MAC TECHNICAL MEMORANDUM 36

# A USER'S GUIDE TO THE MACRO CONTROL LANGUAGE

Steven P. Geiger

December 1973

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

# 1. INTRODUCTION

Although there currently exist many languages used for process-control, they are restricted to special-purpose applications. The Macro Control Language is the base of a new language approach which combines the advantages of compilation from a higher-level language with the automatic scheduling of a pre-programmed real-time system. This language has been implemented on the M.I.T. Electrical Engineering Department's PDP-11/45 DELPHI system, located in Building 38-473. The primitives of the language are structured as macros for the computer's assembly language.

The purpose of this guide is to explain the syntax and semantics of the statements in the Macro Control Language. The guide assumes that the reader is familiar with the assembly language for the PDP-11. Although this guide gives step-by-step instructions on how to write and run Macro Control Language programs, it assumes that the reader is familiar with certain aspects of the DELPHI system. In particular, it is assumed that the reader can log in and out of the system, knows how to use the editor, and is somewhat familiar with the "m" sub-system of DELPHI (see DELPHI User's Manual).

## 2. DAEMONS

The purpose of this section is to consider "daemons", explain what they are and how to use them.

### 2.1) Introduction to Daemons

The central concept of this language is the "daemon" - i.e., a user-specified process exhibiting some "continuity". A daemon is composed of two parts, each with its own user-declared measure of "continuous performance". The first part is some condition to be monitored. If this condition is "true", then the second part of the daemon will take some course of action. For example, in moving a robot arm whose hand is holding a glass of water, it is desirable to keep the hand's angle with the horizontal, $\theta$, equal to zero so as to avoid spilling the water. To accomplish this, a daemon would be created to monitor the angle $\theta$, and to take appropriate action if the glass begins to tilt. In English, the body of the daemon might read as follows: "Recognize within 25 milliseconds if $|\theta|>0.1$; and within 100 milliseconds complete the corrective action of applying procedure P to the actuator controlling the robot's hand." Here, for angle monitoring, the measure of continuity is 25 msec - that is, due to the dynamics of the robot arm, recognition of the angle error within 25 msec of its actual occurrence will produce essentially the same results as if the angle were monitored continuously. If a more sluggish arm were used, the daemon could monitor less frequently

than every 25 msec and still have the same degree of "continuity". Similar comments can be made about the declared time of 100 msec for corrective action at the actuator end of the system.

Thus, the daemon is the heart of a user-defined control loop. The programmer can specify as many daemons as he wishes, and in any order. To his eyes, all daemons run simultaneously, independent of the implementation details of the language.

## 2.2) Creating Daemons

A daemon is created with the following statement:

DAEMON dest,cond,expr,rec-within,serv-within

dest - specifies the destination for the unique address or "name" of the daemon. After creating the daemon as specified, the system will put the daemon's name into this destination. This destination should be specified using the normal assembler destination addressing modes. That is, "R5" specifies that the name will be put into register 5; "DLOC" specifies that the name will be put into the location "DLOC".

cond - specifies the address of the code for the daemon condition to be monitored. The format for a daemon condition will be discussed later. Normal assembler source addressing modes are to be used for this argument. That is, "R5" specifies that the condition address is contained in register 5; "DCOND" specifies that the address is contained in location "DCOND"; and "#DCOND" specifies that "DCOND" is the address of the condition.

expr - specifies the address of the code to be executed when the daemon condition is "true". This code, called the daemon's "expression", will be discussed in Section 2.4. Like cond, expr should

be specified using the assembler's source addressing modes.

rec-within - specifies the measure of continuity for the daemon condition. This measure is referred to as the "recognize-within" time and will be discussed in Section 2.5. The "recognize-within" should be specified using the TIME FORMAT discussed in Section 3.

serv-within - specifies the measure of continuity for the daemon expression. This measure is referred to as the "service-within" time and will be discussed in Section 2.6. Like the recognize-within time, the "service-within" should be specified in TIME FORMAT.

## 2.3) Daemon Conditions

It is up to the daemon condition to perform the appropriate monitoring tasks, and to reply either "true", the daemon expression should be run, or "false", the daemon expression should not be run. There is no special entry format for the actual code of the condition. The user is free to change registers 0 thru 5 at will - they need not be saved upon entry or restored upon exit from the condition. The use of register 6 (the stack pointer) will be discussed in the section on STACKS. The condition can execute any Macro Control Language statement except START, WAIT, or ENDEXPR. To exit from the condition, the ENDCOND statement should be used. The format for this statement is simply:

```
ENDCOND
```

At this time, the system will test register 0 to determine its action. If R0 is positive, this means the condition has returned "true". If R0 is negative, this means the condition has returned "false". Whenever a condition has returned "false", it will be rescheduled to run at a later time.

Users should obey the discipline just outlined. They should not attempt to execute the daemon expression themselves. They should not attempt to re-execute the condition by branching to its beginning. It is desirable that all daemon conditions be short and take little time to execute. Any "work" that takes an appreciable amount of time should be left for the daemon expression.

## 2.4) Daemon Expressions

The expression is the "workhorse" of the daemon. It is here where "corrective" actions are performed before control is again returned to the daemon condition. Like the condition, the expression has no special entry format and registers 0 thru 5 may be used freely. The STACK restrictions that apply to conditions also apply to expressions. The only Macro Control Language statements that expressions may not execute are START and ENDCOND. Although the expression returns no truth value, its exit format is similar to the condition. That is, to terminate an expression's execution, use the ENDEXPR statement. The format

for this statement is simply:

ENDEXPR

## 2.5) Recognize-within

The recognize-within time specifies the measure of continuity for the daemon condition. Very simply, it directs the system to recognize the occurrence of the "true" condition within the specified time after the condition actually becomes true. Practically, this time specifies the frequency of the condition execution. For this specification to be meaningful, it is necessary that the time be larger than the actual code execution time of the daemon condition. If the user does not wish to declare any specific amount of time, he may instruct the system to use a default for the recognize-within time. Default is indicated by specifying a recognize-within time of "zero". The length of default recognize-within time varies dynamically, depending on the number of daemons that need their conditions evaluated. The larger the number of such daemons, the larger the default time.

## 2.6) Service-within

The service-within time specifies the measure of continuity

for the daemon expression. That is, it says to fully service (or execute) the daemon expression within the specified amount of time after the recognition of a "true" condition. In a sense, this time can be used to specify how "important" a particular expression is, with respect to other daemon expressions. This service-within time should be larger than the actual code execution time of the daemon expression. The user can specify a default for the service-within time in the same manner as for the recognize-within. In this case, the length of the default time depends on the number of daemons that need their expressions executed.

## 2.7) Controls over Daemons

As created with the DAEMON statement, a daemon is an idle process which will do nothing until "activated". It is said to be in a "deactivated" state. The usefulness in having control over daemon "activation" can be seen with the earlier example of a daemon keeping a robot's hand level. Suppose one wished to direct the robot to pour the water out of its glass. All attempts would be in vain unless there were some way to "deactivate" the daemon that keeps the hand level. It would be desirable to keep the hand-leveling daemon in a "library" of daemons to be "activated" when wanted, and "deactivated" when no longer needed. Thus, all daemons in the Macro Control Language

are created in the "deactivated" state.

To activate a daemon, the following statement should be used:

ACTIVATE name

name - specifies which daemon should be "activated". This argument uses the assembler's source addressing modes to reference the value returned through the "dest" argument of the DAEMON statement.

The action taken by the ACTIVATE statement depends on the state or "status" of the specified daemon. If the daemon is "deactivated" and idle (for example, after creation), its status becomes "activated" and its condition is scheduled to execute immediately. If the daemon is already "activated", then the ACTIVATE statement performs no useful work. If the daemon has been "deactivated" but has not yet finished executing its code, the statement will set the status to "activated", but will not interrupt the current execution of the daemon; the daemon will proceed as if it never had been "deactivated" in the first place.

To "deactivate" a daemon, the following statement should be used:

DEACTIVATE name

name - specifies which daemon should be "deactivated". This argument has the same format as in the ACTIVATE statement.

The action taken by the DEACTIVATE statement is quite simple. This statement sets the "status" of the daemon to "deactivated".

If the specified daemon were already "deactivated", this statement performs no useful action. If the daemon were executing either its condition or expression, this execution proceeds uninterrupted. However, both the ENDCOND and ENDEXPR statements check the daemon's "status". If either statement discovers that the daemon has been "deactivated", it will return the daemon to its original "idle" state.

## 2.8) Daemon Status

In the previous section it was mentioned that daemons have a "status", and that this "status" can be "activated", "deactivated", or "idle". The status of a daemon can be accessed by a user since it may influence the control strategy to be followed. Thus the following statement is provided:

STATUS name,dest

name - specifies which daemon the user is interested in. This argument has the same format as in the ACTIVATE statement.

dest - specifies the destination for the value of the daemon's current status. The format of this argument is similar to that of the "dest" argument of the DAEMON statement.

The value returned by the STATUS statement is coded in bits. Bit 0 (the lowest order bit) being 1/0 specifies that the daemon is activated/deactivated. If bit 1 is set, the daemon's condition is scheduled to run. If bit 2 is set, the daemon's expression is

scheduled to run. If the user doesn't wish to test individual bits, the following table can be used to understand the daemon's status:

| value | meaning |
|---|---|
| 0 | deactivated and idle, as after creation |
| 1 | impossible, system error |
| 2 | the daemon's condition is scheduled to run. However, after its execution, the daemon will return to the "idle" state since it is deactivated. |
| 3 | the daemon is activated and its condition is scheduled to run normally. |
| 4 | the daemon's expression is scheduled to run. However, after its execution, the daemon will return to the "idle" state since it is deactivated. |
| 5 | the daemon is activated and its expression is scheduled to run normally. |

# 3. TIME - A DISCUSSION

A few statements in the Macro Control Language use the concept of "time". The purpose of this section is to explain the representation of time to the user, explain how to access the time-of-day, and to describe the TIME FORMAT and its various options.

## 3.1) Internal Representation of Time

The Macro Control Language works with 10 microseconds as its basic unit of time. Two consecutive words inside the system represent the current time-of-day. The first word (lower address) contains the high order bits of the time-of-day, while the second word contains the lower order bits of the time-of-day. The representation is a 32-bit unsigned number. This two word size accomodates approximately 12 hours of real time. The time-of-day is initialized to zero and begins counting with the execution of the START statement (see Section 6.2). It is assumed that no user will be running for more than 12 hours at a time. If this should happen, an error message will notify the user that a "TIME OVERFLOW" has occurred.

## 3.2) Accessing Time

During the course of his program's execution, the user may

wish to access the current time-of-day kept by the system. Such an access of time would be useful for timing events external to the computer. For example, consider a researcher who wishes to determine the time required for a human subject to blink in response to a flash of light. Such a calculation is trivial if the computer can record the time at which it flashed the light and the time at which the subject blinked. Thus, the following statement is provided to give the user a copy of the current two word time-of-day:

TIME dest

dest - specifies where the two words of the time-of-day are to be placed. The addressing modes are restricted to the following subset: "Rn" specifies that the time-of-day is to be put into registers Rn and Rn+1; "X(Rn)" specifies that the time-of-day is to be put into locations X(Rn) and X+2(Rn); finally, "DLOC" specifies that the time-of-day is to be put into locations DLOC and DLOC+2.

In all cases, the first word (lower address) will contain the higher order bits of the time-of-day. In the second option for dest, "X" may be blank. The user is reminded to program carries if doing arithmetic operations on the time-of-day, and to use the unsigned conditional branches after any comparisons of "times".

## 3.3 TIME FORMAT

Certain statements, e.g. DAEMON, require the use of

arguments in "TIME FORMAT". Since the user must specify two words worth of time to the system, the TIME FORMAT was developed to aid him. There are three acceptable TIME FORMATS which can be used. In the first two formats, the user must specify two words of "time" data. In the last format, the user can forget that time consists of two words, specifying time in convenient units. The three formats are as follows:

(1) <p>

(2) <p,q>

(3) <amount,:unit>

In the above formats, the characters "<", ">", ":", and "comma" are required.

In the first format, "p" is used as a pointer to a two-word time specified by the user. It is assumed that this "time" is given in two consecutive locations, with the lower address word containing the high order bits of the time. The high order word will be accessed first (in case auto-increment mode is used). There are two notable exceptions to this rule. First, if "p" is of the format "#var", the system assumes a high order word of zero and a lower order word of the number "var". Second, if "p" is auto-decrement mode, it is assumed that the first word "popped" is the higher order word, while the second word "popped" is the lower order word.

In the second TIME FORMAT, the user explicitly specifies

both words of the time. That is, "p" specifies the higher order word while "q" specifies the lower order word. Any assembler addressing modes may be used for "p" and "q". The higher order word will be accessed first.

The third TIME FORMAT allows the user to work with conventional time units. That is, if "unit" is USEC, MSEC, SEC, MIN, or HR, then the user is specifying his time in microseconds, milliseconds, seconds, minutes, or hours, respectively. Here, "amount" specifies the time to be converted, using any legal assembler source addressing mode. This third format shifts the burden of generating a two word time to the system. If "amount" is specified using the immediate addressing mode, this format is as efficient (in execution time and space) as the first two formats, since the conversion is done by the assembler itself. However, if any other mode is used, the conversion must be done at the expense of a larger physical program size and a longer execution time for the statement.

| example | - meaning |
|---|---|
| <LOC> | - the two words of time are in LOC and LOC+2 |
| <#10.> | - the two words of time are: 000000 000012 |
| <(R1)+> | - access high order word first using mode (R1)+, then access lower order word using mode (R1)+ |
| <R3> | - the two words of time are in R3 and R4 |
| <#2,#100.> | - the two words are: 000002 000144 |
| <R1,-(R3)> | - the first word is in R1 and the second word will be "popped" using R3 |
| <#10.,:MSEC> | - the two words equivalent to 10 (decimal) milliseconds will be calculated by the assembler |
| <AMOUNT,:MIN> | - at execution time, the data located in location "AMOUNT" is assumed to specify minutes and will be converted to a two-word time for the system. |

# 4. PROGRAM CONTROL STATEMENTS

The purpose of this section is to describe some statements which help control the execution of a program.

## 4.1) PAUSE Statement

When writing time dependent code a user may wish to suspend the execution of this code for a specified amount of time, and then have its execution resume. For example, consider someone trying to balance an inverted pendulum who needs to know its rate of fall. If angle position is the only value the computer can sample, then the rate of fall must be determined by taking two angle samples spaced in time. Thus, if the user samples the angle, pauses for a period of time, and then makes another angle sample, he can approximate the rate of fall by dividing the difference between the samples by the amount of time paused. This task can be accomplished with the PAUSE statement, whose format is as follows:

PAUSE length

length - specifies the "length" of time the code will remain idle before execution resumes. This argument should be specified using TIME FORMAT.

Phrased more precisely, "length" is the amount of time the processor will ignore the code stream which executed the PAUSE statement. The PAUSE statement has one very important restriction: When it is executed, the R6 stack should be free of

important information since that information will be destroyed (see the section on STACKS).

## 4.2) WAIT Statement

The WAIT statement is a more general means to suspend execution of a sequential program. It is similar to a PAUSL in that normal execution of code is suspended. But while PAUSL "pauses" for a specified amount of time, WAIT "waits" until some specified condition becomes true. The following example will demonstrate the usefulness of the WAIT statement.

Suppose one is interested in controlling the temperature of an oil bath. Using only a thermistor and a heater, a simple control strategy involves turning the heater on when the temperature is too low, and turning the heater off when the temperature is too high. What follows are three sample programming approaches to implement this general strategy.

1) Create two daemons. The first daemon will recognize when the temperature is too low, and then turn on the heater. The second daemon will recognize when the temperature is too high, and turn off the heater. This approach will work, but may cause the computer to thrash. For example, as soon as the first daemon turns the heater on, the temperature will not change noticeably. Thus, the daemon will again turn the heater on (even though it was already on).

2) Create one daemon that will recognize when the temperature is outside an acceptable zone. When this occurs, apply a procedure that turns the heater on if the temperature is low, or turns it off if the temperature is high. Unfortunately, this approach may

also cause thrashing.

3) Create one daemon that will recognize when the temperature is too low. When this occurs, apply the following procedure: turn the heater on; wait until the temperature gets too high; turn the heater off. Such a daemon cannot thrash. It uses a WAIT statement whose condition recognizes when the temperature is too high. This approach requires that the heater is initially off when the daemon is activated.

Example 3) above used a WAIT statement. The format of this statement is as follows:

WAIT cond,rec-within

cond - specifies the condition to be tested, and uses the same addressing modes as the "cond" argument of the DAEMON statement.

rec-within - specifies the measure of continuity for the WAIT condition. Its meaning and format are similar to that of the daemon "recognize-within".

The code for the WAIT statement condition is similar to the DAEMON's condition. That is, there is no special entry format, and it is terminated with an ENDCOND statement. At that time R0 is tested for "true" or "false". If "false", the condition will get rescheduled for execution at a later time. If "true", execution will return to the instruction located physically after the WAIT statement.

The "recognize-within" for the WAIT statement is completely analogous to the "recognize-within" of the DAEMON statement. That is, it specifies the measure of continuity to be used for the WAIT condition. As before, the user can instruct the system

to use a default for this value, by specifying the value zero.

Like PAUSE, execution of WAIT restricts the R6 Stack to be empty. There is an additional restriction on the WAIT statement, in that conditions cannot be nested. Thus, neither daemon conditions nor other wait conditions may execute a WAIT statement. If such an attempt is made, program execution will terminate. Thus, only daemon expressions and the main program (see Section 6) may execute a WAIT statement.

# 5. INPUT/OUTPUT

The purpose of this section is to describe the I/O statements of the Macro Control Language.

## 5.1) Teletype I/O

Any daemon can output to the teletype at any time. However, if more than one daemon outputs messages, they may appear intermixed at the console due to the scheduling process for the daemons. The user must do his own queuing for teletype use among his own daemons. It is also recommended that large messages not be output to the teletype. After the output buffer (50 characters) is full, the Delphi system puts the whole job into I/O wait until at least 15 new characters will fit into the buffer. The daemon scheduler does not know when such I/O waits occur, and thus the response of the system may become sluggish.

Any daemon can input from the teletype at any time; again, queueing must be done by the user. The scheduler does not know when the user intends to do teletype input. An instruction such as READCH puts the system into I/O wait if there are no characters in the buffer yet. Such I/O may degrade the system's response. Thus, the user is advised to check for characters in the input buffer before executing teletype input instructions. Such checking may be done by using NUMCH or interrupts when input appears in the buffer (See Delphi documentation).

## 5.2) Real World Input

Input from the physical process being controlled is obtained using the SENSE instruction. The format of this instruction is as follows:

SENSE line,dest

line - specifies the input port number whose value is to be sampled. This argument uses the normal assembler source addressing modes.

dest - specifies where the input value is to be put. Like all other "dest" arguments, this uses the assembler destination addressing modes.

This single statement is used regardless of what kind of device is connected to the specified input port. If the port has a digital input, it will be read immediately. If the port has an analog input going through an A/D converter, a value is still read immediately. However, in the current hardware configuration (September, 1973) the A/D converter is being multiplexed with "n" input ports (n=4). Thus, due to single convert times of approximately 60 microseconds, the data read will be somewhere from zero to 60n microseconds old.

The system uses only the lower 5 bits of "line" as the port address. Higher order bits will be ignored. Port zero is reserved for the system's programmable clock. This clock may be read by the curious user, but it cannot be set. In the current interface, an input port consists of ten data bits which will be sampled upon command. For example, port #4 corresponds to one of

the motor position sensors on the existing pendulum hardware. This port is sharing an A/D converter with three other input ports. Sensing an unused port will return the value "zero". A list of legal input ports will be kept in 38-473.

## 5.3) Real World Output

Output to the real world is accomplished with the SEND statement. The format of this statement is as follows:

SEND line,value

line - specifies the output port number to which "value" is to be sent. As in the SENSE statement, this argument uses the assembler's source addressing modes.

value - specifies the data to be sent to the specified output port. The format for this argument is specified in the following BNF-like statement:

value:= data | <PAUSE,time> |

<rcount,<value(1),value(2),...,value(n)>>

In the first option for "value", "data" specifies the actual output data, using normal assembler source addressing modes. For this case, the data is output to the appropriate port and control is returned to the instruction following the SEND statement.

In the second option for "value", "time" specifies the amount of time that the SEND statement will "pause" before control is returned to the next instruction. The argument "time"

is specified using TIME FORMAT. The effect of this statement is the execution of a "PAUSE time" statement. No actual data gets sent to the output port.

The third option lets the user specify a string of output values, "value(1),...,value(n)", to be output sequentially, and a repetition count, "rcount", governing how many times the string will be output. The argument "rcount" is normally specified using the assembler's source addressing modes. If the count is zero, the string will not be output. If the count is one, the string will be output once. Any argument of the string, "value(i)", must be in the format of a legal "value". These values will be output starting with value(1) and ending with value(n). The effect is similar to that of execution of the sequence:

```
SEND   line,value(1)
SEND   line,value(2)
  .
  .
  .
SEND   line,value(n)
```

If the count is two, after value(n) is output, the string will be repeated immediately with value(1) for a second pass. Thus, if the count is "m", the string will be output m times. The exception for the argument "rcount" is the option to repeat "forever". This option is specified by putting the single character "*" in place of the argument "rcount". Since the definition of "value" is recursive, a repeated string of values

can be used for any "value(i)". Nesting may be to any level, provided of course that the source statement fits on one line.

It was mentioned that the effect of a "value" in this third form was "similar" to the execution of a sequence of SEND statements. As far as the world can tell, the effect is the same. However, there is a difference to the programmer. Executing a sequence of SEND statements will prohibit the particular daemon from doing any other useful work. However, this third output option actually creates a "temporary output daemon" for the specified line number. Once the temporary daemon is created and activated, control returns to the instruction following the SEND statement. Thus, useful work can be done while a string of data is sent to a device. The daemon is called "temporary" because once the repetition count has decremented to zero, the daemon "disappears". This is important because only one temporary daemon can exist for a given output line at a time. Attempts to send two strings to the same port simultaneously will result in an error.

In the current interface, an output port consists of a ten-bit data register. A port's register will be changed whenever output is directed to that specific port. Currently, port #4 is used to control the motors on the inverted pendulum. A list of legal output ports will be kept in 38-473.

# 6. PROGRAM FORMAT

The purpose of this section is to explain the format for a complete program with several daemons, and discuss a few related statements.

## 6.1) Stacks

The PDP-11/45 is built around a stack orientation. All the Macro Control Language statements use the main stack governed by the stack pointer - register 6. For its use, the system reserves stack space and sets up R6 to point to this stack. Throughout the execution of his program, the user is free to use this stack, provided he obeys the following restrictions. First, he may not specify any use of R6 in any Macro Control Language statement. This restriction comes from the simple fact that the Macro Control Language statements "push" items into the stack. Thus, any specification using R6 will cause unknown errors. Second, he must not have anything remaining in the stack at the time he executes either a PAUSE, WAIT, ENDCOND, or ENDEXPR statement. At these times, the stack pointed must point to the same location that it pointed to upon entry to the daemon condition or expression. All Macro Control Language statements obey this discipline, and "pop" the stack when their execution is finished. If the user forgets this restriction and leaves something in the stack while executing one of the above statements, unpredictable effects will generally occur. The user is reminded that the JSR

instruction "pushes" onto R6's stack. This item must be "popped" before execution of one of the above statements.

The user may use stacks without any of the above restrictions, provided that he reserves his own stack space and uses one of the first six registers (0 thru 5) as a stack pointer. He must initialize his stack pointer upon entry to his condition or expression. All sharing of stack space by multiple daemons must be managed by the user.

## 6.2) Starting a Program

Somewhere in his program, a user must include the following statement:

START (stack)

stack - an optional argument specifying how much system stack space should be reserved. The parentheses are not part of the syntax, but are used to signify that the argument is optional.

The START statement will be the first executable statement in the user's program, regardless of its physical location. This statement reserves space for the system stack governed by R6. Normally, 100(octal) bytes are reserved for this stack. However, the user may override this by specifying the argument "stack". The DAEMON statement temporarily uses 20(octal) locations in the stack. Thus, if more than 4 daemons could be created at the same

time, a stack overflow would result unless more stack space were reserved. Similarly, the user could specify less stack space.

The START statement initializes the system and defines the label "START". This statement belongs to the "MAIN" program, and it creates a "daemon", with no condition, to represent this "MAIN" body of executable code. The service-within time for this "daemon" is infinity, making the code "low priority" compared to real daemons. The START statement also initializes the time-of-day to zero, and starts the system clock.

The statement located physically after the START statement will be the second statement to be executed. If the user should ever forget the START statement, address errors will occur on ".CONTRL", as it will be an undefined symbol. If the START statement is executed a second time (by branching to the label "START") the system will reset itself to its original internal state. That is, no daemons will exist and the time-of-day will be reset to zero. Multiple-definition errors will occur if more than one START statement exists per program.

## 6.3) Finishing a Program

Since the system treats the MAIN program as a daemon expression, the last executable statement (in time) of the MAIN program should be an ENDEXPR statement. The status of a running

MAIN program is deactivated; thus, after the ENDEXPR statement is executed, the MAIN program becomes "idle", and does not affect the execution of other daemons. The only way to re-activate the main program is to re-execute the START statement.

The last physical statement of a complete program is the FINISH statement, whose format is as follows:

FINISH (num)

num - is an optional argument specifying how many daemons (ordinary and temporary) are to be created by the program.

The FINISH statement is a non-executable pseudo-op. It simply provides "clean-up" directives to the assembler and replaces the normal ".END" pseudo-op. One of its tasks is to reserve space for all the daemon structures to be created. The assembler counts the number of DAEMON statements expanded, and adds to this the number of "temporary daemon" expansions (see the SEND statement). The FINISH statement then reserves space for this total number of daemons, and for the main program. The user can override this total by using the argument "num". This procedure might be necessary if a single DAEMON statement is used to create more than one daemon, or if a single SEND statement with option 3 for "value" is used for more than one output port.

If the user forgets to include a FINISH statement in his program, an assembly error will result because the symbol ".FINISH" will be undefined.

## 7. RUNNING PROGRAMS

The purpose of this section is to guide the user through the necessary steps to create, assemble and execute a program in the Macro Control Language. In this section the user is provided with sample command lines. In all the command lines shown, underlined characters are typed by the computer, not by the user.

Since DELPHI's assembler does not have facilities for macros, it is necessary to use DEC's MACRO assembler provided with their Disk Operating System (DOS). The user will never be running pure DOS, since DELPHI provides a Virtual DOS sub-system. To use the MACRO assembler, the user must keep his text files on the system's DOS disk.

### 7.1) DOS Directories

To use the Macro Control Language, the user must have "write" access to DOS directory [40,<x>], where <x> is some integer greater than one. This guide will refer to the directory as [40,<x>], but the user should substitute his particular value for <x> in all the commands shown (i.e., [40,3]). Items in a DOS directory are of the form <filename>.<extension>, where <filename> is a unique one to six character name and <extension> consists of one to three characters. The standard extensions used by DOS are ".MAC" for macro source files, ".BAK" for backup source files, ".LST" for assember listing files, ".OBJ" for

object files, and ".LDA" for load files.

To determine what files exist in his directory, the user should give the Delphi command:

>list [40,<x>]

This command will list the names of all files in directory [40,<x>]. Following each name will be the file length (in disk sectors), and the creation date of the file.

To delete a particular file from his directory, the user should give the Delphi command:

>delete <filename>.<extension>[40,<x>]

This command will delete the file <filename>.<extension> from the directory [40,<x>]. Since ".BAK" files are created after editing, users are encouraged to delete all such files after use. Similarly, all ".LST" files and ".OBJ" files should be deleted after they are used. These files take disk space away from other users. Thus, the user is warned that all ".BAK", ".LST", and ".OBJ" files may be periodically deleted by system programmers if the user becomes negligent.

## 7.2) Editing

To assemble a program, a user's program text must consist of

upper case letters and must reside on the DOS disk with a ".MAC" extension in directory [40,<x>]. To create and/or edit such a text file, use the Delphi command:

>edit filename[40,<x>]

where "filename" is a unique one to six character name for the user's file. Editing is essentially the same as with the normal Delphi editor. The only difference is an automatic case conversion of all characters entered from the console. That is, typing a lower case "a" enters an upper case "A", and vice versa. This enables the user to enter upper case letters without using the "SHIFT" key. As usual, ".BAK" files get created when re-editing a file. However, the ".BAK" files on the DOS disk do not get automatically deleted when the user logs out. Thus, if the user has no interest in the old file, he should execute the command

>delete filename.bak[40,<x>]

to delete the file.

## 7.3) Assembling

To assemble his program, the user must use the MACRO assembler in Virtual DOS. To use the assembler, first enter the Delphi command:

>dos

Once inside Virtual DOS, the system will respond with a "$" whenever it is at DOS command level (instead of the ">" used at Delphi command level). The user should note that DOS will echo lower case characters as upper case. The first DOS command should be:

$LO 40 <x>

This command specifies the user wishes to work in directory [40,<x>]. The next DOS command to be used calls the MACRO assembler:

$MACRO (or $M)

The assembler then types out its current version number, and responds with a "#". This character signifies that the assembler wants a command line. A typical command line would be as follows:

#filename<CTRLM1[40,1],filename

This assembles the user's file, filename.MAC, with the system's macro file, CTRLM1.MAC[40,1], and creates an output file, filename.OBJ, to be used by the LINK Editor. If any assembly errors occur, they will be typed on the console. The user can abort an assembly by entering "CONTROL B", and the system will return to DOS command level. Otherwise, the assembler will

respond with another "#" when finished. To exit from the assembler at this time, first enter "CONTROL B", and then "LINE FEED". This sequence will return the user to DOS command level. To exit from Virtual DOS, enter the following command while at DOS command level:

$QUIT (or $Q)

This will return the user back to Delphi command level.

If the user is only interested in a source listing and symbol table, he should use the MACRO directive

#,filename<CTRLM1[40,1],filename

This will create the file "filename.LST" which contains the program listing and symbol table. To list this file, the user should use the Delphi command:

>p ^nh filename[40,<x>]

After getting his listing, the user should delete this listing file. To obtain only a symbol table, use the directive

#,filename/NL<CTRLM1[40,1],filename

The "/NL" option will not list the source program. To obtain both object and listing files, use:

#filename,filename<CTRLM1[40,1],filename

For other listing options, see the documentation on the DOS MACRO assembler.

## 7.4) Link Editing

Once a user has obtained an error-free object file, he is ready to use the link editor. The link editor is called by giving the following command while at DOS command level:

$LINK (or $L)

This can be done after leaving the MACRO assembler, or after re-entering DOS and giving the "LO" command. After LINK is called, it responds with its current version number and then a "#", as did MACRO. The only command the user should give to LINK is as follows:

#filename<filename,CTRLO1[40,1]/B:0/U/E

This will create a file "filename.LDA" which can be executed at a later time. If any errors occur, the user has not followed all the directions up to this point. Normally, only a transfer address, low limit, and high limit will be typed on the console by LINK. After LINK gives a new "#", exit to DOS command level by giving the "CONTROL B", "LINE FEED" sequence as in MACRO. The user should then return to Delphi command level by giving the "Q" command.

## 7.5) Executing a Program

Once a ".LDA" file has been created, it can be executed by M, as a normal Delphi program. To specify the ".LDA" file, enter the command:

>m filename[40,<x>]

Once inside M, the first command should be "a". This will attach the sensors and actuators to the user's process, locking out other users. Then after the "a" command, enter the "g" command and the program will start executing the START statement. If a program terminates normally, an "END OF JOB" message will be given.

If the user wishes to set breakpoints in his program, at least one must be set before starting his program. When the START statement is executed, the system checks whether or not any breakpoints are set. If none are set at that time, the system will prevent tracing from occurring, even if breakpoints are set at some later time. If one or more breakpoints are set when the START statement is executed, then the system will trace all user code, even if breakpoints are reset at some later time. The user should be warned that tracing slows the execution time of a program and may affect time-dependent code.

## 8. SAMPLE PROGRAM

This concludes the user's guide to the Macro Control Language. The following is a simplified sample program to balance an inverted pendulum.

Assume that the problem can be split in two. That is, the "x" and "y" axes can be controlled independently. In the code that follows, the MAIN program creates all daemons, waits for the operator to press a "start-button", and then activates the axis daemons. It also activates a "DIFF" daemon which finds the angular velocities of the pendulum. The axis daemons use routine "CHECK" to determine if the absolute value of the angle is less than two degrees. If correction is necessary, these daemons use routine "FIX" to control the motors. The code supplied is simplified so that the reader can see a sample Macro Control Language program, without being bored by the details of balancing a pendulum.

```
;DEFINITIONS

XANGLE=14               ;SENSOR PORT # FOR THE "X" ANGLE
YANGLE=3                ;SENSOR PORT # FOR THE "Y" ANGLE

XPOSITION=4             ;SENSOR PORT # FOR THE "X" POSITION
YPOSITION=7             ;SENSOR PORT # FOR THE "Y" POSITION

XMOTOR=4                ;ACTUATOR PORT # FOR THE "X" MOTOR
YMOTOR=1                ;ACTUATOR PORT # FOR THE "Y" MOTOR

BUTTON=1                ;SENSOR PORT # FOR THE "START" BUTTON
```

```
;MAIN PROGRAM
          START
;CREATE "X" AND "Y" DAEMONS
          DAEMON   XDAEMON,#XCOND,#XEXPR,<#5,:MSEC>,<#10.,:MSEC>
          DAEMON   YDAEMON,#YCOND,#YEXPR,<#5,:MSEC>,<#10.,:MSEC>
;CREATE AND ACTIVATE DAEMON TO FIND FALLING RATES
          DAEMON   DIFF,#COND1,#EXPR1,<#1,:MSEC>,<#5,:MSEC>
          ACTIVATE  DIFF
;WAIT FOR OPERATOR TO PRESS START BUTTON
          WAIT      #WCOND,<#1,:SEC>
;ACTIVATE DAEMONS
          ACTIVATE  XDAEMON
          ACTIVATE  YDAEMON
;END MAIN PROGRAM
          ENDEXPR



;WAIT CONDITION
WCOND:    SENSE #BUTTON,R0
          TST   R0
          BEQ   WFALSE
          ENDCOND                   ;RETURN "TRUE"
WFALSE:   MOV   #-1,R0
          ENDCOND                   ;RETURN "FALSE"

;CONDITION FOR DIFF
COND1:    CLR   R0                  ;SET "TRUE"
          ENDCOND

;EXPRESSION FOR DIFF
EXPR1:    SENSE   #XANGLE,XTEMP ;GET VALUES
          SENSE   #YANGLE,YTEMP
          PAUSE   <T,:USEC>     ;PAUSE
          SENSE   #XANGLE,R1    ;GET VALUES
          SENSE   #YANGLE,R3
          SUB      XTEMP,R1     ;CALCULATE DIFF'S
          DIV      R0,T
          MOV      R0,XDIFF
          SUB      YTEMP,R3
          DIV      R2,T
          MOV      R2,YDIFF
          PAUSE   <T,:USEC>     ;DON'T THRASH
          ENDEXPR

;CONDITION FOR YDAEMON
YCOND:    MOV   #YANGLE,R1
          JMP   CHECK
```

```
;CONDITION FOR XDAEMON
XCOND:      MOV    #XANGLE,R1
            JMP    CHECK


;EXPRESSION FOR XDAEMON
XEXPR:      MOV    #XANGLE,R1
            MOV    #XPOSITION,R2
            MOV    #XMOTOR,R3
            JMP    FIX              ;GO TO FIX ROUTINE

;EXPRESSION FOR YDAEMON
YEXPR:      MOV    #YANGLE,R1
            MOV    #YPOSITION,R2
            MOV    #YMOTOR,R3
            JMP    FIX              ;GO TO FIX ROUTINE

;CHECK ROUTINE
CHECK:      SENSE  R1,R0            ;GET ANGLE
            TST    R0
            BPL    1$               ;TAKE ABSOLUTE VALUE
            NEG    R0
1$:         SUB    #2,R0            ;CONDITION "FALSE" IF R0 NEGATIVE
            ENDCOND


;FIX ROUTINE
FIX:        SENSE  R1,R0            ;GET ANGLE
            SENSE  R2,R4            ;GET POSITION
;CALCULATE MOTOR RESPONSE
               .
               .
               .
               .
;CONTROL MOTOR
            SEND   R3,R5            ;R5 HAS CONTROL VALUE
            PAUSE  <#200.,:USEC>
            ENDEXPR



;DAEMON "NAMES"
XDAEMON:    .WORD   0
YDAEMON:    .WORD   0
DIFF:       .WORD   0

;STORAGE LOCATIONS
XTEMP:      .WORD   0
```

```
YTEMP:      .WORD    0
XDIFF:      .WORD    0
YDIFF:      .WORD    0

;PAUSE TIME FOR DIFF
T:          .WORD    500.

            FINISH                    ;END PROGRAM
```