first, then the 'mystery' of the **#32** will be revealed.

```

O9013 (G13 CIRCULAR MILLING CYCLE - CLIMB MILLING)
#31 = ABS[#7]/2           Radius of the circular pocket - guaranteed positive
#11 = #4001              Store current G-code of Group 01 (motion commands)
#13 = #4003              Store current G-code of Group 03 (absolute/incremental)
#32 = #31-#[2000+#20]    Actual radius of the circle to cut - see explanation !!!
IF [#32 LE 0] GOTO998    Generate error if the radius offset value is too large
#33 = #32/2              Calculated lead-in/lead-out radius
G91 G03 X#32 I#33 J0 F#9 Lead-in arc toolpath - Step 1 or R#33 instead of I/J
I-#32                   Full circle toolpath - Step 2 R not allowed for 360°
X-#32 I-#33 J0          Lead-out arc toolpath - Step 3 or R#33 instead of I/J
G#11 G#13               Restore original G-codes of Group 1 and Group 3
GOTO999                 Bypass error message
N998 #3000 = 13 (OFFSET TOO LARGE) Alarm condition - refers to actual offset value setting
N999 M99                 End of macro
%
```

For conventional milling, the **G12** cycle can be developed by making only a very few changes:

```

O9012 (G12 CIRCULAR MILLING CYCLE - CONVENTIONAL MILLING)
#31 = ABS[#7]/2           Radius of the circular pocket - guaranteed positive
#11 = #4001              Store current G-code of Group 01 (motion commands)
#13 = #4003              Store current G-code of Group 03 (absolute/incremental)
#32 = #31-#[2000+#20]    Actual radius of the circle to cut - see explanation !!!
IF [#32 LE 0] GOTO998    Generate error if the radius offset value is too large
#33 = #32/2              Calculated lead-in/lead-out radius
G91 G02 X#32 I#33 J0 F#9 Lead-in arc toolpath - Step 3 or R#33 instead of I/J
I-#32                   Full circle toolpath - Step 2 R not allowed for 360°
```

X-#32 I-#33 J0	<i>Lead-out arc toolpath - Step 1 or R#33 instead of I/J</i>
G#11 G#13	<i>Restore original G-codes of Group 1 and Group 3</i>
GOTO999	<i>Bypass error message</i>
N998 #3000 = 12 (OFFSET TOO LARGE)	<i>Alarm condition - refers to actual offset value setting</i>
N999 M99	<i>End of macro</i>
%	

Detailed Evaluation of Offset Value

Some programmers may be surprised to find that the cutter radius offset **G41** is not required. Does that mean the pocket diameter cannot be controlled for high precision requirements? Not at all. At the machine, the CNC operator will handle the offset *exactly the same way* as in any other application. Here is how it works. There are two main elements that make the offset work, *without* the **G41** command.

- The T (#20) variable in the G13 call refers to the offset *number* that stores the cutter radius
- The definition of variable #32 in the macro that retrieves the actual stored offset *amount*

In the example, variable #20 (T) was stored with the value of 56 (*tool length and radius shared in the same registry*). The pocket diameter is 60 mm, so a suitable cutter should be greater than 60/3 (to clean the pocket bottom), say 22. If the difference between the pocket and cutter radius is too great, the pocket bottom surface will not be fully cleaned. During setup, under ideal conditions, the CNC operator inputs the cutter radius (11 mm), as the radius offset value, into the offset number 56 (assuming a system parameter is set to radius entry). So far, nothing is really new.

The 'problem' is with the way Fanuc handles this information for different features of its control models. Fortunately, at least in this case, the model control numbers do not matter too much, because the same approach applies to the Fanuc 10/11/15/16/18/21 controls (Fanuc 0 is excluded because of its limited applications). The solution is based on these two very important features:

- Tool offset memory type *three groups - A, B, and C*
- The number of available offsets *two groups - 200 and less, and over 200*

Both of these groups have been described earlier in *Chapter 11 - Tool Offset Variables*. In this chapter, the focus is on the offsets for the radius setting, using the two related tables:

200 OFFSETS AND LESS					
Offset Number	Memory A	Memory B		Memory C	
	Geometry / Wear	Geometry	Wear	Geometry - D	Wear - D
1	#2001	#2001	#2201	#2401	#2601
2	#2002	#2002	#2202	#2402	#2602
3	#2003	#2003	#2203	#2403	#2603
4	#2004	#2004	#2204	#2404	#2604
5	#2005	#2005	#2205	#2405	#2605
6	#2006	#2006	#2206	#2406	#2606

...
...
200	#2200	#2200	#2400	#2600	#2800

OVER 200 OFFSETS					
Offset Number	Memory A	Memory B		Memory C	
	Geometry / Wear	Geometry	Wear	Geometry - D	Wear - D
1	#10001	#10001	#11001	#12001	#13001
2	#10002	#10002	#11002	#12002	#13002
3	#10003	#10003	#11003	#12003	#13003
4	#10004	#10004	#11004	#12004	#13004
5	#10005	#10005	#10005	#12005	#13005
6	#10006	#10006	#11006	#12006	#13006
...
...
999	#10999	#10999	#11999	#12999	#13999

From the tables is clear that the definition of variable #32 in the macro body will be *different* for the type of offset memory and the number of available offsets. Actually, the input of variable #32 (as shown in the macro) is only good for memory *Type A*, and less than 200 offsets:

$$\#32 = \#31 - \#[2000+\#20] \quad \text{SHARED OFFSET REGISTRY}$$

For the example, #32 returns the result of:

$$\#32 = 30.0 - \#[2000 + 3] = 30.0 - \#2003 = 30.0 - 11.0 = 19.0$$

Study the calculation - try to understand how Fanuc system evaluates it - understanding many other macro features, including those that are not so obvious, will follow very quickly. A similar example applies to the same settings as above, but is used only for the *Type C* offset memory and more than 200 offsets - note that both the *Geometry* and *Wear* offsets must be considered:

$$\#32 = \#31 - [\#[12000+\#20] + \#[13000+\#20]]$$

Since the *Type C* offset memory is quite common on the modern high end control systems, but the number of available offsets is typically less than 200, let's look at one last example - *Type C* offset memory, with 200 or less offsets available - again, note that both the *Geometry* and *Wear* offsets are considered, but with different variable numbers:

$$\#32 = \#31 - [\#[2400+\#20] + \#[2600+\#20]]$$

In last several examples, many typical macro applications were evaluated, each time with a little different focus. Although the presented applications have been shown using particular programming techniques, the principles described here apply to many similar applications, particularly those that apply to various tool offsets in different ways.

Counterboring Application

The **G13** cycle is ideal to be used for counterboring previously drilled holes using a circular toolpath with an end mill rather than by a plunge cut. Program example that follows the drawing (Figure 54) uses the previously stored circular cutting macro **G13**:

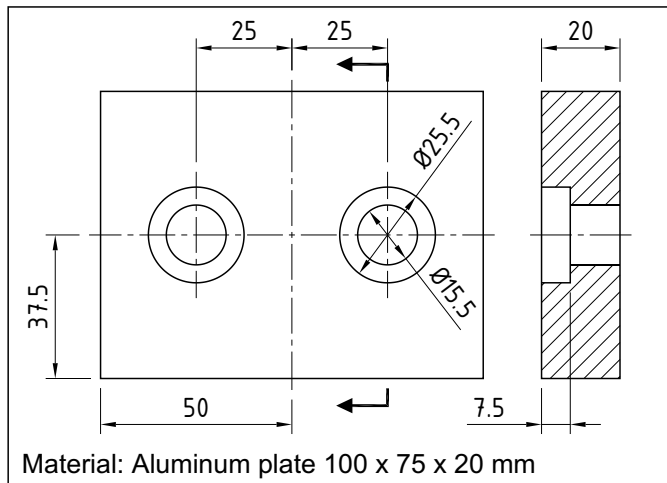


Figure 54

Drawing example for using circle cutting macro (special cycle G13)

```

O0031
N1 G21
(PART ZERO = LOWER LEFT CORNER AND TOP OF PART)
<... spot drill and drill operations ...>
N31 T03                                T03 = 10 mm end mill
N32 M06                                Tool change
N33 G90 G54 G00 X25.0 Y37.5 S750 M03   Left counterbore location
N34 G43 Z2.0 H03 M08                  Clearance above part
N35 G01 Z-7.5 F250.0                  Feed-in to counterbore depth
N36 G13 D25.5 T53 F180.0               Macro call as a cycle appears 'natural'
N37 G00 Z2.0                           Clearance above part
N38 X75.0                               Right counterbore location
N39 G01 Z-7.5 F250.0                  Feed-in to counterbore depth
N40 G13 D25.5 T53 F180.0               Macro call as a cycle appears 'natural'
N41 G28 Z2.0 M09                       Machine zero return
N42 M30                                End of program
%
```

Since the *Offset Memory Type A* selection (shared offsets) is part of the macro, the stored radius offset must have a different number than the stored tool length offset. A difference of 50 is how in the example ($H01 + 50 = > D51$).

22

EXTERNAL OUTPUT

Normally, the current value of any variable can be viewed right on the control display screen. Often, that is not enough. For example, when troubleshooting a macro that contains many variables, it may be necessary to check the current variables in different parts of the macro, at different processing stages. Fanuc controls provide several commands that can output variable values and various characters, through the RS-232C port (I/O interface - *Input/Output*) to various external devices. These commands are called *External Output Commands*, and there are four of them:

POPEN	PCLOS
--------------	--------------

and

BPRNT	DPRNT
--------------	--------------

Port Open and Port Close Commands

In order to make any two computer based communication devices to transfer data, the devices must be set to a single matching mode that allows the data transfer. When transferring data from a Fanuc control system, the output port has to be *initiated*, which means the port has to be set to an *open* state. Once the data transfer had taken place, the control port must be closed - set to a *closed* state. On the receiving end, the device used will have to be set to the state that can accept the data. Term used in communications between devices are *read* and *write*, *reader* and *puncher*, *upload* and *download*, *input* and *output*, *ON* and *OFF*, etc.

The command **POPEN** (*Port Open*) provides the connection to an external *Input/Output* device (I/O unit). The two typical units in this category are the *tape puncher/reader* unit and a *personal computer*. The variable data can be stored on a punched tape (the older method), or on a computer disk as a text file (the modern method). In a custom macro, the **POPEN** command must always be set at the beginning, *before* the transfer data are specified. Fanuc control outputs the *DC2* control code. Think of the **POPEN** as the 'connect' command.

The command **PCLOS** (*Port Closed*) cancels the connection to an external *Input/Output* device (I/O unit). When all data had been transferred, the communication lines must be closed with the **PCLOS** command. Fanuc control outputs the *DC4* control code. Think of the **PCLOS** as the 'disconnect' command that should never be programmed, unless the **POPEN** command is in effect.

A macro containing the POPEN command at the beginning must also contain the PCLOS command at the end

Data Output Functions

BPRNT and **DPRNT** functions must always be used between the **POPEN** and **PCLOS** functions

BPRNT and **DPRNT** functions are used to execute the actual data transfer in two different forms:

- BPRNT** ... transfers the output in a binary format, useful for data only
- DPRNT** ... transfers the output in a plain text format, data & text, (ISO or ASCII text)

In practice, the majority of outputs to external devices are in the **DPRNT** text format, partially because of its output of decimal places. Text format output is easier to read and interpret, whether it is printed as a hard copy or displayed on the computer screen.

Each format takes several arguments, describing:

- One or more characters to be output - usually headings and similar identifications
- Variable number that is being printed
- Control of decimal places

There is a small, but important difference in the format for each command.

BPRNT Function Description

The **BPRNT** function writes a binary output. The programming format is shown in *Figure 55*.

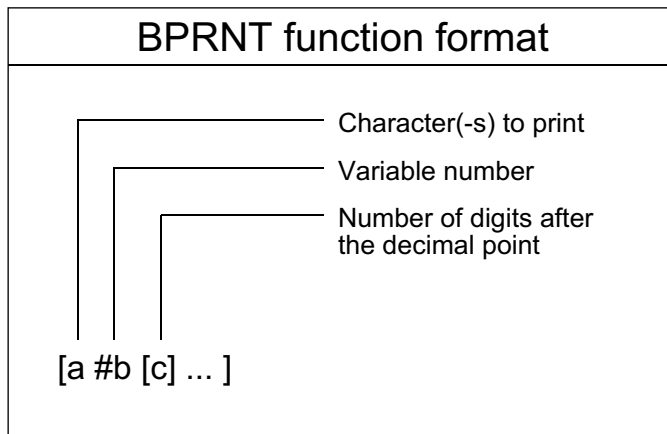


Figure 55

Format structure of the BPRNT function

In the **BPRNT** function, the characters can be the capital letters of the English alphabet (A to Z), all digits 0 to 9, and several special characters (+ - / * ...). An asterisk symbol (*) will be output as the space code. The *End-Of-Block* character (EOB) will be output according to the setting of the ISO code. Variables that are vacant (null variables) cannot be output on older Fanuc System Model 6 (alarm #114 will result in this case), however, they will be output as 0 on Fanuc controls 10/11/15/16/18/21. All variables are stored with a decimal point and the number of decimal places after the decimal point must be specified in brackets, following the variable number (see item *c* in the above illustration).

DPRNT Function Description

The **DPRNT** function writes a plain text output. The programming format is shown in *Figure 56*.

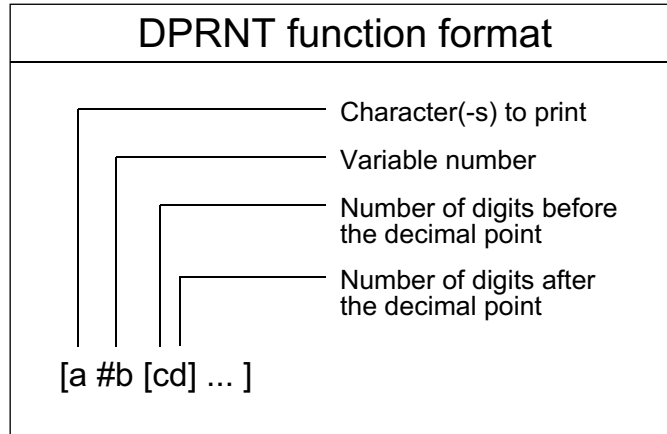


Figure 56

Format structure of the DPRNT function

In the **DPRNT** function, the characters can be the capital letters of the English alphabet (A to Z), all digits 0 to 9, and several special characters (+ - / * ...). Asterisk (*) will be output as the space code. The *End-of-Block* character (EOB) will be output according to the setting of the ISO code. Variables that are vacant (null variables) cannot be output on Fanuc 6 (alarm #114 will result), however, they will be output as 0 on Fanuc 10/11/15/16/18/21. Since the output format depends on the setting of some system parameters, let's look at the settings of the relevant parameters. There is a difference between some controls.

Parameter Settings - Fanuc 10/11/12/15

In order to make the data transfer work correctly, some system related parameters have to be set accordingly. The following parameters have to be set for Fanuc controls 10/11/12/15:

Parameter Number	Setting value	Type
0021	Output device interface number for foreground	Byte

☞ where the *setting value* can be ...

- 1: Puncher to be connected with CD4A of BASE0 (RS-232C interface 1)
- 2: Puncher to be connected with CD4B of BASE0 (RS-232C interface 2)
- 3: Puncher to be connected with CD4 of serial port (RS-232C interface 3)
- 4: DNC1
- 13: Puncher to be connected with CD3 of serial port (RS-222 interface)
- 15: MMC DNC operational interface
- 16: MMC UPLOAD/DOWNLOAD interface

Note that PUNCHER can be any external RS-232 device.

During the control operation, press the *RESET* button when the required parameters are set. Normally, select the suitable '*Puncher to be connected with ...*' setting, depending on the configuration of the communications devices. The actual message may be different.

NOTE - some minor distinctions apply between Fanuc 10/11 and Fanuc 15 !

Parameter Number	Setting value	Type
5001 to 5162	This series of parameters control the various settings of the external interface, such as the I/O device number (up to six), baud rate, stop bits, etc. The parameters in this range should be set in coordination with the setting of parameter 0021.	Byte

Parameter Number	Setting value	Type
7000 - Bit #7 (PRT)	This parameter controls the spacing of leading zeros, for the output generated by the DPRNT command.	Bit

 where *Bit #7* is ...

- 0: A space is output when reading zero with the **DPRNT** command
- 1: Nothing is output when reading zero with the **DPRNT** command

The **DPRNT** function always requires the specification of both number of digits (items *c* and *d* in the illustration) - *before* the decimal point (the integer number of the variable) and *after* the decimal point (the actual number of decimal places).

Metric vs. Inch Format

In typical CNC applications, the metric format (**G21**) takes 5 digits before the decimal point, and 3 digits after the decimal point (eight digits total). Since the decimal point is legitimate in this format, it is often referred to as the 5.3 format or **[53]** format. When the English measurements units are used (**G20**), the inch format takes 4 digits before the decimal point, and 4 digits after the decimal point (also eight digits total). Since the decimal point is legitimate in this format as well, it is referred to as the 4.4 format or **[44]** format. Either of these two commands are applicable to all controls. For example, if the local variable **#100** specified in the macro program contains a value of 123.45678 units (metric or English), and the **DPRNT** command is specified as ...

DPRNT [X-VALUE*#100 [53]]** ... each asterisk outputs a space

.... then the output value will depend on the parameter setting (varies for different controls):

X-VALUE 123457 ... if the parameter #7000 is set to 0 ... **or:**
X-VALUE 123.457 ... if the parameter #7000 is set to 1

Variables must be numeric values with no more than the total of eight digits

It may be necessary to experiment with the output formatting, to match personal preferences.

Parameter Settings - Fanuc 16/18/21

In order to make the data transfer work correctly, some system related parameters have to be set accordingly. The following parameters have to be set for Fanuc controls 16/18/21:

Parameter Number	Setting value	Type
0020	I/O channel: Selection of an input/output device	Byte

 where the *setting value* can be ...

- 0: The device associated with Channel 1 is selected (JD5A connection on main board)
- 1: The device associated with Channel 1 is selected (JD5A connection on main board)
- 2: The device associated with Channel 2 is selected (JD5B connection on main board)
- 3: The device associated with Channel 3 is selected (Option1 board connection)

Select the suitable device setting, depending on the configuration of particular communications devices. Typically, the I/O channel is set within the following range of parameters:

Parameter Number	Setting value	Type
0100 to 0149	This series of 150 parameters control the various settings of the external interface, such as the I/O device number (up to three), baud rate, stop bits, etc. The parameters in this range should be set in coordination with the setting of parameter 0020	Bit and Byte

Reader/puncher interface is set by the following parameters:

I/O Channel	Parameters used	Type
0	#101, #102, #103	Bit and Byte
1	#111, #112, #113	
2	#112, #122, #123	

Note - Fanuc cassette or floppy cannot be used for puncher output !

The last parameter setting is for the control of leading zeros:

Parameter Number	Setting value	Type
6001 - Bit #1 (PRT)	This parameter controls the spacing of leading zeros, for the output generated by the DPRNT function	Bit

☞ where *bit #1* is ...

- 0: A space is output when reading zero with the **DPRNT** command
- 1: Nothing is output when reading zero with the **DPRNT** command

As mentioned earlier, the **DPRNT** command always requires the specification of both number of digits (items *c* and *d* in the illustration) - *before* the decimal point (the integer number of the variable) and *after* the decimal point (the actual number of decimal places). For example, if variable **#100** contains a value of 123.45678, and the **DPRNT** command is:

```
DPRNT [X-VALUE***#100 [53] ]
```

then the output value will depend on the parameter setting (varies for different controls):

```
X-VALUE    123457                ... if the parameter #6001 is set to 0           ... or:
```

```
X-VALUE    123.457              ... if the parameter #6001 is set to 1
```

Variables must be numeric values with no more than the total of eight digits

Structure of External Output Functions

Although the **POPEN** function is required *before* either **BPRNT** or **DPRNT** can be used, it is not necessary to immediately close the receiving device after the data transfer had been completed. If another data transfer is required, just call another **BPRNT** or **DPRNT** command. Program the **PCLOS** function only after *all* transfers have been completed. The two following macro structures are allowed, with the second one as a preferred method:

Macro structure - Version 1

```
POPEN
...
BPRNT or DPRNT          ... with variable specifications
...
PCLOS
...
POPEN
```

```

...
BPRNT or DPRNT          ... with variable specifications
...
PCLOS
...

```

Macro structure - Version 2

```

POPEN
...
BPRNT or DPRNT          ... with variable specifications
...
BPRNT or DPRNT          ... with variable specifications
...
PCLOS
...

```

Output Examples

The following macro example will download the current values of variables within the range of 100 to 149 to an external device, such as a computer disk file (in text format):

➤ Macro call:

```
G65 P8200 I100 J149          Example of macro call - range of variables specified
```

➤ Macro definition:

```

O8200 (VARIABLE SETTINGS PRINT-OUT)
POPEN          Initialize the active communications port
#1 = 0         Reset variable counter
WHILE[#1 LE [#5-#4]] DO1 Limit loop to selected range of variables
#2 = #[#4+[#1]] Current variable number - as a variable
#3 = #4+#1    Current variable number - as a number (no # symbol)
DPRNT[VAR #3[5] ***DATA #2[57]] Formatted output includes text, variable ID and value
#1 = #1+1     Increase the counter of variables by one
END1          End of loop
PCLOSE       Close the active communications port
M99         End - can be M30 if used as main program
%

```

Virtually any data that is stored in the control system can be output as a hard copy or displayed on the screen. Macro programs using the **DPRNT** function can be very useful in keeping records, creating a log of program flow, debugging a troublesome macro, and many other applications. Some common examples are shown in the next section.

23

PROBING WITH MACROS

In the modern world of CNC technology, many machine features have been controlled through a part program or a dedicated macro program. It is quite normal in typical CNC programming to use *Automatic Tool Changer (ATC)*, *Automatic Pallet Changer (APC)*, coolant functions, spindle functions, etc., in addition to the automation of the toolpath itself. With macros - and suitable machine and control features - the possibilities can go a big step further and automate the whole manufacturing process, especially various dimensional measurements. Within such a manufacturing process, tolerances of important part features are extremely important. Various depths, widths, diameters, thicknesses, distances and many other miscellaneous engineering requirements have to be handled as reliably and accurately as possible. With macros, these processes can be automated and high quality machining results can be achieved with little or no human interference during the part production.

The most important aspect of this approach is to perform various measurements and inspections directly at the CNC machine. Operations before, during, or after machining can be made possible with the use of the so called 'probing' devices, consisting of a ball-shaped precision probe (usually), connected electronically to the control system and controlled by suitable macro programs.

This concept of probing requires a solid background in the probing technology, a subject that is not in itself part of the programming process, but serves as the initial building block. Before some probing macros can be introduced, it is important to understand what the probing is all about, and get familiar with the fundamental concepts. This short background will make it easier later, when actually developing real probing macros.

Probing technology changes rapidly - always check for the newest capabilities

The most important part of the probing concept is the interaction between the macro program and the probing device (and its numerous activities). The CNC system, using customized macro programs, fully supports reading and writing of data between the program processing and the machine activities. External CMM machines are only marginally related to this subject.

What is Probing ?

In a machine shop environment, there are two important words that relate to the same objective - to measure certain features of a three dimensional object - a part to be completed. These two words are *probing* and *gauging* (also spelled *gaging*). Often, these two words are used interchangeably, as both apply to similar activities.

Typically, the word *probing* is used when the measuring device includes special spherical stylus that is program controlled to move from one part of the measured object to another part of the same object. The other word - *gauging* - is used for all other types of measurement.

The focus of this chapter is in the area of probing, since a special program is always required. Many operators in a machine shop are familiar with the concept of probing a part before or after the machining process. This type of probing is commonly known as *coordinate measuring*, and the specially designed precision machines that perform this process are called *Coordinate Measuring Machines*, known as the CMM. The CMM method only measures the existing dimensions and can record and register the measured values - but it cannot change any of the measurement. Often it is called the pre-process and post-process method. There is a program involved in running a CMM station, but it has nothing to do with CNC and macros.

The probing method that is applied during CNC machining has many benefits. In this method, the probing device is mounted as one of the tools in the machine tool magazine, using a specific tool number. The actual probing process is controlled by a specially developed macro program. The benefit of this method is that when the measurement is done, the results of the measurement can be evaluated by the macro program and changes can be made while the part is still within the work area of the machine tool. This is done by using the three groups of offsets that are available on various CNC machines (see *Chapter 11*). A thorough knowledge of the work offset, the length offset and the radius offset is extremely important for understanding how probing macros (some can be very complex) perform the measurements and change the individual offset settings. As this method of probing always takes place at the CNC machine during its operation, it is also called the *in-process method* or *in-process gauging*. Whichever name is used, this method of probing is based on the application of a touch-trigger device, called a *touch probe*. Understanding the principles behind probes and their functions is definitely helpful.

Touch Probes

Around the year 1973, the first modern probe was developed. It was called a *kinematic touch-trigger probe*. This probe design functions on the principle of a multi-directional switch. The main part of the probe is called a stylus, which is a precision ball-shaped tip, ranging in ball diameters and located at the tip of the arm-like stem or extension. The key element of this design (and any probe design) is that the spring loaded pivoting stylus deflects when it touches the contact point and returns to its original position after the deflection. The spring loaded stylus is located against three bearing points, and each of these bearing points is also an electrical contact point.

During the contact with the measured surface (or feature), the center of the stylus deflects at one or two of the bearing points. An electrical contact is established and results in a trigger signal that is registered by the measuring system. Probe technology has evolved in the last few decades, and is well advanced. As any other technology, it evolves constantly, and for any macro programmer it is important to keep in touch with its advancements.

Probing Technology Today

Although the kinematic touch probe principles are applied to this day, the original technology was very dependent on the trigger effect caused by the actual pressure on the bearings. This method has been very accurate, but had its disadvantages. The major disadvantage was that the length of the probe (the stem) had to be kept fairly short, thus preventing measurements of hard-to-reach and deep contact areas of the measured part.

The new technology is called the active silicon strain gauge technology, and does not generate the trigger effect by pressure, but by sensing the contact force that builds up during contact between the highly sensitive stylus and the measured point. Since even a very low contact force can be detected using this method, accuracy is guaranteed for probes even with long stems. Other improvements include high accuracy while measuring complex three-dimensional surfaces, higher repeatability, and increased life of the probing device.

Probe Calibration

In order to get the expected - and very necessary - high accuracy readings from a probe, the probing device must be calibrated. When the probe touches the contact point on the measured surface, a certain force is generated, causing a bend, mostly a very slight one. This small amount of travel that is caused by the bending (or stylus deflection) is called *pre-travel*. Calibrating the probe means to compensate the final reading of the measurement for the amount of the pre-travel. In some respects, this phenomena is very similar to the backlash of a CNC machine. Later in this chapter, basic probe calibration issues will be described.

There are many methods of calibrating a probe, one of the most common ones is touching a special precision gauge located on CNC machine tools (sometimes called *artifact*). During the calibration, the effective size of the stylus is determined (as compared to its actual size). Also during the calibration, it is very important to calibrate the probe stylus in *all directions* that will be measured. Keep in mind that the direction of the pre-travel motion is critical to accuracy of the final measurement.

Calibration is necessary not only for a new probe, but in many other instances as well. The following list should help to make an informed decision when a probe calibration is necessary:

- New probe or new stylus of the probe has been installed
- Start of a new machining job that employs probing
- When a probe or stylus had been replaced or repaired
- When the ambient operating temperature has changes dramatically
- When probing feedrate had been changed
- When repeatability deviates from expected values by an unexpected amount

Other considerations may also have an effect.

Feedrate and Probing Accuracy

Note the above comment relating to the feedrate. All probing should be done at the same feedrate in all directions. Some macro programmers prefer to 'hard-code' the feedrate into the macro, others use a variable for the same purpose. Either way, if the feedrate had been changed, the calibration process should be repeated.

Also a very practical approach is to disable the feedrate override on the CNC operating panel, so the feedrate is always guaranteed to be 100% of the programmed value. To disable the feedrate override in the macro, use the system variable `#3004=2` (*feedrate override disabled*) or `#3004=3` (*feedhold and feedrate override are disabled*). Other variations are also possible. Make sure to program `#3004=0` at the end of the macro, to remove the restrictions when not needed.

Probing Devices on CNC Machines

Any machine shop can benefit from measuring the part during a machining cycle. The probe unit is mounted in the tool magazine, with its own number and offset settings, just like any other tool. The major difference from other tools is that a probe does not rotate in the spindle when it is used. In-process gauging is closely tied to the macro program that controls the machining, measuring and adjustments of offsets in the control system. Macro features such as branching, conditional testing, looping, use of different variables and access to control features, are all important for programming the various probing devices.

In-Process Gauging Benefits

There are many benefits in measuring the various part features during the machining cycle - they all relate to the increased productivity and overall accuracy. The most important benefits can be summed up into several items:

- Part location, length, and diameter of the cutter used can be measured automatically
- All three offset memory groups can be calculated automatically and corrected as needed during machining, individually or collectively
- Reduction of machine idle time - setup of the part can be greatly simplified - there is no need to spend time on exact physical setup. The actual position of the part and its alignment with the machine axes and/or fixture datums can be corrected mathematically rather than physically
- Reduction in the level of scrap - since the actual machined dimensions are monitored by the macro program and the probing cycle, any required offset correction is made automatically
- Inspection of the first finished part does not require its removal from the machining area
- Broken tools can be detected and proper action followed as specified by the macro program
- Initial investment in the technology (equipment and skills) is returned much faster than other methods
- CNC operator's confidence level is increased and unattended machining can become a reality

Types of Probes

Various manufacturers offer many different models of probes. When selecting probes, the main issue is, of course, the probe accuracy. However, accuracy of a probe is not usually an issue, which means the customer looks for additional features when selecting probes. In this context, it is very important to understand that a probe in itself does not measure, so the question of probe accuracy is purely academic.

Probe can be evaluated in terms of repeatability, which does influence the accuracy of the probing system. In terms of accuracy, the most important aspect is the accuracy of the complete system, with all its parts, not just its individual components. When selecting probing devices today, the most important feature is the probe purpose - current one and one in the future.

Probe Size

In the touch-trigger method of probing, the size of the probe is generally determined by the nature of the work. Obviously, the probe must fit into the area to be measured. That means a small diameter probe will allow access to more features of the part, such as crevices and small openings. Large probe sizes are usually part of a generally heavier configuration, and a slightly lower system accuracy should be expected.

Probe Selection Criteria

Not all probes are created equal - there are several considerations that must be made when selecting a probe for in-process CNC measurement (gauging). The considerations relating to the probe selection can be summed up in the following few groups:

- | | |
|---|--|
| <input type="checkbox"/> Machined part | <i>... its size and shape</i> |
| <input type="checkbox"/> Control system capabilities | <i>... standard and optional features</i> |
| <input type="checkbox"/> Expected tolerances | <i>... engineering data - are they realistic ?</i> |
| <input type="checkbox"/> Additional and optional features | <i>... may be beneficial in the long run</i> |
| <input type="checkbox"/> Associated costs | <i>... initial costs as well as ongoing costs</i> |

Machined Part

Without a doubt, the nature of the work determines what probe or probes will be used, based on the current technology. The initial nature of the work may require different probes for different jobs. Although most probes can be exchanged with relative ease, the measuring system always has to be able to accommodate all varieties. That means looking ahead to future part designs and be able to establish the need for a measuring system based on current and future requirements.

Areas of focused consideration should also include the actual part size, its geometrical complexity (general shape), the features on the part that require measuring (critical features), the actual probe size (small or large), and its accuracy, if used with a long stem, as well as its capability to be positioned normal (*i.e.*, perpendicular) to the measured surface.

There are also other factors related to the part, such as its material and the thickness of the measured feature. Some soft materials, such as plastics, may be deflected or even deformed by the probing device, so other techniques will have to be used. The most common method in this type of application is a non-contact probe, offered by all major probe manufacturers. Often a twin probe (one with two styluses, sometimes called a bullhorn probe) can be equipped with one contact and one non-contact probe. Some probing heads can be equipped for both types, and changed either manually or automatically. Automated probe change is particularly attractive to high volume manufacturing (such as in the automotive industry).

Control System Capabilities

When the probing device is applied to CNC machines, the capabilities of the system are a very important consideration. Typically, the most mistakes are made at the time of purchase by failing to establish future needs. Many probe manufacturers offer upgrades, but after a few years the compatibility may fade away. It may be difficult - or even impossible - to upgrade a simple equipment at a later date to a more sophisticated equipment.

Equally important consideration are the specifics of the CNC machine. Small parts of today may not cause problems, but large parts of tomorrow may not fit into the thinking of today.

Expected Tolerances

Design engineers place tolerances on critical dimensions. In a brief reminder, a dimensional tolerance is the allowed deviation from the nominal size specified in the engineering drawing. In probing, particularly during the probe selection process, it is important to realize that tolerances too tight (those with a very small allowable range of deviation) make the machining process much more expensive in all respects. However, if the engineering need is there, the cost has to be acknowledged and absorbed. For tight tolerances, probes have to be selected with this initial consideration. Tight tolerances also require longer measuring times, because many more points or hits have to be measured to guarantee such accuracy. For example, a tight tolerance requirement on a hole diameter will require more than the standard three points to measure the diameter.

Additional and Optional Features

Many optional features are available for various CNC probing devices. Some of the most important features are the fixed orientation of the probe versus its capability to be oriented into different (non-standard) directions. Additional flexibility is welcome, if its costs can be justified.

Associated Costs

There are different types of probes available and - more often than not - a higher price indicates a more sophisticated, higher level, probing device - a better device. Always check the unit features and future service availability from the manufacturer or the vendor.

CNC Machine Probe Technology

Every probe installed on the CNC machine must be able to communicate with the control system of the machine tool. On CNC machining centers, the probe is mounted in a holder similar to the tool holder for standard cutting tools, also stored in the tool magazine, and is placed into the spindle the same way as any other tool, when required. On the CNC lathes, the probe is mounted into the tool turret, also occupying one station. In either case, the purpose of the probe is to touch the desired feature on the part, make the measurement and transmit a signal to the control system with the results. There are three major transmission methods available for the triggered signal:

- ◆ **Optical Signal Transmission**
- ◆ **Inductive Signal Transmission**
- ◆ **Radio Signal Transmission**

In the later development, an infrared method of signal transmission has also been available from several manufacturers, although the optical and the inductive probing methods form the majority of applications.

Regardless of which method of signal transmission is applied, the transmission system uses three important components:

➤ **Component 1:**

Probe and the probing module, are mounted on a stem. The optical and the radio method of transmission both work on the same principle - the module receives the signals from the control system and transmits the signals from the probe as well as the status of the internal battery that provides the necessary voltage. The probe and the probing module can be used in a standby mode, or a continuous operating mode. If the setting is a standby mode, the unit acts only as a receiver that is waiting for the transmitting signal. Once the signal is received, the standby mode is automatically changed to the continuous operating mode. Once the operating mode takes over, the status signals from the probe and battery are transmitted to the machine communication module.

➤ **Component 2:**

Machine communication module, used to establish contact with the probe module. Through signal transmission, a cable and its unique wiring configuration, the machine communication module will be power connected to the machine interface unit.

➤ **Component 3:**

Machine interface unit is used to convert the received signals from the probe into a format the control system can interpret. Using the interface unit, several light indicators show the probe and battery status.

Optical Signal Transmission

In the optical signal data transmission, an infrared light beam is used to transfer probe signals collected at the time of contact, from the probing device to the CNC system. Light emitting diodes (LED) installed on the probing device emit signals towards a pre-tuned receiver. The receiver may pickup signal from as far as 10 feet (3 meters). The power source for the probes that use optical signal data transmission is a small battery installed in the probing device body. A small disadvantage of optical signal transmission system is the requirement of a clear light path between the probe and the CNC control. The majority of the CNC machining centers installed today use the optical method of data transmission.

Inductive Signal Transmission

In the inductive signal data transmission area, transmitted signals use electromagnetic induction as they are transferred across a small air gap. The air gap is between the two induction modules, one located on the probing device, the other located on spindle of the CNC machine tool. Both are linked to the CNC system. The inductive signal data transmission is also quite common on CNC machining centers and CNC lathes. Probably the major benefit of the inductive signal transmission probes is the ease of maintenance and the lack of a battery power. The probe module receives power from the machine module and passes back various probe signals.

Radio Signal Transmission

In the radio signal data transmission, the probing device generates a radio frequency signal. The power source for the probes that use radio signal data transmission is a small battery installed in the probing device body. This method of transmission is commonly used on large CNC machines. Using a radio signal data transmission is very practical on CNC machines that are physically large, and in situations where an optical system would not be suitable to function properly over a distance greater than about 10 feet (3 meters).

Probing systems using radio transmission are based on the principle of high frequency radio waves that carry the data signals. The signals are carried between the probing device and the CNC system of the machine tool. Additional benefit of radio signal transmission is the elimination of the direct and clear path of the signal, needed for other probing methods.

In-Process Gauging

In-process gauging has already been mentioned several times in this chapter, but always in a rather oblique way. The next few paragraphs will attempt to throw some light and more details of this extremely important subject.

For many unmanned machining stations, for example in an FMS cell (*Flexible Manufacturing System*), or similar cellular manufacturing, a provision must be made in the CNC program to allow the checking and adjusting of critical dimensions directly on the part, preferably while it is still mounted in the fixture. As the cutting tool wears out, or because of many other causes, the expected dimensions may fall into the 'out-of-tolerance' zone. Using a probing device and a suitable macro program, the *In-process gauging* option offers a very satisfactory solution. The CNC program for the *In-Process Gauging* option will contain some quite unique format features - it will be written parametrically, and will be heavily dependent on the usage of macro programs, often very deeply nested.

If a CNC machine shop is a user of the *In-Process Gauging* feature, there are good chances that other control options are also installed and available to the CNC programmer. Some of the most typical options are probing software, tool life management, macros, various detectors, tool length and tool radius readers, etc. Some of this technology goes a little too far beyond standard CNC programming, although it is closely related. Companies that already use the CNC technology successfully, will be well advised to look into these options to remain competitive in their field.

The idea of making the first manufactured part exactly to specifications developed into the implementation of in-process gauging. In some cases, this method of measuring can totally eliminate off-line measuring systems (CMM), or at least complement them. By employing the full in-process gauging exclusively at the machine, CMM systems can be often eliminated (at least for certain applications) and the cost and downtime of the off-line measurement is also eliminated. There are several technological requirements for this technology to be successful.

- The probing system must be used in the toolholder, just as any other tool
- The probing system is stationary (non-rotating) and often locked in an oriented position
- Macro program option has to be available within the control system
- Proper interfaces between the probe and control system have to be established
- Special macro programs have to be developed and maintained for the measuring

In terms of economics, in-process gauging does extend the total cycle time, often quite significantly. Even if not every part in the batch is measured, the average cycle time must be considered. When the CNC machine tool is used as a measuring device, the equipment involved in the various stages of measurement has to be certified or calibrated. Calibrating of the probing device is done on a verified gauge (gage), using a macro program. The principles of calibration have been discussed earlier.

Many CNC machine tools incorporate a unique design that allows for the inclusion of a calibrated artifact within the working cube of the machine. Working cube is defined by the combined maximum amount of travel along the X-axis, Y-axis and Z-axis. If only two axes are considered, the term 'working cube' is changed into a working area or a working envelope.

Features to be Measured

In order to determine what features of the part can be measured on a CNC machine, a great deal depends on the type of probe used. In their basic applications, virtually every probe can measure the following features of a part or within a particular part:

- Center measurement
- External diameter
- Internal diameter
- External length - width
- Internal length - width
- Depth of a feature
- Angle of a feature

There are many other part features that can also be measured and some are more common than others. The most typical item in this group covers the part feature location, such as a center of a hole, distance between two points, diameter, and so on. Since the typical part feature is normally specified by at least the X and Y axes, it means a macro development that uses a combination of the various results and manipulating them mathematically. All this takes place within the macro body. In many cases, directly returned value are not used as such, but for calculations of a difference in values between two measurements. How this difference is handled depends on the particular application. Normally, the calculated value is used to adjust a particular offset setting.

There are many other simple and complex measuring methods available to probing macros, but the basic concepts remain the same at all times, regardless of the measuring method used. In the next few application examples, various common options will be explored in some detail, along with the evaluation of typical probing procedures. Included diagrams always show the measured object (feature) and the points of contact with the measured object. For example, P1, P2, P3, P4, etc., indicate the probe position at the time of contact along the selected axis, the identification *Cn* indicates a center location number, the letter *W* indicates the width or length (either one can also be applied to the feature depth), the letter *D* specifies the measured depth, and so on. In most probing cases, the probe diameter (commonly known as the *stylus diameter*) must always be known, but in some calculations, the stylus diameter is not an important at all (any reasonable diameter can be used). Most calculations often use the **ABS** macro function, in order to guarantee a returned value that is always positive. In any macro, the positive and negative values of a calculation (the returned value) are very important and must always be applied correctly.

Center Location Measurement

Measuring the center location (position) applies to measurement of flat and circular objects and part features, such as flat walls, edges, holes, bores, rods, shafts, grooves, even cones and tapers. The measuring also includes cylindrical calibrating devices, if required. In terms of implementation, the center location measurement is probably the most common probing operation in macros.

Figures 57 and 58 illustrate one significant mathematical principle that is the very basis of measuring many center locations, especially those applied to part features related to walls (sides) of the part (external and internal).

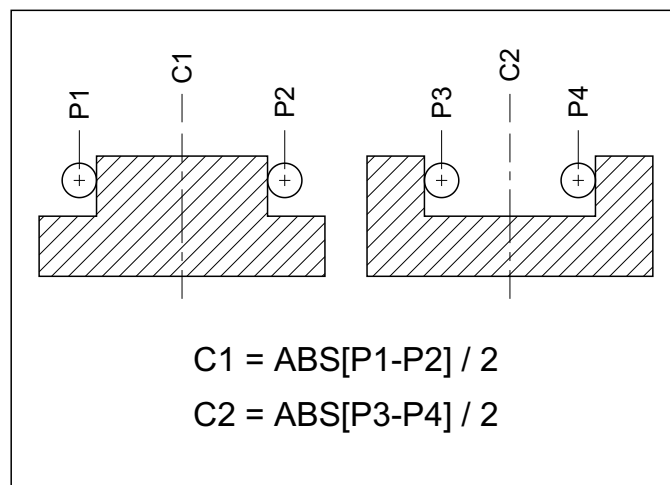


Figure 57

Center measurements C1 and C2 are between two machined features, such as walls:

External (C1) and Internal (C2) walls

*Stylus diameter is not critical
... single axis check is shown*

In both illustrations, two objectives are implied that relate to macro program development. One objective is to find the width measurement between two *external* walls, shown as C1 dimension in the left illustration above. The other objective is to find the width measurement between two *internal* walls, shown as C2 in the right illustration above. The drawing example in *Figure 57* illustrates two parts that are independent of each other and solves the C1 and C2 calculations at the same time. Note that the probe ball diameter (stylus diameter) has no effect on the result.

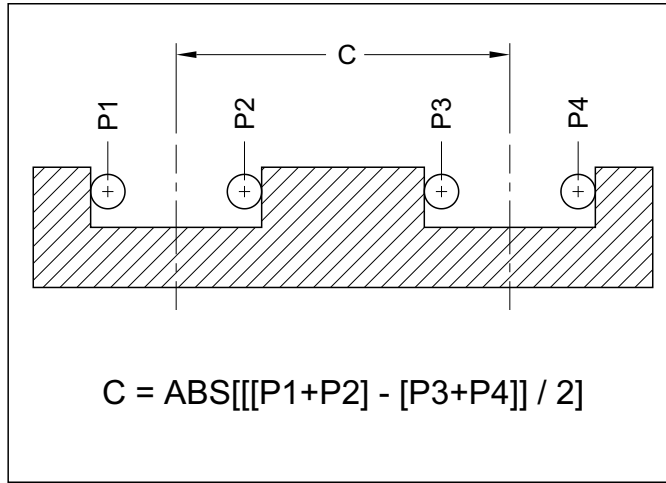


Figure 58

Center measurement C is between two machined features, such as slots

*Stylus diameter is not critical
... single axis check is shown*

A similar approach is equally logical when trying to find the center of a circular feature (radius or diameter), whether it is external or internal. Although one example is shown for finding the center on an external feature (such as a rod, boss, spigot, etc.), and the other example is shown for the internal application (such as a hole diameter), they share the same mathematical formula that can be used in the macro to calculate the center of a circular feature, regardless whether it is external or internal. The formula applies to any single axis.

Figure 59 and Figure 60 illustrate the mathematical principle behind measuring a center location as applied to features related to external or internal part diameters.

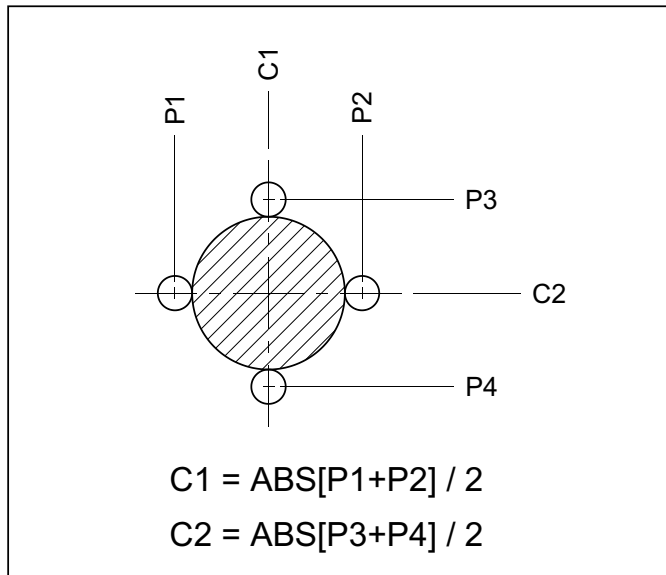


Figure 59

Center measurement C1 and C2 of a circular external feature

*Stylus diameter is not critical
... two axes are shown*

In the previous center finding application applied to two linear features, either axis could be used in the formula, and only one axis measurement was required. The most important element in this example is that only two measurements along a single axis are required, whereby in the two circular examples, an actual point is needed, so both axes must be considered in the calculation of the center of a circular object and four measurements per diameter are required.

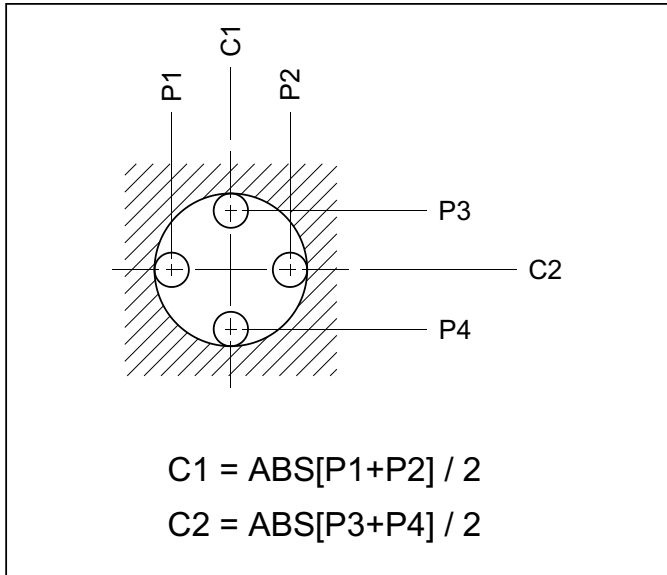


Figure 60

Center measurement C1 and C2 of a circular internal feature

Stylus diameter is not critical ... two axes are shown

As both illustrations show, the calculating formulas are identical. The only difference is the measuring direction, which has to be incorporated in the macro.

Measuring External or Internal Width

The external (outside) or internal (inside) length or width between two features is established by picking and registering two established point positions, one on each end of the measured object. Only a single axis is used for this purpose, the other axes remain idle. Two registered positions (axes X or Y) are normally required. The actual width is found by subtracting one measured position from the other, considering the styles radius.

Figure 61 shows the external feature width measurement on the left and internal measurement on the right - stylus diameter B is very important in both types of measurement.

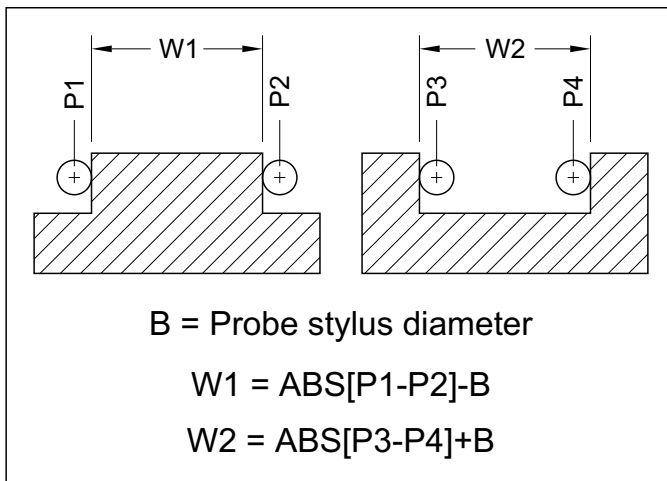


Figure 61

External and internal width measurement

Measuring Depth

Another commonly measured feature of a part is the depth. Depth is normally associated with the Z-axis, but the probing method can be used along the X or Y axes as well, for example, to measure a shoulder depth or a step depth.

Depth is measured by subtracting two measured positions along the axis. Logically, the depth is established in the same way as the external or internal length (width).

Figure 62 shows the mathematical principle behind the depth calculation.

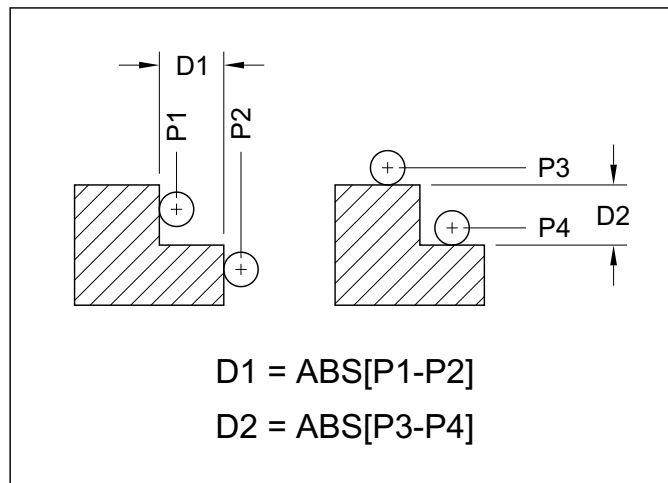


Figure 62

External and internal depth measurement

Other features can be measured by applying the same logical approach. Amongst them, measuring external or internal diameters and measuring angles are the more common applications.

Measuring External Diameter

External (outside) diameter can be a stud, a core, a boss, a spigot, or any other round object that is an external cylinder, including a calibrating device. To measure an external diameter, measured points on the diameter have to be established. Each measured position is registered and the external diameter is calculated mathematically, through the formulas in the CNC macro program. An approximate external diameter must always be known. Typically, an external diameter is measured by establishing three points on the diameter.

Measuring Internal Diameter

Internal (inside) diameter can be any hole, such as a counterbore, circular pocket, or any other internal cylinder, including a calibrating device. To measure an internal diameter, measured points on the diameter have to be established. Each measured position is registered and the internal diameter is calculated mathematically, through the formulas in the CNC macro program. An approximate internal diameter must always be known. Typically, an internal diameter is measured by establishing three points on the diameter.

Measuring Angles

Angle measurement can be used for many useful purposes, one of them being to adjust the coordinate system of the machine tool by rotating it. To illustrate the idea, think of the ease of programming a rectangle where all four sides are parallel to the axes, then programming the same rectangle located at an angle within the machining area. Rotating the coordinate system allows the simplicity of programming a straight rectangle, but locating it at an angle during the machine setup. By rotating the coordinate system to match the part orientation, the setup can be done very efficiently. Fanuc offers optional feature called *Coordinate System Rotation (G68-G69)*. Macro is the best choice if this option is not available.

Changing of Set Values

Since probing is controlled by the CNC macro program, the macro can contain decisions based on the result of the probing process. For example, a tolerance can be entered in the macro body, the probing result registered, evaluated and compared with the stored values. Decision whether to adjust the offset, recut the part, or even reject it, can be made automatically. This is a very sophisticated method of programming that requires a lot practice.

Calibration Devices

Calibration devices - they are sometimes called *artifacts* or *master gauges* - installed on a CNC machine tool are - in their simplest definition - the master reference for all other references. In a more technical explanation, such a device is a physical substitution for the actual machined part whose fixed position within the working cube has been established previously, under precise conditions, such as controlled temperature and humidity. This may sound a bit complicated, but it does indicate the main purpose of a calibration device.

In general terms, there are two basic types of calibration devices design:

Calibrating device - Type 1

Previously established physical substitution of the actual machined part. The benefit of this type is that it can be calibrated directly on the CNC machine and results in a very high precision of the whole machine measuring environment.

Calibrating device - Type 2

A general purpose reference item, such as a high precision calibration sphere or a calibration block. Either design is used by measuring against known relative sizes and locations of the calibration device, for example walls of a cube or a top surface of a sphere.

The main purpose of the calibration device is to evaluate the integrity of the CNC machine tool geometry, before any cutting tool is applied to the production. In addition, the device also serves as means to compensate for even a very slight expansion or shrinkage of the measuring system, due to the effects of heat or cold.

Checking the Calibration Device

During the machining cycle, the critical sizes (features) of the part are established by comparing the desired dimensions with the previously calibrated feature. Typically, the calibration device is measured before any actual machining and any deviation error is established at start. This error can be caused by several factors, mainly those relating to extreme heat or extreme cold.

Once the integrity of the calibration device has been established, the machined part is measured. The result of the part measurement is considered erroneous, if the errors detected during the probing process are compared with the dimensions of the calibration device and found outside of specified limits. This is an important consideration, because it shows that the accurate measurement of the part is determined by the accuracy of the calibration device setting, rather than the whole machine tool system.

Centering Macro Example

As a practical example of a macro using a probing device, the following centering macro is one of the most common applications of a probe. Its purpose is to find a center of a circular object, typically a calibrating ring or a hole in the part that has to be measured. The center of the circular object will be used as the new setting of **G54** work offset (macro can be modified for any other work offset number).

```

00032 (MAIN PROGRAM)
N1 G21                               Metric input
N2 G17 G40 G80                       Startup block
N3 G90 G00 G54 X0 Y0                 XY motion to the center of hole (near center must be known)
N4 G43 Z25.0 H19                     Clear position above work
N5 G01 Z-5.0 F250.0                 Feed to probing depth (*)
N6 G65 P8112 D175.0 F80.0           Macro call with assignments D=measured dia, F=feedrate
N7 G00 Z25.0                         Retract above work
N8 G28 Z25.0                         Return to machine zero (Z-axis)
N9 M30                               End of main program
%

08112 (CENTERING MACRO - METRIC)
(USES 6 MM DIAMETER BALL)
IF[#7 EQ #0] GOTO996                 Check if measured diameter is missing
IF[#9 EQ #0] GOTO997                 Check if probing feedrate is missing
IF[#9 GT 100.0] GOTO998              Max. recommended probing feedrate is F100.0 mm/min
#3004 = 2                             Feedrate override disabled
#10 = #4003                           Store current setting of G90 or G91
#7 = #7/2                              Change diameter of input to radius
#101 = #5041                           Store current X-axis coordinate
#102 = #5042                           Store current Y-axis coordinate
M51                                    Turn Blast ON to clear probe - M-code will vary
G04 X2.0                               Allow 2 seconds for the air blast
M52                                    Turn Blast OFF - M-code will vary

```

```

G91 G01 X[#7-5.0] F[#9*4]      Move 5mm away from right side target (X+ motion) (*)
G31 X[#7+5.0] F#9              Skip the rest of X motion upon contact (X+)
#103 = #5061                    Store X-position at skip signal (X+)
G90 G01 X#101 F[#9*4]          Move back to the start position in X
G91 X-[#7-5.0]                 Move 5mm away from left side target (X- motion) (*)
G31 X-[#7+5.0] F#9             Skip the rest of X motion upon contact (X-)
#104 = #5061                    Store X-position at skip signal (X-)
G90 G01 X#101 F[#9*4]          Move back to the start position in X
#105 = [#103+#104]/2           Average the X+ and X- reading
#106 = #101-#105               Calculate the shift amount for G54 along the X-axis
G91 G01 Y[#7-5.0]              Move 5mm away from top side target (Y+ motion) (*)
G31 Y[#7+5.0] F#9              Skip the rest of Y motion upon contact (Y+)
#107 = #5062                    Store Y-position at skip signal (Y+)
G90 G01 Y#102 F[#9*4]          Move back to the start position in Y
G91 Y-[#7-5.0]                 Move 5mm away from bottom side target (Y- motion) (*)
G31 Y-[#7+5.0] F#9             Skip the rest of Y motion upon contact (Y-)
#108 = #5062                    Store Y-position at skip signal (Y-)
G90 G01 Y#102 F[#9*4]          Move back to the start position in Y
#109 = [#107+#108]/2           Average the Y+ and Y- reading
#110 = #102-#109               Calculate the shift amount for G54 along the Y-axis
(-----)
#2501 = #2501-#106             Update the X-coordinate for G54 work coordinate system
#2601 = #2601-#110             Update the Y-coordinate for G54 work coordinate system
(-----)
#3004 = 0                       Feedrate override enabled
G#10                             Restore original G90 or G91
GOTO999                          Bypass error messages
N996 #3000 = 106 (DIAMETER MISSING) Issue alarm if measured diameter missing
N997 #3000 = 107 (FEEDRATE MISSING) Issue alarm if feedrate missing
N998 #3000 = 108 (FEEDRATE TOO HIGH) Issue alarm if feedrate too high
N999 M99                          Macro end
%
```

Clearances marked with (*) must be greater than the radius of the probe stylus

Note that the specified feedrate applies to the probing feedrate, not the positioning feedrate. Arbitrarily, the positioning feedrate is four times greater, but will not exceed 500.0 mm/min. Flexibility can be added to the macro for more precise control in this area. This is a working macro and satisfies the goals set earlier. The possible changes or improvements to the macro would be handling of the probe ball diameter (radius), feedrate, and the clearances before reaching the target position.

With this probing macro sample, virtually any other probing macro can be developed. Many features of the sample macro will not change from one type of probing to another. The major change will be in the inclusion and manipulation of different formulas.

One critical command that is unique to macros is the **G31** command - *Skip Command*. It will be explained at the end of this chapter.

Probe Length Calibration

Probes are used not only for measuring within the XY plane, they can also check depth, along the Z-axis. Just like centering macro (center position calibration) is designed to update the work coordinate system, this macro sets the *tool length offset* for the Z-axis. There are several methods available, however their basic functionality is the same. *Figure 63* shows the setup for a vertical CNC machining center.

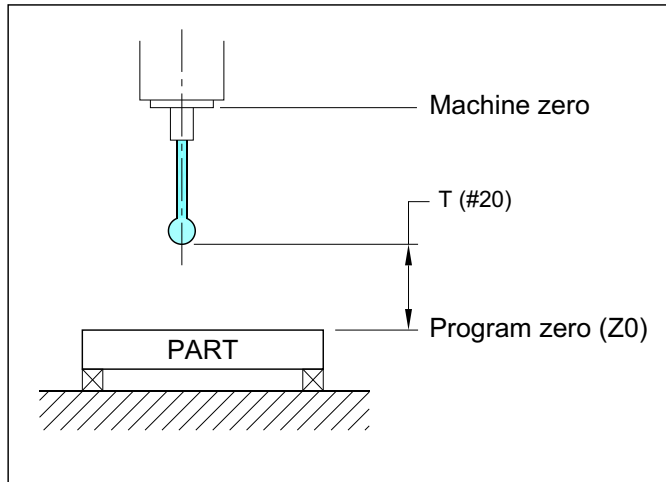


Figure 63

Probe length offset setting

The assignment of variables will be simple. Strictly speaking, only a single variable will be required - the tool length offset number - where the measured length will be stored. For more flexibility, also included can be the Z-axis position to measure, although it will normally be zero. The skip function **G31** will need a feedrate and a small amount of extra travel. The macro will have the feedrate built-in, so the probing will be consistent for all jobs. The extra travel amount (known as the *overshoot*), will also be built-in, which makes it easier to develop the macro for both metric and inch units of measurement.

```

O0033
N1 G21
N2 G17 G40 G80
N3 90 G00 G54 X300.0 Y250.0 T99
N4 M06
N5 G65 P8113 Z0.0 T99
N6 M30
%
```

Main program example
Units of measurement selection
Startup block
XY-position for the probing - also calls Tool 99
Tool 99 to spindle
Call macro for Z0 and tool length offset 99
End of main program

```

O8113 (PROBE LENGTH OFFSET)
(***) DO NOT CHANGE SEQUENCE NUMBERS (***)
IF[#20 EQ #0] GOTO99
G40 G80 G49
IF[#26 NE #0] GOTO98
#26=0
```

Alarm issued if offset number is not assigned
Startup block reaffirmed
Check if Z-position is assigned
If Z-position is not assigned, it defaults to Z0.0

N98 #3004 = 2	<i>Disable feedrate override</i>
#11 = #4001	<i>Store current G-code of Group 01</i>
#13 = #4003	<i>Store current G-code of Group 03</i>
#16 = #4006	<i>Store current G-code of Group 06</i>
IF[#16=20.0] GOTO20	<i>Check if main program is in inches</i>
IF[#16=21.0] GOTO21	<i>Check if main program is in millimeters</i>
N20 #32 = -0.25	<i>Set extra travel in inches</i>
#9 = 2.0	<i>Set probing feedrate in inches per minute</i>
GOTO100	<i>Bypass metric values if inches selected</i>
N21 #32 = -6.0	<i>Set extra travel in millimeters</i>
#9=50.0	<i>Set probing feedrate in mm/min</i>
N100 (PROBING STARTS HERE)	<i>Start of probing routine</i>
#33 = #26+#32	<i>Calculate the final Z-position</i>
G90 G31 Z#33 F[#9*2]	<i>Make the initial probe touch at a faster feedrate</i>
G91 G00 Z[ABS[2*#32]]	<i>Retract twice the amount of stored extra travel</i>
G90 G31 Z#33 F#9	<i>Make the final probe touch at a slower feedrate</i>
#100 = #5063	<i>Register Z-position at skip signal</i>
#[2000+#20] = #100	<i>Transfer the new value to the selected offset</i>
GOTO999	<i>Bypass alarm message if processing normal</i>
N99 #3000=99 (OFFSET NUMBER MISSING)	<i>Issue alarm if offset number not assigned</i>
N999 G91 G00 G28 Z0	<i>Return Z-axis to machine zero</i>
#3004 = 0	<i>Enable feedrate override</i>
G#11 G#13 G#16	<i>Restore previous G-codes of Groups 01, 03, 06</i>
M99	<i>End of macro</i>
%	

Note a few special programming techniques used in the macro. Even when developing a similar custom macro, some of these programming techniques come very handy in the development process of any macro. Some changes may be small, such as change of the variable number that stores the offset, if the control requires it.

Here are descriptions of the techniques used:

Technique 1

The technique that may be controversial is the Z-axis value. If the Z-position is missing in the **G65** call, the macro will define the variable **#26** as zero, automatically. Some programmers may question this approach and choose not to agree with it. Under proper conditions, there is nothing wrong with this technique.

Technique 2

The **IF-THEN** shortcut has been avoided, so the macro is more flexible for a variety of control models (not all models support the **IF-THEN** functions). More **GOTO**n statements will be needed.

Technique 3

The macro will work equally well for metric probing or probing in English units.

Technique 4

Inch and metric values have been built-in. Once verified as suitable, they will be the same for every probe that will need to be measured along the Z-axis. This is one adjustment that may need to be done, to suit particular probing setup.

Technique 5

Controlled feedrate has also been built-in (metric and inch). The reason is that to get consistent results in probing, consistent feedrate is mandatory. Once optimized for the best performance, it will remain the same for any probing.

Technique 6

The probe touched the part twice - once to reset its position in the stem (if necessary), the second time it was the actual probing that was registered as offset.

Other techniques have been used in this and other examples throughout the handbook. The main objective here was to show some probing macros and explain their design. There was no attempt to describe the actual workings of probing devices from different manufacturers. Many manufacturers offer their own macros, suitable for their equipment.

Skip Command G31

Through the probing macros, there was special G-code used - one that is not a part of any other standard program or macro designed for machining. This command is **G31**. In the manuals, it is usually described as the '*skip command*' or '*skip function*'. In many ways, this command behaves the same as the linear motion **G01** - so why cannot **G01** be used? The answer is simple. During a **G01** motion, the target position is established by the XYZ coordinates and the motion takes place at a programmed feedrate. During **G31** motion (also at a programmed feedrate), the XYZ target coordinates are also established. The difference is in the result. **G01** command will normally complete the motion to the target position. This is not acceptable for probing, because the target position is always inside the material and the probe would crash. The target position must be inside of the material, otherwise there would be no guarantee that the probing would take place.

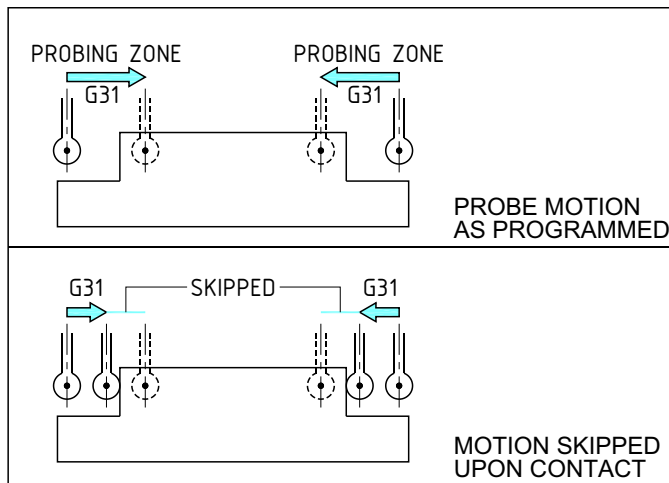


Figure 64

G31 Skip Command

Figure 64 illustrates the general concept of the skip command.

In the top part the programmed motion and direction is illustrated. Note the end of motion is in the material. The bottom illustration show what happens when the probe makes a contact with the material. It skips the remainder of the motion and registers the position, as per macro instructions.

The amount of motion into the material does not have to be large, but it must be greater than the largest dimensional deviation expected.

24

ADDITIONAL RESOURCES

Many valuable resources have been provided in this handbook. No single publication can cover all details of the topics presented, particularly those as complex as *Fanuc Custom Macros*. Several additional resources are suitable to include at the end. One deals with the major macro restrictions and limitations when it is processed at the CNC machine, others suggest additional reading.

Limitations During Macro Execution

This handbook covers the subject of macro program development in a fair detail, which means it has been restricted to the coverage of various programming methods and special techniques. An experienced CNC operator who works with macros on a daily basis knows that there are many significant differences that are unique to macros during their execution (processing):

Single Block Setting

In many ways, the single block switch setting (ON/OFF switch) at the control panel works the same way for macro processing as it does for processing of standard programs. There are exceptions - these are the most notable:

- ◆ **Macro call commands G65–G67** ... *will not stop in single block mode*
- ◆ **Mathematical (arithmetic) expressions** ... *controlled by a parameter setting*
- ◆ **Control commands** ... *controlled by a parameter setting*

As in some other examples, this is one of the control-specific settings and checking the control specifications supplied by the manufacturer is very important.

Block Number Search

When the control system is in the macro execution mode (macro processing mode), the block number search (sequence number search) cannot be done.

Block Skip Function

Block Skip Function is identified in the CNC program with the slash symbol (/). The same slash symbol is also used in macro expression as a symbol for arithmetic division of two values. Normally, the block skip slash code is used at the block beginning, but some controls support a slash code being programmed in the middle of the block. When a macro that contains the slash code for division is processed by a control system that also supports mid-block skip function, the control will evaluate the expression first.

If the slash symbol is part of a mathematical expression enclosed in square brackets, it will be processed as a division symbol, *not* a mid-block skip function:

#31 = 10.0	<i>Initial value set</i>
#32 = [#31/2]	<i>Division operation - #32 = 10.0/2 = 5.0</i>
#33 = #31/2	<i>Mid-block skip - #33 = 10.0 if block skip ON</i>

The simple example shows the possibility of a serious error as a result of not knowing the control system well. Most controls do not allow mid-block skip function, only and the beginning.

MDI Operation

Often it is necessary to test one of more program blocks in the *Manual Data Input* mode (MDI) rather than from the control memory. If the program block to be executed in the MDI mode is a macro call using the **G65 P-** command, the control system will process this request normally and calls the specified macro by its number and assignments, as expected. However, the MDI mode cannot be selected to call a macro program, while the automatic operation is in effect.

Edit Mode

Parameters that control the editing of subprograms or macro programs within the range O8000 to O8999 and O9000 to O9999 can be set to *allow* or *disallow* editing or deletion of programs in these ranges. If a macro (or a subprogram) is proven correct and used frequently, it should be protected by a parameter setting from being accidentally edited or even deleted.

Control Reset

When the *RESET* key is pressed at the control panel, all local variables within the range of #1 to #33 will be automatically cleared. Also cleared will be common variables within the range #100 to #149. When a variable is cleared, it is set to a null value, which means it is equal to #0 (not zero). If it is necessary to keep these variables from being cleared by the RESET key, a parameter can be changed for that purpose. Check the *Parameter Manual* of the control system for details.

Pressing the RESET key also returns the macro processing (execution) to the main program level (top level). In practice, it means all active subprograms, macros, conditions, loops (such as DO statements), etc. will be cleared (cancelled).

Feedhold Switch

The purpose of the feedhold switch (or button) is to stop axis motion in the middle - between the start position and the target position. When the feedhold switch is activated (turned ON) during a macro program execution, the axis motion stops *after* the macro statement has been processed. Unrelated to the feedhold switch setting, the axis motion also stops when the operator presses the RESET key, or when an error condition (alarm) is generated.

Knowledge for Macro Programming

Any publication that offers CNC macro programming is generally considered a publication containing many advanced programming topics. Ideally, only experienced CNC programmers should open this handbook and learn from it. However, that is not always the case, as many junior and less experienced programmers find themselves in the position to develop a macro programs.

All advanced subjects depend on a certain 'basic core' knowledge and understanding of some general principles that are necessary to advance from one level to the next - this handbook is no exception. As a publication, it has been designed to stand on its own - there is no relationship to any other publication or technical text mentioned (all references are only suggestions).

Throughout the handbook, many different subjects were presented including many that have no equivalent in manual programming or even CAM programming. All subjects in this category are new subjects, and the purpose of the handbook was to explain them in details. Hopefully, that has happened. As macro programming is considered a high-level programming, it is heavily dependent on programming practices from various lower levels. These practices are not described in this specialized handbook. For those users who wish to either review or even learn the basics of CNC programming in details, the best selling *CNC Programming Handbook*, also published by *Industrial Press, New York, NY* (www.industrialpress.com), offers all the answers.

In particular, thorough understanding of the following items is absolutely critical as the core knowledge, in order to develop macro programs, even at their lowest level:

- General skills**
- Manual programming experience**
- Math applications**
- Setup practices**
- Machining practices**
- Control and machine operation**

This above list can be expanded quite a bit, but it has addressed the necessary basics as is. Let's evaluate these items individually. Some have been mentioned at the beginning, others are new in this section.

General Skills

Even before entering the field of macro programming, any CNC programmer should possess several skills that are used all the time in the programming process. It should be understood that macro program development is not normally assigned to a person with limited experience, or a person who has no knowledge of the various basic CNC processes and associated machining, including some experience in manual programming. The ability to interpret engineering drawings is important and basic at the same time, but for macros, the ability to see two or more such drawings as possible candidates for a macro development is even more important. There are many features in macro programming that can be closely related to the operation of a fully featured high-level scientific calculator. Building formulas, using variables, programming loops, etc., is no different for a sophisticated calculator then for a macro program running a CNC machine tool.

Manual Programming Experience

Manual programming requires the knowledge of G-codes, M-codes and hundreds of other control supported features that form the structure of any part program. Macro programming method is part of a manual type program development - the only computer involved in the process is the CNC system that processes the macro - a CNC system is not designed to create macros. In this handbook, many macro examples or specific applications have dealt with subjects such as program formats, modal and non-modal commands and functions, subprograms, offsets, fixed cycles, data settings, automatic cornerbreaking, and many others. In essence, these subjects are all part of the standard manual programming. It is impossible to create any significant macro program without a good knowledge of these and other subjects.

Math Applications

CAM style programming needs virtually no math knowledge, while manual programming depends heavily on math. Macros do not require additional math knowledge, only a different way of applying it. Basic arithmetic, algebra, trigonometry, etc., fall into the 99%+ of all math needed in CNC programming. Macros involve math calculations similar to those done on a pocket calculator. As many examples in the handbook have illustrated, mathematical applications are a very important part of macro development.

Setup Practices

How to machine a part for production is a critical skill for any CNC programming, including macro programming. Along goes the knowledge of machine setup, fixtures, tooling, etc. Although macros do not cover any physical setup, many macro activities are setup related at the control level - for example, to assign work offset, tool length offset, cutter radius offset, etc.

Machining Practices

Machining practices include all machine shop oriented subjects that are as basic as general understanding of speeds and feeds, concepts of workholding, fixtures, tools, materials, coolants, etc. Within these skills are other skills, more focused, and more specialized - for example, various machine shop formulas, special machining techniques and operations, unique materials, etc., all help to develop a better macro.

Control and Machine Operation

Operation of a CNC machine is just a little more than an operation of the CNC system. Knowing the control system of a particular machine tool is important to every part programmer or machine operator - knowing the same control system even better and in more depth is absolutely critical to a macro programmer. Even the smallest and least significant feature of the control unit may have significant effect on the macro development, particularly in the areas of defaults, offsets, parameters, system variables, and all other standard and unique features. For any macro development, being familiar with the control and its associated machine is very important.

Complementary Resources

The above section mainly related to the subject of knowledge for macro programming - even while similar topics have already been stated in the first chapter of the handbook. The last section has also presented many new subjects, mainly those covered within the pages of this publication.

Industrial Press, Inc.

For those users that seek either a refresher update or for those that need to learn the basic subjects from ground up, *Industrial Press, Inc.* is an established publisher of many technical books that can help. For the purpose of learning the pre-macro subjects in the CNC programming field, the best-selling *CNC Programming Handbook* offers just about all answers in great details. This very popular publication has been accepted as an excellent in-depth resource for virtually all everyday CNC programming projects.

The complete listing of all *CNC Programming Handbook* chapters is listed on the enclosed CD

Internet

The internet and the world wide web is an excellent source of finding many CNC oriented subjects. There are many resources - some are superior to others, but there are also those that are mediocre and some even outright wrong. Macros belong to all these categories. Most macros posted on the internet are well intended, but may only work for a particular machine/control combination. Many - even those that work - lack any significant documentation. In other words, use the good old *caveat emptor* approach. Internet can be a mine of gold or a pit of gravel.

Practical Programming Approach

No publication that covers the subject as involved as of custom macros (user macros) can thrive on the provided examples only. Hopefully, this handbook has offered a valuable source of information, tips, tricks, shortcuts, as well as thoroughly documented examples relating to *Fanuc Custom Macro B* control option. Although a part of CNC programming for more than two decades, custom macros have been very severely underutilized. Although they are an option of the control system, macros are becoming a very attractive programming method that has a unique place in a CNC machine shop - they enhance current manual programming methods, but they do not replace CAD/CAM systems or conversational type programming. In fact, macros offer features that no other programming method can achieve. Whether new to macros, or just picked up the handbook for reference, you will find that once you start programming with macros, it will be very difficult to go back. That does not mean macros are useful for all applications, but for the suitable applications, macros really bring many superior benefits.

In closing, and in the form of a few final notes, here are some tips summarized from the handbook, tips that may worth keeping handy when developing a macro program:

Macro Programming Tips

- ◆ Always have an objective - decide on the main reason for the new macro
- ◆ A macro should not do everything - shorter macros are better than a large one
- ◆ Plan ahead and plan well - get organized
- ◆ Do not count on various default values and settings
- ◆ Develop a flowchart or at least a pseudo-code before writing the real code
- ◆ Draw sketches, draw views, draw other pictures - visualize each macro stage
- ◆ Assign variables meaningful addresses, if possible
- ◆ Use common variables only when they benefit the macro
- ◆ Write the core of macro first, add 'bells and whistles' when macro is verified
- ◆ Do not sequence every block in a macro - only the reference blocks
- ◆ Do not change sequence numbers in a macro - include warning message
- ◆ Write portable macros - make them compatible with many other controls
- ◆ Write one macro to work for either metric or inch input
- ◆ Watch for endless loops
- ◆ Force alarms for erroneous or missing input
- ◆ Save all current settings before changing them - restore them when macro exits
- ◆ Document macros internally
- ◆ Include programmer's name and date of last revision
- ◆ Protect special purpose macros from editing and deleting
- ◆ Never assume anything - or as they say 'Assume nothing!'

25

MACRO COURSE OUTLINE

The above title should actually read '*Suggested Macro Course Outline*'. Many companies, community colleges and various training facilities do offer different in-house technical courses, developed and conducted by many professionals experienced in the program being presented. This last chapter offers a general curriculum for a comprehensive *Fanuc Custom Macro* training course, based on this handbook. Its purpose is to provide the multitude of important topics necessary to conduct a successful custom macro training program. The outline is only a suggestion and is meant to be modified to suit particular training needs. Table of contents at the beginning of this handbook may help in this respect.

Feel free to adjust the course outline to your needs and fit it to your training schedule. For convenience, the text of the outline is also included on the CD that accompanies this handbook.

Macro Course Outline

Course Title: INTRODUCTION TO FANUC CUSTOM MACROS

Duration: 36 - 42 hours

Prerequisites: General knowledge of manual CNC programming, CNC machining and setup

The typical student is one with a strong knowledge of CNC programming, particularly of the part program structure, G-codes and M-codes, as well as subprograms - these subjects will be of most benefit. Knowledge of a high level programming language is helpful but not necessary. The participating student should also be familiar with the basic operation of a CNC control panel and general machining practices. On an intellectual level, the student should be a quick thinker, able to search for solutions to various problems. The student should also have a strong background in mathematical applications for machine shop.

Course Description and Objective:

This course is the highest levels of CNC Programming training. The student will learn from the beginning, with progressively more advanced subjects relating to the development of customized CNC programs (macros). The main objective of the training program is to familiarize the student with Fanuc macro concepts, their format, structure, as well as the applications in a typical machine shop.

Starting with a brief review of standard CNC concepts, mainly G-codes, M-codes, and subprograms, the student will learn how to understand Fanuc macro structure and develop practical macro applications. The main emphasis of the course will be on correct programming style and applications development for efficient and productive CNC usage.

One of the most important features of this training program is development of actual macro routines the student can use upon completion of the training. Training sessions are designed around various exercises and practical projects the student will be doing.

Access to a control system with the macro option is optional.

Method of Training:

Only an experienced professional CNC instructor will run this course. All sessions in this program will be based on guiding the student to solve a problem, rather than presenting arbitrary solutions. A copy of the comprehensive *Fanuc Custom Macros* handbook (published by Industrial Press, Inc.) will be supplied to each student, to be kept for future reference.

◆ INTRODUCTION TO MACROS

- General Introduction
- CNC Programming Tools
- What is a Macro Programming
- Usage of Macros
- Groups of Similar Parts
- Offset Control
- Custom Fixed Cycles
- Special G-codes and M-codes
- Alarm and Message Generation
- Probing and Gauging
- Shortcuts and Utilities

◆ BRIEF REVIEW OF PROGRAMMING TOOLS

- G-codes and M-codes
- Preparatory Commands
- Miscellaneous Functions
- Default Settings
- Modal Values
- Programming Format
- Rules of Subprograms
- Subprogram Nesting

◆ SYSTEM PARAMETERS

- What are Parameters
- Binary Numbers
- Parameter Classification
- Parameter Data Types
- Setting and Changing Parameters
- Protection of Parameters
- Changing Parameters
- System Defaults

◆ DATA SETTING

- Data Setting Commands
- Coordinate Mode
- Work Offsets
- Memory Types - Milling and Turning
- Geometry Offset
- Wear Offset
- Adjusting Offsets
- Absolute Mode
- Incremental Modes
- Tool Offset Entry
- MDI Data Setting
- Programmable Parameter Entry
- Modal G10 Command
- Effect of Block Numbers

◆ MACRO STRUCTURE

- Basic Tools
- Variables
- Functions and Constants
- Logical Operators
- Defining and Calling Macros
- Macro Definition
- Macro Call
- Arguments
- Macro Program Numbers

◆ CONCEPT OF VARIABLES

- Types of Macro Variables
- Definition of Variables
- Variable Declaration
- Variable as an Expression
- Usage of Variables
- Restrictions
- Custom Machine Features

◆ ASSIGNING VARIABLES

- Local Variables
- Assignment Lists
- Simple Macro Call

- Modal Macro Call
- Main Program and Variables
- Local Variables and Nesting Levels
- Common Variables
- Volatile and Non-volatile Groups
- Input Range
- Protecting Variables

◆ MACRO FUNCTIONS

- Function Groups
- Definition of Variables
- Referencing Variables
- Vacant Variables
- Arithmetic Functions
- Division by Zero
- Trigonometric Functions
- Rounding Functions
- Miscellaneous Functions
- Logical Functions
- Binary Numbers
- Conversion Functions
- Evaluation of Functions

◆ SYSTEM VARIABLES

- Identifying System Variables
- System Variables Groups
- Read Only Variables
- Read and Write Variables
- Displaying System Variables
- System Variables for Various Controls
- Organization of System Variables
- Resetting Program Zero

◆ TOOL OFFSET VARIABLES

- System Variables and Tool Offsets
- Tool Offset Memory Groups
- Tool Offsets and the Number of Offsets
- Tool Offsets and Control Types
- Tool Setting

◆ MODAL DATA

- System Variables for Modal Commands
- Preceding and Executing Blocks
- Modal G-codes
- Saving and Restoring Data
- Other Modal Codes

◆ BRANCHING AND LOOPING

- Decisions in Macro Development
- IF Function
- Conditional Branching
- Unconditional Branching
- IF-THEN Option
- Single Conditional Expressions
- Combined Conditional Expressions
- Concept of Loops
- WHILE Loop Structure
- Single Level Nesting Loop
- Double Level Loop
- Triple Level Loop
- Other Conditions
- Restriction of the WHILE Loop
- Conditional Expressions and Vacant Variables
- Clearing 500+ Series of Variables

◆ ALARMS AND TIMERS

- Alarms in Macros
- Alarm Number
- Alarm Message
- Alarm Format
- Embedding Alarm in a Macro
- Resetting Alarm
- Message Variable
- Timers in Macros
- Time Information
- Timing an Event

◆ AXIS POSITION DATA

- Axis Position Terms
- Position Information

◆ AUTOMATIC OPERATIONS

- Controlling Automatic Operations
- Single Block Control
- M-S-T Functions Control
- Feedhold, Feedrate and Exact Check Control
- Systems Settings
- Controlling Number of Machined Parts

◆ PARAMETRIC PROGRAMMING

- Variable Data
- Benefits of Parametric Programming
- Families of Similar Parts
- Macros for Machining
- Macros as Custom Cycles

◆ PROBING WITH MACROS

- Probing Fundamentals
- In-Process Gauging
- Part Features Measurement
- Calibration Devices
- Sample Program Evaluation

Closing Comments

In no way is the presented program outline offered as the best possible course presentation. In some ways, the outline follows the material in this handbook, but it also deviates from it. Keep in mind that the handbook has been designed mainly as a reference resource, not a particular course material. However, the topics presented can serve as excellent source for building a customized course on Fanuc macros.

Index

A

Abbreviations of macro functions	204
ABS function	122, 284
ABSIO variables	193
ABSKP variables	193
ABSMT variables	193
Absolute mode	59
ABSOT variables	193
ACOS function	116
Additional work offsets	9, 169
ADP function	87, 122, 124
Agile manufacturing	49
Alarms and timers	187-192
Alarm format	188
Alarm generation	8
Alarm messages	187
Alarm numbers	187
Alarms in macro	188
Resetting an alarm	190
Timers	191
Amplitude - Sine curve	185
AND function	176
Angular hole pattern - version 1	221-223
Angular hole pattern - version 2	224-225
Arc hole pattern	233
Arguments	77
Arithmetic functions	113
Nesting	113
ASIN function	116
Assigning variables	93-108
ATAN function	116
Auto mode operations	195-202
Automatic cornerbreak	213
Auxiliary functions	196
Axis position data	193-194
Axis type parameters	43

B

Basic program codes	11-20
Battery power supply	34
Baud rate	270-271
Baud rate setting	68
BCD function	126
BIN function	126
Binary numbers	45, 126
Binary values	200
Bit-type data	37
Interpreting binary value	200
Logical sum	200
Sum of bits	45, 200
Bit	45

Locations	38
Bit type parameters	37
Block number search	295
Block numbers	40, 72
Block skip function	72, 295
Bolt hole circle pattern	229
Boole, George	74
Boolean operators	74, 124, 175
BPRNT function	267-268
Brackets	86, 126
Branching and looping	171-186
Concept of loops	177
Conditional branching	172
Counter in loops	229
GOTO function	173
IF function	172
Unconditional branching	173
WHILE function	179
Byte type parameters	41

C

Calibration devices	288
Centering macro example	289
Circular groove with multiple depth	247
Circular pocket finishing	240, 260
Circular pocket roughing	236, 260
CMM	276
Common variables	83, 106, 133
Protection	108
Volatile and nonvolatile	106
Concept of loops	177
Concept of variables	83-92
Conditional branching	172
Conditional expressions	175-176, 182
Constants	73-74
Control models	138
Control parameters	33
Conversion functions	126
Coordinate measuring	276
Coordinate Measuring Machines	276
Coordinate mode	50
Coordinate system rotation	9, 288
COS function	116
Course outline	301
Current value	112
Curves	
Approximation	185
Sine curve	184
Custom fixed cycles	8
Custom machine features	92
Custom M-codes	12
Cycle Start	4

D

Data output functions	268
Data settings	49-72

Block numbers	72
G10 command	50
MDI	65
Offsets	49
P-address	67
R-address	67
Zeroing machine axes	70
Datum shift	145
Decimal point	87, 229
Decimal point in G65 statement	229
Decision making in macros	171
Default values	11, 33, 47-48, 233
Dimensional tolerances	280
Disallowed addresses	98
Division by zero	115
DNC method	34
Documentation of programs	29
DPRNT function	267-274
Dwell as a macro	192

E

Edit mode	296
Editing macros	203
Emergency stop	94
Empty variables	111, 182
Endless loops	178
ENDn function	179
English units	88
EQ function	182
Evaluation of functions	126
Exact stop check control	197
Executing block	164
EXP function	122, 124
External output commands	267-274

F

FALSE values	37, 125, 172, 176, 182
Family of parts	7, 205, 209-220
Fanuc custom macros	1
Feedhold control	197, 296
Feedrate override control	197
FIX function	121, 136
Fixed cycles	8
Flowchart	177, 208, 300
Formulas in macros	134
Frame hole pattern	224
Functions	73-74, 109
Arithmetic	113
Available groups	109
Conversion	126
Evaluation	126
Logical	124
Miscellaneous	122
Order of evaluation	128
Practical applications	129
Rounding	117

Trigonometric	116
FUP function	121, 128

G

G04 command	12-13, 192
G09 command	12-13, 198
G10 command	13, 17, 50-52, 62, 66-68
G11 command	66, 68
G13 circle cutting cycle	8, 240, 260
G20 command	12-13, 17
G21 command	12-13, 17
G31 command	193, 293
G61 command	14, 18, 198
G65 command	5, 14, 18, 66, 75, 77
G66 command	15, 18, 98
G66.1 command	98
G67 command	15, 18, 98
G68-G69 commands	288
Gauging	3, 49, 275
G-codes	2, 10-11
G-codes for milling	13
G-codes for turning	16
Groups	13, 16, 164, 167
Modal commands	164
Three digit G-codes	15
Types for turning	16
GE function	182
Geometry offset	8, 53, 58, 149
GOTOn function	171
Groups of similar parts	7
GT function	182

H

Hiding macros	9
-------------------------	---

I

IF function	171-172, 182
IF-THEN function	171, 174
Increment system	43-44
Incremental mode	59
Infinite loops	178
Infinite values	128
In-process gauging	3, 9, 49, 276-278, 282
Input/Output	
Interface	133
Metric and inch format	270
Parameter settings	269-271
Printing a blank line	274
Integer numbers	85
Iteration (looping)	30, 179, 181

<hr/>	
K	
<hr/>	
Knowledge for macros	297
<hr/>	
L	
<hr/>	
Lathe offsets	62
LE function	124, 182
Least increment	88
Limitations	295
LN function	122, 124
Local variables	83, 93, 129
Assigning	94
Assignment lists	94-95, 168
Clearing	94
Defining	93
Disallowed addresses	98
Nesting levels	105
Variable as a counter	229
Logical functions	73-74, 124, 175
AND	125
EQ	124
GE	124
GT	124
LE	124
LT	124
NE	124
OR	125
XOR	125
Looping function	179-181
LT function	182
<hr/>	
M	
<hr/>	
M30 function	16, 20, 94, 129
M98 function	16, 20, 25, 75-77
M99 function	16, 20, 25-26, 30, 94, 129
Machining performance	33
Macro call	75
Macro call by G-code	256, 262
Macro call by M-code	258
Macro call command	77
Macro definition	75
Macro execution	295
Macro functions	109-136
Macro in main program	5
Macro programming	4
Macro structure	73-82
Macro unique features	30
Macros	
Calling	75
Course outline	301
Definition	75
Documentation	29
Program numbers	78
Protection	79
Setting definitions	79
Macros as custom cycles	255-266
Macros for machining	224-254
Main program	5, 101
Manual Data Input (MDI)	65, 186, 296
M-codes	2, 10
M-codes for milling	15
M-codes for turning	19
MDI data input	65, 186, 296
Memory	
Nonvolatile	106
Volatile	106
Message generation	8
Metric units	88
Minimal increment	88
Mirror image status check	199
Miscellaneous functions	10, 12
Modal values	12
Executing block	163
G10 command	66
G-codes	164
M-codes	168
Modal data	163-170
Modal macro call	98
Preceding block	163
Saving and restoring	167
System variables	163
M-S-T functions control	196
Multiple level nesting	2, 27
<hr/>	
N	
<hr/>	
NE function	182
Negative variables	89
Nesting levels	179
Double level	180
Single level	179
Triple level	180
Nonstandard tool motions	8
Non-zero variable	89
Null variables	83, 111, 182-183
With arithmetic operations	114
With axis motion	111
Number of machined parts	202
<hr/>	
O	
<hr/>	
Offset control	8, 49
Offset memory types	53, 58
Milling type A	55
Milling type B	56
Milling type C	57
Offsets	
Data setting	49
Lathe	62
Setting adjustments	59
Tool offset entry	60
Updating offsets	54
Valid input range	62
Operating restrictions	295
OR function	176

Out-of-range values 107
 Overflow 107

P

Parameters
 Backup 34, 46
 Changing 46-47
 Classification 35
 Data types 37
 Definition 33
 Display screen 37
 Enable and disable mode 46
 Grouping 36
 Identification 35
 Numbering 35
 Protection 46
 Relationship 40
 Saving 34
 Setting 46
 Parametric programming 7, 205-254
 Benefits 206
 Definition 205
 Planned approach 207
 Variable data 205
 PCLOS function 267
 Pi function 31
 PMC - Programmable Machine Control 126, 258
 Polar coordinates 9
 POPEN function 267, 272
 Port close function 267
 Port open function 267
 Positive variables 89
 Pound symbol 75
 Preceding block 164
 Preparatory commands 10-11
 Probe calibration 277
 Artifact 277, 288
 Feedrate and accuracy 277
 Master gauges 288
 Probe length calibration 291
 Probe pre-travel 277
 Probing 3, 7, 275
 Angle measurement 288
 Center measurement 284
 Definition 275
 Depth measurement 287
 Devices 278
 Diameter measurement 287
 Feedrate and accuracy 277
 Inductive type 282
 Length or width 286
 On CNC machines 280
 Optical type 281
 Probe calibration 277
 Radio type 282
 Selection criteria 279
 Technology today 276
 Touch probes 276
 Types of probes 278
 Probing applications 3
 Probing with macros 275-294

Program numbers 79
 Program portability 67
 Programmable parameter entry 65
 Programming languages 4, 7, 10, 73, 171
 Protecting macros 9, 79
 Protection of parameters 46
 Pseudo-code 208, 300

R

RAM 106
 Random Access Memory 106
 Read and Write variables 138
 Real numbers 85
 Rectangular pocket macro 251
 Redefinition 112
 Replacing control options 9
 RESET key 94, 129, 190, 196-197, 270, 296
 Resetting an alarm 190
 Resetting program zero 145
 Restoring modal data 167
 Restrictions 295
 Returned value 112
 ROUND function 117, 128
 Rounding functions 117
 Rounding to number of decimal places 119
 RS-232C 68, 267-269

S

Safety issues 46-47
 Saving modal data 167
 Scaling function 9
 SETVN function 108
 Shortcuts 9
 SIN function 117
 Sine curve example 184
 Single block 295
 Single block control 195-196
 Skills requirements 10
 Skip command 293
 Slot machining macro 244
 Special cycles 255
 Special G-codes 8
 Special M-codes 8
 Special tapping example 198
 Speeds and feeds calculation 134
 SQRT function 74, 122
 Stylus diameter 284
 Subprograms 2
 Brief review 21-32
 Documentation 29
 General rules 25
 Lathe example 31-32
 Main program and subprogram 26
 Mill example 21-25
 Subprogram nesting 27

Subprogram repetition	27	Variables	73-74
Subprograms vs. macros	30, 248	Addition	114
Substitution	112	Arguments definitions	77
System defaults	47	As expressions	86
System parameters	7, 33-48	Common	83, 106
System variables	83, 133, 137-146	Decimal point usage	87
Alarms	187	Declaration	85, 87
Automatic operations	195	Definition	84
Axis position	193	Division	115
Clearing 500+ series	186	In main program	101
Displaying	138	Input range	107
Fanuc 0 listing	139	Local	83, 93
Fanuc 10/11/15 listing	140	Multiplication	115
Fanuc 16/18/21 listing	141	Non-zero	89
Groups	138	Null variable	83
Modal commands	163	Out-of-range values	107
Modal G-codes FS-0/16/18/21	165	Positive and negative	89
Modal G-codes FS-10/11/15	166	Protection	108
Number of parts	202	Referencing	110
Organization by numbers	144	Restrictions	90
Read and Write variables	138	SETVN function	108
System settings	199	Substitution	114
Timers	191	Subtraction	114
Tool offsets	147	System	83
		Terminology used	112
		Types	83
		Usage	86
<hr/>			
T			
<hr/>			
TAN function	116	W	
Time information	191	Wait for completion signal	196
Timers in macros	191	Wear offset	8, 53, 58, 149
Timing an event	191	WHILE function	171, 179-182
Tolerances	280	WHILE loop restrictions	181
Tool nose tip number	63	Word type parameters	42
Tool offset memory groups	148	Work offsets	51
Type A	148	Additional	52
Type B	149	Common	52
Type C	149	External	52
Tool offset variables	147-162	Standard	51
Fanuc 0	150		
Fanuc 10/11/15/16/18/21	152, 158		
Tool setting	158		
Touch probes	276		
Trigonometric functions	116	X	
Conversions to decimal degrees	116	XOR function	176
TRUE values	37, 125, 172, 176, 182		
Two-word type parameters	42		
Types of probes	278		
<hr/>			
U			
<hr/>			
Unconditional branching	173, 189	Z	
Underflow	107	Zero shift	145
Usage of macros	6		
<hr/>			
V			
<hr/>			
Vacant variables	83, 93, 111, 182		
Value transfer	137		

WHAT'S ON THE CD-ROM ?

The included bonus CD-ROM provides an easier access to most programs covered in the book, including all machining macros and other files, so they can be printed or modified if necessary. Through the menu selection, choose the item of interest.

➤ Requirements:

1. **Windows XP (Home or Professional edition strongly recommended)**
2. **CD-ROM drive**
3. **Adobe Reader version 5 or higher (formerly known as the Acrobat Reader)**

➤ Using the CD-ROM:

1. **Insert the CD into the CD-ROM drive**
2. **The *Main Menu* should appear automatically**
3. **Use the *Main Menu* to navigate through all available options**
4. **Press the *EXIT* button to exit CD browsing**

➤ Notes:

If the *Main Menu* does not appear automatically, the *AutoPlay* function in Windows XP may have to be activated. The free utility *Adobe Reader v5+* is required to view and print all selected PDF files from the CD-ROM. If you encounter a difficulty opening any PDF file from the disk, install the free *Adobe Reader* version provided on the CD-ROM (uninstalling the older version may be necessary).

The CD-ROM also includes all the major programs and macros listed throughout the book. Specifically, the menu selection covers all macros and associated illustrations (drawings) listed in the *Chapter 20 - Macros for Machining*, as well as the macro for a special cycle development listed in *Chapter 21 - Custom Cycles*.

In addition, other menu selections offer several supplementary resources:

- A suggested *Fanuc Custom Macro B* course outline as a *Microsoft Word* file (feel free to modify this file to suit your training needs)
- Several drawings usable for macro development are also included
- Drawings in the PDF format that can be used as handouts or for classroom exercises
- Two probing macros
- Other references

