Metronome
Motion Controller for ELMO Digital Products:
“SAXOPHONE, CLARINET”

Software Manual

Revision **05/01**

Previews Revisions:

Revision	Chapter	Description
05/01	8.1 Homing and Capture	New homing events
	12.1 The Metronome Error Codes	New error codes.
	7 Position Controlled Motion Modes	• Smooth factor for position mode. • New multi table ECAM definition.

Table of Contents:

List of Tables:

List of Program Example Tables:

List of Figures:

1 About This Manual

This manual presents all the relevant data for understanding and using the Metronome software. All the motion modes are described here as well as the programming language and the user programs. Although the Metronome has three levels of firmware they are all covered in this software manual. Wherever a section is applicable to the "plus plus" level the signs ++ appear adjacent to it. Wherever a section is applicable to the "plus" level the signs +, ++ appear adjacent to it. This is because the firmware features that are included in the "plus" level are always included in the "++" level.
In the course of this manual, many specific commands are used as part of the explanation. However, the full command description is only available in the "Command Reference Manual".

2 The Metronome Language

The Metronome has a language that enables the user to communicate with the Metronome. Using this language, the user can

- Setup the Metronome
- Command the Metronome what to do
- Inquire the Metronome status

There are two methods for communicating with the Metronome.
The first of them is to use the communication interfaces, either the RS232/RS485 or the CANopen interface, to pass commands to the Metronome, and accept the Metronome immediate response. This method requires a close cooperation between the Metronome and its host.

The other method is to write a program in the Metronome language and to store it in the Metronome. The Metronome can then run the program with minimal or no host assistance.

The physics and the standards of the RS232, RS485, and CANopen communication methods imply differences in the command syntax used by each method.
This section describes the Metronome language in the basic "RS232" syntax.

The syntax for RS485 communication is quite similar, but is different in the following points:

- RS485 lines are half-duplex, namely no two RS485 network members are allowed to transmit simultaneously. The RS485 syntax allows therefore that several language commands be batched to one "sentence".
- The RS485 lines are designed for long distance communication in noisy environment. Therefore, RS485 communication contains identification and checksum data.
- The RS485 syntax supports networking.

The CANopen communication method conforms the CiA301 standard.
In contrast to the RS485/RS232 methods, the CANopen method accesses the Metronome language interpreter by binary communication. This is the most economical way to communicate with the Metronome, since binary communication minimizes both the communication load and the Metronome CPU load. The binary syntax implies, however, that a part of the Metronome language cannot be used by CANopen protocol.
The CANopen communication method is a broad topic and is beyond the scope of this manual. Please refer the CANopen Communication Manual.

Programs use the "RS232" syntax, with extension that are necessary to support program flow instructions and in line documentation.

The full set of the Metronome commands is documented in the Metronome Command Reference Manual.

2.1 Expressions And Operators

The Metronome language supports operators, which specify a mathematical, logical or conditional operation/relation between two operands or more. Operands (actually parameters) and operators may be combined in almost any way to create an expression. The following paragraphs present the operators supported and the expression syntax rules.

2.1.1 Mathematical And Logical Operators

The following table presents the mathematical and logical operators of the language. The table also specifies the operator precedence, which is discussed in the next paragraph.

Operator	Description	Precedence
!	NOT - Bitwise NOT of an operand.	1
*	MULTIPLY - Multiply two operands.	2
/	DIVIDE - Divide the left operand by the right one.	2
&	AND - Bitwise AND between two operands.	2
+	ADD - Add two operands.	3
-	SUBTRACT - Subtract the right operand from the left one.	3
|	OR - Bitwise OR between two operands.	3
=	Assignment	4

Table 1 – Mathematical and Logical Operators

2.1.2 Relation Operators

Relation operators are used in condition expressions, as defined below. Relation operators operate on two expressions and return TRUE or FALSE. The following table lists the relation operators supported.

Operator	Description
=	EQUAL - Returns TRUE if left expression is equal to the right one.
<	SMALLER THAN - Returns TRUE if left expression is smaller than the right one.
>	GREATER THAN - Returns TRUE if left expression is greater than the right one.
<>	NOT EQUAL - Returns TRUE if left expression is not equal to the right one.

Table 2 – Relation Operators

2.1.3 Expressions

An expression is a combination of operands (parameters) and operators that evaluates to a single value. There are different types of expressions, as described below.

2.1.3.1 Simple Expressions

A simple expression evaluates to a single value. Any parameter and mathematical/logical operator may be used to create a simple expression. Normally, simple expressions are used as a part of other types of expressions.

Example:

```
sp
ip|5
SP*2/5+AC
```

Simple expressions are evaluated according to the priority of the operators, as specified in Table 1 – Mathematical and Logical Operators. In case of equal priority, the expression is evaluated from left to right.

Example:

```
SP=100000
ac=sp+10000*2
AC
120000
dc=sp*2+20000
dc
220000
ia[1]=7
IA[2]=ia[1]|5&2
ia[2]
7
```

2.1.3.2 Assignment Expressions

Assignment expressions are used to assign a value to a parameter or command. The syntax of an assignment expression is:

<Parameter or command name>=<simple expression>

Example:

```
sp=sp*2/5+ac
op=ip|5
```

3 Writing User Programs

A Metronome program is a list of commands in certain order. A user program can be anything from a simple list of commands to a very complicated machine management algorithm.
The section below describes how to write, maintain, and run user programs for the Metronome.

3.1 User Program Organization

A user program is organized as follows:

- Program text, including commands, labels, and comments
- An EN "end of program" directive may be used to terminate the program.

Most of the interpreter commands can be used in the program text. In order to learn if a given interpreter command can be used in a program, please refer the **scope** attribute of that command in the Command Reference Manual.
The Interpreter commands that cannot be used in a program are:

- Commands that upload or download data between the Metronome and its host.
- Commands that store data in the flash memory, or that retrieve data from the flash memory.
- Commands concerning the execution of the user program.
  Program flow commands are commands that manage how the program runs.
- Code branches.
- Subroutine execution commands.
- Conditions.
  Program flow commands may participate in the user program, but the host cant use them.
  In the program text, semicolons or carriage returns separate the commands.

Example:

```
IA[1]=0;IA[2]=0
##LOOP;                 **A label
IA[1]=IA[1]+1
JP##LOOP,IA[1]<10;      **Program flow- Conditional jump to the label
EN                      End of program directive
```

The top four lines in the above code are the program text. We observe there commands separated by semicolons and/or carriage returns, a label, and comments. Comments are marked by a double asterisk (**) and extend to the next carriage return.
The last line of the program includes the EN end of program directive.
The comment after the EN directive does not require the ** comment mark, since all the text after the EN is ignored anyway.

The maximum program size depends in the Metronome model as follows:

Velocity	Plus	Plus-Plus
2K	2K	32K

Table 3 – User Program Size

3.2 Comments

Comments are texts that are written into the code to enhance its readability.
A double asterisk marks comments. A comment starts in a comment marker, and terminates at the next end of line.
The Metronome ignores the comments when running a program.

Example:

```
**my first program
PX=1
**um=5
mo=1; **motor on
```

In the above example program, the first line is a comment used to enhance program readability. The comment terminates at the next end of line, so the next PX=1 instruction shall be compiled and executed.
In the third line, the comment mark tells the Metronome to ignore the UM=5 command. This technique is useful for temporarily masking program lines in the process of debugging.
The last line demonstrates that a comment may start anywhere in the program line. The MO=1 instruction preceding the comment marker shall be compiled and executed.

3.3 Program Flow Commands

The Metronome has a set of commands that manage the flow of the user program.
With the aid of these commands, the user program can make decisions iterate, or respond automatically to some events.
The program flow commands enable user programs to do much more complicated things then just running a set of commands sequentially.

3.3.1 Condition Expressions

Condition expressions are used within condition commands. The condition commands are:
AF – Wait until a condition is satisfied
JP – Jump, optionally subject to a condition
JS – Jump to a subroutine, optionally subject to a condition
The syntax of a condition expression is:

<Command>,<simple expression><relation operator><simple expression>

Example:

```
AF,IP=5
JP##START,IP|5=5
AF,PX>10000
JS#@INPUT_ON,PX/2+IA[1]<AN*IA[2]
```

3.3.2 Labels

Most of the program flow commands refer labels.
Labels are names that define a place in the program.
User may set up to 200 labels.

Example:

This example demonstrates a loop that iterates 10 times.

```
IA[1]=0
##MYLABEL
IA[1]=IA[1]+1
JP##MYLABEL,IA[1]<10
```

The label ##MYLABEL points to the next command (IA[1]=IA[1]+1).
After executing IA[1]=0, the program executes IA[1]=IA[1]+1, ignoring the label.
The command JP##MYLABEL,IA[1]<10 is a program flow command. It branches the program to ##MYLABEL (actually the following instruction IA[1]=IA[1]+1) if IA[1]<10.

Labels have the following syntax:

##<LABEL_NAME>	A label.
#@<SUB_NAME>	Subroutine name.

A maximum of 32 characters (letter and/or digits) may be used for a label.
Labels are not case sensitive. The Metronome converts all the labels to uppercase
A label may appear anywhere in a program line:
The program line
IA[1]=PX;##LOOP;IA[2]=1;
is legitimate, although assigning a special program line for each label helps to maintain program readability.

A subroutine name is in fact a label just like any other label. The special notation for a subroutine is not a must – it just helps to maintain code readability.

The XQ and the XC program launching commands use labels to specify where to start program execution and where to terminate.
The jump commands JP, JZ and JS use labels to specify the jump destination.

Labels beginning with #@AUTO specify automatic routines, i.e. routines that may be invoked by a certain event, for example the occurrence of a digital input.
Labels beginning with #@LIB specify library routines provided by ELMO.

3.3.3 Jumps

The jump (JP) command instructs the program to continue its execution at the label specified by the jump command.
The JP command may be specified with a condition. If a JP command has a condition, the jump is performed only if the condition is true.

Example:

The code

```
##LOOP                         **A label
IA[1]=IA[1]+1                  **Increment IA[1]
JP##LOOP                       **Jump to the label
##EXIT
…
```

Iterates forever. The label ##EXIT will never be reached.

The code

```
IA[1]=0
##LOOP                                  **A label
IA[1]=IA[1]+1                           **Increment IA[1]
JP##LOOP,IA[1]<10                       **Jump to the label
##EXIT
```

Iterates only 10 times. The label ##EXIT will be reached with IA[1]=10.

3.3.4 Subroutines and The Call Stack

A subroutine is a piece of code that may be called from anywhere in the program. After the subroutine is executed, the program resumes from the line just after the subroutine call.

Example:

```
JV=1000
JS#@JUST_SO                             **Subroutine call
BG
…
#@JUST_SO                               **Subroutine label
IA[0]=1;                                **Subroutine body
RT                                      **Subroutine end
```

The above code will execute the sequence
JV=1000;IA[0]=1;BG;
After executing JV=1000, the program jumps to the label #@JUST_SO. Before doing so, it stores its return address. The return address is the place in the code where execution is to resume after the routine is done. In this example, the return address is the line number of the instruction BG, which is just after the subroutine call.

The RT command instructs the program to resume from the stored return address. After the execution is resumed at the return address, the return address is no longer required, and it is not stored any more.

Subroutine may call each other – in fact, they may even call themselves. The return addresses for nested subroutine calls are stored in a call stack, as shown in the example below.

Example:

```
IA[1]=3
IA[2]=1
JS#@FACTORIAL                     **Subroutine call
BG
…
#@ FACTORIAL                      **Subroutine for factorial
IA[2]=IA[2]*IA[1]                 **Recursive algorithm
IA[1]=IA[1]-1
JS#@FACTORIAL,IA[1]>1             **Recursive call
RT                                **Subroutine end
```

The FACTORIAL subroutine of the example calculates the factorial of 3 in IA[2]. The variable IA[1] counts how many times the subroutine #@ FACTORIAL is executed.
The program executes as follows:

Code	IA[1]	IA[2]	Call stack
IA[1]=4	3	Undefined	Empty
IA[2]=1	Unchanged	1	Empty
JS#@FACTORIAL	Unchanged	Unchanged	→BG
IA[2]=IA[2]*IA[1]	Unchanged	3	Unchanged
IA[1]=IA[1]-1	2	Unchanged	Unchanged
JS#@FACTORIAL,IA[1]>1	Unchanged	Unchanged	→RT →BG
IA[2]=IA[2]*IA[1]	Unchanged	6	Unchanged
IA[1]=IA[1]-1	1	Unchanged	Unchanged
JS#@FACTORIAL,IA[1]>1	Unchanged	Unchanged	Unchanged (condition is false)
RT	Unchanged	Unchanged	→BG (Program jumps to the RT which was on top the call stack)
RT	Unchanged	Unchanged	Empty (Program jumps to the BG which was on top the call stack)
BG	Unchanged	Unchanged	Unchanged

3.3.5 Killing The Call Stack

In some rare situations, it is desirable to exit a subroutine without returning to its return address. A simple JP jump can do the job, but the return address will remain in the call stack, consuming storage space. After several repetitions, the call stack may overflow, and the program halted. The JZ instruction solves this problem by emptying the call stack before making a jump.

Example:

In an assembly machine, an axis does a routine programmed row.
An inspection station may assert a "Product defective" digital signal.
The "Product defective" signal is coupled to the digital in#1 input.
An automatic routine is coupled to the digital in#1 input, to stop the part assembly and get ready for the assembly of the next part.

```
##START_NEW                    **Label for starting a new part
…                              **Working code

…                              **Lat line of working code
#@AUTO_I1                      **Subroutine label
PA=0;BG                        **Return axis to origin
JZ##START_NEW                  **Go to do a new part
```

The JZ in the #@AUTO_I1 routine is required since we do not know whether when the routine was automatically called the program executed a subroutine or not. If a subroutine was executing, the JZ prevents junk gathering in the call stack. Otherwise, the call stack was empty and the JZ does no harm.

Note that the #@AUTO_I1 routine is executed after the work code is done for every assembled part. The program proceeds from the last line of the working code to PA=0;BG, which resets the machine for another part assembly. The next instruction is a JZ to the ##START_NEW label. The JZ works in this similarly to a standard JP jump, since the call stack is anyway empty.

3.3.6 Automatic Subroutines

3.3.6.1 List Of Automatic Routines

A special kind of routine is the auto routine. These routines are executed automatically according to system events. These routines will be executed only when their invocation condition is satisfied.

Routine name	Priority	Activated by	Mask (MI)
#@AUTOEXEC		Power up	
#@AUTO_ER	1	A motor fault event, in which MO=0 is set automatically.	1
#@AUTO_FLS	2	Logic level of FLS.	2
#@AUTO_RLS	3	Logic level of RLS.	4
#@AUTO_I1	4	Digital input #1 goes high. It executes once, and it is not called again as long as digital input #1 remains high.	8
#@AUTO_ I2	5	Similar to #@AUTO_I1, except that the routine is activated by the digital input #2 when it goes high.	16
#@AUTO_ I3	6	Similar to #@AUTO_I1, except that the routine is activated by the digital input #3 when it goes high.	32
#@AUTO_ I4	7	Similar to #@AUTO_I1, except that the routine is activated by the digital input #4 when it goes high.	64

Table 4 – Automatic subroutines and their priority

All the automatic subroutines, except #@AUTOEXEC, are activated only if a program is running.

Example:

Consider the program

```
##LOOP                        **A go forever loop
JP##LOOP
#@AUTO_RLS                    **Subroutine call
…                             **Subroutine body
RT                            **Subroutine end
```

The forever loop in the first two lines of the routine is intended to make the program run forever, so that the automatic routine will be able to handle the RLS event.
The #@AUTO_RLS routine will be called if RLS is sensed.
The FLS will not invoke any automatic action in the user program, since no #$AUTO_FLS routine is defined to handle FLS.

3.3.6.2 Automatic Routines Arbitration

Priority and pending automatic routines
Each automatic subroutine has its assigned priority, according to Table 4 above.
When the conditions to the activation of two automatic subroutines arise simultaneously, the automatic subroutine with the higher priority will be called.
The other automatic subroutine will be marked as pending. It will execute at the first time it has the permission to execute – even if the reason for its call does not exist any more.

3.3.6.3 The Automatic Subroutine Mask

Automatic subroutines may be masked, i.e. set inactive. For example, it may be desired to limit the automatic response to a certain digital input will be limited to certain situations.
This is done using the MI command.

Example:

A Metronome is commanded by a PLC. The Metronome has an autoexec routine, which activates its program upon boot. The PLC sends instructions to the Metronome using RS232 communication for task parameters and a digital input to start an action immediately. The Metronome sets output according to the state of the program. For safety, the Metronome is not allowed to perform any task before digital input 2 is set:

```
#@AUTOEXEC              **Initial setup and forever loop
##START
MI=8                    **Inhibit the routine #@AUTO_I1
OP=1                    **Set output to indicate start.
##LOOP
JP##WAIT_PARS,IB[2]=1   **Jump to label if digital input 1 set.
JP##START,IB[3]=1       **Start all (masking digital input 1) if din 3 is set.
JP##LOOP                **Loop until

#@TEST_PARS             **Subroutine
OP=2                    **Set output 2
WT=2000                 **Wait 2 seconds for testing the part
MI=0                    **Enable automatic handling of digital in #1
RT

#@AUTO_I1               **Automatic handler for digital input #1
OP=3…                   **Set output 3 to indicate din 1 sensed.
…                       **Subroutine body.
RT                      **Subroutine end

##WAIT_PARS             **Waiting done
JS#@TEST_PARS           **Jump subroutine to test the part
JP##LOOP                **Start again
```

The mask may also be used for preventing that switch bouncing will generate spurious routine calls

Example:

A machine performs a periodic task.
Digital input #1 is connected to a sensor to which the Metronome should react.
The code below limits the automatic response to digital input #1 is limited to one time per cycle, even though the digital input #1 may bounce.

```
##LOOP                    **Repetitive task
…
MI=0                      **Re-enable automatic routine
…
JP##LOOP

#@AUTO_I1                 **Automatic handler for digital input #1
MI=MI|8                   **Prevent nested calls to #@AUTO_I1
…                         **Subroutine body
RT                        **Subroutine end
```

3.3.7 Waiting Until a Condition Is Satisfied

The AF command suspends the execution of the program until a specified condition is satisfied.

Example:

The instruction
AF,PX>20000
suspends the program until the variable PX exceeds 20000.

The code below has the same functionality

```
##LOOP
JP##LOOP,PX<20001
```

But the latter code is less convenient to use.

3.3.8 Inserting Time Delays

The WT command suspends program execution for the specified number of [ms].

Example:

```
PA=10000;BG
AF,MS=0
WT=20
```

The above program instructs a PTP motion. The AF,MS=0 waits until the position command is stabilized in its new position. The WT command allows more 20 milliseconds for final stabilization.

In order to run a program, the program need first to be loaded to the Metronome and compiled.
The Composer program has a program development environment, and it usually also handles the acts of program downloading and compilation.
Any host can, however, do that using the DL and the CC commands.
The user can start the program by two ways:
Auto execution subroutine (Please refer to Section 3.3.6).
The commands program execution using the XQ or XC commands.

4 Program Development and Execution

The process of program development includes the following steps:

- Program editing – Writing\editing the program.
- Program loading – Load the program to the Metronome RAM memory.
- Compilations – Let the Metronome process the program and find syntax errors.
- Debugging – Observe the program behavior, and correct where necessary
- Save to flash – Make the program reside permanently in the Metronome.

4.1 Editing a Program

The Metronome program is a simple text. Any text editor can be used to write the Metronome program.
The Composer program has a development environment for Metronome programs, which includes a program editor. We recommend using the development environment of the Composer, since it provides several services like downloading the program to the Metronome, compiling the program and running it.

4.2 Downloading a Program

The best practice is to use the Composer development environment.
You can also transmit the program as a text file to the Metronome, using standard communication programs like the HyperTerminal of the Windows operating system.
Please refer the DL command documentation in the Command Reference Manual.

4.3 Uploading a Program

A program that resides in the Metronome RAM can be uploaded for backup or for further editing. The Composer's development environment best does this.
You can also upload the program from the Metronome to a standard terminal – please refer the LS command in the Command Reference Manual.

4.4 Compilation

The Metronome compiles a program in order to produce address maps and other run time data. If in the course of the compilation it finds syntax errors, it stops the compilation and informs the errors it found.
The compilation is best made from the Composer's development environment. The Composer knows to decode the compiler error messages and present them in convenient form.
For compilation without the Composer program, please refer the documentation of the Command in Command Reference Manual.

The compiler catches syntax errors. It cannot catch:

- Out of range command arguments
- Bad command contexts like an attempt to begin a motion while the motor is off.

Such errors must be corrected in the debugging stage.

For a complete list of the error codes that the compiler can generate and for the explanation of these error codes, please refer the EC command in the Command Reference Manual.

4.5 Saving a Program In the Flash Memory

A program that is already loaded to the Metronome may be saved permanently in the Flash memory using the SG command – please refer the Command Reference Manual.

4.6 Clearing the Program

The CP command clears the Metronome's program RAM. In order to clear a program that is stored in the Flash memory, first clear the program RAM using the CP command, and then store the empty program in the Flash memory using the SG command.

4.7 Program Execution

The following paragraphs describe how to run a program.

4.7.1 Initiating a Program

A program is initiated by the XQ command. The XQ command states at which label the execution shall start. It may also specify at which label the execution is to stop.
The XQ command resets all the program variables. In particular, it clears the call stack (refer Section 3.3.4), it kills any pending automatic routines, and it clears the interrupt mask.

Example:

The command
XQ
restarts the program from its first line.

The command
XQ##START
starts the program from the label ##START.

The command
XQ##START,##STOP
starts the program from the label ##START. If the label ##STOP will be encountered before the program terminates, the program will exit there.
Halting and resuming a program
A program may be halted using the HP or the KL commands.
The HP just halts the program; the KL command halts the program and shuts the motor off.
A halted program may be resumed by the XC command. The command XC restarts execution from the point where the program has been halted.

Example:

Consider the program

```
MO=1                                **Start motor
JV=2000                             **Set jog speed
##LOOP                              **Repetitive task
BG
WT=1000                             **Wait & switch direction
JV=-JV
JP##LOOP                            **Repeat
```

This program makes the motor travel at 2000 counts/sec for one second, then reverses the direction for one second, continuing to travel back and forth forever.
Suppose the HP command is applied when the program waits (executing the WT=1000 instruction). The motor will continue to travel at the same direction, for unlimited time.
An XC command shall reverse the direction immediately, since the WT time already elapsed.

Suppose the KL command is applied when the program waits (executing the WT=1000 instruction). The motor will stop immediately.
If an XC command is issued without stating MO=1, the program shall abort since a BG requires MO=1.

Example:

A servo axis in a machine has two different tasks to do, in two different machine modes.
The following routine implements the two tasks.

```
##TASK1
##LOOP                               **Repetitive task 1
…                                    **Task 1 body
JP##LOOP                             **Repeat task 1
##TASK2
##REP                                **Repetitive task 2
…                                    **Task 2 body
JP##REP                              **Repeat task 2
```

The first task is invoked by
XQ##TASK1
In order to switch to the second task, the first task has to be killed before:
HP;XQ##TASK2

4.7.2 Debugging

4.7.2.1 Program Status

The status of the program can be read using the PS and the CS command.
The PS command reports whether the program is running at all, and if it runs, it tells which line is now running.
The CS command prints the entire program line that is next to be executing to the host computer.

4.7.2.2 Single Command Steps

The command SC advances a halted program by a single instruction.
The SC command does not test the conditions for the automatic routines, so automatic routines shall not be jumped into.

4.7.2.3 Why Did My Program Abort?

If a program instruction returns with in error, the program aborts immediately.
A program that aborts stores an error message internally in the Metronome. This error message includes the line number, the erroneous instruction, and the error code.
The MZ command retrieves the error message to the host terminal, and clears the stored error message.
The following error codes indicate run-time errors. These errors are not generated by a specific faulty instruction but by a failure of the program flow management.

Error code	Description	Possible Reason
45	Return Error from Subroutine	Too many nested subroutine calls. Possibly there is a “call stack leakage” – one of the subroutine returns without an RT command, leaving its return address in the call stack.
74	Program Stack Overflow	An RT command encountered with no corresponding return address in the call stack. Check if there is a spurious RT command in the program, or if a subroutine is entered not through the use of the JS command.

Table 5 – Run time errors

5 Communication With the Host

5.1 General

The following communication types are supported.

- RS232: A PTP (Point to Point) full duplex protocol. This is the simplest communication method.

RS485: A multi-drop half-duplex protocol with a distinct master. RS-485 has the following advantages over RS232.
A single RS485 multi-drop network can control many Metronomes.
Only two wires are required.
The communication range can be hundreds of meters.
Enhanced error detection enables the Metronome to be reliably communicated in noisy environments.
The drawbacks are:
The RS485 method requires more care from the side of the host. The RS485 method is half duplex[1], so the host must monitor the communication timing in order to avoid crashes. The half duplex property requires longer communication times.
The enhanced error detection features result in more complex syntax.
CANopen (CiA-301 standard): Much faster and versatile communication method then RS232 or RS485. The CANopen method is suitable for fast, accurately timed tasks. This method requires, however, special host hardware and software.

5.1.1 Using Two Communication Channels Simultaneously

The Metronome may maintain two communication channels simultaneously.
Devices that are equipped with the CANopen option always respond to CAN communications. This is in addition to the RS232 or RS485 communications. In that way, a Metronome that is permanently connected to its host in a CANopen network can be accessed using the RS232 or RS485 for secondary communications. The Metronome status can be asked, or performance evaluation records can be taken, without intervening with the operational Metronome work. The setup data may be changed and commands be entered by the secondary communication channel, but this is not recommended.
Similarly, a Metronome that is connected to its operational host via RS232 or RS485 communications can be accessed using the CANopen interface as a secondary channel for status and performance evaluation.

[1] At a given time, only one message can be transferred. The host cannot send and receive messages simultaneously.

5.1.2 Switching Between RS232 and RS485 Communications

Even though standard RS232 and RS485 communications use different physical communication lines, the SAX & CLA hardware implementation allows the RS485 communications to use RS232 physical lines. Therefore, a PC computer can access an RS485 speaking Metronome using its standard RS232 serial port without any physical conversion.
With physical RS232 lines, the communication syntax can be switched from RS232 to RS485 and vice versa on the fly, by setting the PP[1] parameter.
Note: Changing the PP[1] while using Elmo's Composer will force user to re-open with the correct communication method.

5.2 RS232 Communications

5.2.1 RS232 Basics

The RS232 method can only serve to communicate a host to a single Metronome. No identification information is required, since there are no options to select from.
The RS232 lines are full duplex – they can carry bi-directional communications. This means that the host can transmit to the Metronome whenever it finds it right, without considering the state of the Metronome.
The RS232 communication consists of ASCII printable characters only, with some exceptions:

- The characters 0xd (carriage return) and 0x8 (backspace) are used to terminate strings.
- Many non-printable characters are used as error codes – please see the description of the Metronome messages below.

The basic syntax for RS232 commands is
<Command mnemonic>{[<index>]}{<Equal sign><Value>}<Terminator>
Where

Item	Description
Command mnemonic	Two letters (case insensitive) that are assigned to a command. Please see the reference manual for a complete list of command mnemonics.
Index	The index, if the mnemonic refers to a vectorized parameter or command.
Equal sign	The '=' character (Optional, if the command assigns a value to a parameter)
Value	Parameter value (Optional, if the command assigns a value to a parameter)
Terminator	<CR>[2] or ';'.

Table 6 – RS232 Rx Item Description

No spaces are allowed between the fields of the above table.
Typical examples are:

MO<CR>
Asks the Metronome to report the value of the variable MO.

MO=1<CR>
Sets the value of 1 to the MO variable of the Metronome.

CA[2]=1;

[2] <CR> is a carriage return, which is the character 0xd (13 decimal).

Sets the value of the CA[2] variable of the Metronome. CA[N] denotes a vector of parameters that can be accessed by their index.

The Metronome responds to the host communicated commands. It never initiates an unasked-for message to the host.
The syntax of the Metronome answers is
{<Value>}{<Error code>}<Terminator>
Where

Item	Description
Value	Parameter value (Optional, if the command asks the Metronome to report parameter)
Error code	A binary number that may be interpreted according to the error code tables – please refer the EC and the CC commands in the reference manual.
Terminator	';' if the host command has been successfully executed, else '?'

Table 7 – RS232 Tx Item Description

5.2.2 The Echo

With RS232 communication, setting the echo to on (EO=1, this is the default) causes the Metronome to echo each accepted character immediately on reception. The echo feature helps when typing to the Metronome directly from a terminal.

5.2.3 Background Transmission

When the host enters the BH=n command to the Metronome, the Metronome uploads the recorder data to the host. The uploading process may take few seconds. At the time the Metronome uploads records to its host, it is still listening to the host for new commands.
Consider the command sequence
BH=1;MO=0<CR>
The Metronome will start to transmit the recorder data immediately. Few milliseconds later, while the recorder data is still transmitting, the Metronome will execute the MO=0 command. The Metronome will store the response message to the MO=0 command, in order to transmit it later, immediately after the records upload terminates.
Communication with a Metronome that has unknown communication parameters
If the host does not know the communication parameters of the host, it may force the Metronome to switch to its default communication parameters:

Baud rate = 19200
Stop bits = 1
Parity = None
Character length = 8bit.

This is done as follows:
Transmit 20 breaks. Its break will generate a framing error at the Metronome. After 20 consecutive framing errors, the Metronome will automatically switch to the default communication parameters.
Within 1 second, transmit the characters ‘J’ and ‘Z’[3]. If the characters ‘J’ and ‘Z’ are not identified within 1 second, the Metronome will revert to its programmed communication parameters.
The sequence described in this section forces the Metronome to communicate using the default parameters, but it does not change any of the communication programming parameters of the Metronome (the PP array).

[3] The characters ‘J’ and ‘Z’ are the initials of Jacob Zohar, whose contribution to the Metronome has been invaluable.

5.2.4 Related Parameters

	Description	Range
PP[1]	Type of communication. PP[1] serves as “Enter communication parameters”. All the other communication parameters do not come to effect until PP[1] is written. Please note that the response to PP[1]=x is not the same as the response to all other commands, since the communication type switches while processing the command.	1 for RS232 2 for RS485
PP[2]	RS232 baud rate. This parameter does not have any immediate effect. It is sampled upon changing PP[1].	2 for 19200 1 for 9600 0 for 4800
PP[3]	Reserved	
PP[4]	RS232 parity This parameter does not have any immediate effect. It is sampled upon changing PP[1].	0: None 1: Even 2: Odd
EO	Echo mode	0 off 1 on

Table 8 – RS232 Protocol Parameters

The RS232 communication always uses one stop bit, since other stop bit selections are not supported by the hardware.

5.3 RS485 Communications

The RS-485 network consists of a single master and up to 254 slave Metronomes.
Each slave has its

- Slave ID – A number in the range [0..255] (59 and 126 are excluded[4]) that serves to identify a single slave and talk to it personally. The slave ID is unique to the slave. One and only one slave must be defined as a representative. The representative is the only one to answer the master for commands that are broadcast to the entire network using the ' !!' waking sign.
- Group ID (Optional) – A number in the range [1..255] (59 and 126 are excluded) that does not serve as a slave ID. The group ID serves to access several slaves together, so that several Metronomes can be synchronized. One and only one slave from the group may answer the host. The Metronome in the group that answers the host is called the group representative. The group ID of 0 defines that the Metronome is not a part of any group.

The master can address a slave directly, or address a group of slaves. If a slave is addressed directly, then the slave answers. If the master addresses a group, only the group representative responds. When a group representative responds to the host, it identifies itself with its own unique ID, NOT in the group ID[5].

Important note :
The RS-485 method does not support collision avoidance – only collision detection.
Collisions can be minimized by orderly bus arbitration in which the master talks and then waits for the addressee to respond. Collisions, however, cannot be completely avoided.
The Metronome detects collisions by receiving and comparing the characters that they have transmitted.
It is strongly recommended that the master will also implement collision detection by listening to its own transmissions.

[4] 59 is the ';' terminator character, and 126 is the '~' reset character.
[5] This is applicable only in the "Identify every message" mode where a responding Metronome must identify itself.

5.3.1 Communication Parameters

The following communication parameters define the behavior of the RS-485 protocol:

	Description	Range
PP[1]	Type of communication. PP[1] serves as “Enter communication parameters”. All the other communication parameters do not come to effect until PP[1] is written. Please note that the response to PP[1]=x is not the same as the response to all other commands, since the communication type switches while processing the command.	1 for RS232 2 for RS485
PP[5]	RS485 baud rate This parameter does not have any immediate effect. It is sampled upon changing PP[1].	2 for 19200 1 for 9600 0 for 4800
PP[6]	Reserve	
PP[7]	RS485 Unit ID This is the ID to which the device uniquely responds. This parameter does not have any immediate effect. It is sampled upon setting PP[1]. (PP[1]=PP[1] will set the unit ID active). In the message body, a binary byte represents the ID. The ID of 48 will be shown in the message body ASCII character ‘0’.	0 to 255, 59 and 126 are EXCLUDED Since 59 (ASCII ‘;’) is the terminator character and the character 126 (ASCII ~) serves as a communication reset
PP[8]	Group ID. Using this ID several servo drives that consist a group may be accessed. This parameter becomes effective immediately when set. In the message body, a binary byte represents the ID. The ID of 48 will be shown in the message body ASCII character ‘0’	0 to 255 0 denotes that the servo drive is not a part of any group. 59 and 126 are EXCLUDED Since 59 (ASCII ‘;’) is the terminator character and since the character 126 (ASCII ~) serves as a communication reset

	Description	Range
PP[9]	Reserve	
PP[10]	RS485 parity This parameter does not have any immediate effect. It is sampled upon changing PP[1].	0: None 1: Even 2: Odd
PP[11]	Address/data identification - ID every message With this method, every message includes its own ID as a part of the message. When this method is disabled, slaves are awakened using the %% command. An awakened slave can talk with the master without any further identification until the master sets the slave to sleep by awakening another slave. This parameter becomes effective immediately when set. After setting this parameter to 0, the device is not awake. It must be awakened using %%n; in order to be spoken with.	0: None 1: ID every message
PP[12]	Representative mode. This parameter defines if the servo drive responds to broadcast or group commands. The representative mode can be set only when a single servo drive is accessed. It can't be set in the net broadcast or the group broadcast modes. This parameter becomes effective immediately when set. When a group/network representative responds to the host, it identifies itself with its own unique ID, NOT with the group ID	0: Not a representative 1: Answer broadcasts, don't answer the group ID (Net representative) 2: Answer group ID, don't answer broadcasts (Group representative) 3: Answer both broadcasts and group ID. (Net and Group representative)

Table 9 – RS485 Protocol Parameters

5.3.2 Special Commands for The RS485 Mode

The following characters and commands are defined for the RS-485 communications:

Command	Description	Comment
!!	Listen everybody. For all the listening devices that are not in the "ID every message" mode (PP[11]=0), The next command shall be targeted to all the listening devices. Only the device that is the net representative shall respond.	
%%	Wake, for a single device or for a group. For all the devices that are not in the "ID every message" mode (PP[11]=0): Following the command %%n, all the consecutive commands must be listened by the device whose ID is n, or by all the ID=n group members	
~	Kill communication buffer, and set all the listening Metronomes to "un selected". The ~ command is not answered.	

Table 10 – RS485 Special Commands

5.3.3 Message Construction

5.3.3.1 The "Don't check ID every message" Format

The formatting of messages for ID=n slaves is very similar to RS232 formatting, with the following differences:

- The command must be terminated by a <CR> or by '**;**' character. In the exceptional case of program listing messages, the terminator may be 0x8. Please do not append a <LF> character.
- A slave or a group are awakened by sending **%%n;**, where n is the ID. The slave (if addressed directly) or the group representative must respond with **%n;** .
- The entire network is awakened by sending **!!;** . Only the net representative will answer with **!;** .
- Slave to master response is formatted as shown in the table below.

Field	**Description**
Message body or error code	Described in detail in the section on the ***ID every message format*** below.
Status	'?' For failure ':' For success
Terminator	';' With the exception of the response to the LS,CC, and LD command, for which the terminator is 0x8. A different terminator is used there since for the LS response the message body may legitimately include the characters ';' and <CR>.

Table 11 – Slave To Master Response Message

5.3.3.2 The ID Every Message Format

5.3.3.2.1 Master To Slave Message

The message format is the following:

Field	Description
'%' '%' Or '!' '!'	Sync characters
ID (Only if the sync characters are '%')	Binary target ID byte. The ID of 0 refers the entire network. If the sync characters are '!', this field is not used, since the message is anyway directed to the entire network. The ID is a binary number. The ID of 48 is equivalent to the ASCII character '0'. Please note that the ID is not necessarily a printable character.
Message Body	A command string, or concatenated command strings. Command strings are documented in the software reference manual, which applies as well to RS232 commands. Concatenated commands are separated by a comma ','. **The maximum length of the message body is 200 bytes**. It is not advised to use such long character strings, as: A large processing delay is introduced. The processing of the commands shall start only after the slave has received the last checksum byte. The message and its response shall consume a lot of time in which other slaves cannot be accessed. A typical command string is PA=1000 The above example string sets the position absolute target to 1000. A typical concatenated command string is SP=2000,AC=100000,PA=1000,BG In the above example, the concatenated string sets the point-to-point speed, acceleration, and target position, and asserts motion starting. Concatenated strings are executed continuously without interruption from any other source of interpreter commands, such as the CAN channel and the user program. Commands that cannot be concatenated are: PP[1]=1 communication type switching.

Field	Description
	CC (compilation) DL (Program download) LS (Program listing)
Checksum	Binary checksum, calculated as explained below.
Terminator	‘;’ or <CR> The exception Is the DL command for which the terminator is 0x8. A different terminator is used there since for the DL command the message body may legitimately include the characters ‘;’ and <CR>.

Table 12 – Master To Slave Response Message

The checksum is calculated as follows:

1. Sum all the characters of the message, including the header, not including the ‘;’ terminator.
2. Keep only the least significant (last) byte of the result.
3. Negate the result, using twos complement.

Some values are not allowed as a checksum and should be replaced according to the following table.

Forbidden checksum Hex (ASCII)	Replaced by (Hex)
08	F8
0D (<CR>)	FD
3B (;)	FB
7E (~)	FE

Table 13 – Forbidden Checksums and Their Replacement

Example:
Command SP=100 to the ID 0f 3:

Byte	Value, hexadecimal	Value, decimal
%	0x25	37
%	0x25	37
3 (Binary)	0x3	3
S	0x53	83
P	0x50	80
=	0x3D	61
1	0x31	49
0	0x30	48
0	0x30	48
Checksum	0x42	66
;	0x3B	59

The checksum is calculated as follows

Step	Do	Result
1	Sum message body. Sum(%%3SP=100)= 0x25+0x25+0x3+0x53+0x50+0x3D+0x31+0x30+0x30	0x1be
2	Keep least significant byte only	0xbe = Binary 10111110
3	Invert	Binary 01000001
4	Add 1	Binary 01000010 = 0x42

Important note:
A master to slave command can be terminated anytime by a '%' character. The slave will interpret the '%' the starting character of a new message, discarding the incompletely received message. This feature is useful for: a) Issuing emergency motion termination.
b)Killing a message after RS485 message collision detection.

5.3.3.2.2 Slave To Master Message

The message format (with the exception of the command PP[1]=2) is the following:

Field	Description		
'%'	Sync character		
ID	Binary slave ID The ID is a binary number. The ID of 48 is equivalent to the ASCII character '0'. Please note that the ID is not necessarily a printable character.		
Message body or error code And status	A response to the command string. Recall that the command string is composed of a series of sub commands, separated by commas. The response string is composed of a series of responses to the sub-commands. Each response to a sub-command is structured as follows:		
		Echo	If EO=1 (Echo on), this field includes the echo – a copy of the referred command sub-string. If EO=0, this field is empty. For example, with EO=1, PX is answered (assuming that the axis position is zero) by PX0.
		Answer to sub-command.	A string of alpha numeric characters for a report request (such as **SP**) Empty for setting command (such as **SP=1**) Empty if sub-command is not understood or could not be performed.
		Error code.	Empty if sub-command is understood and could be performed. An error code otherwise. The error code is a single character. The comma, semicolon, and question mark characters (',' , ';' , and '?') do not serve as error codes.
		Success indicator.	':' For success '?' For failure
Checksum	Binary checksum.		
Terminator	';' with the exception of the response to the LS command, for which the terminator is 0x8. A different terminator is used there since for the LS,CC and DL response the message body may legitimately include the characters ';' and <CR>.		

Table 14 – RS485 Message Format

The binary checksum is calculated the same way as the checksum for the master to slave message.

Example:
Consider the following command string
UM=5,SP=1000,GG,PX
The first command sets the Metronome to the position mode, the second sets the point-to-point speed, the third is a non-existent command string, and the last is a request to report the axis position.
If EO=1, the returning message body will be (Assuming the axis position of 0)
UM=5:,SP=1000:,GG[2]?,PX0:
The response to the GG command is the error code 2, which means “Bad Command”. In the above, the notation [2] designates the binary character 2.
If EO=0, the returning message body will be
:,:,[2]?,0:

Important note:
Although generally slave to master strings will be transmitted contiguously, this is NOT guaranteed. For example, a concatenated command string includes commands of long executing time. Consider the command string UM=2;MO=1; . The UM=2 command changes the unit mode, thus invalidating the database. With invalid database, the MO=1 command will recalculate the database before power is applied to the motor – and this may take few msec. The response to UM=2 is ‘;’ and it will be transmitted immediately. Few msec later the response to MO=1 will be also transmitted.

The master must not use the line idle condition to decide that the slave response is complete. It must wait until the terminating checksum is accepted.

5.3.3.3 Exceptions

5.3.3.3.1 Line Errors

Line errors are related to high noise, too long lines, bad earth connections, or wrong baud rates.
A line error may also occur if the Metronome could not handle the incoming communication in its due rate.
The line errors are identified as noise errors, frame errors, parity errors, or character over-run.
Following a line error, the body of the present master command is ignored, and the command is answered with an error code. A line error kills an entire message. If the message was made of several concatenated sub-commands, all of them will be ignored.
The Metronome responds to a corrupted command only if it identified the command addressed to it.
A line error may kill two messages if it occurs at a message terminator.

5.3.3.3.2 Collisions

The RS485 is a half duplex line. When two nodes try to transmit simultaneously, a collision results. A transmitting Metronome can detect a collision with a good probability, since it listens to its own transmission. If a collision occurred, it will not receive exactly the same string it transmitted.
A Metronome that detected a collision sends a couple of break (null) characters and then cuts its transmission off. All what the Metronome desired to say at the time of the collision is lost – this may also be several queued messages.

5.3.3.4 Message Format and Checksum Errors

The Metronome ignores a badly formatted message, or a message with erroneous checksum.

5.3.3.4.1 Related Error Codes

The following table details the error codes relevant for RS485 communication errors.

Error code	Description
15	Collision detected.

32	Line error
33	Over-run error
34	This command must be addressed to a single Metronome. **Example:** Trying to set the representative mode in a command that is addressed to a group or to the entire network.

Table 15 – RS485 Related Error Codes

5.3.3.5 Booting a Network

Composer program with point-to-point connection
The composer can work with a Metronome that is set up for RS485 simply as if this Metronome is set for RS232. An RS232 cable is connected from the serial port of the PC that runs the Composer to the Metronome. The communication runs through the physical RS232 lines, although the protocol is RS485.

5.3.3.6 Network With Metronomes

We assume that the communication parameters of the Metronomes are already in the flash memory.
The following caution measures must be taken.
Every Metronome in the network must have its unique ID. Never assign the same ID to two Metronomes.
If groups are used, one and only one Metronome in each group must be a "group representative".
If broadcasting to the entire network is used, one and only one Metronome must be the "net representative".
The following precautions may help:
Use the "ID every message" method whenever practical.
Remember that if a group or the entire network is addressed, some Metronomes do not give any feedback, even if they could not process the command successfully. Verify the life of each Metronome by its unique ID. Avoid broadcasting and group access whenever practical.
Avoid fetching long records from the Metronome at one time. Break long records to shorter sequences. The reason is that as the line is half duplex, no bus other bus activity can take place until the records fetching shall be over – and this may take few seconds in which the entire network cannot be commanded. In the case of true emergency, do the following:
Issue several time the '~' character. This will cause a "Bad echo" exception at the transmitting Metronome, so it will stop transmitting. All the Metronomes shall reset their communication streams, and go un-awake.
Transmit what you need to transmit.

Please note that all the Metronomes that are setup to work without "ID every message" power up awake. Therefore, if there is more then one Metronome in the network that is setup without "ID every message", a communication crash may result when more then one Metronome tries to transmit a response at the same time instant. This problem is easily solved by transmitting the communication reset character '~'. After the transmission of '~', no Metronome will be awake.

5.4 CANopen Communication

A detailed description of CANopen communications is out of the scope of this manual. Please refer the Metronome CANopen manual.

6 Motion Modes

This chapter presents the different Metronome motion modes. For each motion mode the related Metronome commands are listed and explained, followed by a motion initialization program. In addition, each motion mode is analyzed for theoretical and practical aspects. Finally, this chapter also describes implementation of special motion features, such as searching for home/index/limit.

6.1 Profiling a Speed Command

Consider the following scheme:

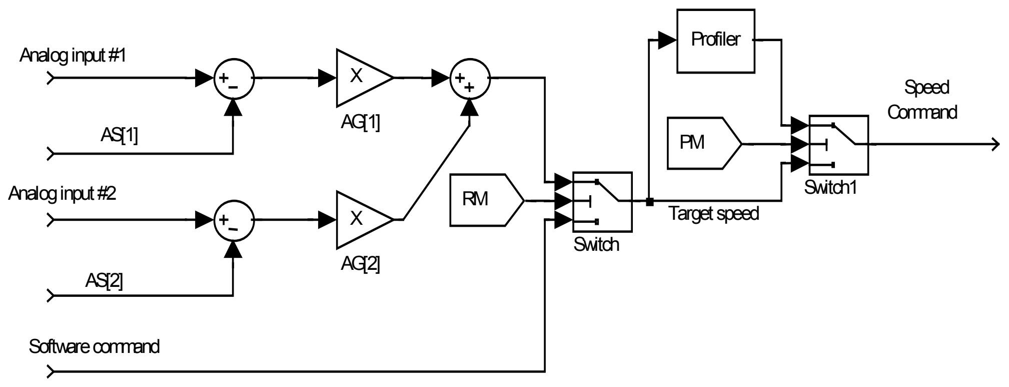


Figure 1 – Speed Reference Generation Model

In this scheme, a target speed is asserted by software or by an analog command, according to the RM parameter. The target speed may serve directly as the reference for the speed controller, or the speed reference to the controller may be profiled, according to the PM parameter.
By profiling, we mean that the reference to the speed controller is derived from the speed target to reach the desired speed within the permitted acceleration and deceleration. The acceleration is not developed at once, in order to avoid shocks. The time it takes to develop the final acceleration is called the “smooth factor”.

Example:

In this example, we apply the sequence
MO=0;JV=4000;AC=100000;BG;
with three different values of SF.
The SF=0 graph displays sharp corners, since its acceleration is not continuous.
The SF=10 graph takes 10 milliseconds more to stabilize the speed reference, but the speed reference profile is much smoother.
For the SF=50 graph, the smoothing is so strong with respect to the total acceleration time that the AC acceleration is never achieved.

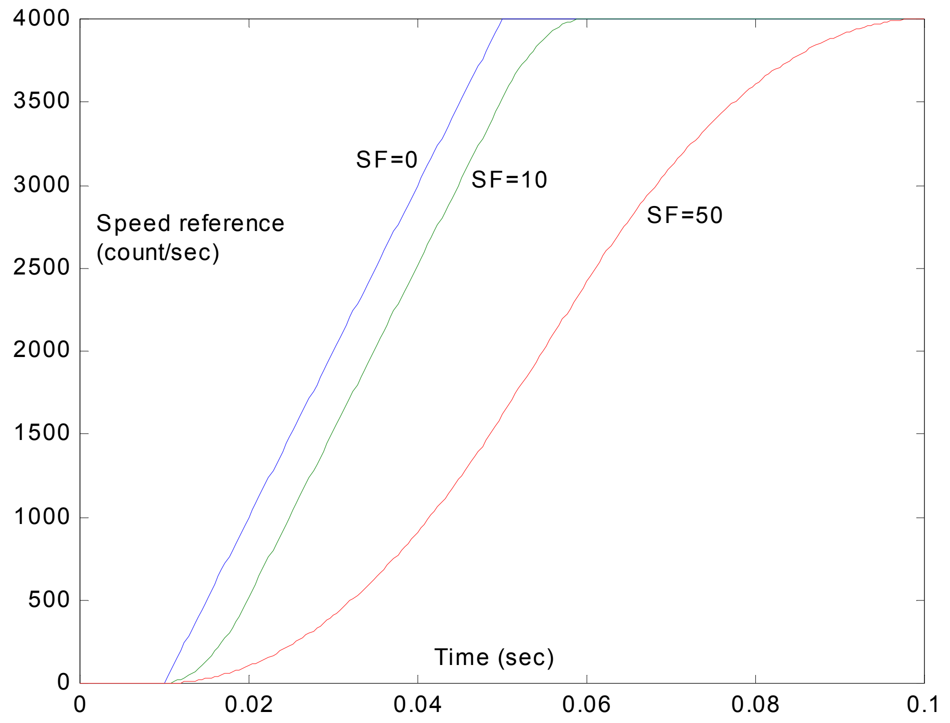


Figure 2 – Smooth Factor Behavior

The profiler smoothes the motion and prevents overshoots, but it also introduce a delay in the controller response. Therefore, it is not recommended to use the profiler if the Metronome implements an inner speed controller that is controlled by external position controller.
The parameters AC and DC control the rate in which the speed command may be changed. The AC parameter is used whenever the target speed is greater (in absolute value) then the present controller reference speed. Otherwise, the DC parameter is used.

Example:

We set AC=100000, DC=20000, SF=0, PM=1.
The figure below depicts how the speed command to the controller tracks the target speed specified by the JV parameter.

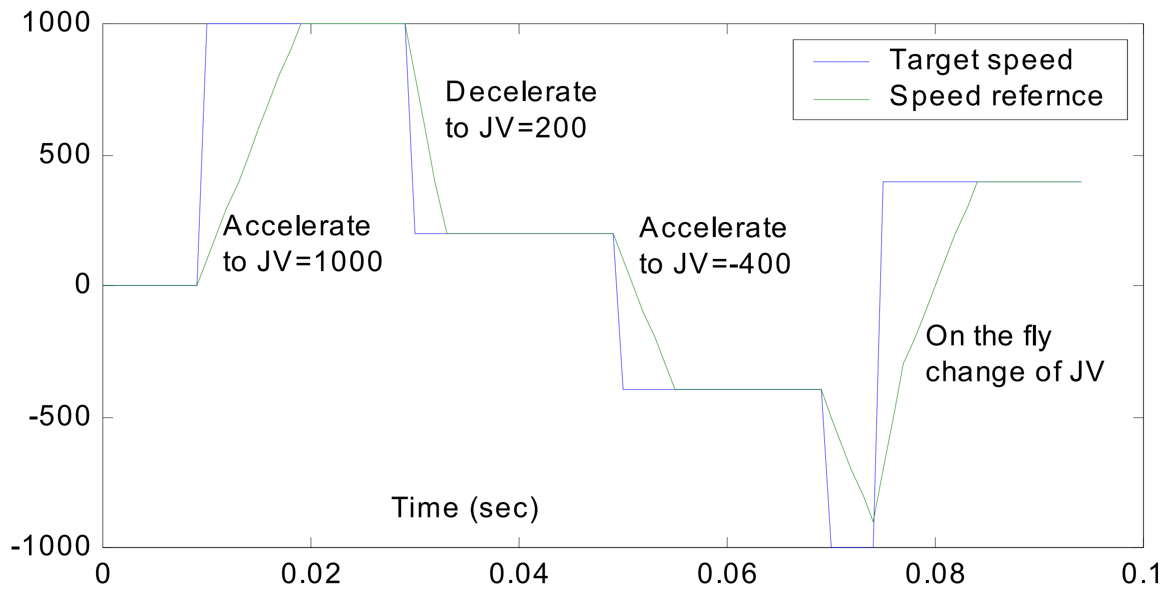


Figure 3 – Speed Profiling

The speed reference to the controller may be changed any time, regardless of the state of the profiler.

6.2 Summary of The Parameters Related To Speed Command Generation Profiling

Parameter	Action
RM	Define if an external reference is used at all RM=0: Don't use external reference RM=1: Use external reference
AG[1],AG[2]	Scale the analog inputs. The units of AG[1,2] are counts/sec/Volt. (RM=1 only)
AS[1],AS[2]	Offset the analog inputs. The units of AS[1,2] are Volts. (RM=1 only)
JV	Software speed command, counts/sec (RM=0 only).
PM	Profiler action. 0: Don't use profiler 1: Use profiler.
AC	Acceleration in $counts/\sec^2$
DC	Deceleration in $counts/\sec^2$
SF	The time it takes to develop the full acceleration of AC or DC, in milliseconds.

7 Position Controlled Motion Modes

This section summarizes how the Metronome generates the motion command for the position controller.

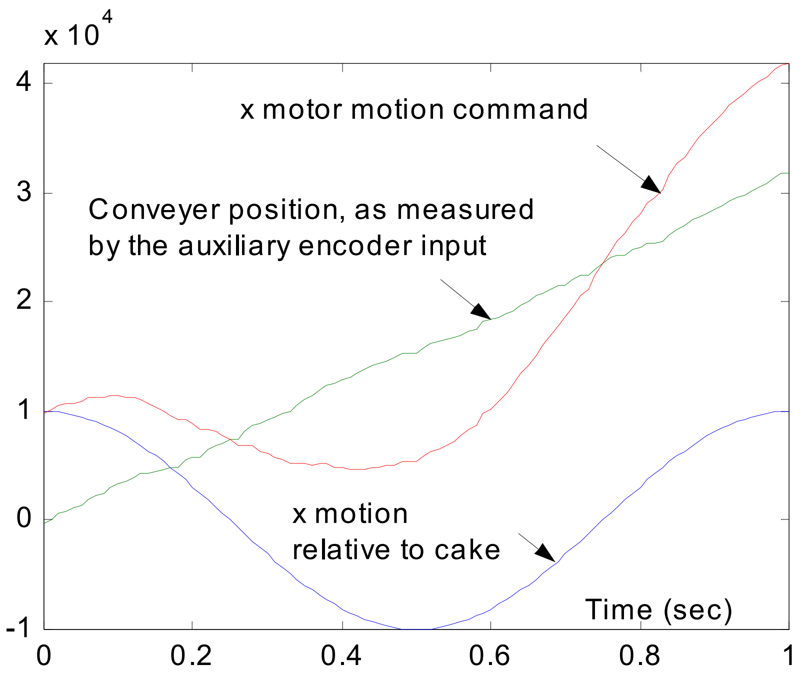
7.1 Position-Reference Generation Model

The position reference for the metronome is generated as the sum of a software reference and an external reference.
The software reference is derived by software commands, either communicated or commanded by a program. An example for a software command is a PTP (point to point) motion, in which the host, or the program specifies the desired absolute position. The Metronome builds a motion trajectory towards the desired absolute position, according to the PTP motion parameters AC, DC, SP and SF.
The external reference is derived from the analog inputs, and from the auxiliary encoder.
This arrangement allows the Metronome to design accurate motions that are relative to a moving, arbitrary frame.

Example:

Consider the application depicted below:

In this application, an x-y stage draws a chocolate picture on a cake while the cake travels on a conveyer.
The drawing has to be accurate with respect to the cake.
In order to draw a circle of radius 10000 encoder counts on the cake in one second, the x-axis motor must follow the trajectory
$x = 1000\cos(2\pi t) + c(t)$
Where c(t) is the position of the conveyer , as depicted below.



The block diagram below describes how the Metronome generates its position reference.

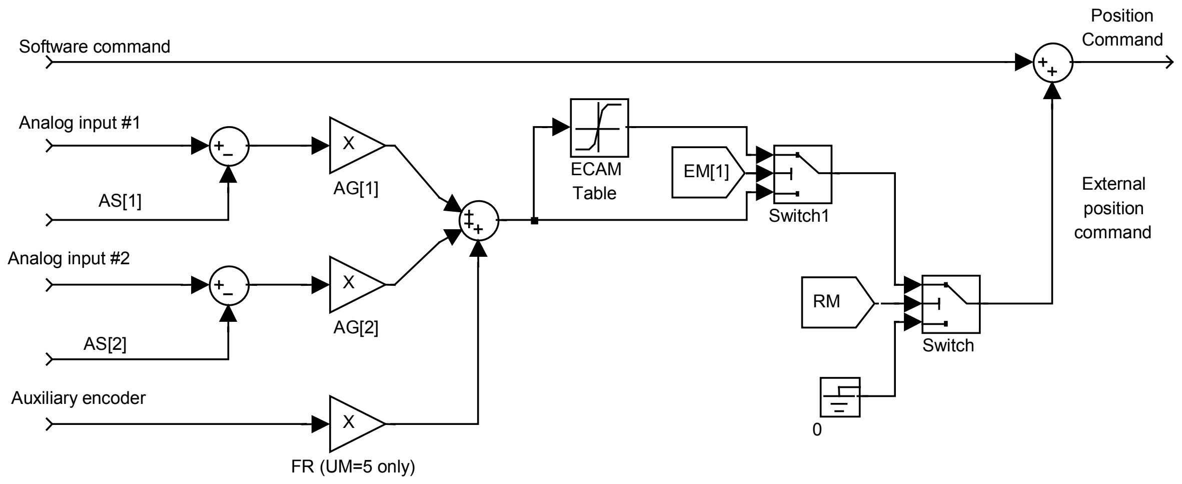


Figure 4 – Position Reference Generation Model

The following parameters determine how the position reference is composed:

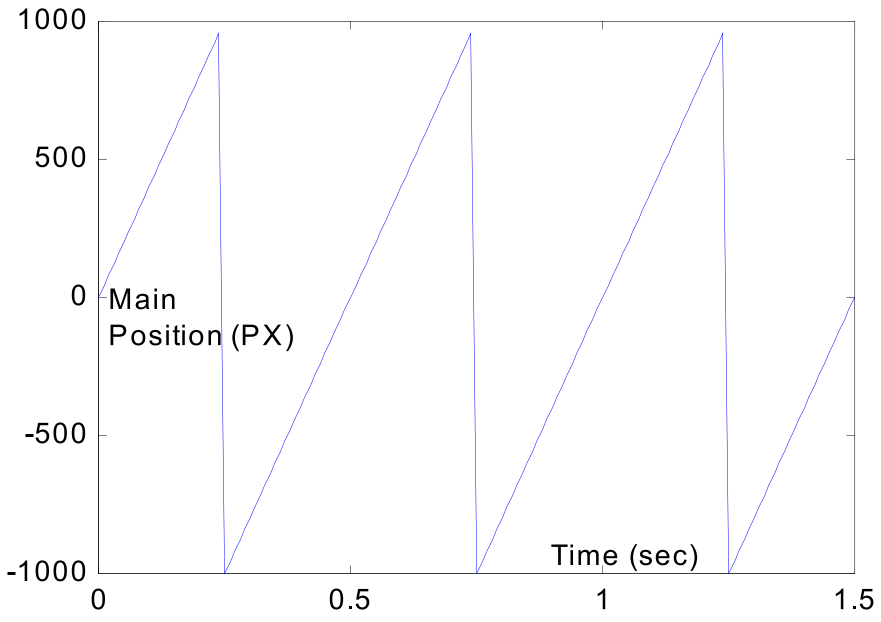
Parameter	Action
AG[1],AG[2]	Scale the analog inputs. The units of AG[1,2] are counts/Volt. Note: AG[1] and AG[2] are unit mode depended (UM).
AS[1],AS[2]	Offset the analog inputs. The units of AS[1,2] are Volts.
FR	Scale the auxiliary encoder input. FR is applicable only if the auxiliary encoder is not used for position feedback.
EM[1]	Define if the ECAM table transforms the external reference or not. EM[1]=0: Do not use ECAM table EM[1]=1: Use ECAM table for transforming the external command.
RM	Define if an external reference is used at all RM=0: Don't use external reference RM=1: Use external reference

7.1.1 Modulo Counting

The variable PX counts the distance traveled by the motor. The PX variable cannot increase indefinitely. After reaching a certain value, specified by the variable XM, the PX variable rolls back.
This type of counting is called "modulo count".

Example:

Consider a motor that rotates in a constant speed of 4000 counts/sec, with XM=2000. The PX variable will behave as depicted below.



The PX variable changes in the range [-XM/2 … XM/2-1], which is in this example [-1000..999].

The largest possible modulo is 2^{31}. With this modulo setting, PX varies in the range $[-2^{30}..2^{30}-1]$.
For easy life, XM=0 is equivalent to XM=2^{31}.

The variable PY counts the distance traveled by the auxiliary encoder. The PY variable is counted modulo YM, similarly to the way PX counts modulo XM.

7.2 Software Position Motions

This section describes how the software position command is generated. The Metronome software knows to generate several types of motion profiles. These are:

- PTP – Point-to-point motions
- Jog – Constant speed motions
- PVT – A tabulated arbitrary motion type, in which the user specifies a sequence of times, and the required positions and speeds at that times.
- PT A tabulated arbitrary motion type, in which the user specifies a sequence of positions, to be visited at equally spaced time values.

The initiation of all the software motions has the same sequence:

- Specification of the motion parameters (For example, the acceleration, the deceleration, and the speed for PTP motion)
- Specification of the next motion.
    - PA=n specifies a PTP motion, to the absolute position n counts
    - JV=n specifies a Jog motion, with the speed n counts/sec
    - PV=n specifies a PVT motion, beginning from the n'th item in the PVT data tables
    - PT=n specifies a PT motion, beginning from the n'th item in the PT data table.
- A BG command (or a BI command with a hardware trigger) initializes the motion.

7.2.1 Switching Between Motion Modes

Any type of motion may be specified on the fly. In special, PTP and jog motions can be commanded to anywhere, any time. This is easy, since the Metronome will calculate the path to be followed so that the desired speed or position is reached, subject to the acceleration limits. Care is required, however, when switching to the tabulated (PT and PVT) motion modes, where the user specifies the position reference directly. For example, consider that the maximum position error ER[3] is set to 1000, and that the present position command is 0, and that a PT motion is commanded to start at the position of 2000. The motion shall be aborted immediately because of excessive position error. It is a good practice to use PTP motions in order to arrive the starting point of a tabulated motion. The MS command can be used to verify that the PTP motion is over.

7.2.2 Tabulated Motion Modes

In tabulated motion modes, the user specifies the entire path followed by the motor. This is in contrast to the PTP and the Jog motions, in which the user just states a steady state target (either position or speed), and lets the Metronome decide how to achieve the steady state target.
Two tabulate motion modes are defined.
Two tabulated motion modes are supported: PT and PVT.
In the PT mode, the user specifies a sequence of points to be visited, with equal time intervals between them. The time interval between the user-specified points, which must be an integer multiple of the controller sampling time, is given only once as a parameter. Between the user specified points, the. The Metronome also calculates the speeds in which the user points are visited, to achieve maximally smooth motion.
In the PVT mode, the user specifies for each visited point not only the position, but also the speed and the time. Between the user specified points, Metronome interpolates with a 3rd order polynomial. For reasonably smooth motion profiles, the PVT method requires less data points communicated to the servo drive.

The following tables compare the PT and the PVT motion modes.

Feature	PT	PVT
Motion buffer length	1024	64
Position points in one CAN message	2	1
Speed command calculation	Automatic, by 3rd order interpolation	Asserted by user
Acceleration command	Automatic, by 3rd order interpolation	Automatic, by 3rd order interpolation
Time between user data points	A fixed multiple of the 1 to 255 sampling times of the controller. Cannot be changed on the fly.	Asserted by the user independently for each motion interval, in msec. In the range [1,255] msec.
Cyclical motion support	Yes	Yes
On the fly motion programming with hand-shaking host protocol	Yes	Yes

Table 16 – Tabulated Motion Difference

Feature	Preferred
High resolution motion programming	PT
Ease of use	PT
Low resolution motion programming	PVT
Variable command sampling time	PVT
Motion design independent of controller sampling time	PVT

Table 17 – Tabulated Feature Preference

7.2.3 Point-To-Point (PTP)

7.2.3.1 Basic Point-To-Point

In this motion mode the motor will move from its present position to a final point. The final point is arrived in zero speed and the motor stays there.
The trajectory to the final point is calculated based on the speed, acceleration, and deceleration limits, as set by the AC, DC, and SP parameters respectively.

The largest PTP motion available is XM/2 (or 2^{30} if XM=0). This is since the PTP motion will always go the short way. For example, if XM=1000, the present position reference is 490, and the command PA=-490;BG is entered, then the position reference will increase and go through 499 to –500 and then to –490. The total length of the movement shall be 20 counts.

The parameters of PTP motion are summarized in the table below.

Parameter	Action
AC	Acceleration in counts/sec^2
DC	Deceleration in counts/sec^2
SP	Maximum speed in counts/sec
SF	Smooth factor, in milliseconds
RP	Relative position in counts
PA	Specifies that the next motion shall be a PTP, and the absolute target position.

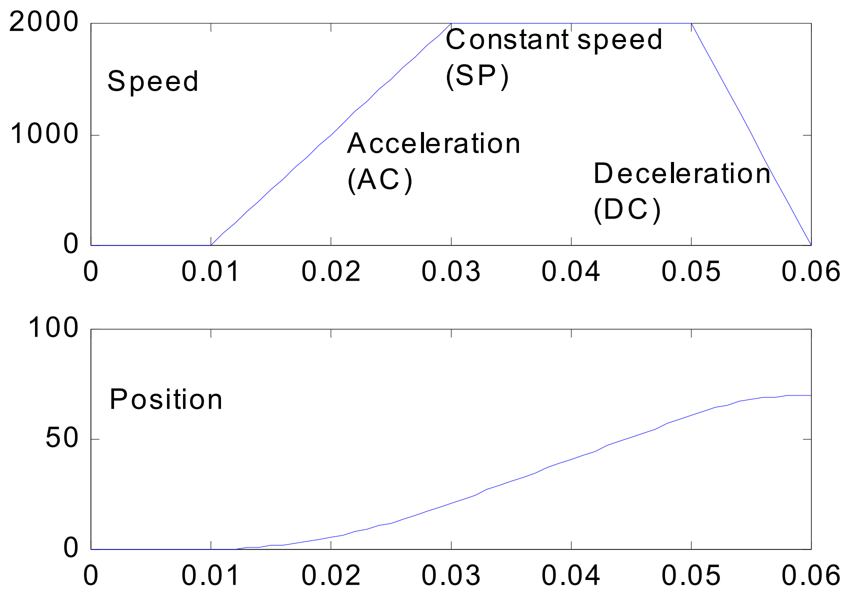
Table 18 – Parameter of PTP Motion

Please note that not like all the other parameters, PA is also a command.
Entering PA=n specifies that the next BG will start a PTP motion, regardless of the present motion type.
The value of PA is incremented automatically with PR every time a new motion is initiated.
That way, the sequence
PA=0;BG;AF,MS=0;PR=1000;BG; AF,MS=0;BG; AF,MS=0;BG
will initiate four consecutive PTP motions, targeted to 0,1000,2000,and 3000 counts respectively.

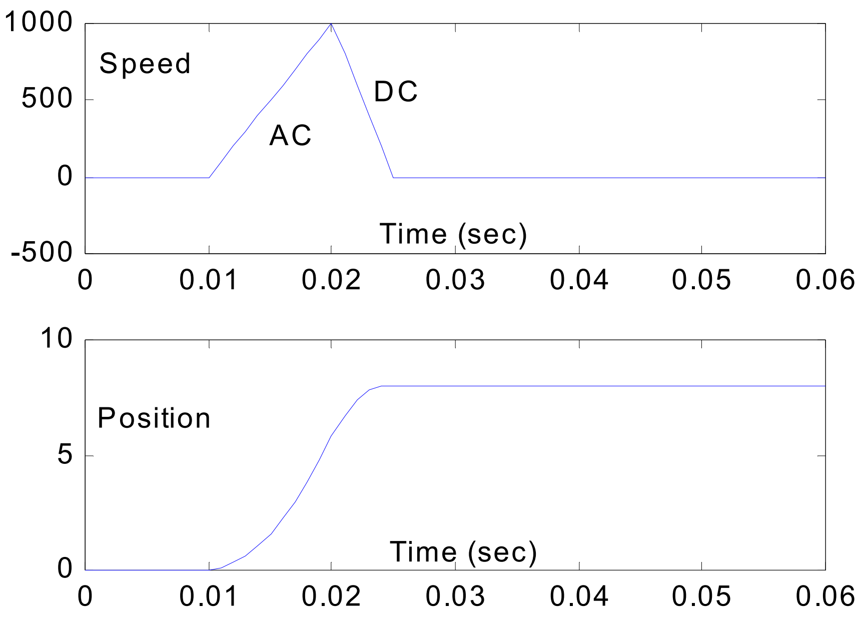
Setting PA clears RP.

7.2.3.2 Example

The simplest point-to-point motion is from a stationary position to another stationary position.
The acceleration and the final speed are AC=100000,DC=200000,SP=2000.
We begin from PX=0 and command PA=70;BG
The speed graph shows clearly the acceleration, constant speed, and deceleration state in the trajectory to the target.



With shorter movement, the deceleration begins before the acceleration achieves the limit speed, so that the SP speed limit is not effective. This situation is depicted in the figure below:



The Metronome sets the total time of motion to the minimum that is possible within the acceleration and the speed limits. Shorter motions take less time.

7.2.3.3 More Complicated PTP Motions

PTP motions may be initiated any time, using the PA command, not necessarily from a stationary state. The PTP decisions, made every position control cycle, are given in the flow chart below.

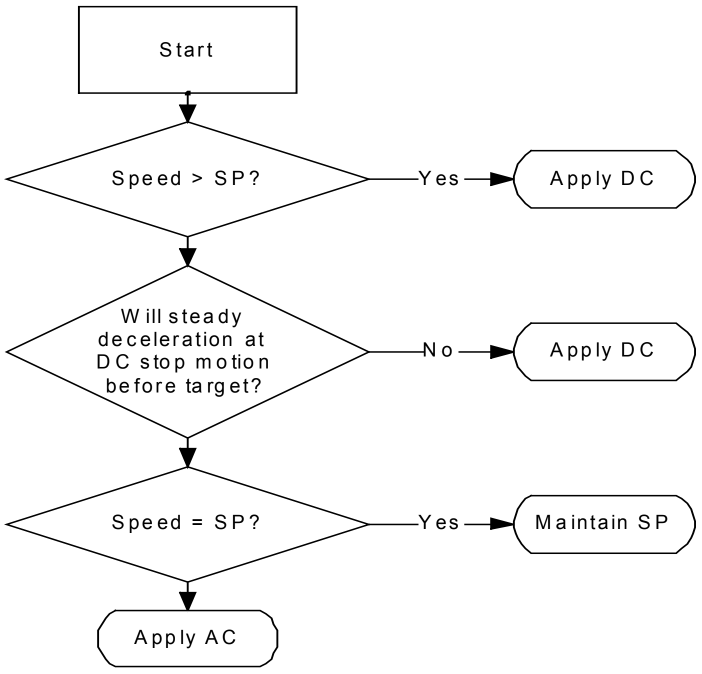
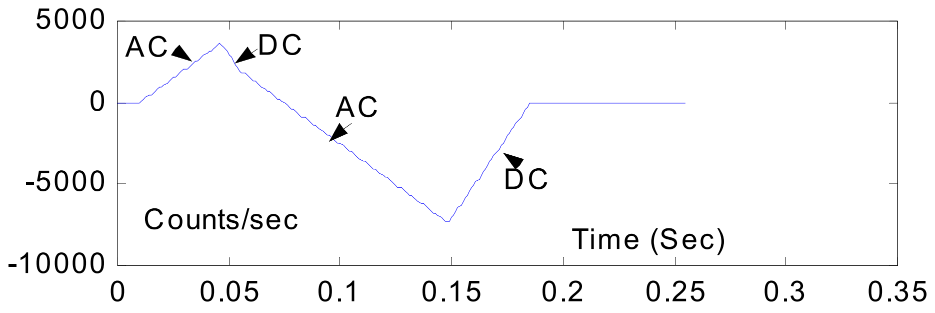


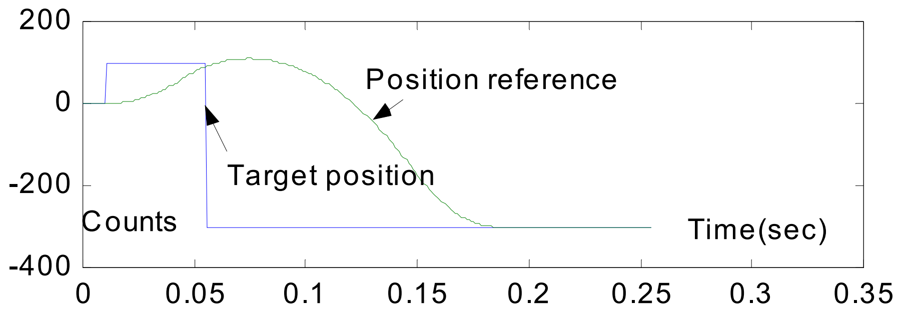
Figure 5 – PTP Decisions Flow Chart

All the parameters of the above flow chart, including AC,DC, SP, and the position target are updated by a BG command, or by its hardware activated version BI.

7.2.3.4 Example – Change of The Position Target On-The-Fly

The acceleration and the final speed are AC=100000,DC=200000,SP=20000.
We begin from PX=0 and set the target position by PA=100;BG.
The position reference has not yet stabilized to the position target, when the target position switched by the command
PR=-400;BG;
To the position target of (PA+PR)=-300.
The Metronome has no problem with that – it calculates the position reference to reach the new target.
Note that in this case the position reference overshoots the original target position of 100. This is since at the time the target position was changed, the Metronome was approaching the target using DC. When a new target is set, the Metronome exits the state of final approach. It therefore uses the AC acceleration, which in this example is smaller, then DC. With the reduced acceleration, it cannot avoid the overshoot.





7.2.4 Jogging

In a jogging motion, the motor is commanded to move in a fixed speed. The acceleration (or deceleration) to the desired speed is made using the AC and the DC parameters.
Jog motions may be initiated any time by using the JV command, not necessarily from a stationary state.
The Jog mode decisions, made every position control cycle, are given in the flow chart below.

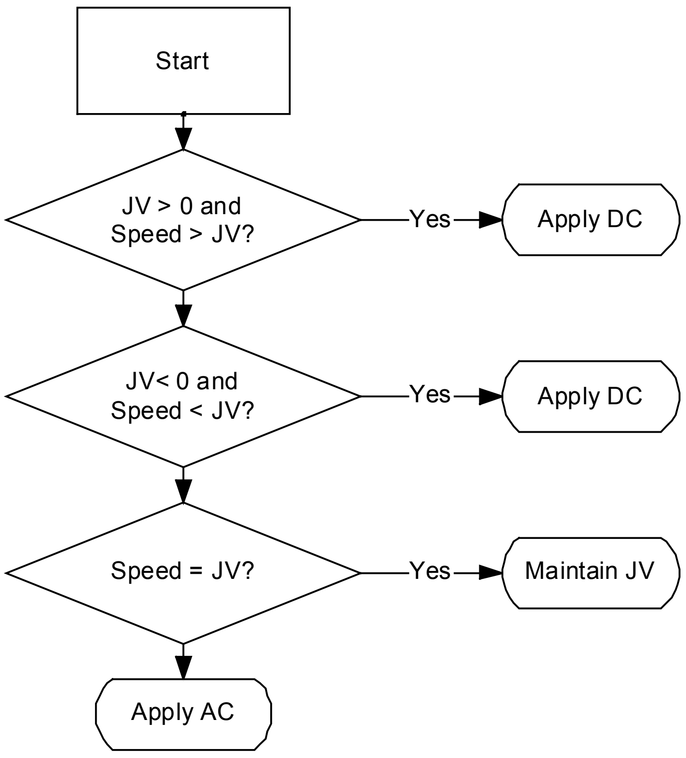


Figure 6 – Jog Decisions Flow Chart

As for jogging motions, all the parameters of the above flow chart, including AC,DC, SP, and the position target are updated by a BG command, or by its hardware activated version BI.

Jog motions can continue forever. When the position reference jumps when it arrives the modulo boundary (+/-XM/2) but speed is kept constant

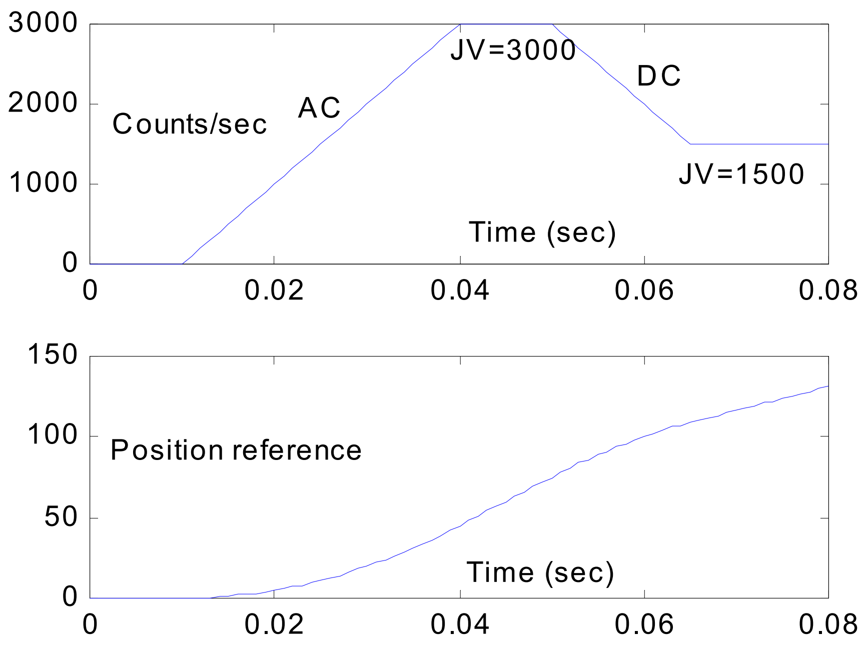
The parameters of PTP motion are summarized in the table below.

Parameter	Action
AC	Acceleration in counts/ sec^2
DC	Deceleration in counts/ sec^2
SF	Smooth factor, in milliseconds
JV	Jog velocity

Please note that not like all the other parameters in the above table, JV is also a command.
Entering JV=n specifies that the next BG will start a jog motion, regardless of the present motion type.

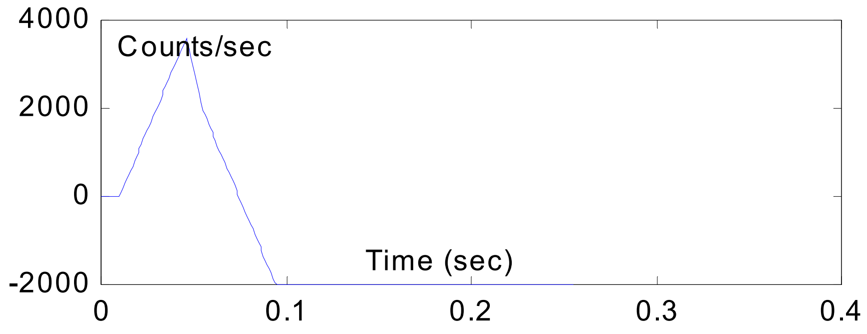
7.2.4.1 Example – Simple jogging

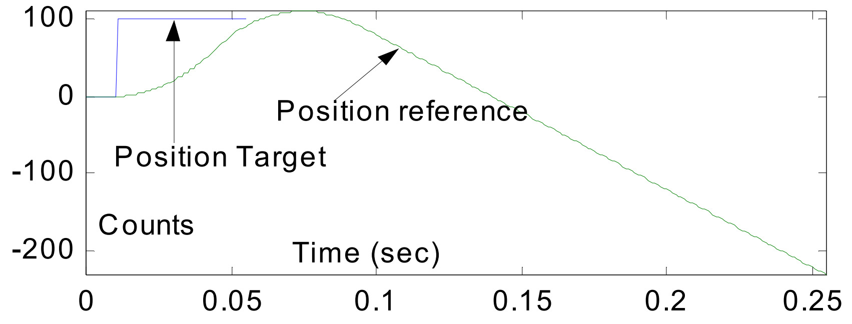
The example is started by the command sequence JV=3000;BG
The position reference starts to accelerate until the jog speed of 3000 is reached.
Later the command sequence JV=1500;BG changes the speed of the position reference to 1500 counts/sec.



7.2.4.2 Example – On the fly mode switching

The example below is started by the command sequence
AC=100000;DC=200000;PA=100;BG;
A PTP motion starts, in which the Metronome brings the reference position to the position target.
At the time of 0.055 sec, where in the picture below the plot of the position target is terminated, the command JV=-2000;BG; is entered. The Metronome uses the AC acceleration to arrive at the desired speed.
The Metronome would use the DC parameter instead of AC, if the speed of the position reference were more negative then –2000 at the time of the BG.


4000
2000
0
-2000
Counts/sec
Time (sec)
0
0.1
0.2
0.3
0.4


100
0
-100
-200
Position reference
Position Target
Counts
Time (sec)
0
0.05
0.1
0.15
0.2
0.25

7.2.5 The PVT and PT - Tabulated Motion Mode

This section defines the basics of ELMO PVT mode of interpolated motion.
In this mode, the user must define the position and the speed of the motor at given time points.
Another interpolation motion mode is the PT mode, in which the user must specify the interpolated positions only, leaving the job of speed calculations to the servo drive.
The PVT mode may be used for generating complex tracking motions, without an excessive communication load.
The PVT mode requires more care in programming then the PT mode. This is since the user has to calculate speeds. Bad speed specification may lead to very weird time-position trajectories.
In contrast to the PT mode, the user specifies the motion time in msec, not in multiples of the Metronome sampling times. This makes the motion plan less vulnerable to a modification of the Metronome settings.
Please refer to Table 16 – Tabulated Motion Difference and Table 17 – Tabulated Feature Preference.

7.2.5.1 What Is PVT?

PVT stands for Position Velocity Time.
In a PVT motion, the user provides the desired position and speed at certain time instances. Between the times specified by the user, the motion controller interpolates to obtain smooth motion.
The position and speed specifications are absolute. The time specification is relative.

7.2.5.2 Example

Suppose we want the system to be at the position of 1000 and the speed of 100000 count/sec at a given time, and to be at the position of 1200 and the speed of 190000 count/sec, 6 msec later.
Then the first point is specified by
Position = 1000,
Speed = 100000,
Time = (Not relevant to the described motion, but to the previous motion segment)
The second point is specified by
Position = 1200,
Speed = 190000,
Time = 6msec

The time is defined with msec units. The time specification does not have to coincide with the specific sampling time of the Metronome. The Metronome interpolates the motion so that the exact desired motion timing is kept, regardless of the sampling time of the controller.

7.2.5.3 Interpolation Mathematics

PVT implements a 3rd order interpolation between the position-speed data provided by the user.
The user provides the following data:

At the time t_0, Position and speed given by P0 and V0, respectively

At the time $t_0 + T$, Position and speed given by PT and VT, respectively

The position for $t \in [t_0, t_0 + T]$ is given by the 3rd order interpolating polynomial:

$$P(t) = a(t - t_0)^3 + b(t - t_0)^2 + c(t - t_0) + d$$

The speed is obtained by taking the derivative:

$$V(t) = 3a(t - t_0)^2 + 2b(t - t_0) + c$$

The four parameters a, b, c, and d are unknown. They can be solved using the four linear equations

$P(t_0) = P0$, Namely $d = P0$.

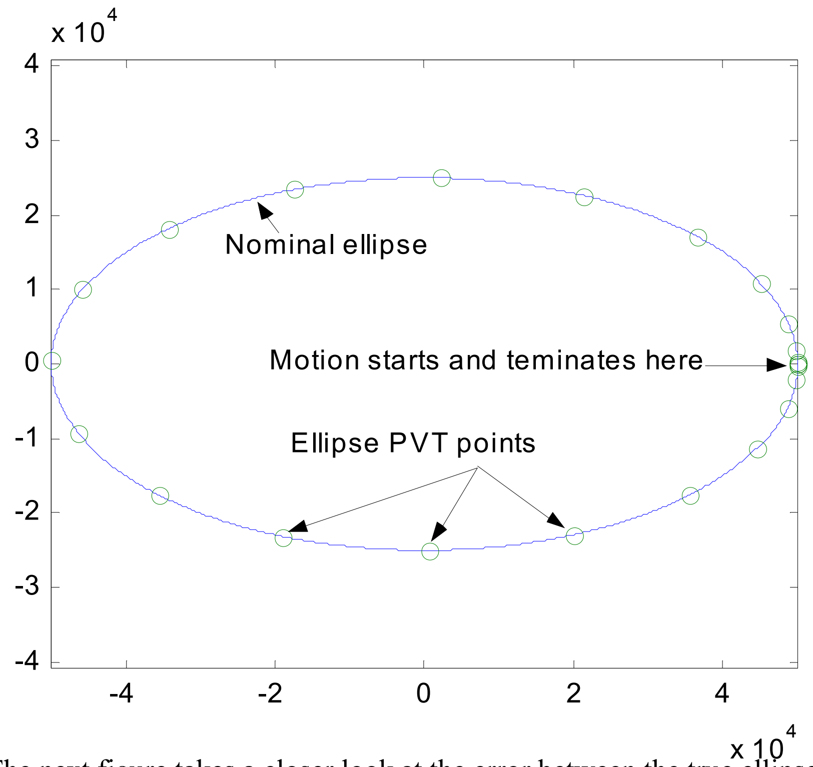
$V(t_0) = V0$, Namely $c = V0$

$P(t_0 + T) = PT$, Namely $PT = aT^3 + bT^2 + cT + d$

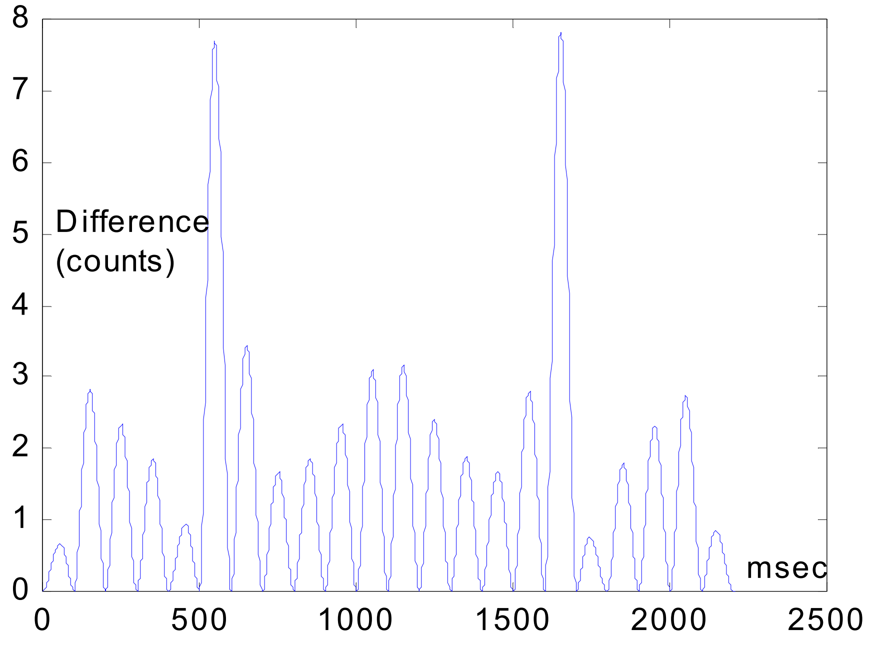
$V(t_0 + T) = VT$, Namely $VT = 3aT^2 + 2bT + c$

7.2.5.4 Example

In this example, we demonstrate how very few points can accurately describe a smooth motion.
Consider two Metronomes, driven synchronously do draw an ellipse. One Metronome drives the x-axis, and the other drives the y-axis (this is readily done with PVT).
The long axis of the ellipse is 100000 counts long, and the short axis of the ellipse is 50000 counts long. The entire ellipse is to be traveled within 2.2 seconds.
We planned a PVT motion with an inter-point time of 100millisec. 23 points thus sample the entire ellipse, as seen in the figures below. The motion is planned so that the tangential speed is accelerated to a constant rate, and then decelerated back to zero speed at the end of the ellipse. Near the starting point of the ellipse, the speed is slow – and therefore the PVT points, which are equally spaced in time, are more spatially dense there. The continuous line in the figure below depicts the true ellipse and also the ellipse generated by the Metronome by interpolating the PVT points.
The true ellipse and the Metronome interpolation of the PVT points are so close, that they can't be resolved on the plot.

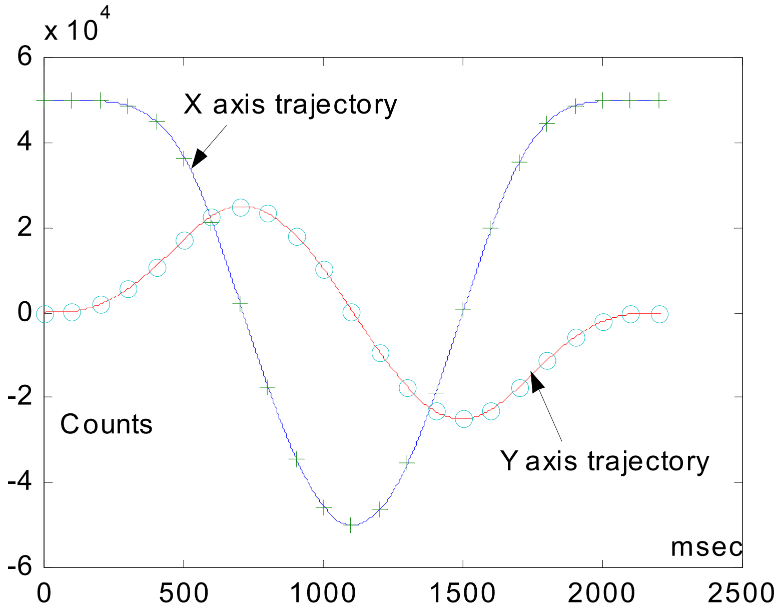


The next figure takes a closer look at the error between the true ellipse and the Metronome interpolated path.



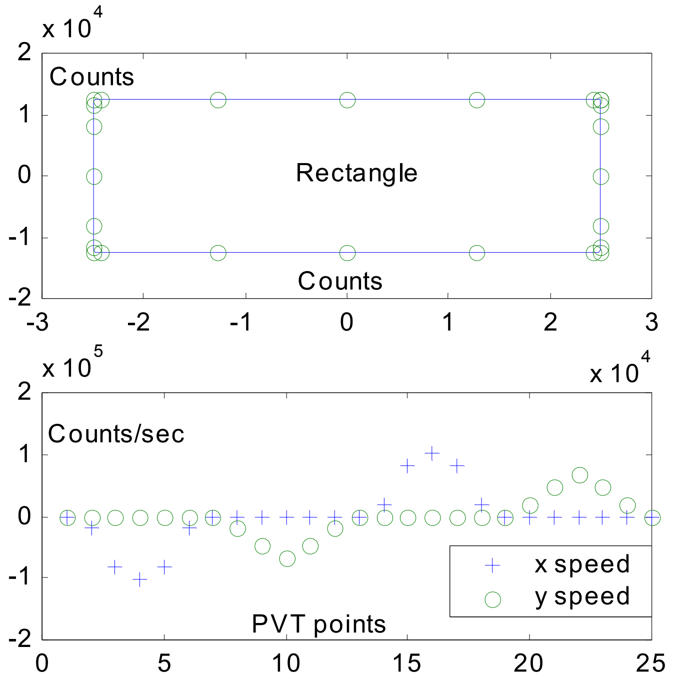
We observe that with 23 PVT points only, the interpolated path never differs from the true ellipse by more then 8 counts (remember that the ellipse axes are 100000 and 50000 respectively)!
At the PVT points, the interpolation error is of coarse zero.

The next figure displays the interpolated trajectories generated by the Metronomes for the x-axis and for the y-axis.



7.2.5.5 Example

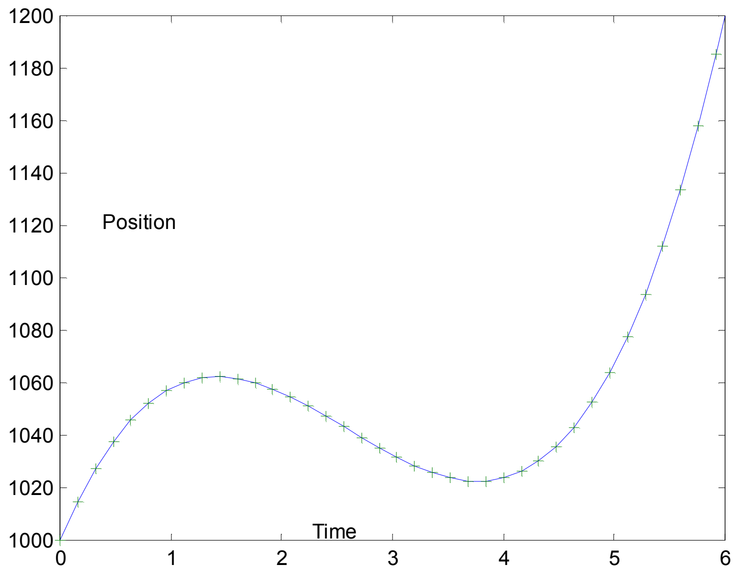
The Metronome normally produces maximally smooth interpolating trajectories. For this reason, the ellipse of the previous example could be interpolated using so few points.
Accurate corners require, however, that the interpolation is not smooth. Specifying zero speed for the corner points generates hard corners. The graph below shows a 2-axes synchronized PVT trajectory of an accurate rectangle.



For the corner points, both the x and the y speeds are specified to zero.
The interpolation error for the entire rectangle is zero.

7.2.5.6 Example

The interpolated trajectory for the data of example1 is depicted in the figure below, for a controller with a sampling time of 160 µsec.
The + symbols show the points at integer multiples of the controller sampling time. At these points, the Metronome evaluates the interpolated motion path.



We note the following:
The end point does not necessarily fall on a controller sampling time.
Weird looking position commands may result if the speed choice is not coherent with the position and time definitions. The position distance and the time between the points imply an average speed of (1200-1000)/0.006 = 33000 count/sec is specified. This average conflicts with the boundary point speed specifications of 100000 and 190000 respectively.

7.2.6 PVT Motion Programming

7.2.6.1 The Basic Mode

7.2.6.1.1 The PVT Table

A three-column table defines the PVT motion.
Each row of the table defines the position and the speed at a single time instant.
The table looks as follows:

#Index	P (32 bits)	V (24 bits)	T (8 bits)
1	P[1]	V1	T1
2	P[2]	V2	T2
…	P…	V…	T…
64	P[64]	V[64]	T[64]

Table 19 – PVT Table

The table has 64 rows, and can thus specify up to 63 consecutive PVT motion segments[6].
The cells of the PVT table may be accessed using the QP, QV, and QT commands.
The QP[N] command sets/reads the n'th row of the P column.
The QV[N] command sets/reads the n'th row of the V column.
The QT[N] command sets/reads the n'th row of the T column.

The positions in the table are limited to range of the position feedback, which is +/-2^{30} counts at most.
The speeds in the table are limited to +/-2^{23} counts/second.
The times in the table are in the range [1,255] msec.

7.2.7 Motion Management

In the PVT mode, the Metronome manages a "read pointer" for the PVT table.
When the read pointer is N, the present motion segment starts at the coordinates written at the Nth row of the table, and ends at the coordinates of the (N+1) row[7].
When the time period specified by QT[N] elapses, the N segment is done, the Metronome increments the read pointer to N+1, and reads the N+2 PVT table row to calculate the parameters of the next motion segment.

The user does not have to use the entire PVT table for a given motion.

[6] 64 segment if the table is used cyclically.
[7] The PVT mode may be cyclical, according to MP[3] – please refer the explanations below. In that case N+1 must be interpreted in the modulo sense.

The use of the PVT table is defined by the following parameters:

Parameter	Use	Comment
MP[1]	The lowest valid row of the PVT table	
MP[2]	The highest valid row of the PVT table	
MP[3]	0: Motion is to stop if the read pointer reaches MP[2]	Cyclical behavior definition.
	1: Motion is to continue when the read pointer reaches MP[2]. The next row of the table is MP[1].	

Table 20 – PVT Motion Parameters

The flow chart of the basic PVT mode is depicted below.

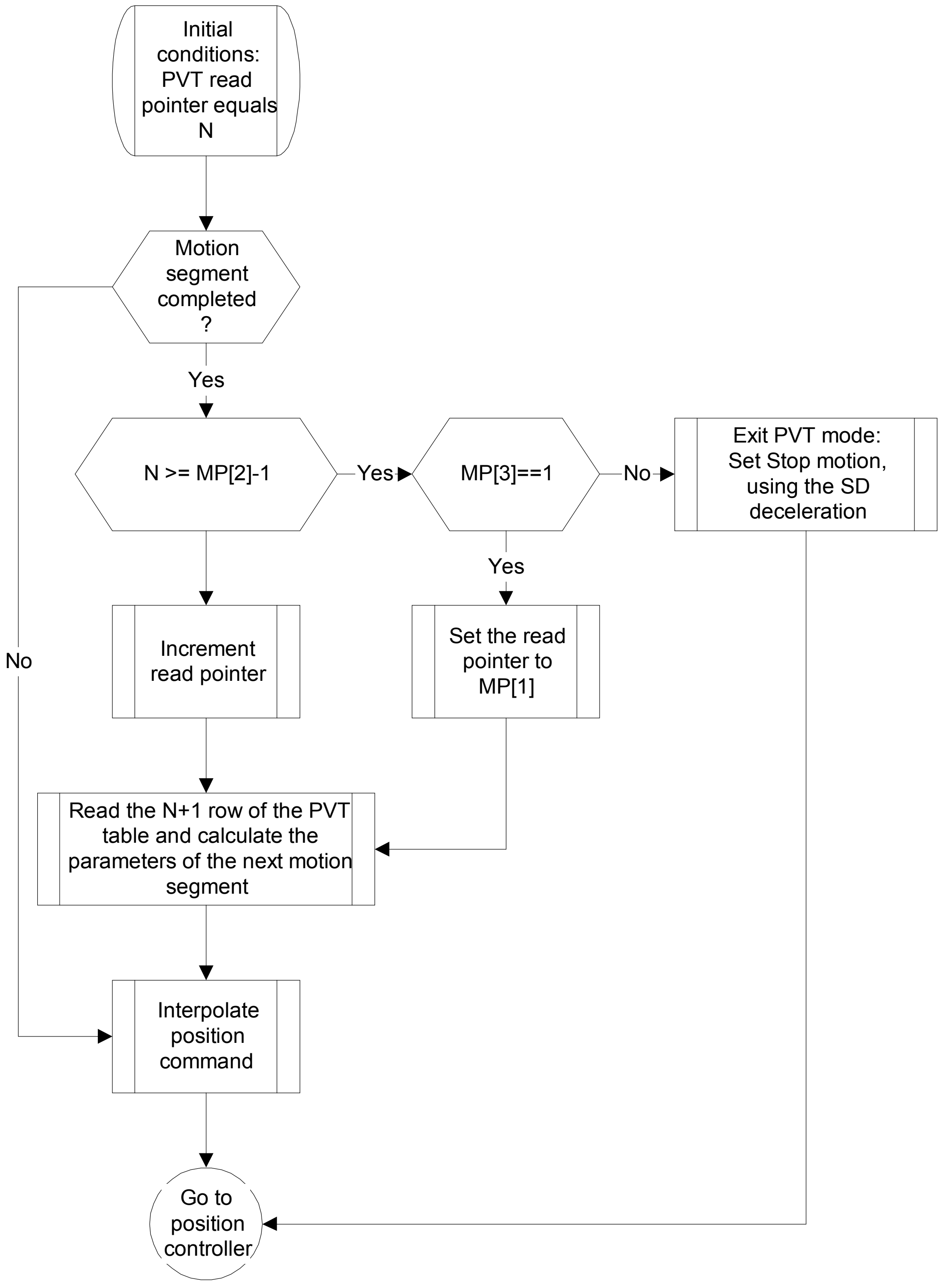

Initial conditions: PVT read pointer equals N
Motion segment completed ?
Yes
No
N >= MP[2]-1
Yes
MP[3]==1
No
Exit PVT mode: Set Stop motion, using the SD deceleration
Yes
Increment read pointer
Set the read pointer to MP[1]
Read the N+1 row of the PVT table and calculate the parameters of the next motion segment
Interpolate position command
Go to position controller

Figure 7 – PVT Decisions Flow Chart

A PVT motion is initiated by stating
PV=N with $1 \le N \le 64$, and BG.
The command PV=N sets the read pointer of the table to N and specifies that the next BG will start a PVT motion. BG starts the motion.
The PVT table may be written on-line while PVT motion is on. This has the following advantages:
An infinite-time non-periodic motion can be generated in the cyclical mode, by programming the PVT table on the fly. The Metronome can be programmed to interrupt the host by a CANopen emergency object when it is time to refill the PVT table with fresh data.
The unused part of the PVT table may be programmed for the next motion while the present motion is executing. An attempt to modify the data of an executing motion segment is an error.

7.2.8 Mode Termination

The PVT motion terminates upon one of the following cases:
The motor is shut down, either by programming MO=0 or by an exception
Another mode of motion is set active, e.g. by programming PA=xxx;BG. In this case, the new motion command executes immediately, without having to explicitly terminate the PVT mode.
The PVT motion manager runs out of data. This happens when the read pointer reaches MP[2] and MP[3] is zero. This may also occur in the auto increment mode (CAN communications only, see below) if the read pointer reaches the write pointer. In that case, the PVT motion is stopped immediately, using the SD deceleration. Note that if the last programmed PVT speed is zero, then the PVT motion terminates neatly, and the stopping at the end of the motion does nothing.

7.2.9 PVT Motion Using CAN

The PVT table allows the performance of pre-designed motion plans, as well as the on-line design of motion plan. On-line motion design is made by writing the PVT table while PVT motion is executing.
The on-line motion design ability is limited by the speed of the communication interface.
Consider an RS-232 ASCII communication interface.
Programming of a single PVT table row has the format
QP[xx]=xxxxxxxxx;QV[xx]=xxxxxxx;QT[xx]=xxx;
Up to 40 characters may be required to program a single PVT table row.
At the communication rate of 19200 baud, this may take 20msec.

The CAN communication option allows much faster PVT table programming, by packing an entire PVT table row into one PDO communication packet. Moreover, for easy synchronization with the host, the Metronome may be programmed to send an emergency object whenever the number of yet unexecuted motion segments falls below a given threshold.

7.2.9.1 The PVT Motion Programming Message

An entire row of the PVT table may be programmed by a single PDO.
The PDO used is 0x300+ID where ID is the node ID of the Metronome.
Note that before using PDO 0x300+ID for PT, its PDO mapping must be first set correctly.
The mapping of this PDO for the PVT mode is listed in the table below.

Object Dictionary Index	0x2001
Type	RECORD, 3 elements
Access	Write only
Structure:	Signed32 Position
	Signed24 Speed
	Unsigned8 Time
PDO mapping	Yes
Value limits	No
Default value	Not Applicable

The PDO does not specify the row of the table to be programmed.
The row to be programmed is specified by a “write pointer”.
The parameter MP[6] initially sets the write pointer. A new PVT CANopen message (Object 0x2001) automatically increments MP[6].

The CANopen auto increment mode has the following flow diagram:

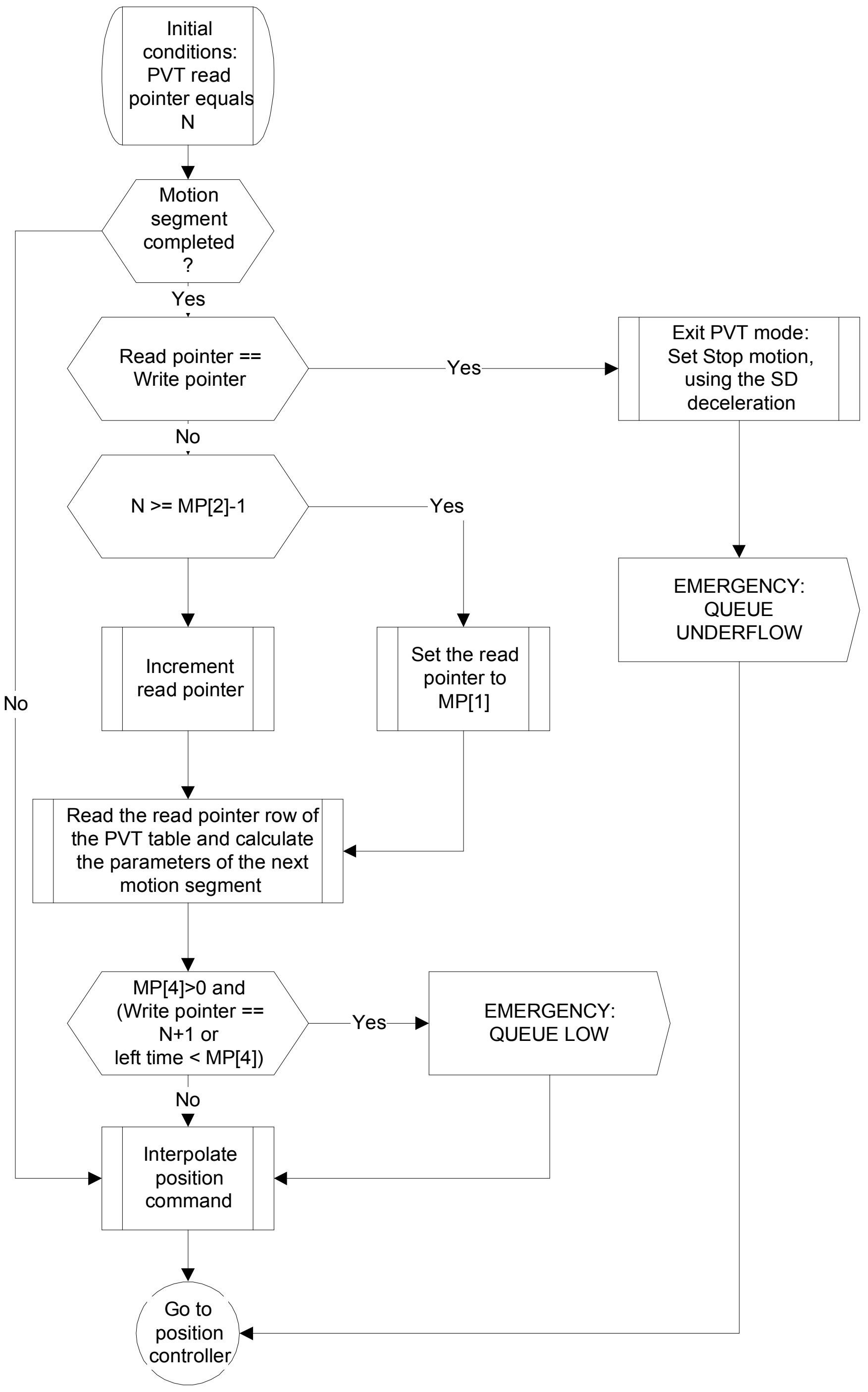

Initial conditions: PVT read pointer equals N
Motion segment completed ?
Yes
No
Read pointer == Write pointer
Yes
Exit PVT mode: Set Stop motion, using the SD deceleration
No
N >= MP[2]-1
Yes
EMERGENCY: QUEUE UNDERFLOW
Increment read pointer
Set the read pointer to MP[1]
Read the read pointer row of the PVT table and calculate the parameters of the next motion segment
MP[4]>0 and (Write pointer == N+1 or left time < MP[4])
Yes
EMERGENCY: QUEUE LOW
No
Interpolate position command
Go to position controller

Figure 8 – PVT Auto Increment Mode Flow Chart

The above flow diagram differs from the basic mode in the following issues:
Motion queue underflow is diagnosed by the read pointer reaching the write pointer
Emergency objects are issued for the queue low and for the queue underflow events.

7.2.10 Programming Sequence for The Auto Increment PVT Mode

PVT motion must start with initial programming of the PVT arrays.
Set
MP[1] = First valid line in the PVT table
MP[2] = Last valid array in the PVT table
MP[3] = 1 for cyclical mode.
MP[4] = Not relevant for PVT (PT only)
MP[5] = Number of left programmed motion rows to issue a “PVT queue low” warning. Set to zero if no “PVT queue low” warning is desired.
MP[6]=the write pointer. This is the next position in the PVT table to be written by the CANopen object 0x2001.

Set the QP[N], QV[N], and QT[N] array elements for at least the first used two rows of the PVT table. This is since PVT requires at least two time points to interpolate the motion trajectory.

Set (if not already set) UM=4 or UM=5, and MO=1
Set PV=N , where QP,V,T[N] and QP,V,T[N+1] are all programmed to valid values.
Start the motion by a BG.
Use the CAN PVT/Auto increment command for the rest of the PVT motion.
Use the queue low emergency signal to synchronize the host and the Metronome.

The queue low emergency message includes the present location of the read pointer and the write pointer. It is safe to send more PVT data PDOs until the write pointer is one location before the read-pointer specified by the queue low emergency message.

The host is well aware to the location of the write pointer, since it can count its own messages. A data message may be, however, rejected since the queue is full, or since a message is lost. In that case, the Metronome will issue an emergency object to the host. After receiving the emergency object the true location of the write pointer may be unclear. The host may then set MP[6] to the possibly rejected table row and continue the writing from there.

Please note:

About 1 msec may elapse between the reason for the emergency object and the time when the emergency object is actually transmitted.
Do not set MP[5] (Number of rows left for queue low emergency) to high. For example, consider a slow host managing a 64 rows long PVT queue in the Metronome. Suppose that the PVT row times are 10 msec each, and that MP[5]=55. The host gets a queue low emergency object telling that there are 64-55+1 = 8 free to program rows in the PVT table. The host takes 50msec to respond, and 5 msec to program each row. Thus, the 8 rows have been programmed in 90 msec. In the meantime, 9 additional rows have been executed, and there are only 54 valid rows in the PVT table. As 54 is lower then MP[5], there will be no more queue-low emergency messages until the PVT table is exhausted, and the mode is terminated. This situation may be remedied by the host asks the Metronome for PV (location of read pointer) after the PVT table writes are complete. If PV<MP[5], the programming process took too much time, and the writing must be continued.
Accurate timing with respect to the host is the essence of multiple-axis synchronized motion. Accurate timing may be achieved by using the CAN SYNC signal, and the CAN synchronized BG service, as described in the CAN communication documentation.

7.2.11 The Parameters of The PVT Motion Mode

The following parameters apply to PVT motion

What	How	Comment

Unit Mode (UM)	Unit modes 4 and 5 select the position mode motion path.	
Stop Deceleration (SD)	The rate of deceleration in the case where motion is killed by queue underflow or by an exception	
Position/Velocity/Time (PV)	Set a PVT motion command	
PVT table entries: QP[N], QV[N], QT[N]	Set values to the PVT table	The PVT table elements can also be set using PDOs as described above.
Motion Parameters (MP)	MP[1] = First valid row in PVT table	Configure a PT or PVT motion. MP[6] and MP[5] are for the CANopen auto increment mode only.
	MP[2] = Last valid row in PVT table	
	MP[3] = Cyclical motion (0 non-cyclical, 1 cyclical)	
	MP[5] = Number of yet unexecuted table rows for queue low alarm.	
	MP[6] = Initial value for the write pointer	

Table 21 – PVT Related Parameter

The following CAN emergencies are supported, all as manufacturer specific:

Error code (Hex)	Error code (Dec)	Reason	Data field
0x56	86	The queue is low: The number of yet unexecuted PVT table rows dropped below the value stated in MP[4]	Field 1: Write pointer Field 2: Read pointer
0x5b	91	The write pointer is out of the physical [1,64] range of the PVT table. The reason may be a bad setting of MP[6].	The value of MP[6]
0x5c	92	The PDO 0x3xx is not mapped	
0x34	52	Ann attempt has been made to program more PVT points then there is place in the queue.	Field 1: The index of the PVT table entry that could not be programmed.
0x7	7	Cannot initialize motion due to bad setup data. Reasons: - The write pointer is outside the range specified by the start pointer and the end pointer.	
0x8	8	Mode terminated, and the motor has been automatically stopped (In MO=1).	Data field 1: Write pointer Data field 2: 1 for End of trajectory in non cyclic mode 2 for A zero or negative time specified for a motion interval 3 for Read pointer reached write pointer
0x9	9	A CAN message has been lost.	.

Table 22 – PVT CAN Emergency Messages

7.2.12 PT Motion

7.2.12.1 What Is PT

PT stands for Position-Time.
In a PT motion, the user specifies a sequence of absolute positions to be achieved by the Metronome with equal time spaces. The time space must be an integer multiple of the Metronome sampling time. Between the user specified positions, the Metronome interpolates to obtain smooth motion.
The position specifications are absolute.
Interpolation Mathematics
PT implements a 3rd order interpolation between the position data points provided by the user.
Let $T = m \cdot T_s$
Where:
T_s is the sampling time of the position controller. The parameter WS[29] reads T_s.
T is the sampling time of the PT trajectory
m (The system parameter MP[4]) is the integer parameter relating T_s and T.

For the case $m = 1$, no interpolation is required.
For $m > 1$, there are sampling instances of the position controller for which the path command must be interpolated. We use a 3rd order polynomial interpolation.

The user provides the position points $P(k), k = 1...N$.
The Metronome calculates the speeds $V(k), k = 1...N$ for the points $P(k), k = 1...N$ as follows:

If k is an ordinary point inside the path: $V(k) = \dfrac{P(k+1) - P(k-1)}{2T}$

If k is the first programmed point in the path: $V(k) = \dfrac{P(k+1) - P(k)}{T}$

If k is the last programmed point in the path: $V(k) = \dfrac{P(k) - P(k-1)}{T}$

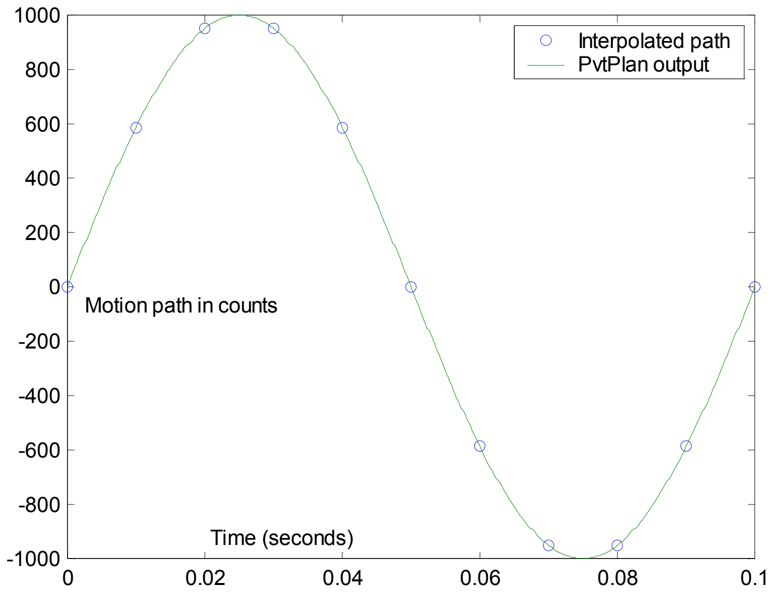
For each motion interval, we have four requirements to satisfy:

- Start position.
- End position.
- Start speed.
- End speed.

These four requirements exactly suffice to solve the interpolating 3rd order interpolating polynomial.

7.2.12.2 Example

We desire that the reference to the position controller will be
$P(t) = \sin(2\pi \cdot 10t)$.
The controller has a position sampling time of 200 microseconds.
The path is sampled once per 50 controller sampling time (10msec).
The points set as PT reference points are depicted in circles.
The path interpolated by the Metronome is shown as a solid line.



7.2.13 PT Motion Programming – The Basic Mode

7.2.13.1 The PT Table

The vector QP[N] defines the position points for PT motion.
Each element of the vector defines the position at a given time.
The QP vector has 1024 elements, and can thus specify up to 1023 consecutive PT motion segments, or 1024 PT motion segments in the cyclical mode.
The positions in the QP vectors are limited by the modulo-count of the controlled axis. At the maximum (XM=0 for UM=5 or YM=0 for UM=4) the QP[N] elements are limited to +/-2^{30} counts.

7.2.14 Motion Management

In the PT mode, the Metronome manages a "read pointer" for the QP[] vector.

When the read pointer is N, the present motion segment starts at the position of QP[N], and ends at QP(N+1)[8]. After MP[4] control sampling times, the Metronome increments the read pointer to N+1, and reads the QP[N+2] to calculate the parameters of the next motion segment.

The user does not have to use the entire PT table for a given motion.

The use of the QP vector is defined by the following parameters:

Parameter	Use	Comment
MP[1]	The lowest valid element of the QP vector	
MP[2]	The highest valid element of the QP vector	
MP[3]	0: Motion is to stop if the read pointer reaches MP[2]	Cyclical behavior definition.
	1: Motion is to continue when the read pointer reaches MP[2]. The next row of the table is MP[1].	
MP[4]	The number of controller sampling times in each PT motion segment.	

Table 23 – PT Motion Parameters

[8] The PT mode may be cyclical, according to MP[3] – please refer the explanations below. In that case N+1 must be interpreted in the modulo sense.

The flow chart of the basic PT mode is depicted below.

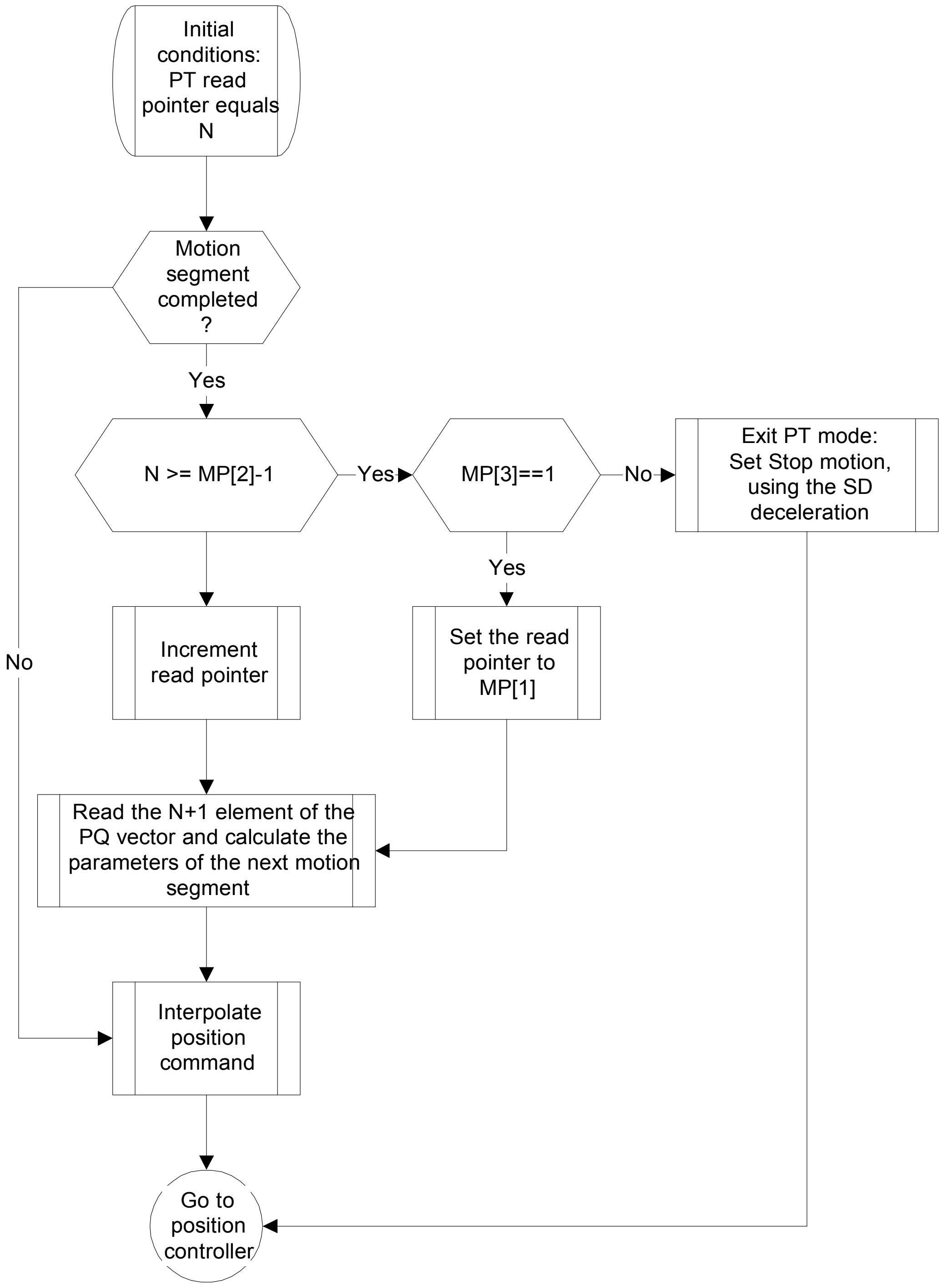

Initial conditions: PT read pointer equals N
Motion segment completed ?
Yes
No
N >= MP[2]-1
Yes
MP[3]==1
No
Exit PT mode: Set Stop motion, using the SD deceleration
Yes
Increment read pointer
Set the read pointer to MP[1]
Read the N+1 element of the PQ vector and calculate the parameters of the next motion segment
Interpolate position command
Go to position controller

Figure 9 – PT Decisions Flow Chart

A PT motion may be started by stating
PT=N with $1 \le N \le 1024$, and BG.
The command PT=N sets the read pointer of the QP vector to N.
BG starts the motion.
The QP vector may be written on-line while a PT motion is on.

7.2.15 Mode Termination

The PT motion terminates upon one of the following cases:
The motor is shut down, either by programming MO=0 or by an exception.
Another mode of motion is set active, e.g. by programming PA=xxx;BG. In this case the new motion command executes immediately, without having to explicitly terminate the PT mode.
The PT motion manager runs out of data. This happens when the read pointer reaches MP[2] and MP[3] is zero. This may also happen in the auto increment mode (CAN communications only, see below) if the read pointer reaches the write pointer. Then the PT motion is stopped immediately, using the SD deceleration. Note that if the last programmed PT speed is zero, then the PT motion terminates neatly, and the stopping at the end of the motion does nothing.

7.2.16 PT Motion Using CAN

The PT table allows the performance of pre-designed motion plans, as well as the on-line design of motion plans by writing the QP vector while PT motion is executing.
The on-line motion design ability is limited by the speed of communication interface.
Consider an RS-232 ASCII communication interface.
Programming of a single QP vector element has the format
QP[xx]=xxxxxxxxx;
Up to 18 characters may be required to program a single position point.
At the communication rate of 19200 baud, this may take 9msec.

The CAN communication option allows much faster PT programming, by packing two position points into one PDO communication packet. Moreover, for easy synchronization with the host, the Metronome may be programmed to send an emergency object whenever the number of yet unexecuted motion segments falls below a given threshold.

7.2.17 The PT Motion Programming Message

Two positions for the QP vector can be programmed in the eight bytes of a single PDO.
The PDO used is 0x300+ID where ID is the node ID of the Metronome. Note that before using PDO 0x300+ID for PT, its PDO mapping must be first set correctly.
The mapping of this PDO for the PT mode is listed in the table below.

Object Dictionary Index	0x2002
Type	RECORD, 2 elements
Access	Write only
Structure:	Signed32 Position1
	Signed32 Position2
PDO mapping	Yes
Value limits	No
Default value	Not Applicable

Note that the PDO does not specify the QP vector elements to be programmed.
The elements to be programmed are specified by a “write pointer”.
The value of the write pointer may be set by the parameter MP[6].
The value of the write pointer may be set once for the entire motion. The write pointer is incremented automatically by two each time it receives a new PT motion-programming message.
The CAN auto increment mode has the following flow diagram:

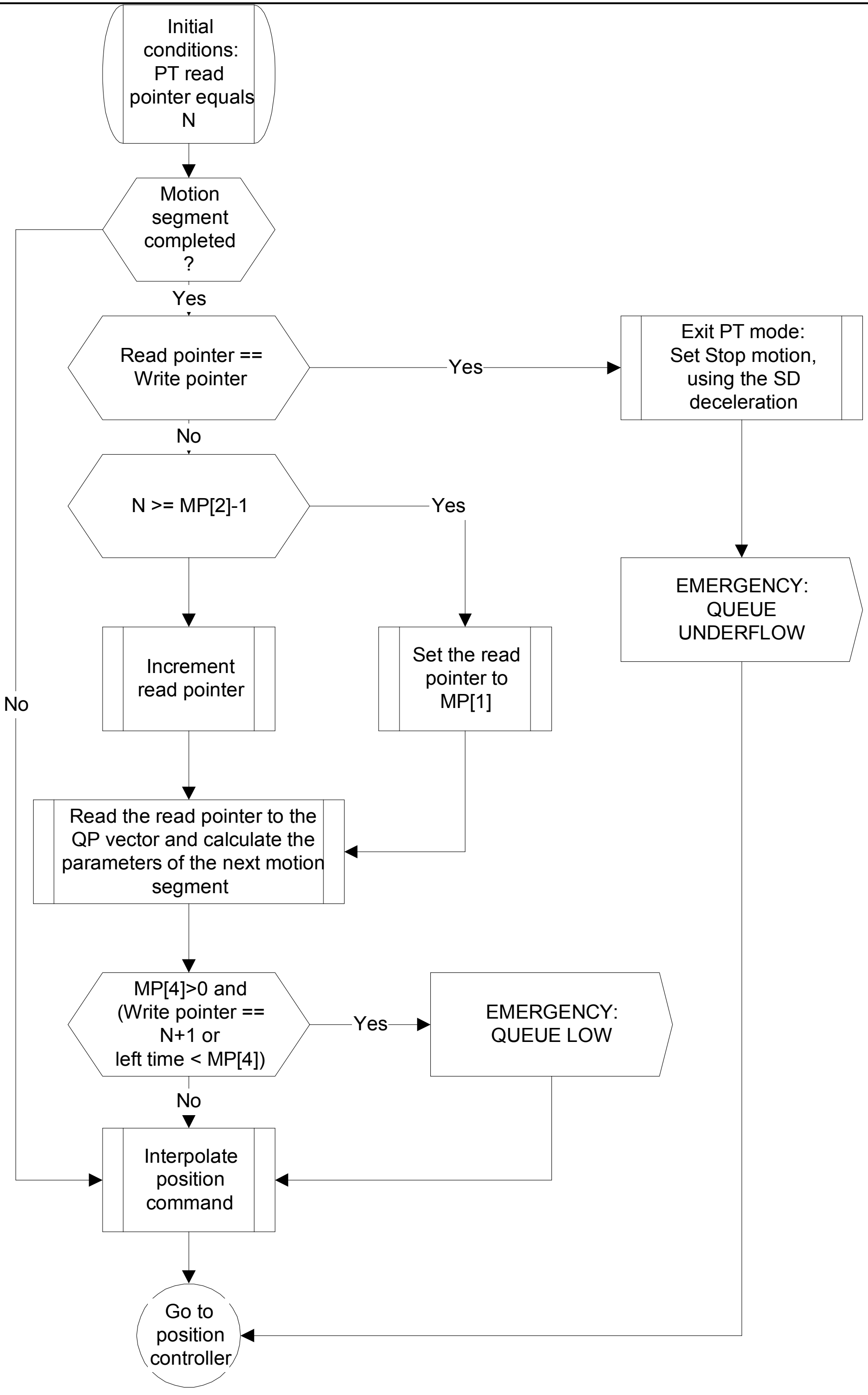

Initial conditions: PT read pointer equals N
Motion segment completed ?
Yes
No
Read pointer == Write pointer
Yes
Exit PT mode: Set Stop motion, using the SD deceleration
No
N >= MP[2]-1
Yes
EMERGENCY: QUEUE UNDERFLOW
Increment read pointer
Set the read pointer to MP[1]
Read the read pointer to the QP vector and calculate the parameters of the next motion segment
MP[4]>0 and (Write pointer == N+1 or left time < MP[4])
Yes
EMERGENCY: QUEUE LOW
No
Interpolate position command
Go to position controller

Figure 10 – PT Auto Increment Mode Flow Chart

The above flow diagram differs from the flow diagram of the basic mode in the following issues:
Motion queue underflow is diagnosed by the read pointer reaching the write pointer
Emergency objects are issued for the queue low and for the queue underflow events.

7.2.18 Programming Sequence for The Auto Increment PT Mode

PT motion must start with initial programming of the PT arrays.
First set
MP[1] = First valid line in the PT table
MP[2] = Last valid array in the PT table
MP[3] = 1 for cyclical mode.
MP[4] = the ration between the length of the PT time interval and the sampling time of the position controller
MP[5] = Number of yet unused QP[N] elements when a "PT queue low" emergency object is sent. Set to zero if no "PT queue low" warning is desired.
MP[6]=the write pointer. This is the next position in the QP[N] vector to be written by the CANopen object 0x2002.

Set the QP[N] for at least the first two points in the PT motion. This is since the PT algorithm requires at least two time points to interpolate a motion trajectory. Set at least 3 points if the speed at the second point is to be continuous.

Set (if not already set) UM=4 or UM=5, and MO=1

Set PT=N , where QP[N] and QP[N+1] and preferably QP[N+2] are all programmed to valid values.

Then use the CAN PT/Auto increment command for the rest of the PT motion.
Use the queue low emergency signal to synchronize the host and the Metronome.

The queue low emergency message includes the present location of the read pointer and the write pointer. It is safe to send more PT data PDOs until the write pointer is one location before the read-pointer specified by the queue low emergency message.

The host is well aware to the location of the write pointer, since it can count its own messages. A data message may be, however, rejected since the queue is full, or since a message is lost. In that case, the Metronome will issue an emergency object to the host. After receiving the emergency object the true location of the write pointer may be unclear. The host may then set MP[6] to the possibly rejected table row and continue the writing from there.

Please note:
About 1 msec may elapse between the reason for the emergency object and the time when the emergency object is actually transmitted.
Accurate timing with respect to the host is the essence of multiple-axis synchronized motion. Accurate timing may be achieved by using the CAN SYNC signal, and the CAN synchronized BG service, as described in the CAN communication documentation.

7.2.19 The Parameters of The PT Motion Mode

The following parameters apply to PT motion

What	How	Comment
Unit Mode (UM)	Unit modes 4 or 5 select the position mode motion path.	
Stop Deceleration (SD)	The rate of deceleration in the case where motion is killed by queue underflow or by an exception	
Position/ Time (PT)	Set a PT motion command	Special features are available for PT using CAN communication
PT table entries: QP[N]	Set values to the PT table	
Motion Parameters (MP)	MP[1] = First valid row in PVT table	Configure a PT or PVT motion. MP[6] and MP[5] are for the CAN auto increment mode only.
	MP[2] = Last valid row in PVT table	
	MP[3] = Cyclical motion (0 non-cyclical, 1 cyclical)	
	MP[4] = Ratio between the command sampling time and the position controller sampling time	
	MP[5] = Time for queue low alarm	
	MP[6] = Initial value for write pointer	
WS[29]	Sampling time, in microseconds, of the position controller.	A read only parameter. WS[29] is an integer multiple of the basic sampling time as set by TS.

Table 24 – PT Related Parameters

The following CAN emergencies are supported, all as manufacturer specific:

Error code (Hex)	Error code (Dec)	Reason	Data field 1
0x56	86	The time for the entire left valid PT program has dropped below the value stated in MP[4]	Time msec left with valid motion program
0x5b	91	Write pointer out of the physical [1,1024] range of the QP vector. The reason may be a bad setting of MP[6].	The value of MP[6]
0x5c	92	The PDO 0x3xx is not mapped	
0x34	52	Ann attempt has been made to program more PT points then supported by the queue.	The index of the PT table entry that could not be programmed.
0x7	7	Cannot initialize motion due to bad setup data. Reasons: - The write pointer is outside the range specified by the start pointer and the end pointer.	
0x8	8	Mode terminated, and the motor has been automatically stopped (In MO=1). Reasons are: End of trajectory in non cyclic mode (MP[3]=0) [Additional code 1] A zero or negative time specified for a motion interval [Additional code 2] Read pointer reached write pointer [Additional code 3]	Read pointer

Table 25 – PT CAN Emergency Messages

7.3 External Reference for The Position Controller

7.3.1 The External Reference Sources

The position reference to the controller is composed of a software command and of an external command as depicted in Figure 4, which is repeated here for convenience.
The Software command is generated by the user program, or by communicated commands.
The external command is generated by the reading of the auxiliary encoder input, and by the two analog inputs.
The rational is that the user may want the motion to perform with respect to a moving reference. For example, a manipulator may work on a product while the product is traveling on a conveyer. The position of the manipulator is the sum of

- The manipulator position with respect to the conveyer
- The position of the conveyer.

The position of the manipulator with respect to the conveyer is known accurately, and is programmed in software. The position of the conveyer is not known exactly in advanced, and it must be measured on line, for example by using the auxiliary encoder input. The reading of the auxiliary input is multiplied by the follower ratio (FR) parameter, and added to the software command.

With the ECAM option enabled, the external reference is not the direct sum of the analog and the auxiliary inputs but a tabulated function of them. The extra flexibility given by the ECAM table can be used for:

- Limit the range of the external input.
- Linearize nonlinear sensors that are directly coupled to the analog inputs
- Design complicated motions that are a function of the external inputs
- Synchronize the motion of two or more Metronomes, all connected to the same auxiliary signals.

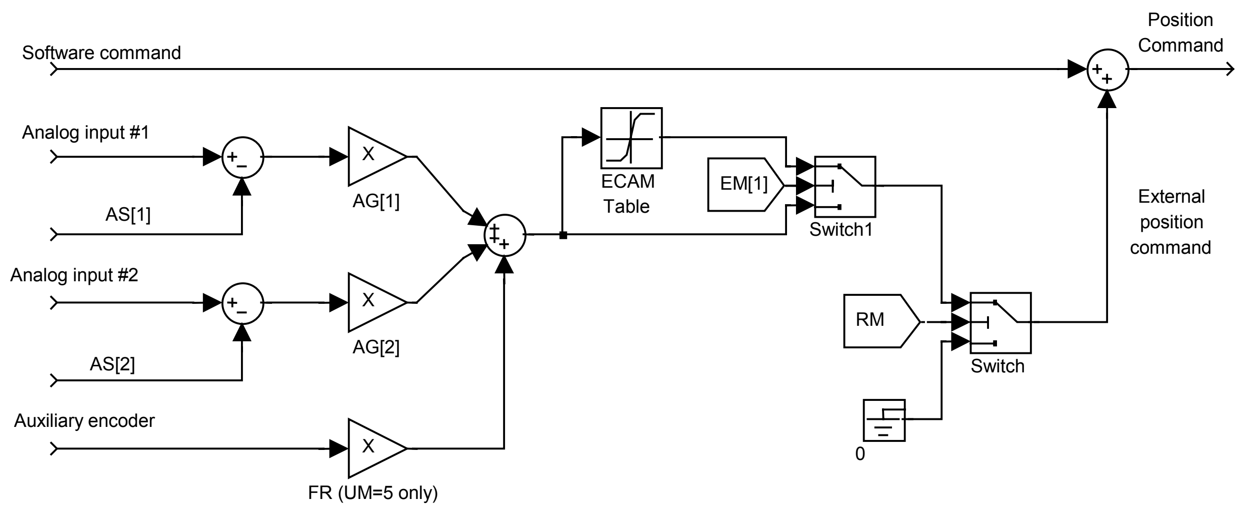


Figure 11 – Position external Reference Generation Model

As we see in the figure, two switches affect the generation of the external position reference:

Switch	Description
RM	Reference Mode. If this switch is set, the external reference is calculated and added to the software command. If this switch is zero, only the software command is used.
EM[1]	ECAM Mode. If this switch is set to 1, the ECAM table will be applied to the external reference. If this switch is zero, then the external reference will be applied directly.

Table 26 – External Position Reference Commands

The formula to obtain the position reference is

- RM=0: EM[1] not applicable:
  Position reference = Software common

- RM=1,EM[1]=0:
  Position reference =
  ECAM [(AG[1] * Analog in 1) + (AG[2] * Analog in 2) + (FR * Auxiliary Encoder)]
  + Software command

- RM=1,EM[1]=1
  Position reference:
  ECAM [(AG[1] * Analog in 1) + (AG[2] * Analog in 2) + (FR * Auxiliary Encoder)]
  + Software command
  In the dual loop position mode (UM=4), the reference to the position loop is generated in a very similar way, with the only difference that the auxiliary encoder is not available for reference input, since it is used for position feedback.

7.3.2 ECAM

ECAM is an acronym for “Electronic Cam”. It means that the position reference to the Metronome is not directly proportional to the summed external inputs, but is a function of them. The ECAM related commands are as follows:

EM[1]	Asserts whether the ECAM function is active. 1 for active ECAM. 0 for direct external referencing. Set EM[1] to 1 when ever a change is needed in EM parameters
EM[2]	The last valid index of the ECAM table. The maximum for EM[2] is 1024.
EM[3]	Starting position – the value of the input to the ECAM function for which the output of the ECAM function will be ET[EM[5]] (ET of EM[5])
EM[4]	Table difference. When the input to the ECAM table will be EM[4], the ECAM function will output ET[2].When the input to the ECAM table will be 2*EM[4], the ECAM function will output ET[3], etc.
EM[5]	The first valid index of the ECAM table.

Table 27 – ECAM Related Command

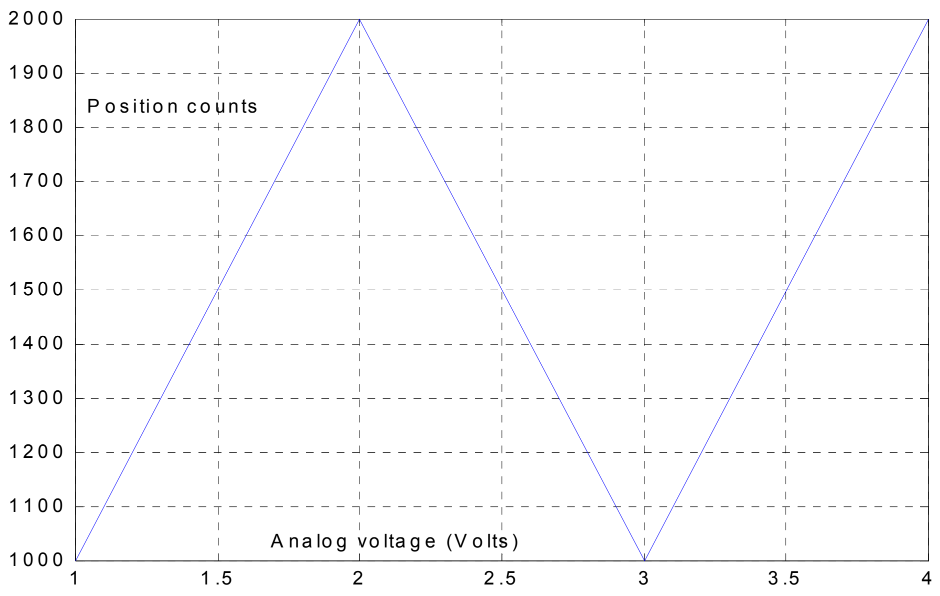
The Input to the ECAM Table (IET) is composed from:
IET = ((AG[1] * Analog in 1) + (AG[2] * Analog in 2) + (FR * Auxiliary Encoder)).

As a result to IET the response may be:

- If IET < EM[3], the external position command (output of the ECAM table) will be ET[1].
- If IET > (EM[3]+(EM[2]-1)*EM[4]), the external position command will be ET[EM[2]].
- If EM[3]< IET < (EM[3]+(EM[2]-1)*EM[4]), the external position command will be derived by linear interpolation of the ECAM table.

Example:

Suppose that we want the following functional relationship between the voltage of analog input 1 and the motor position:



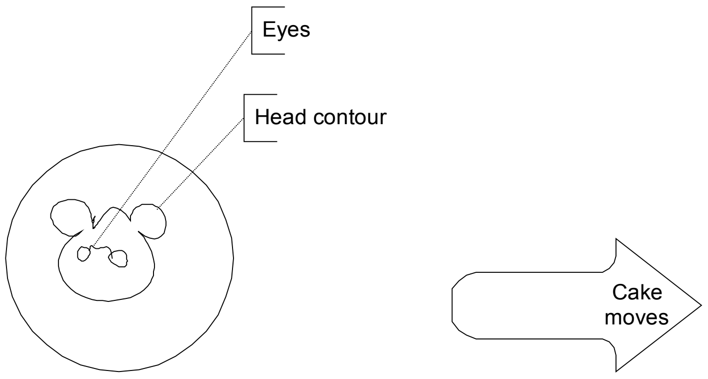
Then we may set:

AG[1]=1000	**Arbitrary – set 1000 counts at the input of the ECAM function for 1v of input
EM[2]=4	**4 ECAM points are programmed
EM[3]=1000	**The first point is at 1v, which by AG[1] is equivalent to 1000 counts
EM[4]=1000	**Each two position break points are separated by 1v that by AG[1] is equivalent to 1000 counts
EM[5]=1	**First index of the table
EM[1]=1	** Activate ECAM.
ET[1]=1000	**ECAM table values
ET[2]=2000	
ET[3]=1000	
ET[4]=2000	
…	

Suppose that the external reference is made only by analog input 1 (AG[2] and FR are zero). Then:
- For every input voltage less then 1v, the external position command will be 1000.
- For input voltage of 1.5v the external position command will be 1500.
- For input voltage of 2.75v the external position command will be 1250.
- For every input voltage greater then 4v, the external position command will be 2000.

Example:

Consider an application, in which a two-axis x-y servo system is used to plot chocolate teddy bears on birthday cakes.



The cakes come from the oven on a conveyer. An incremental encoder measures the place of the conveyer. Each time a new cake arrives, it operates an optical switch, which sets the RLS input of the servo drive. In response to the RLS, the x-y axes begin to contour the head of the teddy bear, drawing it with chocolate. Then the chocolate flow is stopped for a while, while the x-y axis travel toward starting the eyes of the teddy bear. After drawing eyes to the bear, the x-y stage returns to initial position, to be ready for another cake.
Both the Metronomes that manage the x and the y-axes work in the ECAM mode. They get their position reference as a function of the location of the conveyor, via the auxiliary encoder input. The ECAM motion starts when a cake arrives at the plotting station. It continues until the conveyor had traveled 4000 counts.
One of the Metronomes controls, by a digital output, the flow of the chocolate out of the drawing nozzle.

The initial programming of the Metronome includes:

```
EM[1]=1                    **Enable ECAM
EM[2]=200                  **Length of the ECAM vector
EM[3]=0                    **Starting position
EM[4]=100                  **Conveyer encoder counts between two consecutive
                           ECAM table entries
ET[1]=…;ET[2]=…;E          **Program the numeric data of the ECAM table
T[100]= …;
UM=5                       **Set standard position mode
RM=1                       **Enable external referencing
AG[1]=0;AG[2]=0;F          **Kill both the external analog inputs, and enable the
R=1                        auxiliary encoder input with the follower ratio of 1.
MO=1                       **Start motor
PA=1000;BG                 **Go to waiting position
HY[2]=0;HY[3]=5;H          **Null the auxiliary encoder count upon cake arrival
Y[1]=1                     (RLS high)
```

Each Metronome has the following AUTO_RLS routine:

```
##AUTO_RLS
**The RLS has operated an already programmed auxiliary homing process to synchronize
**the follower input
OB[1]=1;                   **Activate chocolate nozzle
##WAITA                    **Wait until end of the head contour
JP##WAITA,PY<2000
OB[1]=0;                   **Stop the chocolate
##WAITB                    **Go to start of eyes
JP##WAITB,PY<3000
OB[1]=1;                   **Restart the chocolate
##WAITC                    **Draw the eyes
JP##WAITC,PY<4000
OB[1]=0                    **Stop the chocolate
PR=-2344;BG                **Return to starting point
HY[1]=1                    **Program the auxiliary encoder to reset again at the
                           next cake
RT                         **End of auto subroutine
```

7.3.3 Starting a Motion – Absolute and Relative Frames

7.3.3.1 Jump-Free Motor Starting Policy

Upon starting a motor by the MO=1 command, the motor should never jump.
The first and most important reason is safety. The other reason is to avoid an excessive position error fault immediately after the motor is started.
In order that the motor will not jump, the initial software reference is automatically set to the present position of the motor, and the software command remains stationary until a motion instruction is accepted. After setting MO=1, no motion mode is defined, so that commanding BG without the prior specification of the desired motion type will not launch any motion.

Example:

Suppose that MO=0, RM=0, and PX=1000 (Motor is off, no external reference, present position is 1000 counts). Entering MO=1 will automatically set the software position reference to 1000 counts. The command sequence PA=0;BG will launch a PTP motion from the position of 1000 counts to the zero position.

Example:

Suppose that MO=0, RM=1, FR=1, AG[1]=AG[2]=0, PY=3000, and PX=1000 (Motor is off, external reference is generated by the auxiliary encoder input only with a unit follower ratio, the auxiliary position is 3000 and the present position is 1000 counts). Entering MO=1 will automatically set the software position reference so that the total position reference will equal PX. Therefore the initial, automatic, software command will be PX - FR*PY= 1000 - 1*3000 = -2000.
If the auxiliary input is stable, the command sequence PA=0;BG will launch a PTP motion from the position of -2000 counts to the zero position.

7.3.3.2 Relative and Absolute Motions

We say that the Metronome tracks an absolute reference signal when the external position reference is superimposed on a known software command.
The Metronome performs only relative tracking when the Metronome, not by the user, sets the software reference automatically. The "shape" of the external reference signal is tracked, but the user does not know the absolute position of the entire motion.
From the above paragraph, it is clear that in RM=1, after MO=1 the Metronome tracks the auxiliary command relatively only – the auxiliary reference is not superimposed on a known software position, but on the motor position as it was sampled on MO=1.
The tracking can be made absolute by specifying a PTP[9] software motion.

A motor may be stopped regardless of RM by hitting a switch (The abort switch, for example) that is programmed to stop the motion by the IL command.
When the motor is stopped in RM=1, it cannot safely resume an absolute tracking. This since at the time the motion is resumed, the external reference value may be different from its value when the motor was stopped. Therefore, after a stop event, the Metronome can only return to track relatively. The user must be well aware that the Metronome is no longer tracking absolutely before it frees the Metronome to move. Therefore, the user must explicitly use the RI (Relative input) to set the Metronome to relative tracking. Only then a BG command can set the Metronome to absolute tracking.

7.3.3.3 Related Commands

RI	RI – Relative tracking. In RM=1, this command tells the Metronome to start relative tracking. After an RI command: The software command is automatically set to (PX-external command) The software command is made stationary. The next motion mode is undefined, so that a motion mode command as PA or JV must be entered before a BG can initiate a motion. If there is a pending BG (by the BI command) it is cleared.
MO	Start the motor. MO=1 performs RI automatically.
BG,BI	Begin – launch a predefined motion. BG and BI cannot start a motion just after MO=1 or RI, since then there is no motion definition that may be launched.
RM	Define if an external reference is used at all RM=0: Don't use external reference RM=1: Use external reference

Table 28 – Absolute and Relative Frames - Related Command

[9] Other software commands will not do the job: the Jog command does not lead to any absolute position, and the tabulated motion modes cannot be safely started before the software reference is well known.

8 Event Handling

This chapter deals with the handling of events.
Events are hardware conditions that affect the Metronome. Sampling and manipulating the digital inputs in the user program could theoretically handle most of the events. The user program is however too slow for that, and some events, as described below, are handled in the real-time process of the Metronome.

8.1 Homing and Capture

8.1.1 What Is Homing?

The Metronome uses incremental position sensors[10]. Therefore, the Metronome can only tell the distance the motor traveled since power on, but not where the position counting started.
The process of teaching the Metronome its absolute position is called "Homing".

Homing of the main position counter (PX) typically serves for:

- Absolute position control in the single-feedback position-control mode.
- Relative work in the single-feedback position-control mode. For example, an axis in a machine may be homed by a sensor of the edge of the product the machine works on. The position referencing of the axis becomes relative to the product.

Homing of the main position counter (PY) typically serves for:

- Absolute position control in the dual-feedback position-control mode.
- Relative work in the dual-feedback position-control mode.
- In the single-feedback position-control mode: Fixing the origin for the external position command- see Section 7.3.3.

In the homing process, the motor travels until an expected event occurs. The event marks that the motor is now in a known absolute position. This known position, if set to the position counter at the time of the event, makes the position reading of the Metronome absolute.
The looked-for event may be one of the limit switches (RLS or FLS), the home switch, the encoder index or the digital inputs.

[10] The position reading of the resolver models is also taken incrementally. Please refer the WI command in the reference manual to learn how the resolver reading can be used to set the absolute position reading.

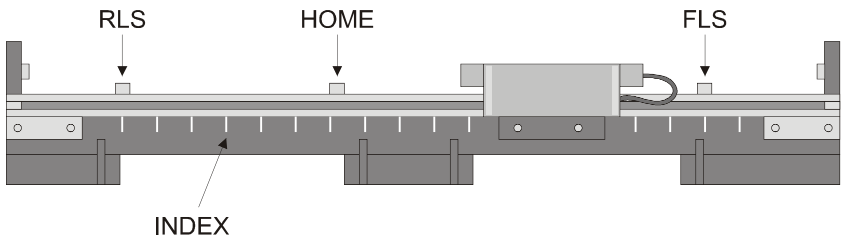


Figure 12 – Switches location

The Metronome can detect RLS and FLS and digital input transition only with the resolution of its sampling time. The accuracy of these homing types is therefore sampling time and speed dependent.

Example:

A Clarinet with TS=200 seeks the RLS at the speed of 10000 counts/sec. Assuming that the RLS switch has no sensing delay, and that it is absolutely repetitive, we still have a homing error of up to 200µsec * 10000 = 1 due to the uncertainty in the event time. The 1 count possible error may be negligible. Higher speeds may introduce significant homing errors.

The HOME and the INDEX are timed accurately with a microsecond precision, so that no timing-uncertainty exists for them.
Please refer to *Homing Examples* for examples.

8.1.1.1 RLS,FLS as Input Logic (IL) Vs Homing Modes(HM)

In position mode UM=5 the limit switches are treated as STOP. A conflict may occur when setting the limit switches to STOP (i.e. IL[3]=3) or using it in homing mode (i.e. HM[3]=7). It is recommended to ignore the functionality of the limit switches during the homing process and to restore it when done. The following example is using the RLS for homing event while restoring its functionality of the switch after procedure is done.

Example:

Following example is using the AUTOEXEC procedure to start homing mode on power on and on digital input 1.

#@AUTOEXEC	Start from here each amplifier reset.
JP##START_HOME	Jump to home sequence
#@AUTO_I1	Each time digital input 1 is set….
JZ##START_HOME	… jump to home sequence and clear stack pointer
RT	
##START_HOME	
IA[1]=IL[4]	Save value of RLS input logic
IL[4]=5	Ignore RLS as a limit switch.
HM[2]=0	PX on home is 0
HM[3]=5	Event on high RLS
HM[4]=0	Stop after event.
HM[5]=0	Set HM[2] to PX when done
HM[1]=1	Activate homing sequece
MO=1	
JV=-1000;BG	Start motion towards RLS switch
AF,HM[1]=0	When switch detected, and homing sequence is done…
IL[4]=IA[1]	…restore RLS switch functionality.

##LOOP	
JP##LOOP	Stay in loop for ever

Program Example 1 - Limit switch input logic change

8.1.2 Capturing

Capturing means the registration of the position in which a certain event occurred.
The event may be chosen from the long list of events supported by the HM or the HY event handling functions.
The capturing may be a part of a homing process, especially when multiple sensors are used for homing.
Capturing may also serve to measure event positions for a host. Consider for example a machine with multiple servo axes. One axis can sense the product that the machine works on. The axis transmits to the host the position captured when the product has been sensed. This data enables the host to generate appropriate position commands for all the other axes.
The Metronome can capture 1 event, and register up to 4 consecutive occurrences of PX and PY of this event.

8.1.3 The homing and capture functions

The homing and capturing functions are made using the HM and the HY commands.
Both function capture both the main and the auxiliary positions.
The main position counter is homed by the HM function and the auxiliary position counter is homed by the HY function.
The HM and the HY functions are very similar, except that they home different position counters.

The following table summarizes the HM command (HY is not given here since it is almost similar – please refer the Command Reference Manual).

HM[N]	HM[N] values
HM[1]	Set to 1 in order to initiate a main homing process. Set to [1..4] to initiate a multiple capture process. HM[1] = n will launch n consecutive captures of the PX and the PY position. HM[1] is decremented after each capture.
	Set to 0 in order to disarm the homing process. HM[1] is automatically reset to zero when homing is done
HM[2]	Homing value, please refer HM[5]
HM[3] Homing event	0=Immediate event home setting.
	1= High level/edge Home switch.
	2= Low level/edge Home switch.
	3= High edge Index event.
	4 = Low edge Index event.
	5 = High level RLS switch.
	6 = Low level RLS switch.
	7 = High level FLS switch.
	8= Low level FLS switch.
	9=In 1 high level
	10=In 1 low level
	11=In 2 high level
	12=In 2 low level
	13=In 3 high level
	14=In 3 low level
	15=In 4 high level
	16=In 4 low level
HM[4] What to do after the event	0= Stop immediately according to SD value.
	1=Set HM[6] to digital output
	2=Do nothing
HM[5] What to set for PX	0=Set HM[2] into PX
	1=Set PX-HM[2] into PX
	2=Do nothing
HM[6]	Output value
HM[7]	The 1st captured value of PX
HM[8]	The 1st captured value of PY
HM[9]	The 2nd captured value of PX
HM[10]	The 2nd captured value of PY

HM[N]	HM[N] values
HM[11]	The 3rd captured value of PX
HM[12]	The 3rd captured value of PY
HM[13]	The 4th captured value of PX
HM[14]	The 4th captured value of PY

Table 29 – The HM Command

Table 29 lists many options for the HM command:
Many possible events may be chosen to trigger the homing or the capturing
HM[4] defines what must be done after the homing/capturing is complete: The axis may proceed as if nothing has happened, or it may be stopped.
HM[5] proposes several methods for updating the position counter.

The HM and HY commands are very flexible, and support many methods for homing. In the sequel, we give few examples for the use of these functions.

Notes:
The HM and the HY events are useful for capturing and setting the position counters in motion.
When the motor is off (MO=0) the values of the main and the auxiliary position counters can be set by simply assigning values to PX and PY.
The capture function captures both the main and the auxiliary position counters simultaneously. This is useful for synchronizing the main and the auxiliary positions.
PY value in the PLUS versions will be 0.

8.1.4 Homing Examples

This section presents basic homing sequences.
In the examples and the attached diagrams, the left side as the minimum position and the right side as a maximum position. RLS is located on the left and FLS on the right. HOME switch is somewhere between.

Example: Using a switch to resolve index ambiguity

This example implements the homing method 5 in CiA DS-402.
The ideas behind Method #5 in CiA DS-402 are as follows:
The index can be much more accurately detected then a switch
For gear systems, the index gives ambiguous position, since the index may be encountered many times when the motor rotates.
Seeking a switch and then looking for the nearby index may resolve the index ambiguity.

With CiA DS-402 Method #5, the motor travels until it identifies the home switch. After the home switch is detected, the motor moves forward until the next index. When the index is found, the absolute location is loaded to the main position counter. After the homing motion is complete, motion stops.

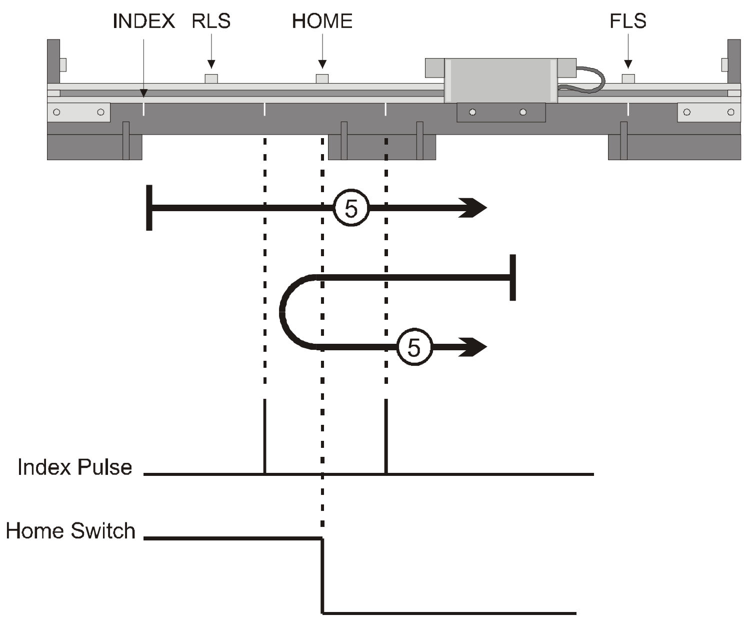


Figure 13 – Homing on Negative (falling edge) Home Switch and Index Pulse

Command	**Description**
MO=0	Motor Off
UM=5	Position loop.
SD=10000000	Deceleration after event occurs.
HM[2]=10000	The required main counter value at the index marker.
HM[3]=1	The event is a high home switch
HM[4]=0	Stop after the event.
HM[5]=2	Do not modify PX after capture
MO=1	Motor on
HM[1]=1	Start home capture process
JV=1000;BG	Jog in 1000 cnt/sec
JP##NO_FLS,IB[10]=0	If FLS is not sensed continue normal
JV=-1000;BG	FLS encountered – reverse direction
##NO_FLS	
JP##SEEK_HOME,HM[1]=1	If the home switch is not yet found, continue.
HM[5]=0	Next event shall put HM[2] into PX
HM[3]=3	The next homing process will look for the index.
JV=1000;BG	Go forward until switch
AF,HM[1]=0	Until the homing process is done.
EN	

Program Example 2 - Method 5 from DS-402

Example: Double homing corrects backlash offsets

This example demonstrates homing on the home switch without using the index.
In many gear systems, the index signal cannot be used for homing. This may be due to:
Motor or gear repairs should not require the tuning of the index position or the Metronome.
Backlash and gear compliance prevent accurate mapping of the motor position to the load.
In order to prevent compliance and timing errors, the position of the home switch is captured two times, with alternating movement directions.
The two captured results are averaged to cancel the error sources.
Suppose that in the middle of the home switch we should have PX=10000.
Then the homing formula is PX=PX+10000-0.5·(PX at right home edge + PX at left home edge)
The user program that does that is as follows:

Command	Description
MO=0	Motor Off
UM=5	Position loop.
SD=10000000	Deceleration after event occurs.
HM[3]=1	The event is a high home switch
HM[4]=2	Do nothing after the event
HM[5]=2	Do not modify PX after capture
MO=1	Motor on
HM[1]=1	Start home capture process
JV=1000;BG	Jog in 1000 cnt/sec
JP##NO_FLS,IB[10]=0	If FLS is not sensed continue normal
JV=-1000;BG	FLS encountered – reverse direction
##NO_FLS	
JP##SEEK_HOME,HM[1]=1	If the home switch is not yet found, continue.
IA[1]=HM[7]	Get captured value into IA[1]
AF,IB[8]=0	Wait until the other side of the home switch
HM[4]=0	Next homing event shall stop motion
HM[1]=1	Start home capture process
JV=-JV;BG	Go the other direction
AF,HM[1]=0	Wait until end of homing
IA[1]=IA[1]+HM[7]	Average positions
IA[1]=IA[1]/2	IA[1]=averaged home positions
HM[2]=10000-IA[1]	Difference for main encoder counter
HM[4]=0	Do nothing after the next homing event
HM[5]=1	1=Set PX-HM[2] into PX
HM[3]=0	Trigger homing immediately
HM[1]=1	Done.

Program Example 3 - Double Sided Homing.

Example: Immediate homing enables the host to home an axis

The host may desire to set the position counter of an axis, possibly in response to an event in another axis. For this purpose, the host must transmit:

HM[2]=-100000	After homing event the location will be –100000
HM[3]=0	Trigger immediately
HM[4]=0	Stop after Event (better assure that axis is already stopped)
HM[5]=0	HM[2] refer to absolute value for PX.
HM[1]=1	Turn on homing mode.

8.2 BG (Begin) and ST (Stop) By Hardware

Motion may be begun and stopped according to hardware events.
The BI command enables the beginning of motion according to the status of the digital inputs. Using the BM command, the host can specify which digital inputs should be watched.

Example – Automatic starting

Consider an assembly machine, in which a servo axis pushes a part from a storage stack to a conveyer.
The pushing is to be synchronized to the arrival of a new assembly. In addition, parts are not to be pushed to an assembly declared defective by a previous inspection station.
The “new assembly” sensor and the “Inspection good” inputs are connected to digital inputs 2 and 3, respectively. The combination of high Digital Input #2 and low Digital Input #3 shall automatically begin a new motion.
The following program code handles this:

BM=6	Consider only the digital inputs that are on in the binary representation of 6. The binary representation of 6 is 0110, meaning that switches 1 and 4 are ignored (zero) and switches 2 and 3 are watched.
PA=1000	Set the target position after the product is pushed
##LOOP	
BI=2	Begin motion when switch 2 is on (1 · binary 0010) and switch 3 is off (0 · binary 0100). The 2 in the BI command is the sum of 2 and 0.
AF,MS=0	Wait until motion terminates
PX=0;BG	Send the axis back, to be ready for another push
AF,MS=0	Wait for stabilization in the back position
JP##LOOP	Repeat.

Program Example 4 - Begin by switch

Event dependent stopping may be programmed similarly using the SI and SM command..

8.3 Dedicated Switch Functions

The user may assign automatic functions to the Abort, Enable, RLS, and FLS switches.
In response to these switches, the Metronome may be told to shut down the servo drive, stop the motion under servo control, or allow only unidirectional motion (FLS and RLS only).
When a switch stops the servo drive under servo control, the RI command must be used to confirm the stopping before any further motion is permitted.
Please refer the IL and the RI commands in the Command Reference Manual.

In addition, the RLS and the FLS may have assigned automatic routines in the user program – please refer the #@AUTO_RLS and the #@AUTO_FLS definitions in chapter on user programs in this manual.

9 Limits, Protections and Faults

9.1 Current limiting

The Metronome protects the motor from over current.
The current protection is applied in two stages:
The motor maximum peak current (Given by PL[1]) is available for the peak duration time (specified by PL[2]). For long time periods, the current is limited to its continuous limit CL[1].

The current limiting process is dynamic. If the current has been close to its continuous limit, then the time allowed for the peak current reduces.

The next paragraphs detail the mathematics of the current limiting process.
The absolute value of the measured motor current (in vector servo drives this is $\sqrt{I_Q^2 + I_D^2}$) is applied to a first order lowpass filter. The state of the filter is compared to two thresholds. When the state of the filter is above the upper threshold, the continuous limit is activated. When the state of the filter is below the lower threshold, the peak limit is activated.

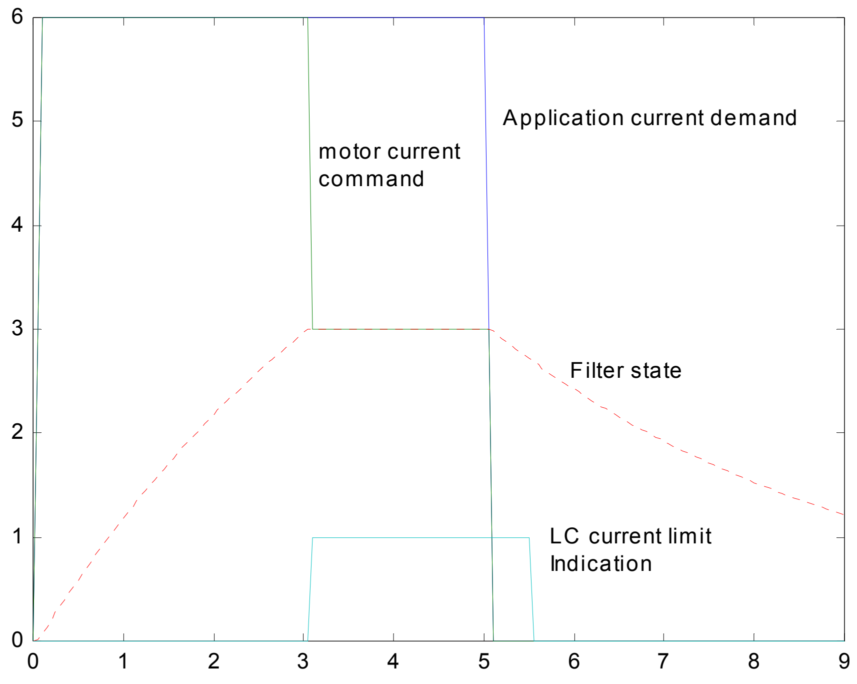
The time constant of the lowpass filter is $\tau = \dfrac{-PL[2]}{\log\left[1 - \dfrac{CL[1]}{MC}\right]}$, where MC is the maximum servo drive current.
The maximum time, for which the peak current can be maintained, after the current demand has been zero for a long time, is $-\log\left[1 - \dfrac{CL[1]}{PL[1]}\right] \cdot \tau$ seconds. If PL[1]=MC, then the maximum time allowed for peak current is PL[2]. Otherwise, the servo drive can provide the peak current for a longer time.

More generally, if a current demand of PL[1] has been placed after the current command is stable at I1 < CL[1] for a long time, then the peak current PL[1] will be available for
$-\log\left[1 - \dfrac{CL[1] - I1}{PL[1]}\right] \cdot \tau$ seconds.

The resolution of PL[1] is about MC/1000, and the resolution of PL[2] is about 0.1 second

Example:

The graph below shows the signals concerned with the current command limiting process for MC=6, PL[1]=6, PL[2]=3, and CL[1]=3.
The application motor current command is increases from zero to 6Amp at the time of 0, and is then decreased to 0 at the time of 5sec.
The state of the filter increases until it reaches the continuous current limit of 3 at the time of 3 sec. At that time, the LC flag is raised, and the motor current command is decreased to CL[1]=3Amp.
After the motor current command is set to zero, the state of the filter begins to drop. When the state of the filter drops to 2.7Amp = 90% of 3Amp, the LC flag is reset and the torque command limiting is again to 6Amp.



9.2 When the motor fails to start

The main reasons for failure to start the motor are:
- The physical conditions are not right
- The database is not consistent

Physical conditions:
The following physical conditions are tested before applying voltage to the motor
- Power supply under voltage
- Power supply over voltage
- Amplifier temperature too high
- Motor not connected to the servo drive
- An active limit switch is programmed to shut the servo drive down.

The motor shall not start if the voltage of the power supply is not within range, or if the servo drive temperature is too high, or an active switch prevents motor on. The MO command will return the error codes 65 or 66. The failure reason may be read through the status (SR) report.
If the voltage is in range, and the temperature is good, the Metronome will try to start the motor.

The first step in starting a motor is to measure and correct the hardware offsets. For this purpose, the Metronome commands very small current (milli-Amperes) to the motor and measures the current. If the offset measurement process fails, and the MO=1 command shall return the error code 75.
The offset correction process may fail if:

- The motor is not well connected to the servo drive.
- The motor is not stationary, or slow enough, when MO=1 is attempted.

Database consistency:
Before enabling the motor, the Metronome tests that all of its parameters make sense.
For example, the variables CA[4], CA[5], and CA[6] define how the Hall sensors are ordered. If CA[4]=CA[5] then two Hall sensors are assigned to the same position, which is an absurd.
With CA[4]=CA[5], the command MO=1 will fail and return the error code of 54, meaning “Bad database”.
In order to find the reason for the “bad database” failure, use the CD command. The CD command will return

```
Null Address=0
Failure address=0
Called Handler=none
Database Status:
CA[4], error code=37
```

The first lines establish that the CPU detected no exception.
The last line states that the parameter CA[4] yielded the error code 37, meaning "Two Hall sensors are defined to the same place" – please refer the EC command in the Command Reference Manual.

9.3 Motion faults

Error conditions may cause the Metronome to shut the motor automatically.
When the motor is shut down automatically,
The MO variable is set to zero
The variable MF is set to reflect the shut down reason.
A flag in the status register (SR) indicates that the motion has been aborted.

The variable MF may reveal why the motor has been shut even if the reason no longer exists. For example, if the power supply has too large impedance, then in full load its voltage may drop and the servo drive will be automatically shut down due to under voltage. When the motor is shut, the under voltage disappears. Another example is that an over voltage is generated due to an insufficient shunt. Again, the over voltage will disappear when the motor is shut down.
The over or the under voltage conditions that caused the fault are recorded in the MF variable.

Please refer the Command Reference Manual for a detailed description of the MF reports.

9.4 Device failures, and the CPU dump

Humans have programmed the Metronome. The programmers, however hard they try, cannot completely avoid bugs in their code. Programming bugs may lead to CPU exceptions, like an attempt to divide in zero, or an attempt to access a variable that does not exist.
Programming errors may also lead to CPU overloading. In that case the CPU will not be able to complete its control tasks on time, and the internal CPU stack shall overflow.

For this reason, the Metronome has traps for CPU exceptions and for stack overflow.
When the Metronome traps an exception (hope it never does), it does the following:
Shut off all the controller activity. Any motion is aborted to freewheeling.
It writes down the exact trap that was activated, and the address in the firmware that caused the exception.
After the exception, communication is still enabled, so the Metronome can be asked what had happened. Them motor, however, cannot be set again to work until the Metronome is rebooted.
A CPU exception is detected using the CD and the SR command.
The SR command only indicates that an exception occurred.
The CD (CPU dump) command gives the details.
The CD command reports as follows:

Null Address	The code address in the firmware when the exception occurred (0 if o.k.)
Failure address	The code address in the firmware when the stack overflow has been detected (0 if o.k.)
Called Handler	A string describing the type of exception. This may be: A divide by zero attempt An attempt to access a non existing variable Many other possible exceptions. This string is “none” if no exception was trapped.

Table 30 - CD Reported Elements
The CPU dump also returns the database status, as described above.

9.5 Sensor faults

9.5.1 The motor cannot move

When the motor is commanded to move to somewhere, and it for some reason it cannot do it, the reasons may be:
- The motion sensor is faulty – the motor moves but motion is not detected. In this case AC motors will be generally brought to stop, since the stator field will remain stationary.
- The motor is faulty, or there is other mechanical failure that prevents the motor from moving.
- The controller filter is poorly tuned. In this case the motor torque may wildly oscillate in high frequency, but the motor will hardly move at all.

These situations are identified by:
- The motor average torque is high
- The motor is stationary, or the movement is very slow.

The facts that the motor is commanded to high torque but is not moving are not always an error indication. In some applications, as thread fastening, it is perfectly legitimate for the motor to reach a mechanical motion limit. The Metronome user is asked to define whether a high torque stopped motor is a fault or not. If the parameter CL[2] is less then 2, a high torque stopped motor is not considered a fault. If the parameter CL[2] is 2 or more, then a high torque stopped motor, detected for at least 3 continuous seconds, is considered a fault. The motor is set to off (MO=0) and MF=0x200000. The time constant of 3 seconds is taken since almost every motion system applies high torques for short acceleration periods while the speed is slow.

CL[2] defines the tested torque level as a percentage of the continuous current limit CL[1].
CL[3] states the absolute threshold main sensor speed under which the motor is considered not moving.

Example
If CL[2]=50 and CL[3]=500, the Metronome will abort (reset to MO=0) if the a torque level is kept at least 50% of the continuous current, while the shaft speed do not exceed 500 counts/sec for continuous 3 seconds.

9.6 Commutation is lost

9.6.1 General

The Metronome uses the feedback (encoder\resolver\tacho) counts to calculate the electric angle of the rotor. This is used to set the currents at the stator so that the magnetic field of the stator will point 90 degrees away from the rotor (90+/-30 degrees at the Clarinet).
The angle between the magnetic field in the stator and the rotor is called the “commutation angle”
The motor torque is

$$T = \frac{3}{2} Ke \cdot I \cdot \sin(\theta_s - \theta_r)$$

Where T is the torque, Ke is the motor constant, I is the motor current, θ_s is the stator field angle, and θ_r is the rotor angle.
The difference $\theta = \theta_s - \theta_r$ is called the commutation angle. Obviously, θ must be kept near 90 degrees for the motor to function properly.
If the commutation angle is incorrectly set, the motor loses torque. For a given torque command (given either directly in UM=1 or by external control loops in other modes)
The motor current remains the same (heating…)
The torque falls, since the proportion between the current and the torque is $\cos(\Delta\theta)$ where $\Delta\theta$ is the commutation angle error.
Note than in extreme cases, where $abs(\Delta\theta) > 90^\circ$, the motor torque becomes reversed with respect to what is expected by the commanded current.

9.6.2 Reasons and effect of incorrect commutation

Incorrectly set commutation errors is disastrous to drive operation.
The most common incorrect commutation behaviors are:

The commutation error is static (i.e. $\Delta\theta$ Does not change in time):
Static commutation error occur because of bad setup data, or by exceptional load in automatic alignment (at MO=1, when the only sensor available is an incremental encoder)
A Static commutation error leads to motor torque reduction, reduced efficiency, degraded dynamic response, and possible speed or position loop instability.

The commutation is static (i.e. θ_s does not change as a function of the motor position):
Static commutation occurs in encoder systems when an encoder wire is broken.
In this situation, the direction of the stator field is constant, and if torque is commanded, the rotor seeks equilibrium, aligned to the magnetic field.
If the motor is driven by an external speed or position controller, it will be commanded to full-torque, and dissipate the corresponding heat, without generating any motion.

The commutation is drifting (i.e. $\Delta\theta$ Changes in time):
Drifting in $\Delta\theta$ occur in two forms:

Slow drift:
The following reasons:

- Excessive noise on the encoder lines
- Damaged or dirty encoder
- Wrong encoder resolution setting

Usually drifts $\Delta\theta$ slowly. When the motor is stopped, $\Delta\theta$ does not drift at all.
Slow drifting will cause the motor to lose torque gradually until fully stopped at the static setting of $\Delta\theta = \pm 90^\circ$. At this point, the current in the motor will not produce any torque. The motor shall overheat, but not move.
Fast drift:
Faulty resolver wires cause the resolver to digital converter to elude motion in its full supported speed. The motor behavior becomes bizarre, normally producing oscillations without much average torque.

9.6.3 Detection of Commutation (feedback) Fault.

The detection of commutation faults is according to the sensors present.
Three categories are dealt:

- Double sensor systems
- Resolver system

Double sensor systems
The situation is the easiest if there are two sensor systems, as happens in encoder + digital hall systems. In this case, there is a rough commutation estimate available by the digital Hall sensors, that measure the electrical angle of the motor directly.
Bad correlation between the encoder-accumulated commutation angle and the Hall sensors angle is a sign of faulty commutation.
If the accumulated commutation angle differs by more then 15 electrical degrees than its closest value that fits the Hall sensor reading, an error is to be declared. The motor is shut (MO=0) and MF is set to 0x4. This way an error is decided when the commutation is 15 to 45 degrees wrong.

Resolver systems
With resolver systems, reading speeds out of the device threshold (+/-1.6 Mcount/sec for the AD2s90 R/D converter used in the Saxophone and the Clarinet) tells us that the sensor is not healthy and that the motor cannot be driven.

After the detection of a commutation fault, the motor must be shut down and a proper MF code set active.
There is no point in trying to stop the motor under control, since we don’t know the correlation between the motor currents and the resulting torque.
At the next motor starting attempt, the motor will seek commutation again.

10 The Control Filter

10.1 General

This Section details the speed and position control algorithms used by the Metronome. For many applications, the details of this document are of no concern. People do not have to understand the internals of a motion controller in order to tune it with a Composer program. This paper is aimed for the advanced user who wants to tune the servo drive manually, or to understand thoroughly what goes on when Composer tunes the servo drive.
This section does not cover the digital current loop.
The rest of this document assumes familiarity with the basic ideas of Digital Control Theory.

The type of the controller used depends on the Metronome usage mode. The “Unit Mode” (UM) software variable defines which control algorithm shall be used, according to the following table.

Unit mode	Control algorithm
1	Open loop
2	Speed control
3	Micro Stepper
4	Position, dual feedback source. One sensor serves for commutation and speed control, the other sensor serves for load position control.
5	Position, single feedback source

Table 31 – Unit Mode Values

The algorithms used for each unit mode are configurable.
In the basic level, the control algorithms are the traditional PI (Proportional-Integral) or PID (Proportional-Integral-Derivative).
The virtue of the PI/PID algorithms is in their simplicity. Few hours of playing will suffice for a technician to percept how the modification of the P, I or D controller parameters affect the performance of the motor.
Life with PID becomes hard as the control requirements become tough.
For this reason, the PI/PID algorithms are extended with the following features:

- A third order linear filter may be used for notching a resonance, or for high passing of high frequency disturbances.
- A continuous gain scheduling scheme enable optimum performance in wide range of speeds.

The third order filter may be tuned manually by the advanced user, but this is not so simple as tuning a PI/PID.
The gain scheduling is programmed by the auto-tuner. Although an extremely advanced user may program it manually.

The following table lists the parameters of the algorithms in this section. Please refer the Command Reference Manual for the details.

UM	Unit Mode. This parameter determines the type of control algorithm to use – speed, position, dual-position, or open loop

KP[N]	Proportional gain.
	N=1: Current loop (only for servo drives with digital current loop)
	N=2: For speed (UM=2)
	N=3: For position (UM=4 and UM=5)

KI[N]	Integral gain.
	N=1: Current loop (only for servo drives with digital current loop)
	N=2: For speed (UM=2)
	N=3: For position (UM=4 and UM=5)

KD[N]	Derivative gain N=3: For single feedback position (UM=4)
KF[N]	Coefficients for the 3rd order linear filter that may be used to low pass high frequency phenomena, or to notch resonance. The parameter KF[0] asserts if the 3rd order filter is used at all – if KF[0] is zero, then the filter is not used.
GS[N]	Gain scheduling. GS[0] controls the proportional gain of the speed controller in very slow speeds. GS[2] controls the gain scheduling process for speed control. GS[3] controls the gain scheduling process for position control. Other members of the GS[] vector are described in the reference manual
MC	Motor current. This is the largest current that the servo drive can control. For an 10 Amp servo drive the peak current MC will be 20 Amp. MC is factory set.
TS	Sampling time of the controller in microseconds. Normally lower TS corresponds to better control performance, penalized by less CPU resources and slower execution of user program commands. **Notes** 1. For servo drives with digital current loop, TS represents the sampling time of the current control loop. The speed and the position loops are four times slower. For example, if a Saxophone servo drive is set to TS=90, then its speed (or position) controller will sample at 360 microseconds. 2. The minimum value available for TS depends on UM.

Table 32 – Parameters for GS

Notation:
We use standard math notation. The time derivative is denoted by the letter s.
The expression sx is equivalent to dx/dt, and the expression x/s is equivalent to $\int xdt$, where t is the time and x(t) is any signal.
The operator z denotes a time advance of a single sampling time. For a digital system with the sampling time of T_s we have $zx(kT_s) = x[(k+1)T_s]$, or simply $zx(k) = x(k+1)$.

10.2 Speed Control

This is the most basic closed loop control form.
The basic control block of the speed controller is the PI. The optional blocks are the third order linear filter and the gain scheduler.

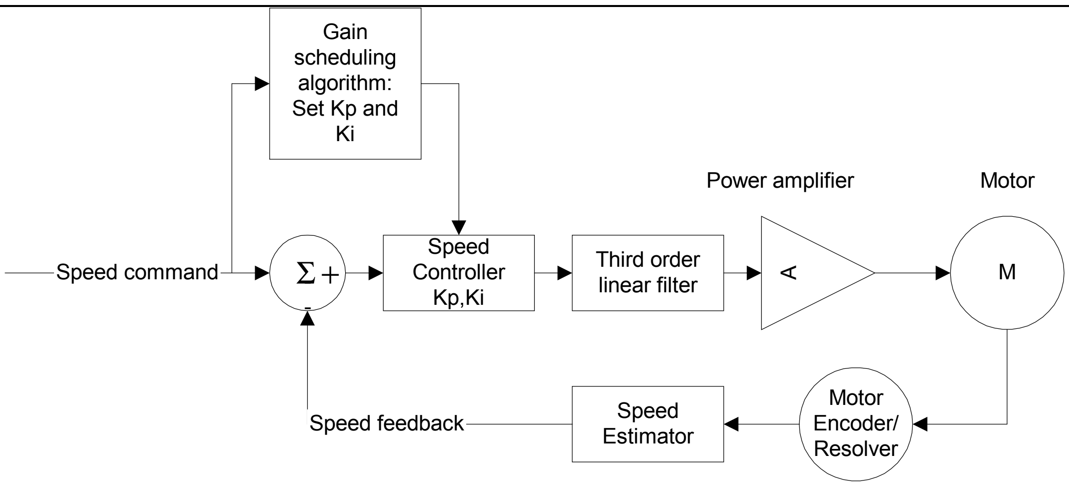


Figure 14 – A block Diagram of the Speed Controller

10.2.1 The Basic PI Controller

The basic continuous time PI controller is $\cdot \dfrac{K_I + K_P s}{s}$.

Where:

K_I is the integral parameter

K_P is the proportional parameter

In general, these parameters are the functions of time, defined by the gains scheduling algorithm.

The input of the PI element is speed error $e_{SPEED}(t)$ [count/sec].

The output is the current command $I(t)$ [A]:

$$\mathrm{I(t)} = \mathrm{K}_{\mathrm{NormSpeed}} \cdot \left(\int_0^t \mathrm{K_I} \cdot \mathrm{e_{SPEED}}(\tau)\mathrm{d}\tau + \mathrm{K_P} \cdot \mathrm{e_{SPEED}(t)} \right)$$

where

$$K_{NormSpeed} = \frac{0.064}{The\, number\, of\,\, position\, counts\, in\, 1\, motor\, revolution}$$

In a digital servo drive, the PI element is implemented as

$$I(kTs) = K_{NormSpeed} \cdot (X(kTs) + K_P \cdot e_{SPEED}(kTs) \cdot G(kTs))$$

$$\mathrm{X((k+1)Ts) = X(kTs) + K_I \cdot Ts \cdot e_{SPEED}(k)}$$

where

$X(t)$ is the state of speed error integrator,

and where

Ts is the sampling time, $Ts = \begin{cases} TS \cdot 10^{-6} & \text{Amplifiers with analogue current loop} \\ TS \cdot 4 \cdot 10^{-6} & \text{Amplifiers with digital current loop} \end{cases}$

The parameter $\mathrm{K_{NormSpeed}}$ does the converts speed in [count/sec] to current in [Amp].

$$G(kTs) = \begin{cases} 1, \text{if at least at one of the last GS[0] sampling times the position reading changed} \\ 0, \, else \end{cases}$$

The parameter GS[0] represents a little dirty trick, which helps to stabilize the motion in very slow speeds. Consider for example a 200-count/sec speed reference. A new encoder count is available once per 5 msec – which are about 15 sampling times of the speed controller. If a new encoder pulse comes only once per 5 msec, then the speed readout is delayed by at least 2.5msec, which in turn may have fatal effect on the control stability. The $G(kTs)$ trick, with N=7 (factory default) will reduce the proportional gain of the controller by half (7 is about a half of 15). The reduced gain implies reduced bandwidth and increased stability.

The price of the trick is that it slows down the acceleration from a complete rest.

The trick may be canceled by setting GS[0] = 100.

10.2.2 User Interface

The PI parameters may be set automatically by the gain scheduler, or manually by the user. If gain scheduling is not desired, set GS[2]=0. Then the PI parameters K_I and K_P are the constants directly set by the user parameters KI[2] and KP[2] respectively.
Position Control
The basic control block of the position controller is the PID. The D (derivative) term is required to stabilize the position loop, since the transfer function from the motor current to the motor position normally includes a double integral.
The optional blocks are the third order linear filter and the gain scheduler.

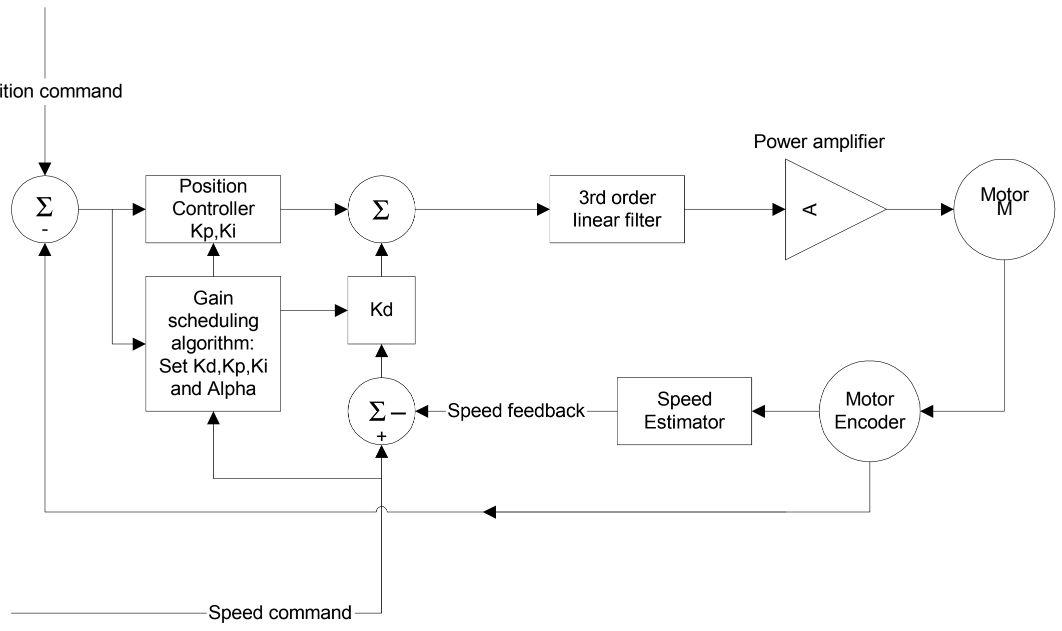


Figure 15 – Bloc Diagram of The Position Controller

10.3 The Basic Position Controller

The basic continuous time PID controller is $\cdot \dfrac{KI + KPs + KDs^2}{s}$.

The input of the element is position error $e_{POS}(t)$ [count], and speed error $e_{SPEED}(t) \cong \dfrac{\partial}{\partial t} e_{POS}(t)$ [count/sec].

The output is current command $I(t)$ [A]:

$$\mathrm{I(t)} = \mathrm{K}_{\mathrm{NormPos}} \cdot \left(\int_0^t \mathrm{K}_{\mathrm{I}} \cdot \mathrm{e}_{\mathrm{POS}}(\tau) d\tau + \mathrm{K}_{\mathrm{P}} \cdot \mathrm{e}_{\mathrm{POS}}(\mathrm{t}) + \mathrm{K}_{\mathrm{D}} \cdot \frac{\partial}{\partial \mathrm{t}} \mathrm{e}_{\mathrm{POS}}(\mathrm{t}) \right) =$$

$$= \mathrm{K}_{\mathrm{NormPos}} \cdot \left(\int_0^t \mathrm{K}_{\mathrm{I}} \cdot \mathrm{e}_{\mathrm{POS}}(\tau) d\tau + \mathrm{K}_{\mathrm{P}} \cdot \mathrm{e}_{\mathrm{POS}}(\mathrm{t}) + \mathrm{K}_{\mathrm{D}} \cdot \mathrm{e}_{\mathrm{SPEED}}(\mathrm{t}) \right)$$

It is implemented as

$$\mathrm{I(kTs)} = \mathrm{K}_{\mathrm{Norm}} \cdot (\mathrm{Y(kTs)} + \mathrm{K}_{\mathrm{P}} \cdot \mathrm{e}_{\mathrm{POS}}(\mathrm{kTs}) + \mathrm{K}_{\mathrm{D}} \cdot \mathrm{e}_{\mathrm{SPEED}}(\mathrm{kTs}))$$
$$\mathrm{Y((k+1)Ts)} = \mathrm{Y(kTs)} + \mathrm{K}_{\mathrm{I}} \cdot \mathrm{Ts} \cdot \mathrm{e}_{\mathrm{POS}}(\mathrm{kTs})$$

where

$Y(t)$ is the state of the position error integrator,

$\mathrm{K}_{\mathrm{NormPos}} = \mathrm{K}_{\mathrm{NormSpeed}}$, and Ts are the same as defined previously for the speed controller.

10.3.1 Acceleration Limiting

The use of automatic gain scheduler programming provides a capability of acceleration limiting. The last is defined by the factor α, which is given in the range of 4..128. In this case advanced position controller is implemented:

$$I(t) = K_{NormPos} \cdot \left(\int_0^t K_I \cdot f(e_{POS}(\tau))d\tau + K_P \cdot \frac{f(e_{POS}(t)) + e_{POS}(t)}{2} + K_D \cdot \frac{\partial}{\partial t} e_{POS}(t) \right) =$$

$$= K_{NormPos} \cdot \left(\int_0^t K_I \cdot f(e_{POS}(\tau))d\tau + K_P \cdot \frac{f(e_{POS}(t)) + e_{POS}(t)}{2} + K_D \cdot e_{SPEED}(t) \right)$$

It is implemented as

$$I(kTs) = K_{NormPos} \cdot (Y(kTs) + K_P \cdot \frac{f(e_{POS}(kTs)) + e_{POS}(kTs)}{2} + K_D \cdot e_{SPEED}(kTs))$$

$$Y((k+1)Ts) = Y(kTs) + K_I \cdot Ts \cdot f(e_{POS}(kTs))$$

The function f(x) is calculated as

$$f(x) = \operatorname{sign}(x) \cdot \min(|x|, \alpha \cdot \sqrt{|x|}),\ 0 \le |x| \le 32767$$

Use of such element enables to limit current command and since to acceleration as well, when position error is larger than α^2.

The manual programming implies the use of $\alpha = 128$, i.e. no acceleration limiting for $|e_{POS}| < 16384$. In this case we have $f(e_{POS}) = e_{POS}$ and the advanced controller it is equivalent to the basic PID controller.

10.3.2 User Interface

The PID parameters may be set automatically by the gain scheduler, or manually by the user. If gain scheduling is not desired, set GS[3]=0. Then K_I, K_P and K_D are directly set by the user parameters KI[3],KP[3] and KD[3], respectively. The α may be set by GS[9]= x, where $x = \alpha \cdot 256$.
Note that a speed controller with KP[2]=A and KI[2]=B (A,B are any pair of positive numbers) is equivalent to the position controller with KD[3]=A, KP[3]=B, and KI=0.

10.4 Position Control With Dual Loop

The dual loop position controller is build from an auxiliary position loop closed over the basic speed controller. The reference to the speed controller is composed of the derivative of the position command and of the output of the auxiliary position controller.
The derivative of the position command is multiplied by the FF factor, in order to compensate the gear ratio.

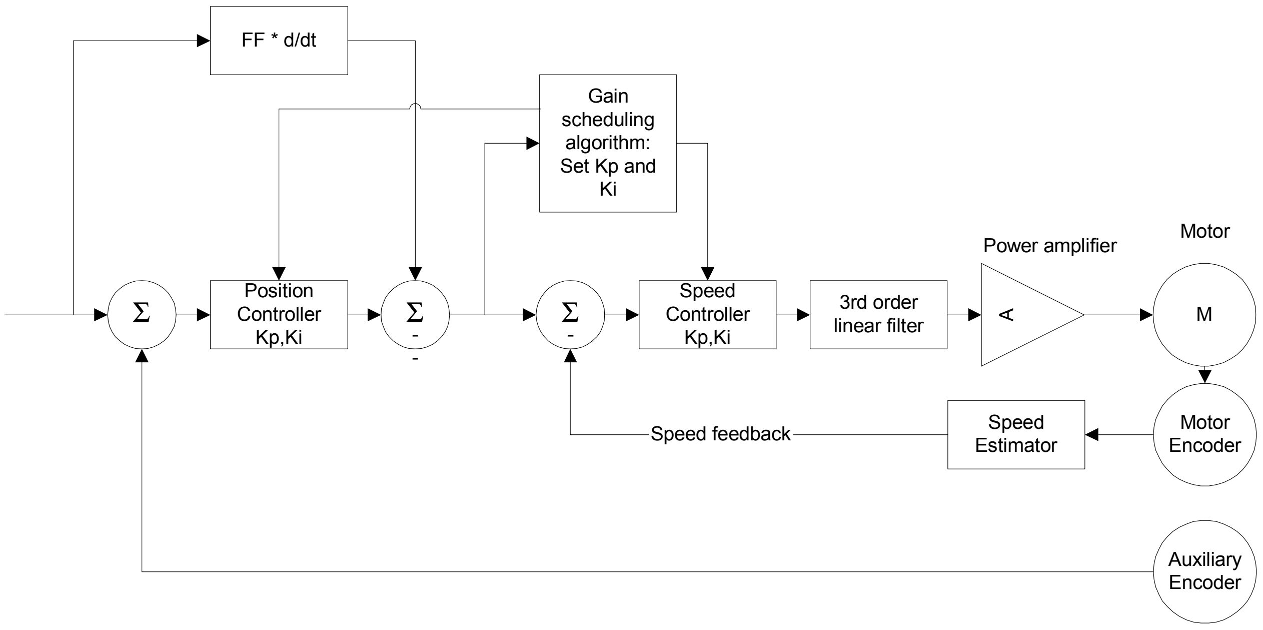


Figure 16 – Block Diagram of The Dual Loop Controller

The output of the speed command generator is calculated as

$$r_{SPEED}(t) = K_{FF} \cdot \frac{\partial}{\partial t} r_{POS}(t) + K_{NormDual} \cdot \left(\int_0^t K_I^{Dual} \cdot e_{AUX\_POS}(\tau) d\tau + K_P^{Dual} \cdot e_{AUX\_POS}(t) \right)$$

where

$r_{SPEED}(t)$ is speed reference in [count/sec],

K_{FF} is feed forward coefficient which must be ratio of main and auxiliary encoder resolutions,

K_I^{Dual} is integral parameter

K_P^{Dual} is proportional parameter

$$K_{NormDual} = \frac{1}{The\,number\,of\;position\,counts\,in\,1\,motor\,revolution \cdot Ts \cdot 4000}$$

Ts and '*the number of position counts in 1 motor revolution*' are the same as described for the speed controller.

10.4.1 User Interface

The PI parameters K_I^{Dual} , K_P^{Dual} of the speed reference generator are set manually by the user parameters KI[3] and KP[3], respectively. The PI parameters of the speed controller may be set automatically by the gain scheduler, or manually by the user. If the gain scheduling is not desired, set GS[2]=0. Then K_I , K_P are directly set by the user parameters KI[2],KP[2] , respectively. The K_{FF} may be set by FF=x, where $x = \frac{The\,number\,of\;position\,counts\,in\,1\,main\_motor\,revolution}{The\,number\,of\;position\,counts\,in\,1\,aux\_motor\,revolution}$.

Note that a dual loop controller with KP[3]=0 and KI[3]=0 is equivalent to the speed controller with KI[2], KP[2] set up.

10.5 The 3rd Order Linear Filter

The 3rd order linear filter is divided into two blocks connected in series. The first block has the first order. It implements a lead-lag or a lag element.
Another block has the second order. It implements a notch filter or a 2nd order low pass.
The first-order (Lead – Lag) element.

The basic continuous time lead-lag element is $\frac{b}{a} \cdot \frac{s+a}{s+b}$

The frequency $a/(2\pi)$ [Hz] is the lead corner frequency.

The frequency $b/(2\pi)$ [Hz] is the lag corner frequency.

The digital lead-lag Pade approximation is

$$\frac{1-\beta}{1-\alpha} \cdot \frac{z-\alpha}{z-\beta} \qquad (1)$$

$$\alpha = \frac{2-aT_s}{2+aT_s},\ \beta = \frac{2-bT_s}{2+bT_s}$$

where T_s is the sampling time.

The element is implemented by the state equations:

$$\begin{aligned} y_{k+1} &= (1-p)u_{k+1} + ps_k \\ s_{k+1} &= s_k + q(u_{k+1} - s_k) \end{aligned}$$

Where

$$q = 1-\beta = \frac{2bT_s}{2+bT_s},\ p = \frac{\beta-\alpha}{1-\alpha} = \frac{2(a-b)}{a(bT_s+2)}$$

A simple pole is obtained in the special case of a = 0. There

$$p = \exp(-b \cdot T_s), q = 1-p$$

10.6 The Second Order (Notch or Low Pass) Element

The basic continuous time second order element is $\frac{D}{B}\cdot\frac{s^2+As+B}{s^2+Cs+D}$.

The numerator of the element has a complex pair of zeros, with corner frequency of ω_1 and with damping D_1. If $D_1>1$, the pair of zeroes become real.

The denominator has a complex pair of poles with corner frequency ω_2 and with damping D_2. If $D_2>1$, the pair of poles become real.

The coefficients of the 2nd order filter are

$A=2D_1\omega_1$

$B=\omega^2{}_1$

$C=2D_2\omega_2$

$D=\omega^2{}_2$.

For a notch, we set $\omega_1=\omega_2=2\pi\cdot f_{notch}[Hz]$, $0<D_1<0.2$, and $0.6<D_2<1$

For a low pass, we set $\omega_2=2\pi\cdot f_{corner}[Hz]$, $0.6<D_2<1$, $A=0, B\to\infty$.

$\frac{b_0z^2+b_1z+b_2}{z^2+a_1z+a_2}$ The digital implementation of the continuous second order element is with the unity DC gain condition

$b_0+b_1+b_2=1+a_1+a_2$

For frequencies below ¼ of the Nyquist frequency, the behavior of the digital implementation well approximates the behavior of the continuous filter with

$$b_0=\frac{D(1+AT_s)}{B(1+CT_s)}$$

$$b_1=\frac{D(BT_s^2-AT_s-2)}{B(1+CT_s)}$$

$$b_2=\frac{D}{B(1+CT_s)}$$

$$a_1=\frac{DT_s^2-CT_s-2}{(1+CT_s)}$$

$$a_2=\frac{1}{1+CT_s}$$

Note that for a low pass design with $B\to\infty$ only b_1 exists. In that case the resulting coefficient b_1 is advanced to the place of b_0 to prevent an unnecessary delay.

The bilinear approximation $s=\frac{2}{T_s}\cdot\frac{z-1}{z+1}$ leads to better approximation.

The design of high frequency notches requires pre-warping of the approximated continuous time filter. Bilinear transformations and pre-warping are covered by almost all standard digital control text books.

The 2nd order element is implemented by

$y_{k+1}=y_k+k_1(u_{k+1}-y_k)+k_2(u_{k+1}-x_k)+k_3(u_k-u_{k-1})a_2+k_4(y_k-y_{k-1})$

where

$$k_1 = (b_0 + b_1 + b_2)$$
$$k_2 = -(b_1 + b_2)$$
$$k_3 = -b_2$$
$$k_4 = a_2 \ .$$

10.6.1 User Interface

KF[0] enables / disables the 3rd order linear filter. It may only be changed when MO=0.
To disable the filter:
KF[0]= 0.
To enable the filter:
KF[0]= 1.
Access the lead-lag element parameters p, q is available through the KF[5],KF[6] user command:
KF[5]= $p \cdot 2^{15}$
KF[6]= $q \cdot 2^{15}$
The setting
KF[5]=0,KF[6]=0 disables lead-lag.
Access to of notch filter parameters $\mathrm{k}_1, \mathrm{k}_2, \mathrm{k}_3, \mathrm{k}_4$ is available through the KF[1]..,KF[4],KF[7]..KF[9].
KF[1]= $\mathrm{k}_1 \cdot 2^{14+\mathrm{l}_1}$, -16383.. 16384
KF[2]= $\mathrm{k}_2 \cdot 2^{14-\mathrm{l}_2}$, -16383.. 16384
KF[3]= $\mathrm{k}_3 \cdot 2^{14-\mathrm{l}_3}$, -16383.. 16384
KF[4]= $\mathrm{k}_4 \cdot 2^{14}$, -16383.. 16384

KF[7]= l_1, -8..32
KF[8]= l_2, 0..8
KF[9]= l_3, 0..8

The setting KF[1]=16384, KF[2]=0, KF[3]=0,KF[4]=0,KF[7]=0,KF[8]=0,KF[9]=0 disables the notch filter.

11 Gain Scheduling Implementation

11.1 The Gain-Scheduling Algorithm

The gain scheduling (GS) is implemented for speed and position controllers. This means, that the controller gains (KI/KP/KD/ α) are changed automatically when the working point of the servo drive changes. The gain scheduling of a speed controller is affected only by the level of the command reference. The gain scheduling of a position controller is affected by the position error, and by the derivative of the position reference.

11.2 Speed Controller Gains Scheduling

The control input to the gain scheduling is the speed reference. The goal of the algorithm is to decrease the gains when the speed is low, since in this case the feed back data is accepted with the very large delay.
The gains are scheduled according to the reference speed. If the desired speed is high, then the encoder data is to update in high rate, so that the speed sensing delay will be minimal. If the desired speed is low, then encoder pulses shall be slow. For example, suppose that a speed of 200 count/sec is required. Then we have a new encoder count once per 5 msec. The calculated speed will have a 2.5msec delay which may kill the stability of the control algorithm.
For that reason, if the speed goes low, the controller bandwidth must be narrowed, so that the extra sensing delay becomes tolerable.

The speed command is filtered, and the filtered value of the speed command serves to fetch the relevant controller parameters KP and KI from a table.
The filtering is required to avoid abrupt changes of controller parameters, since abrupt changes can spoil the guaranteed stability of the gain scheduled controller (Assuming that each table entry corresponding to a given speed leads to a stable controller in this speed)
The speed range where the scheduling is effective is the range of about 0..20000 count/sec.

The Ki and the Kp parameters are tabulated in the vectors
$SpeedKiTable[0:63], SpeedKpTable[0:63]$.
The tabulated gains are not tabulated linearly by the commanded speed, since the dependence of the parameters in the speed is very nonlinear. The controller parameters are very sensitive to the command speed about the zero speed (At zero speed the feedback delay approaches infinity). Close to the upper end of the table, at about 20000 count/sec, the controller parameters become insensitive to the speed.
For this reason, the speed command does not index the gains table directly. Instead, the parameter tables are indexed by a nonlinear, monotone, function of the commanded speed. This indexing function is tabulated in the vector
$SpeedIndexTable[0:62]$

The actual parameter indexing process goes as follows:

Saturate speed command	$X = \max(\lVert \text{speed reference} \rVert, 20000 \text{ count/sec})$
Filter commanded speed Note that the filter does not respond symmetrically to rising and falling speed commands. High speed commands are responded immediately, for swift response, where as low speed commands are accepted slowly to guarantee stability.	$Y(m+1) = \beta \cdot Y(m) + (1-\beta)X$ where $\beta = \begin{cases} GsFilterUpGain, \text{ if } X \ge Y(m) \\ GsFilterDnGain, \text{ if } X < Y(m) \end{cases}$
Get the index to the gains table by interpolating the indexing function	$n = [Y/4096],$ $q = (Y/16 \bmod 256)/256$ $k = SpeedIndexTable[n] +$ $(SpeedIndexTable[n+1] - SpeedIndexTable[n]) \cdot q$
Get the gains	$KP = SpeedKpTable[k]$ $KI = SpeedKiTable[k]$

Table 33 – GS Parameter Indexing Process

11.3 Position Controller Gain Scheduling

The control inputs to the gain scheduling are the position reference derivative and the position error. The goal of the algorithm is to decrease the gains when the speed is low and the position target had been reached, since in this case feed back data is accepted with the very large delay. The speed range where the scheduling is effective is the range of about 0..20000 count/sec. Before choosing the appropriate gains the index to the gain tables is filtered. The enables the quick increase of gains when position error or position reference derivative is going up and to decrease the gains slowly when the latter are going down, so that stability remains guaranteed.

The Ki, Kp, Kd, and Alpha (acceleration limit) parameters are tabulated in the vectors
$PosKiTable[0:N], PosKpTable[0:N], PosKdTable[0:N], AlphaTable[0:N]$
The tabulated gains are not tabulated linearly by the commanded speed, since the dependence of the parameters in the speed is very nonlinear. The controller parameters are very sensitive to the command speed about the zero speed (At zero speed the feedback delay approaches infinity). Close to the upper end of the table, at about 20000 count/sec, the controller parameters become insensitive to the speed.
For this reason, the speed command does not index the gains table directly. Instead, the parameter tables are indexed by a nonlinear, monotone, function of the commanded speed. This indexing function is tabulated in the vector
$PosIndexTable[0:62]$

The actual parameter indexing process goes as follows:

Saturate speed command (the speed command is obtained by decomposing the position controller so that it may be viewed as a proportional gain controller closed over a PI speed controller)	$X = \max(\lVert \text{Position command derivative} \rVert + K_{SC} \cdot \lVert \text{position error} \rVert, 20000 \text{ count/sec})$
Filter commanded speed Note that the filter does not respond symmetrically to rising and falling speed commands. High speed commands are responded immediately, for swift response, where as low speed commands are accepted slowly to guarantee stability.	$Y(m+1) = \beta \cdot Y(m) + (1-\beta)X$ where $\beta = \begin{cases} GsFilterUpGain, \text{ } if \text{ } X \geq Y(m) \\ GsFilterDnGain, \text{ } if \text{ } X < Y(m) \end{cases}$
Get the index to the gains table by interpolating the indexing function	$n = [Y/4096]$, $q = (Y/16 \bmod 256)/256$ $k = PosIndexTable[n] +$ $(PosIndexTable[n+1] - PosIndexTable[n]) \cdot q$
Get the gains	$KP = PosKpTable[k]$ $KI = PosKiTable[k]$

Table 34 – GS Actual Parameters Indexing

11.4 Gain Scheduling Programming

To access the GS data the GS,KI,KP,KD user commands are to be used.
To access SpeedKiTable[k]/SpeedKpTable[k], k=0..N,N<63:
Set GS[3]=0 or 64
Set GS[2]=k+1
Use KI[2]/KP[2] to access the appropriate table entry

To access PosKiTable[k]/PosKpTable[k]/ PosKdTable[k]/AlphaTable[k]:

Set GS[2]=0 or 64
Set GS[3]=k+1
Use KI[3]/KP[3]/KD[3]/GS[9] to access the appropriate table entry

To access SpeedIndexTable[n], n=0..62:
Set GS[6]=n
Use GS[7] to access the table entry (the value = 0..N)

To access PosIndexTable[n] , n=0..62:
Set GS[6]=n
Use GS[8] to access the table entry

To access GsFilterUpGain (0<= value <= 1):
Use GS[4] = GsFilterUpGain*2^15

To access GsFilterDnGain (0<= value <= 1):
Use GS[5] = GsFilterDnGain*2^15

To access to K_{SC} (0 <= value <= 32767):
Use GS[10] = K_{SC}
To switch speed gain scheduling:
Set UM=2 or 4
Use GS[2] = 64 to turn on
Use GS[2]=0 to turn off
To switch position gain scheduling:
Set UM=5
Use GS[3] = 64 to turn on
Use GS[3]=0 to turn off

11.5 Programming The Gains for Non-Scheduled Mode

To access KI/KP for speed:
Set GS[2]=0
Set GS[3]=0 or 64
Use KI[2]/KP[2] commands

To access KI/KP/KD/Alpha for position:
Set GS[3]=0
Set GS[2]=0 or 64
Use KI[3]/KP[3]/KD/GS[9] commands

Default value for GS[9] is 32767.
To access KI/KP for current:
Set GS[2]=0 or 64
Set GS[3]=0 or 64
Use KI[1]/KP[1] commands

12 Appendixes

12.1 The Metronome Error Codes

This appendix summarizes all the error codes the Metronome uses for RS232/485 communications and programming. CANopen abort messages and error codes are covered in the CANopen reference manual.

12.1.1 Compilation Time Errors

The compiler issues compilation time errors in response to syntax errors.

Error Code	Description/Example	Example/Remedy
29	Program With Too Many Errors. Compilation was stopped before scanning the entire program, because too many errors were found.	
31	Compilation failed	Error code for the CC command if compilation revealed errors
43	Not Valid Label. The label name does not obey the syntax rules.	#%LOOP is an illegal label name.
49	Two Similar Labels.	The two labels **Example1:** ##LOOP and #@LOOP exist in the same program. **Example2:** A program section has been duplicated by cut and paste, without modifying the labels at the copied part, so that the program contains two identical labels.
79	No Variable No variable to be assigned was found in an IN command.	The program line IN DON'T ENTER; Will return that error.
80	More Than 200 Labels	The program contains more then 200 unique labels, excluding the names of auto routines.
81	Label too long	If the program terminates by ##LASTLABEL Without a terminator or an EN command, the compiler will not be able to find the end of the label before the end of the program.
83	CMD Not For Program. This command can't be used in a program.	The command XQ, when included in a program, shall invoke this error since a program cannot execute itself.

Table 35 – Compilation Time Errors

12.1.2 Run Time Errors

The Metronome issues run time errors in response to program management errors.

Error code	Description	Possible Reason
45	Return Error from Subroutine	Too many nested subroutine calls. Possibly there is a “call stack leakage” – one of the subroutine returns without an RT command, leaving its return address in the call stack.
74	Program Stack Overflow	An RT command encountered with no corresponding return address in the call stack. Check if there is a spurious RT command in the program, or if a subroutine is entered not through the use of the JS command.

Table 36 – Run Time Errors

12.1.3 Errors In Response To Commands

The command processing error codes reflect why the Metronome was unable to perform a certain command.

Error Code	Description/Example	Example/Remedy
2	Bad Command. The interpreter has not understood the command.	XF=2; is an error since there is no XF command. MC=2; is an error since the MC setting is a privileged command. If the password is not entered, the MC set command is not identified
3	Bad index. Attempt to access a vectored variable out of its boundaries.	Observe the index range for the used command. IL[6]; is an error since the index range for the IL command is [1..4].
4	A Program Flow Command. This command manages the control flow in a user program. It can't be used via communication.	As a terminal command, JP##AAA is an error. Use the XQ command to set the program counter from the terminal.
5	No Interpreter Meaning. Un-recognized character has been found were a command was expected.	A*=3 is an error since a command mnemonic made of two alphabetic characters has been expected.
6	Program is not running.	MI return this error when called if the program is not running, since the MI parameter will be automatically set to zero when the program starts.
12	Command not available in this unit mode.	PA=1000 is an error if UM=2, since in this mode the position cannot be commanded.
13	Cannot reset communication - UART is busy. An attempt to modify the parameters of the serial communication while the communication line is busy.	
16	Collision of RS-485 messages. The Metronome tried to transmit a message, but characters have been destroyed by electrical interruption – probably the network master, or another servo drive tried to transmit on the RS485 lines at the same time.	Test that: No two servo drives in the network have the same ID. No more then one servo drive is defined as network representative. No more then one servo drive in a group is defined as a group representative Network time-outs are correctly set.
17	Sampling time is too short for the mode. Some modes use less CPU, so they can run with higher sampling rate. An attempt has been made to set a CPU consuming motion mode while TS is defined too low.	If UM=2 and the sampling time is set as the minimum for UM=2, attempting UM=4 will return this error.

Error Code	Description/Example	Example/Remedy
18	Empty Assignment. The right side of an equation is missing.	AC=; is an error, since the interpreter expected a numerical value to appear after the '=' sign.
19	Bad Command Format	An unresolved syntax error in the command – please refer this manual for the correct command syntax.
20	Operand Error. Attempt to read a command that do not return a value.	
21	Operand Out of Range. Attempt to assign an illegal value to a parameter.	JV=100000000 returns this error since the required speed is too large for the Metronome.
22	Zero Division.	JV=0; IA[1]=1000/JV Will return this error.
23	Command Cannot Be Assigned.	BG=3000 returns this error. BG is an execution command that do not have a value.
24	Bad operator. An un-recognized character has been found in an expression where an operator has been expected.	IA[1]=3$VX is an error since $ is not a recognized operator.
25	Command Not Valid While Moving.	PV=n while in PVT motion is an error since the PV=n command sets the read pointer of the PVT table manually; while in the PVT mode this pointer is set automatically.
26	Motion Mode Not Valid. A Begin Motion attempted while the parameters of the motion has not been properly set.	PV=n;BG is an error if the first valid line of the PVT table is smaller then the last valid line.
27	Out Of Modulo Range.	XM=1000;PA=2000 is an error. XM=1000 defines that the maximum encoder count be 500. Therefore the position of PA=2000 can never be reached.
28	Out Of Limit Range. A command has been specified out of its permitted limits.	VH[2]=1000;SP=2000 is an error. The latter command specified that point-to-point motions shall reach the speed of 2000 counts/sec, where the first command limited the maximum speed command to 1000.
29	Program With Too Many Errors. Compilation was stopped, since the compiler already recognized as many errors as it can count.	
30	No program to continue	An XC command has been issued but no halted program is there to be continued

Error Code	Description/Example	Example/Remedy
32	Communication parity, noise, or framing error	Test that Communication lines are well connected with adequate ground. Baud rate and other communication parameters are set consistently for the master and the slave.
33	Communication problem due to receiver overrun. Arriving communicated characters are normally stored by hardware until processed. Too many characters arrived without being processed, and the hardware storage capability has been exceeded.	CAN messages are sent in too high rate. For RS232 and RS485, this message is not expected.
34	This command cannot be broadcast to a group of servo drives.	The command PP[12]=3 sets the servo drive to be a group representative. If this command is broadcast to a group, it is an error. If accepted, all the listening servo drives will become representatives and answer simultaneously. The network shall crash.
36	Bad commutation table The data points in the back EMF table HV do not form a valid back EMF function.	This error is normally located while attempting MO=1. At MO=0 the HV table can be updated so its consistency is not required.
37	Two Hall sensors are defined to the same place.	One of the following: CA[4]=CA[5], CA[4]=CA[6] or CA[5]=CA[6].
42	No Such Label. The program does not contain a label with the specified name.	XQ##FOO will return that error if the label ##FOO does not exist in the user program.
47	Program Not Compiled	XQ shall return this error if no program had been loaded to the Metronome and successfully compiled.
50	Stack Overflow A CPU exception detected. This error reflects one of the following: A bug in the Metronome A hardware problem. A faulty power supply.	Please use CD to find out the details of what happened. If “Called handler” is “none” and “Failure address” is non-zero, then TS is too short and there was a real-time overflow. Please record the entire string of CD and call your service center for technical support.
51	Empty Program	LS is an error if no program has been loaded to memory – nothing to list.
53	Only For Current. Command is applicable only in torque control modes UM=1 or 3.	TC=2 (Set torque to 2A) is an error in UM=2, since in this mode the torque command is set automatically by the controller so as to achieve the desired speed.

Error Code	Description/Example	Example/Remedy
54	Bad Database. Cannot start the motor, as the setup data is not consistent.	If CA[4]==CA[5] then two physical Hall sensors are defined to be the same logical sensor. This inconsistency prevents powering the motor.
55	Bad Context. A command that is not applicable in the present context has been attempted.	This error is caused by privileged commands that are used in auto-setup sessions.
56	RM Must be Zero. A software reference has been attempted in modes where the reference signal is composed only from analog inputs.	For UM=1;RM=1, TC=1 is an error since the current command is measured at the analog input and not taken from the software.
57	Motor Must be Off. This command cannot be used when the motor is ON.	CA[25]=1 sets the order of firing the motor phases, thus controlling the motor direction. This parameter can't be set while the motor is ON, since it will immediately de-stabilize the feedback loop.
58	Motor Must be On. This command cannot be used when the motor is OFF.	PA=1000 is an error If MO=0. The absolute position reference is automatically set to the present position at MO=1, so that setting PA at MO=0 is pointless.
60	Bad Unit Mode Attempt to do something that is not supported in this unit mode.	PT[4]=5 is an error in UM=1 as PT motion requires position control.
61	Data Base Reset. The database has restored to factory defaults after the parameters loaded from the Flash memory failed a consistency check.	This error may occur after upgrading the Metronome version, if the newer version uses a different database structure.
62	Stepper mode only. This command is supported in UM=3 only.	SA=0 at UM=2 is an error, since SA sets manually the field angle, and in all modes but UM=3 the field angle is set automatically.
65	Disabled By SW. Motion could not start since a switch programmed to abort motion was active when MO=1 was tried.	Check the IL[N] switch definition settings and compare to the actual switch reading (use the IP command)
66	Servo drive Not Ready. Motor could not be powered because of: Over or under voltage Over temperature Short circuit (A shorted motor or a hardware problem) Hall sensors problem	Check the servo drive status (SR command)
67	Recorder Is Busy. A recording process is on, and the recorder setting can't be changed. Recorded data cannot be fetched.	Let the recorder finish its job, or use the command RR=0 to kill the recording process.

Error Code	Description/Example	Example/Remedy
69	Recorder Usage Error. Something illegal attempted with the recorder.	RC=2;RR=2 and later BH=1 is an error, since an attempt is made to bring a vector that was not recorded.
70	Recorder data Invalid. Cannot upload recorded data since the recorder contains no valid data.	Reasons: Recorder settings (like RC=n) have been changed since the last records has been made The recorder has not been operated at all since power-up.
72	Must Be Even. Only even numbers are valid for this value.	XM=5 is an error since the modulo-count of the shaft position must be even.
73	Please Set Position. An attempt to set the position counts modulo to a smaller number then the present position reading.	PX=2000; XM=500 is an error since if XM is 500, the position reading of 2000 will become invalid.
75	Can't Get Offsets Motor could not be started since the offsets of the motor current sensors could not be measured.	Reasons: The motor is not connected to its terminals, or is open circuited. Hardware problem
76	Program Reset. The user program could not be loaded from the flash memory	No program in the flash memory. The program in the flash could not be compiled. This may happen only if the stored program has been compiled by an earlier firmware version.
77	Buffer Too Large. A too long string command has been sent. (More then 255 characters in one command)	Check command syntax
78	Out of Program Range. An attempt to load a program that is larger from the storage capabilities of the Metronome.	
82	Program Is Running. Cannot load a new program, compile a program, or start program execution.	Wait until program is done, or use the HP or the KL commands to stop the program.
84	The System Is Not In Point To Point Mode.	A PR (position relative) cannot be set when not in the PTP mode, since it has no reference position to start from.
85	Must resume from a stop. The system has been stopped by the abort switch, or by a limit switch. The RI command must be used to release the emergency trap and let the system go.	
87	RI applied before motor has come to a complete stop.	

Error Code	Description/Example	Example/Remedy
90	Gain Scheduling not ready. An attempt made to program a controller while the gain-scheduling table of another controller is programmed.	KP[1]=x when GS[2]=5 will return this error.

Table 37 – Processing Errors

12.2 Metronome Timing

The Metronome accepts commands from up to three sources simultaneously.
The command sources are:

- The RS232/RS485 communication channel
- The CANopen communication channel
- The user program

The Metronome scans all the three command sources repeatedly, as described by the flow diagram below.

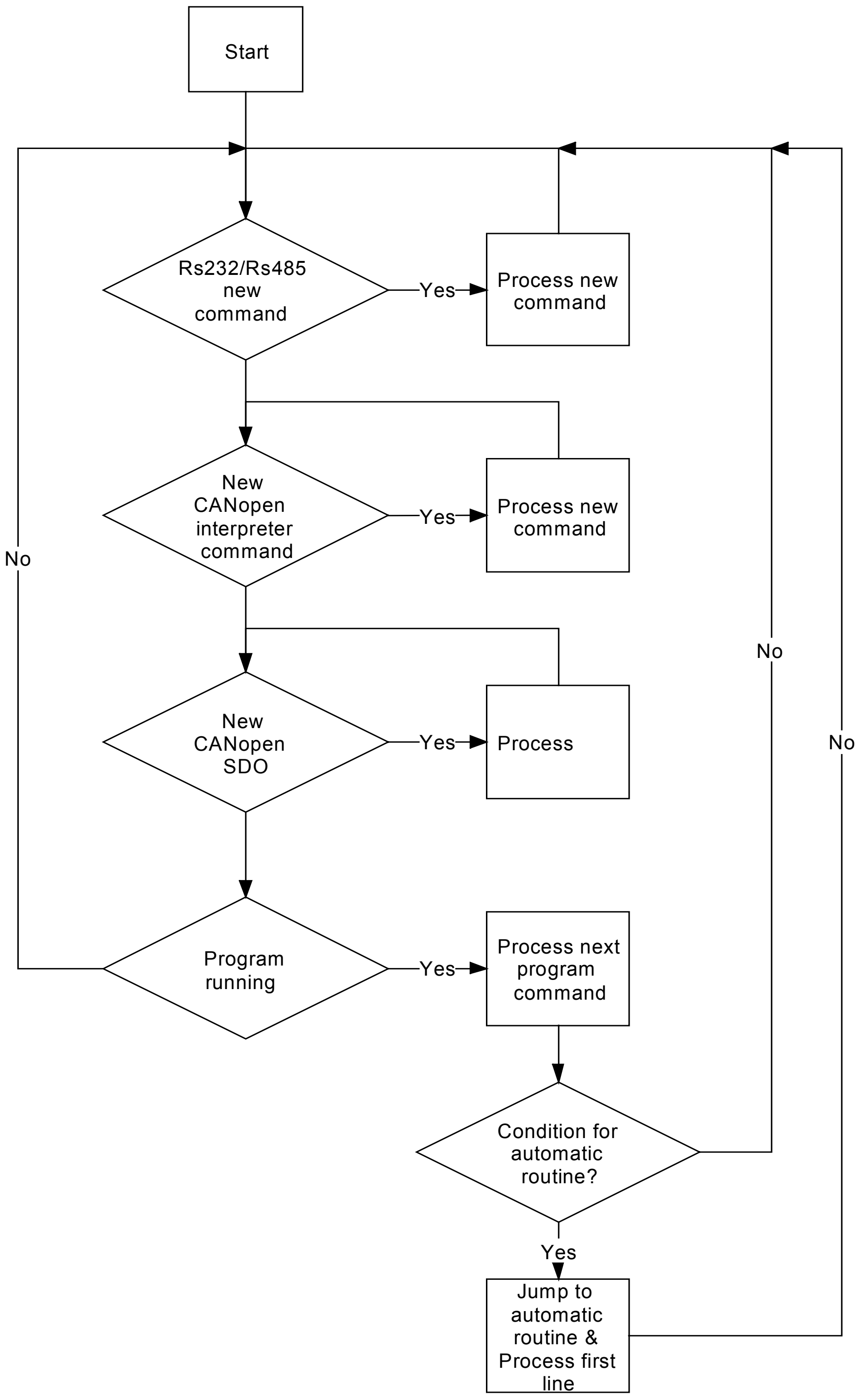

Start
Rs232/Rs485 new command
Yes
Process new command
New CANopen interpreter command
Yes
Process new command
New CANopen SDO
Yes
Process
Program running
Yes
Process next program command
No
Condition for automatic routine?
No
Yes
Jump to automatic routine & Process first line
No

Figure 17 – The Scan Loop

Notes:
Program commands that suspend program execution do not stop the scan, and does not prevent the response to communication commands.
All the commands return to the scan loop immediately. For example, the recorder data upload command (BH) may take seconds to complete. Upon accepting the BH command, the interpreter merely initiates the upload process. It than carries on the scan loop, with the upload process continuing in the background.
The typical scan time of the Metronome is 1msec to 2msec. The scan time depends in the load of the 3 input channels, and in the commands used. Most of the commands have very short processing times. Commands that recalculate the database, like MO=1, may however consume tens of milliseconds, and commands that store data in the Flash may consume hundreds of milliseconds. In general, longer controller sampling times (TS) lead to faster program execution.

The above scheme gives no priority to any communication channel. Some CANopen commands are, however, privileged and treated in the real time process, without being queued for the process

12.3 Loading New Software Version

New firmware may be loaded to the Metronome via the communication lines. Loading new firmware replaces all the software excluding the Flash boot sector[11].
The firmware comes as a text file, formatted as standard S-records. The firmware file can be loaded using the Composer program.
If loading does not complete successfully, the Metronome wakes up in a state which downloading firmware is the only possible operation.
After loading new firmware, the Metronome must be reset by powering it of and on again.

After the Metronome is rebooted with the new firmware, it tests the parameters and the user program that are stored in the flash memory.
If the parameters in the flash can be recognized, and they all have reasonable values, then they are retained.
Otherwise, the parameters in the flash are discarded, and the new firmware automatically loads the reset (RS) parameter values. After the parameters are reset, it is necessary to setup and tune the Metronome from the beginning. The factory default will not allow the starting of the motor, before at least the current limits are set.

If there is a user program in the flash and that program can be recognized and compiled, it is retained. Otherwise the program is lost.

[11] The boot sector includes only the software required to download new firmware.