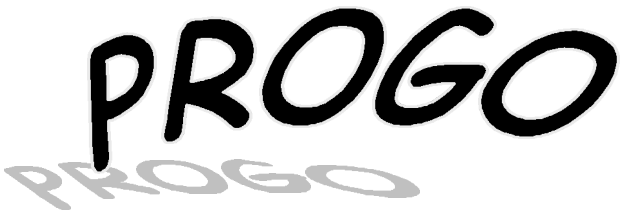


# LANGUAGE REFERENCE MANUAL

## B Y

## Keith L. Doty

Gainesville, Florida Phone 407-672-6780

# AGREEMENT

This is a legal agreement between you, the end user, and Mekatronix™. If you do not agree to the terms of this Agreement, please promptly return the purchased product for a full refund.

1. **Copy Restrictions. N**o part of any Mekatronix™ document may be reproduced in any form without written permission of Mekatronix™. For example, Mekatronix™ does not grant the right to make derivative works based on these documents without written consent.

2. **Software License.** Mekatronix™ software is licensed and not sold. Software documentation is licensed to you by Mekatronix™, the licensor and a corporation under the laws of Florida. Mekatronix™ does not assume and shall have no obligation or liability to you under this license agreement. You own the diskettes on which the software is recorded but Mekatronix™ retains title to its own software. You may not rent, lease, loan, sell, distribute Mekatronix™ software, or create derivative works for rent, lease, loan, sell, or distribution without a contractual agreement with Mekatronix™.

3. **Limited Warranty**. Mekatronix™ strives to make high quality products that function as described. However, Mekatronix™ does not warrant, explicitly or implied, nor assume liability for, any use or applications of its products. In particular, Mekatronix™ products are not qualified to assume critical roles where human or animal life may be involved. For unassembled kits, you accept all responsibility for the proper functioning of the kit. Mekatronix™ is not liable for, or anything resulting from, improper assembly of its products, acts of God, abuse, misuses, improper or abnormal usage, faulty installation, improper maintenance, lightning or other incidence of excess voltage, or exposure to the elements. Mekatronix™ is not responsible, or liable for, indirect, special, or consequential damages arising out of, or in connection with, the use or performances of its product or other damages with respect to loss of property, loss of revenues or profit or costs of removal, installation or re-installations. You agree and certify that you accept all liability and responsibility that the products, both hardware and software and any other technical information you obtain has been obtained legally according to the laws of Florida, the United States and your country. Your acceptance of the products purchased from Mekatronix™ will be construed as agreeing to these terms.

# MANIFESTO

Mekatronix™ espouses the view that the personal autonomous agent will usher in a whole new industry, much like the personal computer industry before it, if modeled on the same beginning principles:

- Low cost,
- Wide availability,
- Open architecture,
- An open, enthusiastic, dynamic community of users sharing information.

Our corporate goal is to help create this new, exciting industry!

Gainesville, Florida Phone 407-672-6780

**WEB SITE:** http://www.mekatronix.com
**Address technical questions to** tech@mekatronix.com
**Address purchases and ordering information to an authorized Mekatronix Distributor** http://www.mekatronix.com**/distributors**

## TABLE OF CONTENTS



## LIST OF FIGURES



## LIST OF TABLES

# 1 SCOPE

This manual is intended for teachers, educators, university and advanced high school students, hobbyists, researchers and anyone else interested in advancing the infrastructure of robotics and advanced technology in our society. The goal of this manual is to enable teachers and professors, from middle school through the university level, to develop exciting, stimulating, and entertaining instructional materials for hands-on laboratories using embodied, intelligent, autonomous mobile robots.

This manual assumes that the individuals learning PROGO™ have little to no knowledge of the *C* programming language. Teachers who plan to teach PROGO™ to middle school students and do not know *C* can use PROGO™ as a vehicle to learn *C* themselves. Those who do know *C* will immediately understand how #define macros define PROGO™ so the *C* preprocessor can translate valid PROGO™ programs into valid *C* programs.

PROGO™ provides a vehicle for beginners to write sophisticated programs for intelligent, autonomous mobile robots after less than an hour of instruction. Ultimately, PROGO™ will enable young learners to move on to *C* and, later still, to JAVA, with ease and high motivation.

Mekatronix has presented PROGO™ to eighth graders with success. Don't let that surprise you. Eighth graders are really smart! A specific corporate goal is to help middle school and high-school technology teachers introduce this exciting, fun, highly motivating technology to children. We believe Mekatronix robots and PROGO™ represent a substantial step in the right direction.

## *1.1 Language Specific PROGO™*

The underlying *C* languages, *ICC11* and *Interactive C*, that support PROGO™ differ from one another since neither realize standard *ANSI C*. The *ICC11* version of PROGO™ will work with any *ANSI C*, but not with *IC*. *IC* requires a separate PROGO™ implementation.

From the PROGO™ programmer's perspective, there is only one difference between an *ICC11* or *ANSI C* supported PROGO™ and an *IC* support PROGO™:

NOTE BENE:
*Standard C and ICC11 require the User Function List (defined below) while IC insists the User Function List be omitted!*

You can use either the DOS or the Wind95 version of *ICC11* compiler or the *IC* interpreter to run PROGO™. The *IC DOS* version is freeware.

This manual assumes you have an *ICC11* supported PROGO™ and that you have purchased a copy of the *HSSDL11* and will use that program to load PROGO™ code into the robot. For *ICC11* PROGO™ Mekatronix does not support PCBUG11 downloading.

The manual applies equally well to the *IC* DOS or WINDOWS supported PROGO™ versions. In those rare instances where the *IC* version forces different actions or coding procedures from the *ICC11* version, an italicized note will highlight the differences.

*Caution*

*IC and ICC11 are not Standard ANSI C, so be careful of language limitations using them. The differences are discussed in the IC and ICC11 Users Manuals, respectively. ICC11 is closer to the standard and has fewer limitations.*

### *1.2 Requirements to run PROGO™*

This manual provides the definition of the PROGO™ programming language syntax and semantics coupled with exercises and applications realized with the kernel set of robot functions. You will need *ICC11* compiler and *PROGO.c*, or the

***IC VARIANT***

*You will either have either IC DOS (freeware) or the commercial WINDOWS version of IC, if you do not have ICC11.*

To do the downloading you will need the robot, a 6-wire Mekatronix serial cable and a serial cable to connect your personal computer to the Mekatronix D25 connector on the MB2325 communications board. Any of these items can be obtained through a Mekatronix distributor. For distributors check

(http://www.mekatronix.com/distributors).

### *1.3 Robot Specific PROGO™*

Each Mekatronix robot requires separate underlying robot sensor and actuator driver software to support PROGO™. Be sure you get the correct version of PROGO™ for your robot. The language itself does not change from robot to robot, although robot specific functions will. All Mekatronix robots will share a core set of drivers. That core will be explained in this manual along with PROGO™ proper.

If you own a Mekatronix robot, you may also want to supplement this reference manual with an appropriate *PROGO™ Applications Manual*. For example, the applications manual for the TJ PRO™ robot is called the *PROGO™ Applications Manual for the TJ PRO Robot.*

## 2 MOTIVATION BEHIND PROGO™

Programming behaviors for intelligent autonomous mobile robots (IAMRs) presents a significant challenge to the beginner. To become an IAMR practitioner, you must learn and become familiar with the software development cycle and the specific capabilities of the robot you program. The software development cycle may be described as a set of activities, not necessarily sequential, yet with obvious dependencies:

**Software Development Cycle**

1. Specify the problem
2. Develop algorithms and data structures for the problem solution
3. Implement a solution
4. Test, Verify and Document solutions

While sharing traditional techniques and software development activities, robot programming requires additional considerations:

1. The program must interface with the outside world through machine actuation and sensation
2. A robot will not, in general, execute commands with the precision of a digital computer
3. Robots must respond in real-time for most operational situations

Together, robot specific learning and program development activities establish a rather demanding potential barrier to individuals wanting to gain access to the exciting world of intelligent autonomous mobile robots. Mekatronix has developed robots and PROGO™ to lower those barriers and make programming IAMRs easier and fun.

PROGO™ reads something like a highly structured programming language like Pascal, but, unfortunately, does not have the compiler enforcement features of Pascal. PROGO™ may seem a bit wordy, and certainly limited, for *C* programmers. But, the purpose of PROGO™ is not to convert *C* programmers to a new language, but rather assist beginners to write robot behavior programs immediately in a readable, maintainable manner.

PROGO™ maps directly into a valid C language program using C's macro definition feature. Since PROGO™ builds on C through a standard C-Preprocessor, you can mix C statements among the PROGO™ statements and, as you learn more C, migrate to C completely, if desired. PROGO™ eliminates obscure C punctuation, structures and notation and adds English like phrases to make statements easier to read and remember. PROGO™ also supports a number of robot motion and sense commands, the *robot kernel*, to make programming Mekatronix robots easy. What is the price paid for these advantages? PROGO™ statements typically, but not always, require a few additional keystrokes. The additional keywords also prevent users from naming variables with those keywords. These consequences appear insignificant and well worth the advantages gained, especially to beginning programmers who do not know the C language.

PROGO™ requires a *C* compiler for the target processor, which, for the Mekatronix robots, is the Motorola MC68HC11 A or E series chips.

# 3 A BRIEF DESCRIPTION OF PROGO™

This section provides a global perspective of a PROGO™ program. Later sections will give graded examples applicable to the TJ PRO™ robot for purposes of illustration.

### 3.1 Esthetic Code Writing

Since PROGO™ syntax is white-space insensitive, carriage returns, line feeds, spaces and tabs have no affect on the interpretation of a PROGO™ statement. Judicious choice of white spaces makes a program more readable, hence, correctable. Several useful rules for white-space usage follow:

**White-Space Usage**

1. One PROGO™ statement per line.
2. Statements can be blocked into "paragraphs", meaning that tightly coupled statements can be on adjacent lines. Statements that begin another "paragraph" of computation can be separated from a previous statement by a blank line. Such breaks give the reader visual cues about the progress of the computation.
3. A bracketing keyword (Table 1) should have its own private text line to be easily visible!
4. Statements should be indented 2 or 3 columns from the first letter of the bracketing keywords containing them.

**Table 1 Bracketing keywords in PROGO™**

| | | | |
|---|---|---|---|
| Program_begin | Program_end | perform | end |
| Function_begin | Function_end | Repeat_begin | Repeat_end |

### 3.2 Commenting PROGO™ Programs

The special symbol sequence in the quotes "/*" begins a comment and the sequence "*/" ends a comment. Comments cannot be nested. Doing so will often create subtle, hard-to-find program bugs. A popular way of commenting, which avoids many of the compiler problems generated by incorrect commenting, has the following form:

```
/*
   The comment begins here. Anything typed
   between the comment markers will have no
   affect on the program in which it is embedded.

   You can use as many lines as needed to
   say what needs saying.

*/
```

With this style of commenting, you can scan the left-side of the listing and easily detect missing closing comment marker "*/". Comments written at the same column position as the bracketing keywords (Table 1) surrounding it provides a visual cue as to the scope of the comment, assisting in further readability.

Comments should be succinct, informative, and descriptive. A good way to judge a comment, "Does the comment make understanding the program much easier?" Leave out comments that produce a "No" answer to this question!

### *3.3 Structure of a PROGO™ Program*

Every *ICC11* supported PROGO™ program (Figure 1) begins with a *`User Function List`* of user defined functions, a **`Dictionary`** declaration of user-defined integer variables. A PROGO™ program, consists of PROGO™ (or *C) statements*, between **`Program_begin`** and **`Program_end`**. All user-defined functions must be listed after the **`Program_end`** statement.

***IC VARIANT***
*An IC supported PROGO™ program (Figure 2) has the same structures as an ICC11 version, except that the `User Function List` **must** be left out.*

```
/*User Function List*/
   Function <function_name1> used
    ...
   Function <function_nameK> used

/*Declare integer variables*/
   Dictionary
     <variable>,
     <variable>,
     ...
     <variable>
   ok


/*Program Brackets*/
  Program_begin
   <block of PROGO™ statements>
  Program_end

/*Function Defintions*/
   Function <function_name1>
      Function_begin
        <block of PROGO™ statements>
      Function_end

    ...

    ...
   Function <function_nameK>
      Function_begin
        <block of PROGO™ statements>
      Function_end
```

Figure 1. A schematic view of a PROGO™ program supported by *ICC11* or *ANSI C*.

```
/*Declare integer variables*/
   Dictionary
     <variable>,
     <variable>,
     ...
     <variable>
   ok


/*Program Brackets*/
  Program_begin
   <block of PROGO™ statements>
  Program_end

/*Function Defintions*/
   Function <function_name1>
      Function_begin
        <block of PROGO™ statements>
      Function_end

    ...
    ...
   Function <function_nameK>
      Function_begin
        <block of PROGO™ statements>
      Function_end
```

Figure 2. A schematic view of a PROGO™ program supported by *IC*.

### *3.4 Simple PROGO™ Programs*

The simplest PROGO™ program, a program that does nothing, consist only of two keywords

```
Program_begin
Program_end
```

A program that moves a robot forward 20 inches and stops has an added statement

```
Program_begin

   Forward 20 inches

Program_end
```

To move a robot around a 20 inch square in a clockwise direction code

```
Program_begin

   Forward 20 inches
   Turn_right 90 degrees
```

```
      Forward 20 inches
      Turn_right 90 degrees

      Forward 20 inches
      Turn_right 90 degrees

      Forward 20 inches
      Turn_right 90 degrees

Program_end
```

The last **Turn_right** command attempts to turn the robot into its original orientation and has no affect on the square itself.

The PROGO™ *Repeat* statement simplifies the *square traverse* program above,

```
Program_begin

   Repeat 4 times

      Forward 20 inches
      Turn_right 90 degrees

   Repeat_end

Program_end
```

### *3.5 User Function List : ICC11 or ANSI C Only*

For the *ICC11* or *ANSI C* supported PROGO™ the programmer must list all the user-defined functions before the variable dictionary. This corresponds to the *C* language requirement for listing the function prototypes. While the requirement probably satisfies some compiler efficiency requirement, the practice also benefits readability of the code. The list serves as a function dictionary and alerts the reader to the specific functions developed for the program.

The *User Function List* consists of a list of statements of the form

```
Function <function_name1> used
      ...
Function <function_nameK> used
```

The keyword **Function** and **used** must appear before and after each user-defined function name, respectively.

***IC VARIANT***
*Do NOT include A* `USER FUNCTION LIST` *for IC.*

### 3.6 PROGO™ Dictionary

Formally, PROGO™ only supports integer variables in the dictionary. The variables in the list between the keywords **Dictionary** and **ok** must be separated by commas (refer to Table 2). All variables in the list will be typed *integer* (*C* data type *int*). Advanced users can use any *C* data type declaration, *but those declarations must be outside a dictionary declaration.* A dictionary declaration before **Program_begin** will be global to the user program and all the user-defined functions. You can define dictionaries within the main program and user-defined functions, too.

Table 2 Dictionary Syntax and Semantics

| PROGO™ Syntax | C-Semantics |
|---|---|
| **Dictionary**<br>variable_1,<br>variable_2,<br>. . .<br>variable_n<br>**ok** | int variable_1,<br>variable_2,<br>. . .<br>variable_3; |

In the main program the dictionary must follow immediately after **Program_begin**. For user defined functions the dictionary must immediately follow **Function_begin**. The scope of dictionary variables is limited to the function, including the main program, in which it is defined. Dictionaries defined outside of functions, but after **Program_end**, will be global to all functions listed subsequent to it.

### 3.7 PROGO™ Functions

Anywhere after **Program_end** the user may define a function with the following syntax.

```
Function <function_name>

  Function_begin

    <block of PROGO™ statements>

  Function_end
```

To call a function in a program or another function, use the function's name followed by the keyword **call**.

```
<user_defined_function_name> call
```

The above structure is called a *function statement* in PROGO™.

The next version of the *square trace* program (Figure 3) uses a function to produce an angle and a side of a square. The main program calls this function four times to produce the square. For this particular program there is no obvious advantage to write a function. However, function encapsulation often naturally leads to generalization and possible *reusable code*, code useful for

programming other applications. Writing functions, even in situations where nothing in the problem solution seems to suggest doing so, often proves to be good programming practice.

```
/*
  Make the robot trace a square in the clockwise direction.
*/

Function square_side used /* Delete this line if you are using IC */

Program_begin

  Repeat 4 times

    square_side call

  Repeat_end

Program_end

/*
  What follows next is the definition of the function square_side
*/

Function square_side

  Function_begin

    Forward 20 inches
    Wait 1000 ms

    Turn_right 90 degrees
    Wait 1000 ms

  Function end
```

Figure 3. A robot control program to trace out a square in the clockwise direction. The program employs a user-defined function.

The programmer must define the function *square_side* after the **Program_end** keyword and declare its usage before the **Program_begin** keyword in the *User Function List*. The *Repeat* statement calls, or executes, the function four times.

The *Wait* statement makes an appearance in Figure 3. After each move the *Wait* statement stops the robot for 1000ms (milliseconds), i.e., one second, to make each move crisper and more precise. If you attach a colored felt pen to the robot and watch it trace the square on blank newsprint taped to the floor, you will actually observe less rounding of the corners when a one second pause separates the turning and forward motions.

If a PROGO™ function alters the value of any variable global to the function, that value persists after the function completes execution. If the function alters a variable defined in a dictionary local to it, neither the variable nor its last assigned value exists after the function completes execution.

As in *C*, PROGO™ functions can return an integer value with the *Return* statement,

```
Return <integer> now
```

For example, the TJ PRO™ function **RIGHT_IR** returns the value output by the IR detector on the right underside of the robot.

# 4 PROGO™ STATEMENTS

PROGO™ programs consist of a sequence or `<Block>` of *statements*. Statements have several functions in a program

1. Control the execution flow, i.e., what statements execute next,
2. Perform computations,
3. Govern input and output,
4. Actuate robot motors, a special type of output,
5. Read robot sensors, a special type of input,
6. Keep track of time.

All PROGO™ statements begin with a capital letter. The only exception would be a user-defined function where the user has chosen to begin a function name with a lower case letter, for example, the function *square_side* in Figure 3.

The syntax of the various PROGO™ statements appears in Table 3. There you will find the following statement types

1. *Assignment*
2. *If_then_or_else*
3. *If_then*
4. *Repeat*
5. *While*
6. *Do_forever*
7. *Function_call*
8. *Return*
9. *Repeat*
10. *Program Start*

**Table 3** PROGO™ Language Statements

*User Function List*
```
Function <function_name1> used
 ...
Function <function_nameK> used
```

*Declare integer variables*
```
Dictionary
  <variable>,
  <variable>,
  ...
  <variable>
ok
```

*Program Start Statement*
```
Start
```

*Program Brackets*
```
Program_begin
Program_end
```

*Function Defintion*
```
Function <function_name>
   Function_begin
     <block>
   Function_end
```

*Function Call Statement*
```
<function_name> call
```

*Function Return Statement*
```
Return <integer> now
```

*Assignment Statement*
```
Set <variable> to <expression> ok
```

*Repeat Statement*
```
Repeat <number> times
  <repeat_block>
Repeat_end
```

*While Statement*
```
While <test_expression>
perform
  <while_block>
end
```

*Endless While Statement*
```
Do_forever
  <do_block>
end
```

*If Statement*
```
If <test_expression>
then
  <then_block>
end
or_else
  <or_else_block>
end
```

*If Short Form: If without else*
```
If <test_expression>
then
  <then_block>
end
```

The previous section discussed the syntax and usage of the *Function_call* and *Return* statements. The next few paragraphs will elaborate on the other statement types.

### *4.1 Assignment Statement*

The *Assignment* statement in a procedural language stores the value of an expression into a variable. The PROGO™ version employs three key words, **`Set, to`** and **`ok`** with the form:

*Assignment Statement Syntax*

```
Set <variable> to <expression> ok
```

While **Set** is optional, the **to** and **ok** must be present. After execution the value of <variable> is the same as the value of <expression>. The next paragraph explains what can be substituted for the metavariable <expression>.

### *4.2 Arithmetic and Logic Expressions*

The metavariable <expression> is either an arithmetic or logic expression. The arithmetic expression can be any valid combination of plus (+), minus (–), times (*), divide (/) and *C* library functions or user-defined functions with integer return statements. As you learn more about *C* you can employ floating-point functions and variables in your PROGO™ code. With proper data declarations, PROGO™ permits any valid *C* expression. For a complete, technical description of a *C*-expression, refer to any book on *C*. Informally, any valid algebraic expression written from a keyboard using the PROGO™ operators and functions will compile correctly. However, expressions like x + y/2 can be ambiguous. Does it mean (x+y)/2 or x + (y/2)? In PROGO™, as in *C*, it is the latter. To avoid guessing or learning the detailed precedence and associativity rules, use parentheses liberally to eliminate ambiguity in your own mind. Fortunately, one does not have to understand all the complexities of writing expressions to do it successfully.

*Example use of an expression:*

```
Set x to b – a*(x+y)/2 ok
```

*If, at the time of evaluating the expression, b=20, a=4, x = 10, y= 8, then after execution of this assignment statement the new value of x= –16.*

Relational and logic operators (Table 4) can be mixed with arithmetic ones, but requires more advanced understanding. A 0 indicates a false result and 1 a true result in logic expressions. Whether the expression produces logic or arithmetic results depends upon the structure of the expression.

*Example use of relational and arithmetic operators that produce logic results:*

```
Set x to b greater_than (x+y)/2 ok
```

*If, at the time of evaluating the expression, b=20, x = 10, y= 8, then after execution of the assignment statement x =1 meaning true, since 20 is greater than 9.*

*The following, however, produces an arithmetic answer x= 3. The logic value of 1 is treated as an integer 1 when used in an arithmetic expression :*

```
Set x to (b greater_than (x+y)/2) + 2 ok
```

**Table 4 PROGO™'s Relational, Logic and Arithmetic Operators**

| *Relational Operators* | *C-Symbol* | *Bitwise Logical Operators* | *C-Symbol* |
|---|---|---|---|
| **greater_than** | **>** | **bit_and** | **&** |
| **greater_than_or_equal_to** | **>=** | **bit_or** | **\|** |
| **less_than** | **<** | **bit_not** | **~** |
| **less_than_or_equal_to** | **<=** | **bit_xor** | **^** |
| **equal_to** | **==** | *Arithmetic plus, minus, times, divide and modulus* `+, -, *, / , modulo` *(Same symbols used in C)* | |
| **not_equal_to** | **!=** | | |
| **and** | **&&** | | |
| **or** | **\|\|** | | |
| **not** | **!** | | |

The bitwise operators facilitate the control and manipulation of bits, both in data structures and hardware registers that control robot resources.

*Example use of an expression:*

```
Set x to b bit_and a*(x+y)/2 ok
```

*If, at the time of evaluating the expression, b=20, a=4, x = 10, y= 8, then after execution of this assignment statement the new value of x=4 can be computed as follows:*

$a*(x+y)/2 = 36 = 0x24 = (0010\ 0100)_2$ and $b = 20 = 0x14 = (0001\ 0100)_2$

$(0010\ 0100)_2$ **bit_and** $(0001\ 0100)_2 = (0000\ 0100)_2 = 0x04 = 4$

*The notation `0x24` means that the number following `0x` is a hexadecimal number. The binary equivalent of a number is expressed with the notation of* $(\langle number \rangle)_2$*. A bitwise operator is one that manipulates corresponding bits in each argument independently. In this example, the logical* ***And*** *operation takes place.*

### *4.3 If Statement*

The conditional statement determines the execution of the `<then_block>`, a list of PROGO™ statements, or the <or_else_block>, another list of PROGO™ statements. If <test_expression> has a logic value of 1 or a non-zero numerical value the `<then_block>` executes, otherwise the `<or_else_block>` executes.

*If Statement Syntax*

```
If <test_expression>
 then
   <then_block>
 end
 or_else
   <or_else_block>
end
```

Many situations do not require an **or_else** clause. For convenience in those cases, the **or_else** clause can be dropped altogether.

*If Short Form Syntax (If without or_else):*

```
If <test_expression>
then
  <then_block>
end
```

### 4.4 Repeat Statement

The *Repeat* statement provides a convenient method for coding iteration for a fixed number of times. The syntax is shown below. Figure 3 provided an application of the *Repeat* statement.

*Repeat Statement Syntax*

```
Repeat <number> times
    <block>
Repeat_end
```

### 4.5 While Statement

The *While* statement provides for conditional iteration. Technically, only the *While* statement is needed for any kind of iteration since other forms of iteration, the *Repeat* statement for example, can be implemented by the *While* statement.

*While Statement Syntax*

```
While <test_expression>
perform
  <while_block>
end
```

Suppose, in a particular situation, you want a robot to continue moving until it bumps into an object with its front bumper.

```
Go
While not FRONT_BUMP
perform
end
Stop
```

The **Go** command moves the robot forward at full speed. The effects of the **Go** command persist even after the program terminates execution of the instruction and moves on to the next one. As long as the robot does not detect a front bump, the program executes…what? Nothing! The <while_block> of statements is empty ! Essentially, the robot program hangs up on the *While* statement, doing nothing while the robot continues to move forward until it detects a front bump.

When the program detects a front bump, execution proceeds to the next instruction in the program. In this example, the program executes **`Stop`** and the robot halts.

A special form of the *While* statement useful in robotics is the *Endless-While.* Typically, a robot program executes "forever", or at least until you stop it manually!

*Endless While Statement*

```
Do_forever
  <block>
end
```

Finally, we mention the

*Program Start Statement*

```
Start
```

The **`Start`** statement hold up further program execution until the back bumper switch is depressed.

### *4.6 Input and Output Functions*

PROGO™ provides for elementary input and output and a few screen commands (Table 5). All terminal screen IO executes the RS232C serial protocol.

*Serial IO Protocol*

*9600 baud, 8 data bits, 1 stop bit, no parity bit, and no flow control.*

With a robot connected to your Personal Computer through COM1 and your PC running Hyper Terminal or some other VT100 terminal simulation program, execution of the sequence of statements

```
Clear_screen
Home_screen
Display "The value of x is printed below." on_screen
Move_cursor"4" row "6" column
Write x on_screen
```

will clear the terminal screen on your personal computer, move the cursor to *Home* (1st row, 1st column on the screen), display the line of text between the double quotes, move the cursor to the 4th row and 6th column, and then print the integer x starting at the 4th row and 6th column.

When using **`Display`** and **`Move_cursor`** be sure to bracket the string of characters with double quotes.

**Table 5. Input and Output for PROGO™ Programs**

| *Output from Robot to Personal Computer* | *Input to the Robot from the Personal Computer* |
|---|---|
| **Display** "<character string>" **on_screen**<br><br>**Clear_screen**<br>**Home_screen**<br><br>**Write** <variable> **on_screen**<br><br>*Cursor Control VT100 Type Terminal*<br>**Move_cursor**"<x>"**row**"<y>"**column** | **input_number**<br>**input_character** |

In some instances you will want to input parameters to your robot before turning it loose on the floor. The command statements **input_number** and **input_character** allow you to input an integer or a character, respectively, into the robot.

The sequence

```
Display "Type in a character to get its numerical value: " on_screen
   Set x to input_character ok
Write x on_screen
```

displays the characters between the double quotes. If you type the character *'A' (capital A)*, the value of the integer x = 65 and the screen will display the number 65 just one blank space after the colon, like so,

```
Type in a character to get its numerical value: 65
```

On the other hand,

```
Display "The value of x = " on_screen
   Set x to input_integer ok
Write x on_screen
```

will not accept non-numeric characters. In fact, any non-numeric character terminates the integer as far as **input_integer** is concerned.

***IC WINDOWS VARIANT***
*To use IO with IC WINDOWS, you must exit IC before invoking Hyper Terminal.*

***IC DOS VARIANT***
*To use IO with IC DOS, you must exit IC and enter Kermit or some otherVT100-terminal simulator.*

This completes the list of PROGO™ statements.

# 5 ROBOT KERNEL

A basic set of robot functions, called the *robot kernel*, associates with the PROGO™ language. These functions apply to all Mekatronix robots, with the exception of the *wheel move* commands. Those commands obviously do not apply if the robot has no wheels, and are ambiguous when a robot has two or more powered wheels per side.

The **forward** and **backward** commands were discussed in an earlier example.

The functions **Go, Reverse, Stop** command a robot to just what they say. The effect of these commands persists after their execution. To change the robot's motion requires the execution of another motion command. The wheel commands

```
Left_wheel <number> percent

Right_wheel <number> percent

Move <number> lws <number> rws
```

and the spin commands **Spinccw** and **Spincw** also have this persistent property. All the remaining motion commands execute for a finite time or distance.

**Table 6. Robot Motion Function Kernel**

| | |
|---|---|
| *Move Robot Forward a Specified Distance*<br>**Fwd** <number> **inches**<br>**Forward** <number> **inches**<br><br>*Move Robot back a Specified Distance*<br>**Back** <number> **inches**<br>**Backward** <number> **inches** | *Turn Commands*<br>**Pivot_right** <angle> **degrees**<br>**Pivot_left** <angle> **degrees**<br>**Turn_right** <angle> **degrees**<br>**Turn_left** <angle> **degrees**<br>**Spinccw** *Spin counterclockwise*<br>**Spincw** *Spin clockwise* |
| *Robot wheel move commands*<br>**Left_wheel** <number> **percent**<br>**Right_wheel** <number> **percent**<br><br>**Move** <number> **lws** <number> **rws**<br>(**lws** means left wheel speed)<br>(**rws** means right wheel speed) | *Simple Motion Commands*<br>**Go**<br>**Reverse**<br>**Stop**<br><br>*Time Delay Command*<br>**Wait** <time_in_ms> **ms** |

**Spinccw** spins the robot counterclockwise about the robot's center axis until the program explicitly changes the motion. **Spincw** spins the robot clockwise about the robot's center axis until the program explicitly changes the motion.

The wheel speed arguments in the three *wheel* commands express a percentage of 100%. Thus, 50 means 50% of full speed and 100 means 100% of full speed, etc.

The *pivot* and *turn* commands differ as follows. The turn functions rotated the robot about its center axis so there is no net translation. The *pivot* functions turn the robot about the axis perpendicular to the floor and passing through the point of contact of the opposite wheel. Hence,

```
Pivot_right 90 degrees
Turn_right 90 degrees
```

both turn the robot 90 degrees, however, the pivot command translates the center of the robot some because the pivot point is the left-wheel point-of-contact.

### *5.1 Timing Variables and Functions in PROGO™*

The *Wait* command statement permits the programmer to time the duration of a given motion or action in milliseconds. The maximum wait time you can specify equals 65,535 milliseconds, that is, 65.535seconds. This time is sufficiently long for most behaviors. Implementing delays for longer time periods can be realized by the predefined global variables,

*PROGO™ Time Variables, ICC11 or ANSI C*

| | |
|---|---|
| `seconds` | `0 ≤ seconds ≤ 59` |
| `minutes` | `0 ≤ minutes ≤ 59` |
| `hours` | `0 ≤ hours ≤ 23` |
| `days` | `0 ≤ days ≤ 65,535` |

The day variable cannot exceed 65,535 days or almost 180 years! These variables reset each time the robot is reset.

***IC VARIANT***
*IC time variables are not compatible with PROGO™ functions and statements without data type casting. The user must understand type casting and data declarations* long *and* float *to use IC time variables.*

Mekatronix robots also support an interrupt-driven millisecond counter variable whose range varies from 0 to 65,535:

| *Variable Name* | *Robot* |
|---|---|
| `timertj` | TJ |
| `timertjp` | TJ PRO |
| `timertk` | TALRIK |

The `timert_` program variable serves as a free running timer or counter that can be used to define different timing functions. The *Wait* function uses this timer to implement the wait delay.

Use `timert_` as *read-only*. Do not assign values to it as it will generate subtle side effects and errors to code that depend upon the timer.

### 5.2 TJ and TJ PRO™ Sensor Functions

The sensor functions tend to be robot specific and part of its own *Robot Kernel*. Table 7 lists the sensor functions for the TJ PRO™ robot. Technically, **IRE_on** and **IRE_off** do not sense anything. They actuate the IR emitters, i.e., make them shine with 940nm light. They are listed here for convenient reference.

Some of the sensor functions may be shared with other robots. For example, the TJ™ robot shares all the sensor functions in Table 7 except **BUMPER**. The TALRIK II™ robot shares

**FRONT_BUMP** *True if a front bump*
**BACK_BUMP** *True if a back bump*
**IRE_on** *Turn on all IR emitters*
**IRE_off** *Turn off all IR emitters*

with both the TJ™ and the TJ PRO™.

**Table 7. TJ PRO™ Sensor Functions**

| | |
|---|---|
| *Analog Bumper function* | *Read IR sensor values* |
| **BUMPER** *Returns bump contact reading* | **RIGHT_IR** *Read right IR sensor value* |
| | **LEFT_IR** *Read left IR sensor value* |
| *Logical bumper tests* | |
| **FRONT_BUMP** *True if a front bump* | *Control IR emitters* |
| **BACK_BUMP** *True if a back bump* | **IRE_on** *Turn on all IR emitters* |
| | **IRE_off** *Turn off all IR emitters* |

The numerical value returned by **BUMPER** permits the TJ PRO™ to identify six different regions of contact. The other two bumper functions just return logic 0 if no bump occurs when the function tests the bumper output and logic 1 if a bump occurred during that time. The **RIGHT_IR** and **LEFT_IR** sensors read a value between 84 and 127, depending upon the amount of light reflected by an object in front of the robot.

## 6 FINAL COMMENTS

This manual specifies the syntax and semantics of the PROGO™ programming language. A *C* preprocessor takes the user code and includes it into a PROGO.c program and replaces the #define constants with the appropriate character string substitutions to convert a valid PROGO™ program into a valid *C* program.

***IC VARIANT***
*IC uses PROGOIC.c instead of PROGO.c*

The claim is that PROGO™ is easier to read, understand and remember than *C*, making it a more desirable beginner's language, but powerful enough not to have any severe limitations. Limitations that do exist can be overcome with *C* constructs, if necessary.

For further details concerning the sensor functions of a specific robot and PROGO™ application programs, refer to the *PROGO™ Applications Manual* for that robot.

# 7 PROGO™ LANGUAGE SUMMARY

PROGO™ Language Statements

*User Function List*
```
Function <function_name1> used
 ...
Function <function_nameK> used
```

*Declare integer variables*
```
Dictionary
  <variable>,
  <variable>,
  ...
  <variable>
ok
```

*Program Start Statement*
```
Start
```

*Program Brackets*
```
Program_begin
Program_end
```

*Function Defintion*
```
Function <function_name>
   Function_begin
     <block>
   Function_end
```

*Function Call Statement*
```
<function_name> call
```

*Function Return Statement*
```
Return <integer> now
```

*Assignment Statement*
```
Set <variable> to <expression> ok
```

*Repeat Statement*
```
Repeat <number> times
  <repeat_block>
Repeat_end
```

*While Statement*
```
While <test_expression>
perform
  <while_block>
end
```

*Endless While Statement*
```
Do_forever
  <do_block>
end
```

*If Statement*
```
If <test_expression>
then
  <then_block>
end
or_else
  <or_else_block>
end
```

*If Short Form: If without else*
```
If <test_expression>
then
  <then_block>
end
```

**PROGO™'s Relational, Logic and Arithmetic Operators**

| *Relational Operators* | *C-Symbol* |
|---|---|
| **greater_than** | **>** |
| **greater_than_or_equal_to** | **>=** |
| **less_than** | **<** |
| **less_than_or_equal_to** | **<=** |
| **equal_to** | **==** |
| **not_equal_to** | **!=** |
| **and** | **&&** |
| **or** | **\|\|** |
| **not** | **!** |

| *Bitwise Logical Operators* | *C-Symbol* |
|---|---|
| **bit_and** | **&** |
| **bit_or** | **\|** |
| **bit_not** | **~** |
| **bit_xor** | **^** |

*Arithmetic plus, minus, times, divide and modulus*
`+, -, *, / , modulo`
*(Same symbols used in C)*

## Input and Output for PROGO™ Programs

| *Output from Robot to Personal Computer* | *Input to the Robot from the Personal Computer* |
|---|---|
| **Display** "<character string>" **on_screen**<br><br>**Clear_screen**<br>**Home_screen**<br><br>**Write** <variable> **on_screen**<br><br>*Cursor Control VT100 Type Terminal*<br>**Move_cursor**"<x>"**row**"<y>"**column** | **input_number**<br>**input_character** |

## Robot Motion Function Kernel

| | |
|---|---|
| *Move Robot Forward a Specified Distance*<br>**Fwd** <number> **inches**<br>**Forward** <number> **inches**<br><br>*Move Robot back a Specified Distance*<br>**Back** <number> **inches**<br>**Backward** <number> **inches** | *Turn Commands*<br>**Pivot_right** <angle> **degrees**<br>**Pivot_left** <angle> **degrees**<br>**Turn_right** <angle> **degrees**<br>**Turn_left** <angle> **degrees**<br>**Spinccw** *Spin counterclockwise*<br>**Spincw** *Spin clockwise* |
| *Robot wheel move commands*<br>**Left_wheel** <number> **percent**<br>**Right_wheel** <number> **percent**<br><br>**Move** <number> **lws** <number> **rws**<br>(**lws** means left wheel speed)<br>(**rws** means right wheel speed) | *Simple Motion Commands*<br>**Go**<br>**Reverse**<br>**Stop**<br><br>*Time Delay Command*<br>**Wait** <time_in_ms> **ms** |

## TJ PRO™ Sensor Functions

| | |
|---|---|
| *Analog Bumper function*<br>**BUMPER** *Returns bump contact reading*<br><br>*Logical bumper tests*<br>**FRONT_BUMP** *True if a front bump*<br>**BACK_BUMP** *True if a back bump* | *Read IR sensor values*<br>**RIGHT_IR** *Read right IR sensor value*<br>**LEFT_IR** *Read left IR sensor value*<br><br>*Control IR emitters*<br>**IRE_on** *Turn on all IR emitters*<br>**IRE_off** *Turn off all IR emitters* |