HARmonica
Software Manual

June 2002

Elmo Motion Control

www.elmomc.com

Important Notice

This guide is delivered subject to the following conditions and restrictions:

- This guide contains proprietary information belonging to Elmo Motion Control Ltd. Such information is supplied solely for the purpose of assisting users of the HARmonica servo amplifier.
- The text and graphics included in this manual are for the purpose of illustration and reference only. The specifications on which they are based are subject to change without notice.
- Information in this document is subject to change without notice. Corporate and individual names and data used in examples herein are fictitious unless otherwise noted.

Doc. no. HARSFEN0602

The Elmo Motion Control warranty for this product covers only the elimination of problems due to manufacturing defects that result in impaired function. Specifically excluded from this warranty is the elimination of problems caused by abuse, damage, neglect, overloading, wrongful operation and unauthorized manipulations of the product. *This product warranty is for a period of 12 months from the time of operational startup but no longer than 18 months from shipment from the manufacturing plant.* Damage claims that exceed the warranty obligation – including consequential damages – will be rejected in all cases. If any term or condition in this warranty is at variance or inconsistent with any provision or condition (whether general or special) contained in or referred to in the Terms of Conditions of Sale outlined at the back of Elmo's Standard Acknowledgement form, then the latter shall prevail and be effective.

Elmo Motion Control Ltd.
64 Gisin St., P.O. Box 463
Petach Tikva
49103
Israel

Tel: +972 3 929-2300
Fax: +972 3 929-2322

Elmo Motion Control Inc.
900(H) River St.
Kennedy Industrial Park
Windsor, CT 06095
USA

Tel: +1 860 683-0095
Fax: +1 860 683-0036

Elmo Motion Control GmbH
Steinbeisstrasse 41
D-78056
Villingen-Schwenningen
Germany

Tel: +49 07720 8577-60
Fax: +49 07720 8577-70

Table of Contents:

List of Tables:

List of Figures:

1 About This Manual

1.1 Scope

This specification presents relevant data for understanding and using the Harmonica software.
The Software Specification complements the Command Reference Manual. The Command Reference Manual lists each single command with all its options. The Software Specification Manual is designed to give wider view of the Harmonica system. The topics are covered functionally, with reference to many commands.

1.2 Relevant documentation

Document	Comments
The Harmonica Command Reference Manual	A complementary document. The Command Reference Manual lists the system variables and interpreter commands of the Harmonica.
The Harmonica CAN Manual	A Complementary document. The CAN Manual explains the CAN communication objects of the Harmonica, and their usage.
The Harmonica User Manual	A Complementary document. This document describes how to select the right Harmonica model, how to install it, and how to perform initial setup
The Composer Manual	This manual explains the Composer IDE and how to best use it with the Harmonica.

1.2.1 Glossary

The following abbreviations and terms are used in this document:

ALU	Arithmetic Logic Unit, the part of the CPU that makes math and address calculations.
ICD	Interface control document. A definition for the external world relations of a product. For software, the ICD defines external I/O relations.
CSCI	Computer Software Configuration Item. A stand alone program that as long as it conforms its ICD, may be freely maintained.
CSU	Computer Software Unit. A functional block in the software.
DSP	Digital Signal Processor.
EDS	The list of CAN object supported by a device, in a standard form suitable for standard configuration software.
ID	Direct current. The current component that generate magnetic field in parallel to the fixed magnet of the rotor
IDE	Integrated development environment
IQ	Quadrature current. The current component that generate magnetic field perpendicular to the fixed magnet of the rotor
PDO	Process Data Object. A CAN message type. A PDO avoids the need to allocate data payload for object addressing by a pre-agreement (called PDO mapping) about the message contents.
PPR	Pulse Per Revolution

2 The Harmonica

2.1 Introduction

The Harmonica a sophisticated and capable network-oriented single-axis amplifier, with:

- State of the art control algorithms, including high order filters and gain scheduling
- State of the art reference generation algorithm, including absolute timed interpolated motion, auxiliary signal following, and ECAM.
- Synchronization capability for network operation
- Conformance to CANopen standards
- User friendly programming
- Advanced analyzing tool for setup behavior
- Built in auto-tuning facilities
- Built in database maintenance tools.
- Built in firmware maintenance tools.

All these are implemented in the extremely small environment of the DSP.

2.2 Software Organization

The DSP software is divided into three parts.
The first part is the boot software. The boot software is permanently burnt into the code flash internal to the DSP, and cannot be upgraded during product life. The boot software includes some data that helps the firmware identify the exact amplifier model it is operating. The data includes the maximal motor phase current, the nominal bus voltage, the hardware of the communication sensors and I/O interface, and the grade of the amplifier (basic or with extended capabilities).
The second part is the operational software (firmware). The firmware may be updated at the user site if some upgrading or modifications are required.
The third part is a supportive database that must be loaded to the serial flash. This supportive part provides a file system for personality description, storage of application database, and factory or user provided programs.

2.2.1 The Boot software

The Boot software has the following functions:

- Initialize some of the registers of the DSP
- Test code validity automatically. If code validation fails, pass automatically to the "Download Firmware" mode.
- Degenerated RS232 communication handling and interpreter, only in the level required for firmware downloading functions.
- Support firmware downloads into the on-chip flash memory.
- Pass control to start of firmware.

2.2.2 The Firmware

The firmware implements all the other software functions, as described in this manual and in the Command Reference manual.
The firmware will pass control to the boot software when a DF command initiates a firmware version upgrade. At the end of the firmware downloading process, the Harmonica reboots.

2.2.3 The Personality

The personality data is loaded to the serial flash memory. The personality data includes a file allocation table and several files. The personality files include data about the Harmonica. A partial list is:
List of supported command
List of error code
CAN EDS
All these data items enable an IDE to deal with the Harmonica. The File Allocation table reserves place for storage of application parameters and of user programs. The personality data is burnt into the serial flash using the firmware software.
The firmware can boot without personality data, but it will not become fully functional before the personality data is programmed in place.
Full explanations of the personality data are given in the chapter "Appendix A: The Harmonica Flash Memory Organization".

2.3 Related Software

The Harmonica requires supporting software for setup, tuning, programming, and performance assessment. The support software is called "Composer". It runs on a PC computer, under Windows.
The Composer includes the following functions (among many others):

- Terminal for direct user interface by RS232 or by CAN
- A recorder with advanced scope controls. You can observe up to 8 signals simultaneously, triggered by a selection of events.
- Setup and tuning tools:
  - Menus for entering basic application data and limits
  - Tools for associating functions to I/O connector pins
  - Automatic tuning of current controller
  - Automatic tuning for commutation
  - Manual and automatic Speed controller tuning
  - Manual and automatic Position controller tuning
- Application database maintenance
  - Save and load application database
  - Edit application parameters, with help
- Advanced IDE for user program development:
  - Editor
  - Compiler
  - Downloading and uploading of user programs
  - Debugger, with:
    - Various break point and stepping options
    - Watches for local and global variables
    - Call stack watch

The Composer software reads the personality data from the Harmonica, and can therefore adapt to the specific amplifier model you bought.

2.4 Units

This section describes how the Harmonica measures time, position, speed, voltage, and current.

2.4.1 Position units

The Harmonica refers to position by sensor counts. The sensor counts may be related to physical units using the following commands:

Command	Description
CA[18]	For rotary motors, CA[18] is the number of sensor counts per one full

	mechanical revolution.
CA[23]	For linear motors, CA[23] stores the number of counts per user unit (Meter, or any other unit the user may select). CA[23] is stored only for convenience – the Harmonica software does not use this number for any internal calculation. For rotary motors, set CA[23]=0.
YA[1],YA[3]	YA[1] is the auxiliary feedback resolution, counts/physical unit. YA[3] sets what a physical unit is: Revolution, Meter, or other. YA[1] and YA[3] are stored only for convenience – the Harmonica software does not use these numbers for any internal calculation.

2.4.2 Speed and acceleration units

The speed units are counts/sec, and the acceleration units are counts/sec^2. The speed units may be related to physical units by converting the counts to revolutions, Meters, etc, as explained in the section "Position units" above.

2.4.3 Current and torque

Currents are measured in Amperes. That is not so simple as it sounds, since there is no single accepted method to specify the current of three-phased motors.
For sinusoidal motors, RMS phase current normally specifies the motor current. The RMS is taken over a mechanical revolution; so that the phase currents are the "Motor current" only if the motor revolves with constant speed.
For trapeze-wound motors, the conventional six-steps drive leaves one motor phase open circuited, and only one current flows through the two driven motor phases. This driven phases current specifies the "Motor current". For trapeze-wound motor running six-step commutation continuously at 1Amp, the RMS current is 0.92Amp.
The Harmonica has one motor current definition, although it can run equally well sinusoidal, Trapezoidal, or free form motor windings.
We defined the motor current as the maximum winding current in a mechanical revolution. This definition is consistent with the traditional current definition for six-step motors and it readily extend to other winding forms.
The user of sinusoidal motors must multiply the motor current reported by the Harmonica by the factor of 0.71 to obtain the RMS phase current.

2.5 Internal Units and Conversions

In order to optimize the use of its CPU, the Harmonica works inside with local units for time, for current, for DC bus voltage, and for electrical angle.

- The user normally thinks of time in terms of seconds. Most of the control algorithms view time by counting controller sampling-times.
- The user thinks of currents in terms of Amperes. The Harmonica thinks internally in the terms of A/D bits.
- The user may think of electrical angles in terms of degrees or radians. The Harmonica divides the electrical cycle to 1024.

Normally the internal representation of time and currents is transparent to the user. For example, the user demands motor current in Amperes, and accepts motor current reports in Amperes as well. The Harmonica does the translation. The following situations, however, require the user to consider the internal data representation.

- Uploading data from the real time recorder - Refer "The Recorder" Chapter.
- Interpreting CAN mapped synchronous PDO's
- Specifying motions in the micro stepping mode.

Some user interface commands retrieve the conversion factors for user convenience.

2.5.1 Current and torque:

The internal representation of phase currents is by A/D bits.
Torque commands, and the active and the reactive parts of the current have another representation, since the coordinate transformations from phase current to torque introduce scaling.
The relation between internal phase currents, internal torque, and Amperes is given by the following commands:

Command	Description
MC	The full current range of the amplifier. The amplifier represents MC Amperes by 14000 bits internally. (The maximum A/D reading is 16384 and some spare is left for overshoots and over-current detection)
WS[22]	Torque command multiplier. Internal torque command = WS[22]×DV[1].
WS[5]	Phase current multiplier Internal phase current = WS[5]×Phase Current[Amp]

2.5.2 Speed:

Speeds are represented internally by the units of counts/(2^{16} speed controller sampling times).
For example, if the sampling time of the speed controller is 200usec (5000Hz sampling frequency), then 1000 counts/sec will be represented internally as $\frac{1000 \cdot 2^{16}}{5000\text{Hz}} = 13107$.
The relation between speeds and internal commands is given by the following commands

Command	Description
TS	Sampling time of the current controller. The sampling time of the sped controller is 4×TS.
WS[53]	Conversion from internal units to counts/sec Speed (counts/sec) = WS[53]×Speed (Internal representation)

2.5.3 Electrical angle:

The electrical angle is measured by 1024 counts per electrical revolution.
Think of a motor with 3 pole pairs. To rotate this motor a full mechanical revolution in the stepper mode (UM=3) you have to specify the movement of (3×1024)=3072 position counts.

2.5.4 Power DC voltage

Power DC voltage is represented internally in A/D units.
The relation between the internal representation and the bus voltage is given by:

Command	Description
WS[54]	Conversion from internal units to Volts. Power DC voltage = WS[54]×Power DC voltage (Internal representation)

2.6 Peripherals

This section summarizes the set of Harmonica peripherals.

2.6.1 Position decoders

There are two position decoders, termed main and auxiliary. The main and the auxiliary decoders are similar. Both decoders are timed (through the timer sets A and B) for accurate speed information.
A position decoder measures quadrature or Pulse/Direction. The maximum counting rate of

a decoder is 20mHz, without input filter.
If input filter is applied, the maximum pulse rate is reduced, as explained at the EF[N] command in the Command Reference Manual.
The encoder input is not protected. There is no hardware to identify illegal transitions. Exceeding the maximal pulse rate will cause loss of counts that cannot be detected.

2.7 A/D converter

The A/D samples the following:

Signal	Description
Ia,Ib,Ic	The three phase currents, sampled simultaneously
Ain,ref	The analog input and the reference voltage are sampled simultaneously to form a differential measurement
Bus voltage	The bus voltage is sampled to correct the current loop gain.

Table 2-1: Analog sampled signals

The resolution of all the measurements is 12bit, and in practice the last bit is noisy.
The motor currents are measured offset-free, because of a special measurement mechanism.
The analog inputs cannot avoid an offset, due to electronic inaccuracies in the Harmonica circuits. This offset can be corrected using the AS[1] parameter, to the resolution of about 5mv.
The ability to correct offsets by AS[1] is limited to the resolution of 5mv, and about 10mv in practice. This means that, for example, if AG[2]=10000, the offset correction quality to speed analog reference will be limited to about 100count/sec.

2.8 Digital inputs

The Harmonica has six digital input connector pins. All the input pins are routed to a digital input port. In addition two pins (5 and 6) are routed to high speed capturing input for main and auxiliary homing. You can associate special functions to digital input pins, like Enable, Stop, RLS, and FLS – refer the IL command in the Command Reference Manual.
The digital input response time is limited by the speed of the optical couplers, and also by the input filters.
The encoder index and home input are filtered similarly to the position decoders. The timing of the position decoder filters is explained in the EF[N] documentation in the command reference.
The other digital inputs are filtered in software only. The timing of software filtering is explained in the IF[N] documentation in the command reference.
The use of the digital inputs is detailed in the chapter on Sensors, I/O, and Events.

2.9 Digital output

The Harmonica has two digital output connector pins.
These two pins can be used for non-committed digital outputs, or programmed by the OL command for special functions, like activating external brakes.

3 Communication With the Host

3.1 General

3.2 RS232 Communications

The Harmonica can communicate by RS232 with baud rates up to 115200baud/sec.
The maximum baud rate depends in the sampling time. The baud rate of 115200baud/sec works only when TS equals or is less than 70. The communication rate of 57600 baud/sec works with all sampling time options.
Please refer to PP[] command in the Command Reference Manual.

3.2.1 RS232 Basics

The RS232 method can only serve to communicate a host to a single amplifier. The RS232 lines are full duplex – they can carry bidirectional communications. This means that the host can transmit to the amplifier whenever it finds it right, without considering the state of the amplifier.
The RS232 communication consists of ASCII printable characters only, with some exceptions:

- The characters 0xD (carriage return).
- Some non-printable characters that are used as error codes. Please refer to EC command in the Command Reference.

The basic syntax for RS232 commands may be an assignment,
<Command mnemonic>{[<index>]}{<Equal sign><Value>}<Terminator>
Or a free evaluation:
<Value><Terminator>
Where

Item	Description
Command mnemonic	Two letters (case insensitive) assigned to a command. Please see the Command Reference manual for a complete list of command mnemonics.
Index	The index, if the mnemonic refers to a vector 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) The parameter value may be any legitimate arithmetic or functional expression, as explained later in this section.
Terminator	<CR>[1] or ';'.

Table 3-1 – RS232 Rx Item Description

An assignment evaluates an expression and stores the result in a variable.
A free evaluation evaluates an expression and sends the result to the terminal.
Typical examples are:

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

MO=1<CR>

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

Sets the value of 1 to the MO variable.
CA[2]=1;
Sets the value of the CA[2] variable. CA[N] denotes a vector of parameters that can be accessed by their index.

The free evaluation
(5+sin(PX))*sqrt(abs(VX))
Returns a numerical value to the terminal.
More detail about text interpreting is given in the chapter "The Interpreter Language".

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

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

Table 3-2 – RS232 Tx Item Description

3.2.2 The Echo

When using RS232 each character that is received by the amplifier is echoed back to the host. The echo is immediate, per each received character.

3.2.3 Background Transmission

When the host enters the BH=n command to the amplifier, the amplifier uploads the recorder data to the host. The uploading process may take few seconds. At the time the amplifier 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 amplifier will start to transmit the recorder data immediately. Few milliseconds later, while the recorder data is still transmitting, the amplifier will execute the MO=0 command. The amplifier will store the response message to the MO=0 command, in order to transmit it later, immediately after the records upload terminates.
Note:
If the host does not know the communication parameters, it may transmit a series of terminators and try several baud rates, until it can receive a matching sequence of echoes

3.2.4 Errors and exceptions in RS232

If an error is intercepted (Overrun, noise, parity, framing), then the entire message including this error is discarded, and answered by the "communication error" code response.
Although the communication is defined as 8 bits per character, only the byte values 0-127 are used.
The characters 127-255 are treated as UART errors.
Empty strings with terminators are echoed back, but otherwise ignored.

3.3 CANopen Communication

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

4 The Interpreter Language

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

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

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

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

The physics and the standards of the RS232, and CANopen communication methods imply differences in the command syntax used by each method.
This section describes the Amplifier language in the basic “RS232” or CAN “OS” syntax.

The CANopen communication method can access simple numeric interpreter get and set commands very efficiently. The CAN binary interpreter uses PDO objects to issue interpreter commands and to collect the response.
This is the most economical way minimizes both the communication load and the Amplifier CPU load.
The CAN OS (command prompt) method can be used to access the entire set of interpreter services, including those inaccessible by the binary CAN interpreter, using a text format.
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 interpreter syntax, with extension that are necessary to support program flow instructions and in line documentation.

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

4.1 The command line

The Interpreter evaluates input strings, called “expressions”. An expression is a sequence of characters, terminated by a semicolon ‘;’, a line feed, or a carriage return.
A command line may include the “Comment marker” – two consecutive asterisks. All the text from the comment marker to the next line feed or carriage return is ignored.
The maximal expression length is 512 symbols.

The comment marker feature enables the preparation of documented batch files, to be later sent directly to the amplifier.

Example:
a = 3 ; b = 2 ; c = a + b ** Ignore this text

This line consists of three expressions separated with semicolon. The comment “Ignore this text” is ignored.

4.2 Expressions And Operators

The Amplifier 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 and the expression syntax rules.

4.2.1 Numbers

The Amplifier has two number types: 32 bit integers and 32 bit floating point numbers. As text inputs, numbers that contain a decimal point or an exponent notation are interpreted as floats. Other numbers are interpreted as integers.

The range for integers is [-2.1475e+009,+ 2.1475e+009].
Positive integers may be written as decimal or as hexadecimal.
The hexadecimal notation 0x10 is equivalent to the decimal number 16.

A 32-bit floating-point number has 24 significant mantissa bits and 8 bits of exponent.
The range for floating point numbers is [-1e20, 1e20].
A floating-point number may be written with or without an exponent.
2.5e4 is equivalent to 25000.0. It is not equivalent to 25000, since the latter number is interpreted as integer.

If an integer number overflows integer range, it is interpreted as an error.
If a floating-point number overflows floating-point range, it is interpreted as an error.

Logical operators yield 0 or 1 as a result. The results of logical operators are integers.

4.2.2 Mathematical And Logical Operators

The following table lists the mathematical and logical operators of the language. The table also specifies the operator precedence, which is discussed in the next paragraphs. The larger number has the higher precedence.

Operator	Description	Precedence

+	Add two operands.	9
-	Subtract the right operand from the left one.	9
*	Multiply two operands.	10
/	Divide the left operand by the right one.	10
%	Remainder after division two integers	10
^	XOR	1
~	Bitwise NOT of an operand.	11
&	Bitwise AND between two operands.	5
|	Bitwise OR between two operands.	4
<<	Bitwise Shift left	8
>>	Bitwise Shift right	8
==	Logical equality	6
!=	Logic not equal	6
<	Logic smaller than	7
<=	Logic smaller or equal	7
>	Logic greater than	7
>=	Logic greater or equal	7
&&	Logical AND	3
||	Logical OR	2
!	Logical negation	11
-	Unary minus	11
=	Assignment	
()	Parentheses, for expression nesting and function calls	
[]	Square parenthesis, for array indices and multiple value function returns	

Table 4-1 – Mathematical and Logical Operators

4.2.3 General rules for operators

4.2.3.1 Promotion to float and truncation to integer

Most the arithmetic operators work on both integers and floats. An arithmetic operation between integers will yield integers, an operation between floating point numbers, or between an integer and a floating-point number, will yield a floating-point result. For example, all the expressions below are legitimate:
1+2 (The result is 3, integer)
1+0x10 (The result is 17, integer) Note that 0x10 is treated as a standard integer.
1+2.0 (The result is 3.0, float)
2.1+3.4 (The result is 5.5, float)
The operation of division between two integers may yield a floating-point result in a case of division with remainder. For example:
8/2 (The result is 4, integer)
9/2 (The result is 4.5, float)
If a result of multiplication between two integers overflows integer range [-2.1475e+009,+2.1475e+009], this result will be converted into floating point, but won't be truncated. For example:
100000 * 100000 (The result is 10000000000, float)
Bit operators require an integer input. Floating-point inputs to bit operators are truncated to integers.
7.9 & 3.4 is equivalent to 7 & 3, since before applying the operator & (Bitwise AND) the floating point number 7.9 is truncated to the integer 7, and 3.4 is truncated to the integer 3..

4.2.4 Operator details

The following paragraphs describe the operators in detail.

4.2.4.1 Addition

Notation	+
Description	Arithmetic addition
Number of arguments	2
Output type	According to the section Promotion to float and truncation to integer
Example	4+5=9 3.45+2.78=6.23

4.2.4.2 Subtraction

Notation	-
Description	Arithmetic subtraction
Number of arguments	2
Output type	According to the section Promotion to float and truncation to integer
Example	4-5=-1 3.45-2.78=0.67

4.2.4.3 Multiplication

Notation	*
Description	Arithmetic multiplication
Number of arguments	2
Output type	According to the section Promotion to float and truncation to integer
Example	PA=PA*2 doubles PA 5*4=20 1.5*2=3.0

4.2.4.4 Division

Notation	/
Description	Arithmetic division
Number of arguments	2
Output type	According to the section Promotion to float and truncation to integer
Example	20/4=5 3/1.5=2.0

4.2.4.5 Remainder

Notation	%
Description	Remainder after division two integers
Number of arguments	2
Output type	According to the section Promotion to float and truncation to integer
Example	20%4=0 5%2=1

4.2.4.6 XOR

Notation	^
Description	Exclusive or, bitwise.
Number of arguments	2

Output type	32 bit long integer.
Example	if x=1 as x^3=2

4.2.4.7 Bitwise NOT

Notation	~
Description	Bitwise NOT
Number of arguments	1
Output type	32 bit long integer.
Example	~3 is 0xfffffffc, which is actually –4 ~3.2 is the same as !3

4.2.4.8 Bitwise OR

Notation	|
Description	Bitwise OR
Number of arguments	2
Output type	32 bit long integer.
Example	PA = 0x2 | 0x5 is equivalent to PA = 7 PA = 0x2 | 5.1 is the same

4.2.4.9 Bitwise AND

Notation	&
Description	Bitwise AND
Number of arguments	2
Output type	32 bit long integer.
Example	PA = 0x7 & 0x3 is equivalent to PA = 3 PA = 0x7 & 3.1 is the same

4.2.4.10 Logical Equality

Notation	==
Description	Logical equality
Number of arguments	2
Output type	0 or 1 (0 equals to false, 1 equals to true)
Example	if x=3 and y=3 as x==y yields 1 if x=3 and y=5 as x==y yields 0

4.2.4.11 Logical Inequality

Notation	!=
Description	Logical inequality
Number of arguments	2
Output type	0 or 1 (0 equals to false, 1 equals to true)
Example	if x=3 and y=3 as x!=y yields 0 if x=3 and y=5 as x!=y yields 1

4.2.4.12 Logical Greater than

Notation	>
Description	Logical greater than
Number of arguments	2
Output type	0 or 1 (0 equals to false, 1 equals to true)
Example	if x=3 and y=3 as x>y yields 0 if x=3 and y=2 as x>y yields 1 if x=1 and y=2 as x>y yields 0

4.2.4.13 Logical Greater than or equal

Notation	>=
Description	Logical greater than or equal
Number of arguments	2
Output type	0 or 1 (0 equals to false, 1 equals to true)
Example	if x=3 and y=3 as x>=y yields 1 if x=3 and y=2 as x>=y yields 1 if x=1 and y=2 as x>=y yields 0

4.2.4.14 Logical Less than

Notation	<
Description	Logical less than
Number of arguments	2
Output type	0 or 1 (0 equals to false, 1 equals to true)
Example	if x=3 and y=3 as x<y yields 0 if x=3 and y=2 as x<y yields 0 if x=1 and y=2 as x<y yields 1

4.2.4.15 Logical Greater than or equal

Notation	<=
Description	Logical greater than or equal
Number of arguments	2
Output type	0 or 1 (0 equals to false, 1 equals to true)
Example	if x=3 and y=3 as x<=y yields 1 if x=3 and y=2 as x<=y yields 0 if x=1 and y=2 as x<=y yields 1

4.2.4.16 Logical AND

Notation	&&
Description	Logical AND. The result is 1 if both arguments are nonzero, 0 if any is zero. The arguments are not truncated to integers before evaluations
Number of arguments	2
Output type	0 or 1
Example	1 && 5 yields 1 0.21 && 2 yields 1 0 && 2 yields 0

4.2.4.17 Logical OR

Notation	||
Description	Logical OR. The result is 1 if any of the arguments is nonzero, 0 if both are zero. The arguments are not truncated to integers before evaluations
Number of arguments	2
Output type	0 or 1
Example	1 || 0 yield 1 0 || 0 is 0.

4.2.4.18 Logical NOT

Notation	!
Description	Logical NOT. The result is 1 if the argument is zero, and 0 else The arguments are not truncated to integers before evaluations
Number of arguments	1
Output type	0 or 1
Example	!4 yields 0 !0 yields 1 !0.0004 yields 1

4.2.4.19 Unary Minus

Notation	-
Description	Unary MINUS. The result is negative if the argument is positive, and vise versa. The arguments are not truncated to integers before evaluations
Number of arguments	1
Output type	The same as argument
Example	-4.5 yields –4.5 -4 yields –4 (-4) yields 4 -5+5 yields 0

4.2.4.20 Bitwise Left Shift and Right Shift operators

Notation	<< or >>
Description	The bitwise shift operators shift their first operand left (<<) or right (>>) by the number of positions the second operand specifies. The arguments are truncated to integers before evaluations.
Number of arguments	2
Output type	32 bit long integer.
Example	8>>2 yields 2 8<<2 yields 32

4.2.5 Mathematical functions

The following table presents the mathematical built-in functions of the language.
Function names are case sensitive.

Operator	**Description**	**Returns**

sin	Sine	Floating point
cos	Cosine	Floating point
abs	Absolute value	Same type as input argument
sqrt	Square root, or zero if argument is negative	Floating point
fix	Truncate to integer. fix(3.8) is 3 fix(-3.8) is -3	Integer
rnd	Truncate to nearest integer. rnd(3.8) is 4 rnd(-3.8) is –4	Integer
tdif	Time difference x=TM tdif(x) returns the time in msec since x=TM has been sampled.	Floating point

4.2.6 Expressions

An expression is a combination of operands (parameters) and operators that is evaluated in a single value. Expressions work with immediate numbers, Amplifier commands, Amplifier and global user-program variables.
There are different types of expressions, as described below.

4.2.6.1 Simple Expressions

A simple expression is evaluated in a single value. Any parameter and mathematical/logical operator may be used to create a simple expression. Normally, simple expressions may be used as a part of other types of expressions.
Simple expressions are evaluated according to the priority of the operators, as specified in Table 4-1 – Mathematical and Logical Operators. In case of equal priority, the expression is evaluated from left to right.
The use of parentheses is allowed to 16 nesting levels.

Example:

AC=100000
SP*2/5+AC
SP= SP*2/5+AC
IP|5
(SP+SP)*IA[1+AC*0]
IA[1]|5&2
2+3
1400000

4.2.6.2 Assignment Expressions

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

<Parameter or command name>=<simple expression>

Example:

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

If the variable or the command is a vector, the assignment is allowable only for its single member. The syntax of the vector member assignment is:
<Parameter or command name>[index]=<simple expression>

The index is an index of the relevant member vector. Indices are enumerated from zero.

Example:
ca[1] = 1

4.2.6.3 User variables

User variables are defined within a user program. The description and syntax rules of the variable definition see at User variables of User Programming Language
User program variables may be used within the command line only if a program was compiled successfully and downloaded to the Amplifier.
Through the Interpreter a user may inquire user variable value or change it.
A user has to pay special attention to the scope of a variable. A variable may be defined at the global or local scope. The local variables are available only within the function where they are defined, while the global variables are available within any function and outside a function.
A user variable may be inquired or changed when the program is in the running or halt status.

For example, suppose that a compiled program includes the following lines at the global scope:
int ZEBRA,GIRAFA[3];
float GNU;
Then the expression
GNU=ZEBRA*GIRAFA[1]+2*sin(GIRAFA[2]);
is valid. User program variables are case sensitive.

4.2.6.4 Built-in Function Calls

The built-in function call may be used in a single expression. For a list of built-in-functions refer Mathematical Functions
The built-in function call may be a part of a single expression.

Example:
sin(3.14/2)
AC=abs(DC)
SP=SP+sin(3.14/2)

4.2.6.5 Time functions

Use the TM command to read the system 32 bit microsecond counter.
The time difference from the present time to an older sampling of TM can be taken in two methods:
Example
QP[1] = TM ; ** QP[1] is used just as storage
…. ** Do something
QP[2] = QP[1]-TM ; ** QP[2] is time difference in microseconds.

Example
QP[1] = TM ; ** QP[1] is used just as storage
…. ** Do something
QP[2] = tdif(QP[1]) ; ** QP[2] is time difference in milliseconds.

Time differences cannot be taken longer then 31 minutes.

4.2.6.6 User Function Calls

The XQ command enables a user function call. See Running of a User Program
A user function cannot be called from the command line without the XQ command.

4.3 Comments

Comments are texts that are written into the code to enhance its readability.

A double asterisk marks comments. The comment starts in a double asterisk comment marker, and terminates at the next end of line.
The Amplifier ignores the comments when evaluating an expression.

Example:

mo=1; **motor on

This example demonstrates that a comment may start anywhere in the program line. The MO=1 instruction preceding the comment marker shall be executed.

Remark: Comments in the command line differ from comments in a user program. User Program Language has three methods to write comments, while the Interpreter Language has the only one.

5 The Harmonica User Programming Language

This chapter describes the Harmonica user language in detail.
This chapter is somewhat out of place, since the user program language is not really a feature of the Harmonica.
The Harmonica can only understand virtual assembly commands – see commands at the chapter The Virtual Machines: Short reference.
The Compiler translates user language into virtual assembly commands. It reads user program and transforms it to the sequence of simple virtual assembly commands. The compilation process runs off-line in the PC, not inside the Harmonica. Before the Harmonica executes a user program, it must be preliminarily compiled and the compiled code must be downloaded to the serial flash of the Harmonica. A little more about the Compiler see at the section Compilation
An Amplifier 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 Amplifier.

5.1 User Program Organization

A user program is organized as follows:

- Integer and floating point variable declarations
- Program text, including expressions, commands, labels, and comments
- An **exit** 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 Amplifier 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.

In addition to the interpreter commands, a program may include program flow statements that manage how the program runs.

- Iterations.
- Subroutine execution commands.
- Conditions.

In the program text, semicolons, commas, line feeds or carriage returns separate the commands.
A single program line is executed as a one unit, preventing intervention from the interpreter or a CAN command.
For example in the sequence
UM=5;
MO=1;
It is possible that an interpreter command shall be executed between the execution of the two program statements. If the interpreter statement was UM=2, then MO=1 is specified for a wrong unit mode. The sequence
MO=0;UM=5;MO=1;BG
Guarantees that MO=1 is executed with the correct unit mode.

Example 1:

int x,k;	Variable declarations
##Foo	Label definition

X=0;	Initialize
for k=0:10	Iterate
X = x + 1 ;	Do something
end	End of iteration
exit	End of program directive
##Moo	
...	More code

The program defines two variables named x and k.

##Foo is an entry point.

After compilation, it is possible to run this piece of code by commanding XQ ##Foo to the interpreter.

When the program starts at ##Foo, it clears the user variable x. It then iterates 11 times, incrementing x with every iteration. Finally, x = 11.

The exit command terminates program execution.
Another code section can be executed by commanding XQ ##Moo.

Example 2:

switch (IP & 3)	Select according to the two low input bits
case 1	
PR=1000;	If value is one...
case 2	
PR=500;	If value is two ...
otherwise	
PR=100;	Otherwise...(Last two bits are 0 or 3)
end	
BG	Begin motion

This example moves an axis with a step that depends in the state of the digital inputs.

5.2 Common

A user program contains lines of text code. This text must have defined syntax for the Compiler to recognize it. This section contains some common description of a program text.

5.2.1 Line and Expression Termination

A line has the following terminators: carriage return, line feeds or their combination.
A line can contain a single expression or sequence of expressions. Sequence of expressions in the same line can be separated with semicolon or comma (not inside round or square parentheses).

Example:
a = 3 , b = 2 , c = a + b , ** This line consists of three expressions separated with comma
a = 3 ; b = 2 ; c = a + b ; ** This line consists of three expressions separated with semicolon
a = 3 , b = 2 ; c = a + b ** This line consists of three expressions separated with comma, semicolon and terminated with line feed
[a,b] = foo (23, c, 3.14) ** In this expression comma is not expressions separator, because it is inside parentheses

5.2.2 Line Continuation

A user program may contain too long line and its representation at the screen may be not convenient, because not all symbols are shown at the screen. In order to improve readability of the program, a user may continue this expression at the next line. Three points at the end of line indicates that this line will be continued.

Example:
Assume that a user program contains the next expression
c = 12 * a + sqrt(2) – sin(3.14 / 2) + 7 ^ 3 * …
(6 + b) * 34
Three points at the end of the first line shows that this expression is not finished and is continued at the next line.

5.2.3 Limitations

Every line of a user program text may contain 128 characters as maximal length (for better readability at the screen). For too long text line (more than 128 symbols) the Compiler sets an error.
Expressions also have limitation: the maximal admissible length of expression is 512 symbols, not including comments. If a program contains complex expression that takes some lines and the summary length of expression without comments and three points exceeds 512 symbols, then the Compiler sets an error.

There are some limitations of a user program text depending on the specified Harmonica device. The list of setup parameters of the Harmonica, which are limiting a user program, is represented below.

- Maximal length of user program text
- Maximal number of routines, including functions, labels and auto routines
- Maximal number of variables, including as global as local
- Maximal length of the data segment – space for storage of global variables
- Maximal length of the code segment – space for compiled code
- Maximal depth of the stack – working space of the program

The IDE enables a user to see these parameters (not implemented yet). Ask Ilia and Nicolay

5.3 Expressions And Operators

5.3.1 Numbers

The number syntax is the similar to the Interpreter Language (see Number of the Interpreter Language), but there are some differences in evaluation of immediate numbers. User program is compiled at the PC in off-line mode, so it has more resources, than the Interpreter.
The range for floating point numbers is [-1e38,+1e38], it is more than the Interpreter language numbers have.
If an integer number overflows integer range, it is converted to float (the Interpreter interprets it as an error).
If a floating-point number overflows floating-point range, it is interpreted as an error.

5.3.2 Mathematical and Logical Operators

The description and syntax is the same as in the Interpreter Language. See Operators of the Interpreter Language

5.3.3 General rules for operators

The description and syntax is the same as in the Interpreter Language. See General rules of the Interpreter Language

5.3.4 Operator details

The description and syntax is the same as in the Interpreter Language. See Operator details of the Interpreter Language

5.3.5 Mathematical functions

The description and syntax is the same as in the Interpreter Language. See Mathematical functions of the Interpreter Language

5.3.6 Expressions

5.3.6.1 Simple Expressions

The description and syntax is the same as in the Interpreter Language. See Simple Expressions of the Interpreter Language

5.3.6.2 Assignment Expressions

The description and syntax is the same as in the Interpreter Language. See Assignment Expressions of the Interpreter Language
User programming language allows multiple assignment to multiple output of the function. Syntax and descrition see at the section

5.3.6.3 User variables

User variables are defined within program, using the int or the float declarations.
The keywords for variable types are ‘int’ and ‘float’.
Syntax:
int int_var1, int_var2[12], …, int_varN;
float flt_var1[13], flt_var2, flt_varN;
int a;
float b;

Variable must be declared before its use (in the expression or assignment).
The definition line of variables consists of type name (int or float) and variable names.
Variables at the definition line must be separated with comma or every variable may be declared at the separate line. Variables may be scalar, or vector. If a variable is a vector, it must be declared its dimension in square brackets after its name.
Only global variable may be one-dimensional array. Neither local variable nor input/output argument of a function can be a vector.
Dimension of the vector must be positive constant number. If dimension is defined as floating-point number, it will be truncated to integer. The dimension that is less than one is illegal.
For example,
int arr[12.5] ;
The floating-point number that defines dimension will be truncated to 12.
int arr[-2] ;
This variable definition is illegal, because dimension is negative.

Local variables must be defined at the beginning of function bodies. Any local or global variable definition after some executable code of the function is illegal.
For example,
function foo (int a) ** Function definition
int b ; ** Local variable definition
b = a ; ** Executable code
float c ; ** Local variable definition

Definition of the variable ‘c’ is illegal, because it is after some executable code.

The names of variables may include ASCII letters, digits (not leading) and underscores (not leading) only.
It's case sensitive. The maximal variable name length is 12 characters.
A variable name cannot be a keyword.
The list of the keywords is as follows:

"int",
"float",
"if",
"else",
"elseif",
"for",
"while",
"switch",
"case",
"otherwise",
"break",
"end",
"return",
"function",
"global",
"keyboard",
"exit",
"virtual",
"all",
"reset",
"wait",
"until",
"goto",
"nargout",
"halt"

The all keywords are case sensitive.
Variable name must be distinct from some function or label name.
After a program is compiled, all the program variables may be used within the command line.
For example, assume that a compiled program includes the following lines:
int ZEBRA,GIRAFA[3];
float GNU;
Then the expression
GNU=ZEBRA*GIRAFA[1]+2*sin(GIRAFA[2]);
is valid. Variables are case sensitive.
More about global and local variables see at the sections Automatic Variables and Global Variables

5.3.6.4 System Commands

The Harmonica system commands are described at the Command Reference.
System commands description in this section is designed only to explain their use in user program.

System command has two-letter mnemonic notation (only English letters, case not sensitive).
For example,
ac = 100000
AC = 100000
Both these expressions have the same meaning, in spite of different notation.

Every command has the 16-bit flag, every bit of which defines any feature.
For example, the fourth bit (PostProcess flag) defines whether this command can be used to set value or not.

List of commands depends on the specified Harmonica.

Consider some examples using system commands.

Example	Explanation
a = AC	This expression assigns value of the system command AC to the variable 'a'. This expression is valid if the AC command is allowed to 'get value', i.e. it has PreProcess flag
AC = a	This expression assigns value of the variable 'a' to the system command AC. This expression is valid if the AC command is allowed to 'set value', i.e. it has PostProcess flag
BG	This is executable command. It cannot be assigned, i.e. it has neither PreProcess nor PostProcess nor Assign flags For example, the following expressions BG = 1 or a = BG are not legal
LS	This expression is not legal, because it uses command that has NotProgram flag. The 'NotProgram' commands are not allowed in user program, while they are available for the Interpreter language.

5.3.6.5 Built-in Function Calls

The description and syntax is the same as in the Interpreter Language. See Built-in function call of the Interpreter Language

5.3.6.6 User Function Calls

Functions perform a pre-written and compiled code. Functions get a list of arguments from their caller, and return a list of return values.
Functions get their list of input argument by value – they do not directly modify them.
The general syntax of a function call is
[OUT1,OUT2,…OUTN]=FUNC(IN1,IN2,…IN_N)

If only one value is returned, the square parentheses are not required and the syntax is following:
OUT=FUNC(IN_1,…,IN_N)

Not all the output variables need be assigned.
For example, if the function FUNC returns two values,
[IA[1],IA[2]]=FUNC()
assigns the two returned values to IA[1] and IA[2].
IA[1]=FUNC()
Returns the first return value to IA[1], and the other returned values remain unused.
If the returned value has another type than variable that receives it, the result shall be type cast to the type of the variable.
Numbers of input arguments during function call are strict according to its definition.

If function call is a part of expression, then multiple output of the function is illegal.
For example,
[a, b] = 5 + foo(c)
or
[a, b] = foo(c) + 5
Both these expressions are not legal: function call is a part of expression and must be evaluated in a single value.

5.4 Comments

Comments are texts that are written into the code to enhance its readability. There are three methods to write comments.
A double asterisk marks comments. The comment starts in a double asterisk comment marker, and terminates at the next end of line.
The Amplifier ignores the comments when running a program or evaluating an expression.

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 Amplifier 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.
A percent sign (the MATLAB style) marks comments in the same way as a double asterisk. The comment starts in a percent sign comment marker, and terminates at the next end of line.
Example:

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

In this example a percent sign marker just changes a double asterisk marker from the previous example to show that both of them have the same use.

Another commenting method is the C style. A C style comment starts with the start comment mark /* and terminates with the end comment mark */. The C style enables closing a text in the middle of an expression, or closing several text lines.
Example:

```
/*
This is a multiple line comment.
All this text is ignored.
*/
if ( 1 /* x == 1 */ )
        y = 1;
end
```

The expression y = 1 will be always executed. The x==1 condition enclosed by the comment markers is ignored.

5.5 Program Flow Commands

The Amplifier 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.
The program flow commands are:
while – end : Iterate as long as a condition is satisfied.
until : Iterate (suspend the execution of the program) until a condition is satisfied.

wait : Iterate (suspend the execution of the program) until a specified time is elapsed.
for - **end** : Iterate counted times
break : Break an iteration or a switch expression (**for, while, switch**)
if – **else** – **elseif** – **end**: Conditional expression.
switch-case-otherwise-end: Case selection
goto – Go to some point in the program
reset – Kill the state of the executing program and jump to some point in the program.
function-return – Declare a function and its return point
##: Declare a label or an auto routine
#@: Declare a label or an auto routine
#@ - **return**: Declare an auto routine and its return point
exit – Terminate program execution.

5.5.1 Labels (Entry points)

Labels denote that program execution can start from that place, or that program execution can be branched to that place.
Label definition has the following syntax:
##<LABEL_NAME>
or
#@<LABEL_NAME>
A maximum of 12 characters (letter and/or digit (not leading) and underscore (not leading) only) may be used for label. A name of label must be distinct.
Labels can reside inside or outside function bodies, but not within a program flow structure, such as a **for** iteration.
Labels inside function bodies regard as local of the function and serve as targets for **goto** instructions within the same function.
Labels in the global text scope serve as possible execution starting points, and also as targets for **reset** and global scope **goto.**
The XQ and the XC program launching commands use labels to specify where to start program execution and were to terminate. For example, if user wants to start execution at ##LOOP2, he can do it by sending the command
XQ ##LOOP2

Example:

##START;	The program start
##LOOP1;	A label
….	The body code A
goto##LOOP1;	
##LOOP2;	
…..	The body code B
##LOOP3;	
…	

According to above example, if the program runs from label ##START, the body code A will be performed forever. The ##LOOP2 will never be reached.

5.5.2 For iteration

Perform an indexed iteration in a program.
Syntax:
for k=N1:N2:N3
…
end
or
for k=N1:N2
…
end
where N1, N2, and N3 are numbers or simple expressions.

The syntax:
for k=N1:N2:N3
…
end
iterates k from N1 to N3 with a step of N2.

The syntax:
for k=N1:N2
…
end
iterates k from N1 to N2 with a step of 1.

Notes:
1. If the iteration step is zero, the program is aborted with the error code INFINITE_LOOP.
2. If N1, or N2, or N3 are variables, they are evaluated once before the iteration begins. If the variable changes within the "for" loop, the iteration process will not be affected.
3. Iteration variable k must be declared as a variable.

Example:

….	Start user program or function
float ra[20];	Float array declaration
int ia[20];	Integer array declaration
int k;	Variable declaration
…	
for k=1:10	Start of iteration loop
ia[k]=100;	Update the first ten elements of the integer array
ra[k]=55.55;	Update the first ten elements of the real array
…	
end	End of iteration loop
….	

5.5.3 While iteration

Syntax:
while(*expression*)
…
statement
…
end

The **while** keyword executes *statement* repeatedly until *expression* becomes 0.
The expression can be logical or/and numerical.
Note: expression may be within round parentheses or without ones.

Example:

OB[1]=0	Digital output 1 is OFF
while(IB[1])	
OB[1]=1	While the digital input 1 is ON, the digital output 1 is ON
end	
OB[1]=0;	Digital output 1 is OFF
….	
…	
While (IB[2])	
end	
MO=1;	This command will be performed only after digital input 2 is OFF.

5.5.4 Until iteration

Syntax:
until (*expression*) ;

The **until** keyword suspends the execution of the program until *expression* becomes true (nonzero).
The expression can be logical or/and numerical.

Example:
Assume that user wants to suspend the program until the variable PX exceeds 20000 and digital input 1 is ON.
The code below has the same functionality

…	
until ((PX==2000)&IB[1]) ;	
…	

The **until** expression may be useful to synchronize threads for amplifiers that support multithread programs. For example, we have two threads. Assume that the second thread must start after the first thread finishes some work. We define a global variable, which shows whether the first thread finished or not. The second thread is suspended by until expression.
We first define a variable:

int IsFirstFinished ;	**The global variable definition

The variable is initially set to zero.
The code of the first thread shall be:

…	

… Do some work …	
IsFirstFinished = 1 ;	**Signal that some work is completed

The code of the second thread will include

… Prior to suspension code	
until (IsFirstFinished) ;	**The second thread suspended until **signal
… Continue program.	

If we want to suspend thread without terminating it, we can use **until** with a false expression.

5.5.5 Wait iteration

Syntax:
wait (*expression*) ;

The **wait** keyword suspends the execution of the program until the specified time is elapsed.
The expression may be within round parentheses or without ones.
The expression specifies the waiting time in milliseconds. It can be numerical expression only, which is evaluated in a single value.

Example:

PA=10000	
BG	
until (MS == 0)	
wait (20) ;	Wait 20 milliseconds

The above program instructs a PVT motion. The until (MS == 0) waits until the position command is stabilized at the new position. The wait (20) allows additional 20 milliseconds for final stabilization.

5.5.6 IF condition

Syntax:
if(expression1)
…
statement1
…
elseif (expression2)
…
statement2
…
else
…
statement3
…
end

The **if** keyword executes *statement*1 if *expression*1 is true (nonzero); if **ifelse** is present and *expression*2 is true (nonzero), it executes *statement*2. The **ifelse** keyword may repeat scores of times during to **if** condition. The *statement*3 will be executed only if all *expression*1, *expression*2, … *expression*N are false(or zero).

Example:

if (IB[4])	
PR=1000;	PR=1000 only if digital input 4 is ON
elseif(IB[3])	
PR=5000;	PR=5000 only if digital input 3 is ON
elseif(IB[2])	
PR=3000;	PR=3000 only if digital input 2 is ON
else	
PR=500;	PR=500 only if digital input 2,3 and 4 are OFF
end	

5.5.7 Switch selection

Syntax:
switch(expression)
case (case_expression1)
statement1
case (case_expression2)
statement2
……
otherwise
statement
end

The **switch** statement causes an unconditional jump to one of the statements that is the "switch body," or to the last statement, depending on the value of the controlling expression, the values of the **case** labels, and the presence or absence of an **otherwise** label.
The switch body is normally a compound statement (although this is not a syntactic requirement). Usually, some of the statements in the switch body are labeled with **case** labels or with the **otherwise** label. Labeled statements are not syntactic requirements, but the **switch** statement is meaningless without them. The **otherwise** label can appear only once. Before the **otherwise** label must be at least one **case** label.
In contrast to the **case** label, the **otherwise** label cannot be followed by any expression for evaluation.

The **switch** and **case expression** may be any logical or/and numerical expression.
The **case-expression** in the **case** label is compared for equality with the **switch-expression**.
If the **switch expression** and the **case expression** are equal, then this case is selected and the statements between the matching case expression and the next case or otherwise label are executed. After the execution of the statements may appear the keyword 'break'. But it is not necessary to finish the statements of a single case with **break**, because after the execution of the statements an unconditional jump to the end of the switch is performed automatically.

If there are some case expressions that match to the switch expression, then the first matching case is selected.

Example:

The following example selects the size of a point-to-point motion according to the value of the variable k.

int k	The variable declaration
…	

switch (k)	For example, k=2
case 1	
PA=1000;	
case 2	
PA=2000;	This statement will be performed
otherwise	
PA=500;	If k doesn't equal to 1 or 2 , pa=500
end	

5.5.8 Break

Syntax:
break

The **break** statement terminates the execution of the nearest enclosing **for**, **switch** or **while** statement in which it appears. Control passes to the statement that follows the terminated statement. The break statement is outside of **for**, **switch** or **while** statements is illegal.

Example:

…	
while (IB[2])	Loop until digital input2 is ON
if (!IB[1])	
break;	Break the loop when digital input 1 is OFF
end	
MO=1;	This command will be performed only after digital inputs 1 or 2 are OFF.
…	

5.6 Functions

5.6.1 Function definition

Functions are program sections that can be parameterized and called from anywhere in the program.
Function declaration consists of the following parts:

1. Reserved keyword **function**
2. List of output arguments with their types in square brackets– optional
3. Assignment sign – only if there is some output argument
4. Function name
5. List of input arguments with their types in round brackets. List of input arguments may be empty.

For example,

1	function [int y1, int y2] = func1 (float x1, int x2)	This function is named func1, it gets two input arguments x1 and x2 and returns two output arguments

List of output argument may be empty. In this case 2 and 3items are absent or it can be empty square brackets. If there is the only output argument it can be without square brackets.
Maximal admissible number of input and output arguments is 16.

For example,

2	function func2 (struct float x1)	This function is named func2, it gets the only input argument of float type and doesn't return any output.
3	[int y1] = func3 (int x1)	This function declaration is illegal, because keyword **function** is absent
4	function y1 = func5	This function prototype is illegal, because type of the output variable is absent.

The valid function name observes the same rules as the variable name. It must be distinct from any variable, function or label name.
Definition of dimensions of the output and input arguments at the function declaration is illegal.

For example,

5	function [int x[100]] = func3 ()	This function declaration is illegal, because dimension of output argument is defined.

Function may have prototype before its declaration. The prototype has the same syntax with function declaration, but it must end with semicolon.

For example,

6	function [float y1] = func4 () ;	This is prototype of function func4 that doesn't has any input argument and returns the only output argument
7	function float y1 = func4 ;	It's the same as the previous example 6.
8	function float = func4 ;	This function prototype is the same as example 6 and 7. Name of input or output argument of the function prototype may be absent, while during definition of the function it's error.
9	function [int, int] = func1 (float, int) ;	This is prototype of the function from the example 1. It's the same as function [int y1, int y2] = func1 (float x1, int x2) ;

It is allowed to write the prototype of the same function several times (multiple prototype), but all these prototypes must be identical. The names of the input/output arguments in the function prototype may be omitted, or may be different from the name of the corresponding argument names in the function declaration.

For example,

10	function [int y1, int y2] = func (float x1, int x2) ; function [float y1, int y2] = func (float x1, int x2)	The first expression is the function prototype and the second is the function definition. It's not legal, because types of the first output argument in the prototype and declaration do not match.
11	function [int y1, int y2] = func (float x1, int x2) ; function [int a, int b] = func (float x1, int x2) ; function [int y1, int y2] = func (float x1, int x2)	The first two expressions are the function prototype and the third is the function definition. It is legal

Body of a function resides below the declaration and must end with the keyword **return**. If the keyword **return** is inside some control block within the function body, it is not the end of the function body.

For example,

12	function [int y1, int y2] = func (float x1, int x2)	** Function definition.

	y1 = x1;	** Function body
	y2 = x2;	
	if x2 > 0	** If block
	return	** return inside block is not the end of the function
	end	** End of if block
	y2 = y1 + y2 ;	** Some executable code
	return	** Function end

Before function call, it must be declared. It may be either function prototype or function definition.

For example,

13	function [int y1, int y2] = func (float x1, int x2) ;	** Function prototype.
	…	
	function main()	** Function main definition
	int a, b;	** Local variable definition
	[a,b] = func(2.3, -9.0);	** Function call
	return	** Function main end
	…	
	function [int y1, int y2] = func (float x1, int x2)	** Function definition
	…	** Function body
	return	** Function end

If any function declared without body, its call is illegal.

For example,

14	function [int y1, int y2] = func (float x1, int x2) ;	** Function prototype.
	…	
	function main()	** Function main definition
	int a, b;	** Local variable definition
	[a,b] = func(2.3, -9.0);	** Function call
	return	** Function main end
	…	

In this example function func has prototype, but has no body, so its call is illegal.

5.6.2 Dummy variables

The input arguments and the output arguments of a function are called dummy variables. A true variable is substituted for the dummy variable when a function is called.

Example: In the example of the **statistic** function from the next chapter, variables **mean** and **std** are the dummy variables.

5.6.3 Count of output variables

A function may return multiple values. In some cases, not all the outputs need be computed. A function can use the **nargout** keyword –to know how much of its results it actually needs to evaluate.
Number of outputs is the number of items in the left hand side expression during the function call. If its number exceeds the number of maximal defined output arguments of this function or the maximal admissible number of output arguments, it’s an error.

If some function does not return any output by its definition, then zero value as output will be inserted to the stack actually. Assume that the function **foo** is declared with no output arguments. So the following expression:
foo () + 3 is legal, because **foo** returns zero by default.

Example:

float vec[11], RA[100];	Declare the global variables
float value;	Declare the global variable
function [float mean, float std]=statistic();	Prototype of the function
...	
[RA[1],RA[2]]=statistic()	Calling the STATISTIC function. After executing RA[1] will be equal to variable MEAN and RA[2] will be equal to variable STD
[value]= statistic()	In this case, after executing the STATISTIC function VALUE will be equal to variable MEAN
…	
function [float mean, float std]=statistic()	Declaration of a function that calculates mean and standard deviation of the global vector vec.
int k;	Declare k as automatic variable
global vec[11];	Redeclaration for vec variable (vec is the global variable that is declared before)
mean=0;	
for k = 1:10	
mean = mean + vec[k];	
end	
mean = mean/10;	Calculate mean of vec[]
if (nargout > 1)	Only if the standard deviation is asked...
std = 0 ;	
for k =1:10	
std = std + vec[k] * vec[k] ;	
end	
st = (1/10)* sqrt(std – 10 * sum)	
end	
return	End of the function body
…	

Count of input arguments
The number of input arguments during function call must be the same as it is declared during function definition.

5.6.4 Automatic variables

A variable declared within a function is automatic. It is generated when the function is called, and it ceases to exist when the function exits.
At the time of function call, all its automatic variables are set to zero. When the function

exits, the value of the automatic variables is not saved.
Automatic variable cannot be a vector.

Example: In the example of the STATISTIC function, variable k is the automatic variable

5.6.5 Global variables

A function can refer a persistent variable. In this case the referred persistent variable must be declared **global** inside function and must be defined above function definition. The **global** keyword is obligatory; otherwise the variable will be referred as an automatic variable of the function.
The dimension of a persistent variable is defined once a program during its definition and the same for this global variable in all functions where it is used. The legal way to declare the dimension of a persistent variable inside function is empty square brackets after its name or absence any brackets.
Example: In the example of the STATISTIC function, variable vec[11] is the global variable. It is defined above the function definition and inside the function body it is redeclared as global.

float temp;	Declare the global vector variable
int vec[10] ;	Declare the global vector variable
…	
function [float a]=func(int b)	Declaration of a function **func**
float temp ;	Declare **temp** as an automatic variable, because the **global** keyword is absent.
…	Function body
return	The end of the function
…	
function [float a]=func1(float temp)	Declaration of a function **func1**. In this case **temp** is not global variable, but an input argument.
…	
return	The end of the function
…	
function [float a]=func2(float b)	Declaration of a function **fun2**.
global float temp ;	Redeclaration of the global variable **temp**. In this function **temp** is the global variable.
global int vec[] ;	Redeclaration of the global variable **vec**. Notice that its dimension is omitted during redeclaration and its actual dimension is 10 as it is defined above.

5.6.6 Jumps

Syntax:
goto ##LABEL1

The jump (**goto**) command instructs the program to continue its execution at the label specified by the jump command.

The **goto** command may be specified only for destinations within the present function scope.
It is not possible to jump to labels that are inside another function.
The jump to the global label from within some function is illegal.

Example:

…	**Working code
if (PX>1000)	**Condition for jump
goto##LABEL1;	**Go to LABEL1 if the condition is true
else	**Return axis to origin
goto##LABEL2;	**Go to LABEL2 if the condition is false
end	
…	**Working code
##LABEL1;	LABEL1 declaration
….	**Working code
##LABEL2;	LABEL2 declaration
…	**Working code

5.6.7 Functions and The Call Stack

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

Example:

function JustSo ;	**function prototype
…	
JV=1000	
JustSo()	**function call
BG	
…	
function JustSo	**function definition
IA[0]=1;	**function body
return	**function end

The above code will execute the sequence
JV=1000;IA[0]=1;BG;
After executing JV=1000, the program jumps to the subroutine JustSo. 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 return 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.

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

Example:

function factorial() ;	**Function prototype
…	
IA[1]=3	
IA[2]=1	
factorial()	**Function call
BG	
…	
function factorial()	**Function for factorial
global int IA[] ;	**Definition of array as global inside function
IA[2]=IA[2]*IA[1]	**Recursive algorithm
IA[1]=IA[1]-1	
if (IA[1] > 1) factorial() ; end	**Recursive call
return	**Function end

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

Code	IA[1]	IA[2]	Call stack
IA[1]=3	3	Undefined	Empty
IA[2]=1	Unchanged	1	Empty
factorial	Unchanged	Unchanged	→BG
IA[2]=IA[2]*IA[1]	Unchanged	3	Unchanged
IA[1]=IA[1]-1	2	Unchanged	Unchanged
if (IA[1] > 1) factorial() ; **end**	Unchanged	Unchanged	→RT →BG
IA[2]=IA[2]*IA[1]	Unchanged	6	Unchanged
IA[1]=IA[1]-1	1	Unchanged	Unchanged
if (IA[1] > 1) factorial() ; **end**	Unchanged	Unchanged	Unchanged (condition is false)
return	Unchanged	Unchanged	→BG (Program jumps to the **return** which was on top the call stack)
return	Unchanged	Unchanged	Empty (Program jumps to the BG which was on top the call stack)
BG	Unchanged	Unchanged	Unchanged

5.6.8 Killing The Call Stack

In some rare situations, it is desirable to exit a function without returning to its return address. The **reset** instruction solves this problem by emptying the call stack before making a jump.

Syntax:
reset <JUMP_NAME>

The valid jump after the 'reset' keyword is one of the following:

1. label
2. auto routine
3. user function with no defined input arguments

All other expression or absence of expression after the keyword 'reset' is illegal.

Remark: Label in reset instruction must be global. Local label is illegal, because the stack will be emptied and all local variables and return address of the function to whose the local label belongs will be erased.

Example:

Assume that an Amplifier (an axis) runs a programmed routine.
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
…	**Last line of working code
#@AUTO_I1	**Subroutine label
PA=0;BG	**Return axis to origin
reset START_NEW	**Clear the stack and go to the beginning

The **reset** in the #@AUTO_I1 routine is required since we do not know if any function calls are executing when Digital In #1 is asserted. If a function was executing, the **reset** prevents junk gathering in the call stack. Otherwise, the call stack was empty and the **reset** does no harm. Note that after the **reset** control does not return to the function which was executing before Digital In# is occurred. Stack was cleared and the return address to the interrupted function was removed from the stack.

Note
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 **reset** to the START_NEW label.

5.6.9 Automatic subroutines

5.6.9.1 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. These automatic routines doesn't have output and input arguments.

Syntax:
There are two options to define an auto routine: either as a function or as a label.
If it's defined as a function, then all syntax rules for function definition are relevant (see the chapter Function definition).
An auto routine may be defined as a label: its name follows the sequence of characters ## or #@ in the definition line. The body of the auto routine is between the definition line and the keyword 'return' unless this return is inside some flow control block.
Note:
The keyword 'return' instructs after execution of an auto routine to return to the next line in the code, where execution was halted by interrupt event.
After call such routine and performing its body, the program will return to the next line in the code, where execution was halted by interrupt event.

There are no default handlers for auto routines. If a user does not define automatic routine, then any handling will not be activated when an automatic event is asserted.

Routine name	Priority	Activated by	Mask (MI)
AUTOEXEC	0	An autoexec code is executed automatically upon power on. An autoexec function can be called later any time.	1
AUTO_ER	1	A motor fault event, in which MO=0 is set automatically.	2
AUTO_STOP	2	Called when a digital input configured to the "Hard-Stop" function is activated. Refer the IL command.	4
AUTO_BG	3	Called when a digital input configured to the "Begin" function is activated. Refer the IL command.	8
AUTO_RLS	4	Called when a digital input configured to the "RLS" function is activated. Refer the IL command.	16
AUTO_FLS	5	Called when a digital input configured to the "FLS" function is activated. Refer the IL command.	32
AUTO_ENA	6	Called when a digital input configured to the "Enable" function is activated. Refer the IL command.	64
AUTO_I1	7	Called when a digital input #1 configured to the "GPI (General purpose Input)" function is activated. Refer the IL command.	128
AUTO_I2	8	Called when a digital input #2 configured to the "GPI (General purpose Input)" function is activated. Refer the IL command.	256
AUTO_I3	9	Called when a digital input #3 configured to the "GPI (General purpose Input)" function is activated. Refer the IL command.	512
AUTO_I4	10	Called when a digital input #4 configured to the "GPI (General purpose Input)" function is activated. Refer the IL command.	1024
AUTO_ I5	11	Called when a digital input #5 configured to the "GPI (General purpose Input)" function is activated. Refer the IL command.	2048

AUTO_ I6	12	Called when a digital input #6 configured to the "GPI (General purpose Input)" function is activated. Refer the IL command.	4096

Table 5-1 – Automatic subroutines and their priority

All the automatic routines, except AUTOEXEC, are activated only if a program is running.

Example:

Consider the program

##LOOP gotoLOOP	**A go forever loop
#@AUTO_I3	**Subroutine definition
…	**Subroutine body
return	**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 digital input #3 event.
The #@AUTO_I3 routine will be called if digital input #3 is sensed and not masked.
The digital input #4 will not invoke any automatic action in the user program, since no #@AUTO_I4 routine is defined to handle digital input #3 event. A user has an option for explicit call of an automatic routine, if it is desired. The syntax for an auto routine call is the same as for a user function.

5.6.9.2 Automatic Routines Arbitration

Priority and pending automatic routines
Each automatic subroutine has its assigned priority, according to Table 5-1 above.
When the conditions to the activation of two implemented 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.

5.6.9.3 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:

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

function autoexec ()	**Declare the autoexec function
mi=8;	**Inhibit the routine #@AUTO_I1
op=1;	**Set output to indicate start.
….	
while (!IB[2])	** Wait for setting the digital input 2
end	
goto #@TEST_PARS;	**Jump to subroutine
##LOOP	**Label LOOP
…	**Label LOOP body

goto ##LOOP;	**Endless loop
#@TEST_PARS	**Subroutine
op=2;	**Set output 2
wait 2000;	**Wait 2 seconds for testing the part
mi=0;	**Enable automatic handling of digital in #1
goto ##LOOP;	
return	**End of the autoexec function
#@AUTO_I1	**Automatic handler for digital input #1
op=3;	**Set output 3 to indicate din 1 sensed.
	**Subroutine body.
return	**Subroutine end
…	
exit	**Exit from the program

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 Amplifier should react.
The code below limits the automatic response to digital input #1during to executing the #@AUTO_I1 routine code, even though the digital input #1 may bounce.

##LOOP;	**Repetitive task
…	
MI=0;	**Re-enable automatic routine
…	
goto ##LOOP;	
#@AUTO_I1	**Automatic handler for digital input #1
MI=MI|8;	**Prevent nested calls to #@AUTO_I1
…	**Subroutine body
return	**Subroutine end

6 Program Development and Execution

The process of program development includes the following steps:

- Program editing – Writing\editing the program.
- Compilation – Let the Compiler process the program and find errors
- Program loading – Load the program to the Amplifier Flash memory.
- Debugging – Observe the program behavior, and correct where necessary
- Running the program
- Save to flash – Make the program reside permanently in the Amplifier.

6.1 Editing a Program

The Amplifier program is a simple text. Any text editor can be used to write the Amplifier program.
The Composer program has a development environment for Amplifier 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 Amplifier, compiling the program and running it.

6.2 Compilation

Each user program after editing must be compiled.

The Compiler is described here, although id does not reside in the DSP software. The compiler is external stand-alone software that may be called, for example by the Composer software. User can write and compile program in the off-line mode (without communication establishing with Amplifier)
The Compiler compiles a program in order to produce address maps and run-time code. If in the course of the compilation it finds syntax errors, it stops the compilation and informs the user about errors and presents them in convenient form.
The compiler accepts the user program as text file and several files with Target Harmonica Information. This information is required to assure that the produced compiled code can run on the Harmonica.
The Compiler outputs are two files: CompileStat.txt and CompileCode.img.
The CompileStat.txt file tells the compilation status. It reports success or failure and if the compilation failed it tells what were the errors.
The CompileCode.img exists only if compilation succeeded. It contains the compiled image file, to be downloaded directly to the Harmonica. The description of the contents image file see at User Partition of the Flash

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.

6.2.1 Compilation Error List

The list of the compilation errors is represented below:

Error code	Error string	Meaning and example
0	No errors	Successful compilation without errors
1	Bad format	This is general error for bad syntax at right or left hand expression. For example, for k = 1::10 There is double colon symbol between ‘1’ and ‘10’ b = a + ()

		There is empty expression inside round brackets k = 1:2:20 Colon expression is used out of ‘for’ statement
2	Empty expression	Expected right hand expression is absent For example, a = ; After assignment symbol the right hand expression is absent
3	Stack is full	Stack overflows its depth
4	Bad index expression	This error appears if an index expression of a variable is not evaluated in a single value. For example, a(2,3) The result of evaluation of the expression in round brackets is two values, not a single value. One more example, a() The index expression is empty
5	Bad variable type	The expected variable type is neither ‘float’ nor ‘int’. This error may appear either after the keyword ‘global’ or input/output arguments at the function definition. For example, global floa a; After the keyword ‘global’ the variable type is expected. The type ‘floa’ is unknown. function foo (long a) In round parentheses the type of the input argument is expected. The type ‘long’ is unknown.
6	Parentheses mismatch	The number of opening parentheses does not match to the number of closing parentheses. It regards to as round as square parentheses. For example, b = a(1)) There is unnecessary closing parenthesis.
7	Value is expected	This error appears if right or left hand expression is not evaluated in a single value or it failed during evaluation of a bad syntax expression For example, b = ^ a ; Before operator ‘^’ a value is expected.
8	Operator is expected	This error appears during right hand expression evaluation, if after successful value evaluation there is neither operator nor terminator of a simple expression. For example, b = a c ; After successful evaluation of ‘a’ an operator or expression terminator is expected, but ‘c’ is not recognized neither as an operator nor as a terminator.
9	Out of memory in the data segment	This error appears during memory allocation for global variable: there is no enough place at the data segment.
10	Bad colon expression	Error during evaluation of colon expression. The colon expression may appear only in ‘for’ statement. Bad syntax of colon expression may cause to this error: more than three values or less than two values in the colon expression. For example, for k = 10:-1:5:9 The colon expression contains more than tree values. This error also appears if expected colon expression is absent at the ‘for’ statement. for k = a After the symbol ‘=’ colon expression is expected

11	Name is too long	Variable or function name exceeds 12 characters. For example, int iuyuafdsf_876234 ;
12	No such variable	Left hand side error: left value is recognized neither as variable nor as system command. For example, de = sin(0.5) de is neither variable nor function.
13	Too many dimensions	Dimension of array exceeds maximal admissible number of dimensions (syntax allows only one dimensional arrays). For example, int arr[12][2]; It is attempt to define two dimensional array
14	Bad number of input arguments	The number of input arguments during function call does not match to the number of input arguments at the function definition. For example, function foo(float a); ** Function prototype … ** Some code foo(a,2); ** Function call The number of input argument during function call is two, while the function 'foo' is defined with the only input argument.
15	Bad number of output arguments	Bad syntax of left hand expression: multiple output without square brackets, or multiple output exceeds maximal admissible number of outputs (it is allowed maximal 16 outputs). For example, a, b = foo(1,2) ; Multiple output must be within square brackets.
16	Out of memory	Out of memory during compilation. This error may occur if user program is too large or too complex and there is no enough space in the Code Segment or Symbol Table.
17	Too many arguments	The number of input or output arguments exceeds maximal admissible number of input or output arguments (16). For example, function foo (int a1, int a2, int a3, int a4, int a5, int a6, int a7, int a8, int a9, int a10, int a11, int a12, int a13, int a14, int a15, int a16, int a17, int a18, int a19, int a20) The number of input arguments exceeds 16.
18	Bad context	It appears if the compiler finds any error that disorders a context of program. For example, it may be mismatched parenthesis or improperly closed flow control statement
19	Write file error	Some error occurred during writing to a file.
20	Read file error	Some error occurred during reading from a file
21	Internal compiler error: bad database	It is internal compiler error because of corrupted database. In a case of internal compiler error, please e-mail for technical support, asusid@elmo.co.il. Please attach the Composer date and version (In the help menu in the toolbar) and the program you tried to compile.

22	Function definition is inside another function or flow control block	Illegal function definition: inside another function or inside flow control block.
		For example,
		if a < 0
		  a = 0 ;
		  function foo (int a);
		end
		Attempt to define function inside 'if' block.

23	Too many functions	User program contains too many functions and there no enough space for

		them in the database
24	Name is keyword	Variable or function name is the same as some keyword. This error may appear if variable name is identical to some auto routine name. For example, int switch; 'switch' is keyword, so its use as variable name is illegal.
25	Name is not distinct	This error occurs if some variable or function name is not distinct. For example, int foo ; function foo (int a) Function and variable have the same name 'foo', One more example, function foo (int a) ** Function definition int a ; ** Local variable definition Definition of the local variable 'a' is illegal, because this function is already contains local variable 'a' as input argument.
26	Variable name is invalid	This error occur in the following cases: 1. Variable or function name starts from digit or underscore, but not from letter 2. Variable or function name is empty 3. At the variable definition line there is a comma as a separator between variables, but variable name after a comma is absent. For example, int _abc ; function (int a) int a, b, In the first example variable name has leading underscore. In the second example variable name is absent after a comma. In the third example function name is absent after the keyword 'function'.
27	Bad separator between variables	The only legal separator between variables at the variable definition line is comma. After variable name either variable separator (comma) or expression terminator is expected. Any other symbol causes to this error. For example, int a b; A command as variable separator is absent between 'a' and 'b'.
28	Illegal global variable definition	Global variable must be declared inside function with the keyword 'global' and must be defined before this function. This error appears only if the keyword 'global' was used in the bad context: 1. The keyword 'global' was used outside function. 2. Variable that is declared as global inside function is not defined before. 3. Type of the variable at the definition outside the function differ from the type of the declaration inside function For example, int a1 ; ** Global variable definition function foo (int a) ** Function definition global float a1; ** Declaration of the global variable inside function Type of variable a1 at its definition is 'int', while inside function is declared as 'float'.
29	Bad variable definition	All local variables must be defined at the beginning of the function. Any variable definition after some executable code in the function is illegal. For example, function foo (int a) ** Function definition int b ; global int a1; b = a ; ** Executable code

		float c, d; The definition of float variables ‘c’ and ‘d’ is illegal, because it occurs after executable code ‘b = a’
30	Variable is undefined	This error appears if iteration variable at ‘for’ statement is not defined before it. For example, function foo (int a) ** Function definition for k = 1 : 10 ** Start ‘for’ loop a = k ; end ** End ‘for’ loop return ; ** Function end The iteration variable ‘k’ is not defined before its use.
31	Bad separator between dimensions	Bad separator between dimensions (not comma). This error is unused, because currently it is allowed only one-dimensional arrays and no needs in separator between dimensions.
32	Bad variable dimension	Legal variable dimension must be a positive number inside square brackets. If the expression inside square brackets is not evaluated into a number or this number is less then one (zero or negative), it’s illegal. For example, int arr[-12]; Variable dimension is negative.
33	Bad function format	This error appears at the function definition in the following cases: 1. Illegal function name – not distinct or empty, etc 2. Function definition does not match to its prototype For example, function foo (int a) ; ** Function prototype function foo (float a) ** Function definition Type of input arguments at the function definition and its prototype does not match.
34	Illegal minus	Minus is illegal in the following cases: 1. Minus before function call with multiple output arguments 2. Minus before round parentheses with multiple expressions inside it. For example, [a,b] = -foo(c) ; -(2+3, c/5); The first example has illegal minus before function call with multiple outputs. The second example has illegal minus before multiple expressions within round parentheses.
35	Empty program	User program is empty
36	Program is too long	User program exceeds maximal admissible length
37	Bad function call	The following reasons cause this error: 1. Attempt to jump at the ‘goto’ statement to the function with non zero number of input or output arguments 2. Consider example: [a,b,c] = foo(x,y) + 5; This sentence is illegal. The Compiler checks if there is an expression terminator straight after function call with multiple outputs. Otherwise it sets this error.
38	Expression is expected	This error appears when expected expression is absent. It may occur at ‘wait’, ‘until, ‘while’, ‘if’, ‘elseif’, ‘switch’ and ‘case’ statements. For example, if a = 0 end An expected expression after ‘if’ is absent.

39	Code is too complex	User program contains very complex code that includes too many nested levels (actually this expression contains more than 100 nested levels). Nested expression means one flow control block inside other, e.g. ‘if’ block inside ‘while’ loop.
40	Line compilation is failed	General error during attempt to compile expression.
41	Case must follow switch	After ‘switch’ statement the only legal statement is ‘case’, otherwise this error occurs. For example, switch a b = 0 ; case 1, b =1 ; end The expression ‘b = 0’ is illegal, because ‘switch’ must be followed by ‘case’.
42	Illegal case after otherwise	‘otherwise’ statement must be the last statement of ‘switch’ block. Any ‘case’ after ‘otherwise’ is illegal. For example, switch a case 1, b =1 ; otherwise b = 0 ; case 2, b = 1 ; end The statement ‘case 2’ is illegal, because ‘otherwise’ must be the last statement of ‘switch’.
43	Bad nesting	This error treats flow control block contradictions: ‘else’ without ‘if’, mismatched ‘end’, some flow control without ‘end’ etc.
44	Code is not expected	This error occurs if there is some unexpected code for evaluation: after ‘otherwise’ or after ‘end’ or after ‘else’ etc.
45	Bad flow control expression	This error occurs in ‘for’ statement if there is no a sign '=' after the iteration variable name. For example, For k a = 0 ; end After ‘k’ sign ‘=’ is absent.
46	Too many errors	This error appears when buffer with compilation error is full.
47	Expression is out of function	Some executable code must be inside either function or label, otherwise this error occurs. For example, int a1; ** Global variable definition a1 = 0 ; ** Executable code function foo (int a) ** Function definition Executable code is illegal, because it is out of function.
48	Otherwise without any case	This error occurs if straight after ‘switch’ statement ‘otherwise’ appears. For example, switch a otherwise b = 0 ; end After ‘switch’ statement ‘case’ is expected, not ‘otherwise’.
49	Misplaced break	‘break’ is legal only inside ‘switch’, ‘for’ or ‘while’ block, otherwise, this

		error occurs. For example, if a < 0 break ; end 'break' from 'if' statement is illegal.
50	Too many outputs	This error appears when number actual output arguments during function call exceeds the number of output arguments during function definition. For example, function [int b] = foo (int a) ** Function definition … ** Function body return ; ** Function end … [c,d] = foo(a) ; ** Function call Function call is illegal, because number of outputs is two, while this function is defined with a single output argument.
51	Line is too long	This error occurs if user program contains too long line (more than 128 characters).
52	Clause is too long	This error occurs if user program contains too long expression to evaluate (more than 512 characters). This expression may take several lines of user program text.
53	Cannot find end of sentance	End of sentence not found within range
54	Open file failure	Attempt to open non-existing file: file name or its path may be not correct.
55	Bad file name	The full path file name is too long.
56	No such function	After 'goto' and 'reset' statement must be auto routine, user function or label name, otherwise this error appears.
57	Variable is array	This error occurs during attempt to assign the entire array variable, not its single member. For example, int arr[10] ; ** Global variable definition ##START ** Label definition arr[1] = 0 ; arr = 0; The last expression is not legal: attempt to assign the whole array.
58	Variable is not array	Attempt to assign a scalar variable according to an index, as an array For example, int a1 ; ** Global variable definition (scalar) ##START ** Label definition a1[1] = 0 ; The last expression is not legal: attempt to assign a scalar variable according to an index.
59	Mismatch between left and right hand side expressions	This error appears if the number of left values does not match to the number of values in right hand side expression. For example, [a,b] = 12 + c ; The number of left values is two, while the number of values after evaluation of right hand expression is one.
60	Illegal local array	Syntax does not allow to define local array. Array must be global. For example, function foo (int a) ** Function definition int arr[10] ; … ** Function body return ; ** Function end Local array is illegal
61	Function already has	This error occurs if function has more than one body.

	body	For example, function foo (int a) wait 2000 return ; … function foo (int a) until a end This text is illegal, because the function 'foo' has multiple bodies.
62	Opcode is not supported by the Harmonica	This error occurs if the specified version of the Harmonica does not support some virtual command.
63	Internal compiler error	In a case of internal compiler error, please e-mail for technical support, asusid@elmo.co.il. Please attach the Composer date and version (In the help menu in the toolbar) and the program you tried to compile.
64	Expression is not finished	User program contains unfinished sentence: user may use three points to show that this expression will be continued at the next line. If this continuation is absent, this error appears. For example, assume that the last line of user text is: a = b + 8 / 12 / (8^2*sqrt(2) – sin (3.14/2)) … After three points the next line supposed to appear, but it is absent.
65	Compiled code is too long	The compiled code exceeds maximal allowable space for the Code Segment in the serial flash of the Harmonica
66	Corrupted the Harmonica setup files	Ant file with the Harmonica's setup parameters does not have defined format.
67	Too many variables	User program contains too many functions and there no enough space for them in the database
68	Variable name length mismatch to the Harmonica setup	The allowed variable name length is not equal to the defined length of the Compiler
69	Auto routine has argument	Auto routine cannot has any input or output argument, otherwise this error appears. For example, function AUTOEXEC (int a) Definition of auto routine with input argument is illegal.
70	Label definition is inside flow control block	Definition of label inside flow control block is illegal otherwise this error appears.
71	Function without return	Function has not finished with the keyword 'return'
72	Block comments is not finished	Comment block has no end. For example, assume that it is the last line of user program text: /* Stam comment Comment block is not closed.
73	Bad function after reset	In 'reset' statement must be either auto routine or global label or user function without input arguments, otherwise this error appears. For example, reset foo(12) After 'reset' keyword there is user function with input argument.
74	Bad jump to label	This error appears in following cases: 1. Attempt to jump to global label from within some function or attempt to jump to local label from within some global space 2. Attempt to jump to global label from within some function or attempt to jump to local label from within some global space at 'goto' statement 3. Attempt to jump not to label at 'goto' label

		4. Attempt to jump to local label at 'reset' statement For example, function foo () ** Function definition … ** Function body return ** Function end … goto ##foo The last expression is illegal: foo is not label.
75	Illegal nargout	The keyword 'nargout' is used outside function For example, ##START ** Label if nargout > 2 Use the keyword 'nargout' outside function
76	Function without body	Attempt to call function which is defined but does not have body
77	Bad goto statement	After the keyword 'goto' must be ## or #@ before the name of a label, otherwise this error appears. For example, goto START Between 'goto' and label name ## oe #@ is absent.
78	Auto routine is local	Auto routine is defined as local label. For example, function start (int a) ** Function definition … ** Some code #@AUTOEXEC … return ** Function end The auto routine AUTOEXEC is defined inside function as local label.
79	Command has 'not program' flag	Program refers to command that has 'not program' falg, i.e. it cannot be used inside user program. For example, LS Attempt to use in a program the LS command that has 'not program' flag.
80	Image file is too long	This error occurs if image file length exceeds user code partition size
81	System function tdif is not supported by the Harmonica	During evaluation of the wait flow control, the tdif system function is must be defined inside the Harmonica, otherwise this error appears.
82	Unknown error	Unknown error

6.3 Downloading and Uploading a Program

Note: In this step of the program development the communication between Composer and Harmonica must be established.

After successful compilation the CompileCode.img may be downloaded to Harmonica.
Composer's development environment supports the downloading process. Before each download, Composer automatically clears the memory Flash sector, which is used for saving the user program.

This section describes how the serial flash down/up loads.
The flash is interfaced using two commands:
DL and LS.
Both commands use the assisting command LP.
LP is a vector integer command.
The user flash area may be cleared by the command CP.
The user flash area may be checksum-verified, and program ready flag set, by the CC command.

6.3.1 Binary data

The Flash is interfaced with binary data. Sending binary data on the RS232 lines is a problem, since they complex the differentiation between data and delimiters.

The characters that are problematic to send on the RS232 lines are:

- All the high numbers, 128 to 255.
- All possible terminators: 0, <CR>, <LF>, ';', ','
- Equating sign '='
- Backspace
- Escape: <ESC>

In order to prevent this problem we use hex binary format during up/down load of data, even so it increases amount of data to be transmitted.
Every byte in the hexadecimal form consists of two numbers (e.g. 0x12). The hex binary format consider every of these two numbers as a character, e.g. the 8-bit number 0x12 in the hex binary format is the sequence of two characters '1','2'.

Representation of numbers in the DSP Flash is different from its representation inside a personal computer. A 8-bit number is represented in the same way as inside a personal computer.
Consider a 16-bit number. For example, the hexadecimal 16-bit number is equal to 0x1234. It is represented in the DSP memory in the following two bytes:

Order number of the bytes	Value of the byte in the hexadecimal form
1	0x12
2	0x34

Now consider 32-bit number. For example, the hexadecimal 32-bit number is equal to 0x12345678. It is represented in the DSP memory in the following four bytes:

Order number of the bytes	Value of the byte in the hexadecimal form
1	0x56
2	0x78
3	0x12
4	0x34

Binary data to be loaded to the serial flash is represented in the described above format.

Examples:

Number in the hexadecimal form inside a personal computer	Sequence of characters in the hex binary form to be transmitted
0x12	12
0x1234	1234
0x12345678	56781234

6.3.2 The Assisting Commands For Down/Upload

6.3.2.1 The LP[N] command

LP[1] defines the byte in which the next action is to start (out of 128k bytes of the flash)
LP[2] defines how many bytes to send (LS command).
The LP command is used together with the LS and the DL commands. Please refer the DL and the LS explanations below.
In addition,
LP[3] specifies the start address (bytes) of the user program partition in the flash.
LP[4] specifies the size (bytes) of the user program partition in the flash

6.3.2.2 The CP command

The CP command clears the entire user area in the serial flash. It may take a significant time.
In special, CP sets the Program Valid flag to –1.

Possible Execution Failures:

1) Motor is on.
2) Program is running.

6.3.2.3 The CC command

The command CC=xxxx does as follows:
Read the actual length of the user partition from the TOC.
Calculate 32-bit checksum for the entire actual user partition.
The checksum shall be calculated by summing all the consecutive 2-bytes sequences (short int numbers) that form the user program space and the checksum number itself. The total summing result must be zero to pass.
If the resulting checksum matches xxxx then:

- Set the program ready flag.
- Copy functions and variables symbol tables from the user partition to internal dsp flash memory

Else an error code is return.
CC returns 1 if the program ready flag in the serial flash is set
Otherwise it returns zero.

Possible Execution Failures:

1) Actual length in the TOC is less then 2 flash pages, or is beyond user program address limit.
2) Calculated checksum doesn't match to xxxx.

6.3.3 Downloading a Program

6.3.3.1 The DL command

The Dl command executes downloading a program.
Downloading to a non-protected area in the Flash goes as follows:
LP[1]=start;
DL##xxxxxxxxx<ESC>CS;
where above xxxxxxxxx denotes the escape-sequenced data payload.
Start denotes the byte address in the user program flash.
CS denotes the 16 bits checksum for the message, including the DL##.
DL takes time, since it has to burn & verify.

Possible Execution Failures:

1) Attempt to write to protected area in the flash. It may happen that DL will start to write legally to the Flash, but its last bytes will attempt an illegal (protected) write. In any case DL command will be rejected, and the contents of the serial flash is unpredictable.
2) DL is used when motor is on.
3) DL is used when program is running.
4) Faulty checksum – DL will be rejected but creates no harm.
5) Verify error – If DL attempts to write to a previously written area in the flash, probably the writing will fail by verify error. In that case the contents of the Flash is unpredictable and the Flash has to be cleared and completely re-written.
6) DL string is too long. The maximum length of a DL string is 500 bytes, due to internal Dama limits.
7) Program Valid flag is not –1. In that case DL command will not be executed (CP must be issued before DL).

6.3.3.2 Program downloading process

In order to download a program image to the Dama, follow the steps below.

- Read the location (loc) and the length (len) of the user code partition from the main TOC. For this, use the LP[3] and the LP[4] commands.
- Verify that your image block will fit inside the allocated space.
- Clear the program flash, using CP
- Download your image file by the sequence
    - LP[1]=loc;
    - DL##…100 bytes of payload
    - LP[1]=(loc+100)
    - DL##…next 100 bytes of payload
    - .. Until the end of the image
- Use the CC=checksum command to declare end of loading and to verify entire download process.

In order to upload a program image from the Dama, follow the steps below.

- Read the location (loc) and the length (len) of the user code partition from the main TOC. For this, use the LP[3] and the LP[4] commands.
- Upload your image file by the sequence
    - LP[1]=loc;
    - LP[2]=100
    - Use LS to get next 100 bytes of payload
    - LP[1]=(loc+100);
    - Use LS to get next 100 bytes of payload
    - .. Until the end of the image

6.3.4 Uploading a Program

6.3.4.1 The LS command

A program that resides in the Amplifier Flash can be uploaded for backup or for further editing. This option is disable when program is running. After program uploading, user can correct it and returns to the compilation step.

The LS command executes uploading a program.
Uploading from a non-protected area in the Flash goes as follows:
LP[1]=start;
LP[2]=payload net length
xxxxxxxxx<ESC>CS;
where above xxxxxxxxx denotes the escape-sequenced data payload.
Start denotes the byte address in the user program flash.
CS denotes the 16 bits checksum for the message.
Possible Execution Failures:

1) LS sequence (including <ESC>'s) exceeds 200 characters. The LS output will be terminated with proper checksum and terminator. The data transmitted in the payload is good and meaningful, only not to the expected length.
   The 200 bytes limit is due to internal Dama buffer management.

6.4 The program execution

Note: In this step of Harmonica's development we disable possibility to run more than one task, the virtual machine, simultaneously. But in the future, this feature will be possible.

The following paragraphs describe how to run a program.

6.4.1 Initiating a Program

A program is initiated by the XQ command. The XQ command states at which label the execution shall start.
The XQ command resets all the program variables. In particular, it clears the call stack (refer the chapter the Functions and the Call Stack), it kills any pending automatic routines, and it clears the interrupt mask.
The description and syntax of the XQ command see at the chapter Debugging: running, breaking and resuming

6.4.2 Halting and resuming a program

A program may be halted using the HP Interpreter command.
The HP command stops the execution of the user program and the automatic routines. The HP command freezes the status of the program, and does not reset it.

A later XC command will resume the program from the instruction where the program was halted. Pending interrupts will remain pending.
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;	
wait(1000)	**Wait & switch direction
JV=-JV;	
goto##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 wait(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 waiting time already elapsed.

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
goto##LOOP;	**Repeat task 1
##TASK2;	
##REP;	**Repetitive task 2
…	**Task 2 body
goto##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;

6.4.3 Automatic program running with power up

If the user program includes the autoexec function, the next program line after function declaration will be performed with powering up.

6.4.4 Save to Flash

Since a program is downloaded to a non-volatile memory, it is always saved. Information isn't lost in cases of power down.

6.4.4.1 Clear user program from Flash

The CP command clears the entire user area in the serial flash. The running program must be killed (not halted) before.
This procedure may take a significant time.

6.5 Debugging

This method allows a debug the user program that is downloaded to the Harmonica's flash. This may be useful during development the user program and examination the program flow.

6.5.1 Running, breaking, and resuming

The XQ command starts program execution from a label, or executes a function.
XQ##MYFUNCTION(a,b,c) runs the function MYFUNCTION(a,b,c).
XQ cannot return values from a function.
XQ##LABEL runs from ##LABEL
XQ## runs from the starting of the user program code.
The XQ command without a parameter is illegal.
XQ does not return value.
Note: XQ## without label or function name is designed for running the program written in the Saxophone/Clarinet style, i.e. without function definitions, local variables, etc. The virtual machine executes virtual assembly commands according to their order in the compiled code. If the program contains function definition, it enters inside function and executes function body. It is problematic, so we try to prevent it. If the program starts from a function definition, the XQ command without label or function name causes to the error: NO_SUCH_FUNCTION. If the function starts from label, the dummy start label will be inserted to the function symbol table and running will be from the start of the program.

KL=0 kills all the virtual machines, if it is running.
KL stops the motor.

HP halts all the virtual machines. They can be continued later by the XC command.
If HP halts inside a wait statement, the wait time stops to run while the program is halted.
XC continues all the virtual machines.

6.5.2 DB command

The DB command is designed to help analyze the user program. It allows the following functionalities:

- set and remove break points
- get information about existing VAC machines
- get running status of the program
- ask for or change a value of local variables
- get call stack of the program

The DB command is designed for the IDE Manager use. The syntax of this command is strict, namely the spaces and white characters are not allowed to simplify the treatment of the command and all values must be numbers, expressions are illegal.

6.5.3 Machine status

The command DB##MS returns status about all existing VAC machines.

Syntax:
DB##MS

The command DB##MS return a string in hex binary format containing 16-bit number. Every 4 bits characterizes the status of the VAC machine, so the command can give a status of four VAC machines as maximum. The returned status may get the following values:
0: Halted
1: Running
2: Idle/Not running
3: Aborted/Fault
4: Not existing

6.5.4 Program status

The command DB##PS returns the status of user program.

Syntax:
DB##PS[N]
where N is a handle of a specified VAC Machine.

The command DB##PS returns a string containing hex binary data with the following information:

	Name	Meaning	Type of data	Size in bytes
1	Status	Running status. See 6.4.	signed short	2
2	Error code	The last error code or 0 for no error	signed short	2
3	Program counter (PC)	The program counter of the present executing line	unsigned short	2
4	Base pointer (BP)	The current base pointer. The base pointer is the saved stack pointer, which marks function entry point. BP is used to refer local variables of a function.	unsigned short	2
5	Stack pointer (SP)	The current stack pointer	unsigned short	2

The non-zero error code indicates that any error occurred.
How to define which of VAC machine caused to an error?

When any error occurs inside a specified VAC machine, it returns with error to the main loop that manages the running of the entire set of VAC machines. This manager stops all other VAC machines with the error ABORTED_BY_OTHER_THREAD. In order to define in which VAC machine the error occurred, find the error code different from ABORTED_BY_OTHER_THREAD.

The DB##PS command returns information an analysis of which allows various debug functionality described below.

6.5.5 Setting and clearing break points

Dama supports up to 6 breakpoints simultaneously. A user can set 5 breakpoints and one for inner use of the IDE.
Breakpoints can be set anytime, anywhere, regardless if the program is running or not.
The syntax is
DB##BP=xxx
Sets a breakpoint at program counter xxx.
DB##BP=xxx,n
Sets a breakpoint at line program counter, activated only after n repetitions.
DB##BC=xxx
Removes a breakpoint at program counter
DB##BC
Removes all the breakpoints.
xxx is unsigned long – 4 bytes, n is signed short – 2 bytes.

6.5.6 Continuation of the program

When the program reaches the break point, it stops running of the entire set of existing virtual machines. To cause the program to continue the running, the command DB##GO is used.

Syntax:
DB##GO
and
DB##GO[N]
where N is a handle of a specified VAC machine.

It continues the program running from the current program counter.
The DB##GO command continue running of all the VAC machines.
The DB##GO[N] command continue running of a specified VAC machine.

How to define a handle of a specified VAC machine, which was halted at the break point?
It must be found out the existing VAC machine whose program counter is equal to the program counter of the break point. First the DB##MS command must be sent to define which VAC machines exist. After that for all existing VAC machines we have to send the DB##PS command to get the program counter for comparison with the break point program counter. It may occur the situation when several VAC machines have the same program counter that is equal to the break point program counter. Every of them may be chosen as a specified.

6.5.7 Single step

When a program reaches a break point, a user may desire to continue running in the single step mode.

6.5.7.1 Run to Cursor

The treatment of *Run to Cursor* is very easy: the IDE sets at the relevant line fake break point with the number of repetitions as one. When a break point is reached, it has to be removed from the break point list. There is no need to define special debug command.

6.5.7.2 Step Over

The DB##SO command executes the treatment of *Step Over*. It runs up to the nearest end of line.

Syntax:
DB##SO[N]
where N is a handle of a specified VAC machine.

This command is implemented inside the Harmonica. The algorithm is following:

- Save the current base pointer
- Start the loop
- Run to the nearest end of line
- Compare the current base pointer with the saved base pointer
- If both of them are the same, the step is over - jump to the end the loop, otherwise go to the start of the loop.
- End of the loop

When the nearest end of line is reached, the VAC machine enters to the halted state. In case of jump or infinite loop or other reason inside line execution, the program may not reach end of line, then it just runs and does not enter halted state.

6.5.7.3 Step In

The DB##SI command executes the treatment of *Step In*. It enters inside function body and runs up to the nearest end of line.

Syntax:
DB##SI[N]
where N is a handle of a specified VAC machine.

This command is implemented inside the Harmonica. The algorithm is following:

- Start of the loop
- Run to the nearest end of line
- End of the loop

When the nearest end of line is reached, the VAC machine enters to the halted state.

6.5.7.4 Step Out

The DB##SU command executes the treatment of *Step Out*. It returns from function body and runs up to the nearest end of line.

Syntax:
DB##SU[N]
where N is a handle of a specified VAC machine.

This command is implemented inside the Harmonica. The algorithm is following:

- Save the current base pointer
- Start of the loop
- Run to the nearest end of line
- Compare the current base pointer with the saved base pointer

- If the current base pointer is less than the saved base pointer, the step is out - jump to the end the loop, otherwise go to the start of the loop.
- End of the loop
  When the nearest end of line is reached and the current base pointer is less than the saved base pointer, then the VAC machine enters to the halted state. Otherwise it just runs and does not enter halted state.

6.5.8 Getting stack entries

The DB##GS command returns relevant entries of the stack.
Syntax:
DB##GS[N]=N1,N2
where N is a handle of a specified VAC machine whose stack is inquired,
N1 is an index of the first stack entry and
N2 is an index of the last stack entry (not including).
The command returns a string containing hex binary data with sequence of stack entries from N1 to N2.
If we are interested in the only stack entry, then N2 = N1 + 1.
The command returns a string in the hex binary format.
Every stack entry is a structure contains the next fields:

	Name	Meaning	Type	Size in bytes
1	value	Stack entry value	float or long depending on type	4
2	type	Type of the stack entry: 0 for long, 1 for float	signed short	2
3	unused	Offset for alignment	signed short	2

This command is very useful during debugging.

6.5.9 Setting stack

The command DB##ST sets stack entry. This entry must be local variable or input/output argument of the function.
Syntax:
DB##ST[N]=N1,N2
where N is a handle of a specified VAC machine (signed short –2 bytes),
N1 is stack pointer (signed short – 2 bytes) and
N2 is new variable value (long or float – 4 bytes)
A user may want to change a value of the local variable. The command DB##SS sets new value to the relevant stack entry for a specified VAC machine.
The type of the stack entry will be set according to the type of the sent value.

6.5.10 Getting call stack

There is no direct command to get call stack. In order to reduce complexity of the Dama, the IDE Manager will execute the all debug analysis.
Consider stack during function call (for more details see 8.2).

Stack pointer relatively to the base pointer	Meaning	Remark
BP-5	Saved previous BP	For the first called function BP is 0

BP-4	Return address	Program counter of the next program line after the function call
BP-3	Index of the current function in the Function Symbol Table	See 3.5.5
BP-2	Number of input arguments	
BP-1	Number of actual output arguments	Number of left hand side values during function call

Let's try to understand how this information can help us to restore call stack.
When the first function is called, its BP is zero. Every next function during its call saves the previous base pointer in the stack at BP-5, so it can restore BP during return from the function. When we want to get the entire call stack, we have to roll back the previous base pointer in the loop until BP becomes zero.
The algorithm to restore call stack is in the table below:

- DB##PS[N] ; ** Get current status
- Check the current status: if it not halted, return error.
- BP_last = BP ; ** Get current BP from the data returned by DB##PS
- N1 = 0 ; ** An index of the bottom of the stack
- N2 = SP ; ** An index of the top of the stack
- DB##GS=N1,N2; ** Get the entire stack
- do **Start do-while loop until BP_last is not 0
- funcIndex = BP_last – 3 ; ** Get called function index in the Symbol Table **and insert it to the call stack
- if (BP_last == 0) break ; ** It is the first called function
- BP_last = BP_last – 5 ; ** Update BP_last
- while (1) ** End of do-while loop

6.5.11 View of global variables

The global variable may be accessed through the Interpreter. There is no need in the special debug command.

6.5.12 View of local variables

Local variable cannot be accessed through the Interpreter. We don't define special debug command to access a local variable in order to prevent excessive complexity of the Dama. The IDE Manager can access a local variable by using of existing debug commands and an analysis of their results as during getting call stack.
The algorithm is as follows:

- DB##PS[N] ; ** Get current status
- Check the current status: if it is not halted, return error.
- DB##GS[N]=BP-3,BP-2; **Get an index of the function in the Function Symbol Table. It is located in the stack at (BP – 3).
- Search in the Variable Symbol Table for all local variables of this function. Variables in the Symbol Table are sorted first according to an index of the function to whose they belong and after that according to a variable name (case sensitive). The Symbol Table contains full information about type and location of the variable (see 3.5.6).
- To view a specified local variable use the DB##GS command.
- To set new value of a specified local variable use the DB##ST command.

Note:
The allocation of place for local variables is executed by the command LINK. It inserts N entries to the top of the stack and zeros them, where N is the number of local variables. Usually LINK is the first op code of the function. It may occur that access to a local variable takes place before the op code LINK is executed. In this case the relevant stack entries contains garbage.
The behavior of the debugger may be developed according to two scenarios:

1. Don’t analyze current state of the stack and just access to the relevant stack entries and get garbage.

Try to analyze current program counter to find out whether the LINK was executed. If access to local variable occurred before LINK, then return error (something like VARIABLE_NOT_INITIALIZED).

7 The Virtual Machines

7.1.1 Introduction Alla please complete where necessary

The Harmonica can run a user program, as explained in the Chapter "The Harmonica User Programming Language". Although the user program runs syntactically the same statements as the interpreter, the processing mechanism is totally different.
The interpreter analyses text on line. The user program does not – it runs a compiled code. The most important advantage is that all the text analysis can be spared in real-time, boosting the user program performance. Another advantage is that future user syntax improvements are possible without upgrading the amplifier software.
An external tool, called "The Harmonica Compiler" compiles the user text. The compiler outputs a file of short and simple binary commands.
Take for example the user text
DC=AC+1000;
This code is translated to
.... Alla please fill.
The texts of the compiled version above are in a mnemonic form. Each mnemonic represents an opcode of fixed structure, as described below. In hexadecimal form, the example piece of code is
.... Alla please fill.
The compiled code looks much like an assembly language. The "CPU" that runs this assembly language is written in DSP software. This software that implements the "User Program CPU" is called a "Virtual Machine".
A virtual machine behaves much like a CPU – it has a stack, a code segment with program counter, and a data memory segment.
The Harmonica is designed with multiple virtual machines, so that several user programs may be run in parallel.
The machines are not completely similar. Only one machine, called the Main machine, is interrupted by automatic routines. The other machines are not interrupted by automatic routines. If one desires to interrupt another machine, he may write a software interrupt to that machine from the automatic routine handler of the main machine.

The first release allows the user to activate only one (the Main) virtual machine.

7.2 Virtual Machine registers

A virtual machine has the following registers:

Register	Description	Comment
SP	Stack pointer	Points to top of stack.
BP	Base pointer	Freezes the value of the stack pointer when a function is entered. BP is used to refer local variables of a function.
PC	Program counter	Points to the code position now executing.

7.3 Call Stack During Function Call

Consider a call stack during a function call (the opcode is USRSUBJ).
Assume that a user defines a function :
function [float x, float y, float z] = f (int a, int b, int c, ind d) ; % Prototype
Assume that a function call is:

[x,y] = f (a,b,c,d) ;
Assume that BP before function call is 12 and SP is 15 (just number not better and not worse than every other).
Before the opcode USRSUBJ is executed the stack contains:

2. n entries for **actual** output arguments (according to the function call, not prototype)
3. n entries for input arguments (a number of input arguments during function call must be the same as in the prototype)

In our example the stack pointer before USRSUBJ is equal to 21:
is was 15 + 2 of actual output arguments + 4 of input arguments.
Now the opcode USRSUBJ is executing. During USRSUBJ execution stack is increased by 5 service entries (from SP = 21 to SP = 25) and number of maximal output arguments according to the function definition. The meaning of these service entries is described in the table below. The number of maximal possible output arguments in our example is 3: *x*, *y* and *z*, from SP = 26 to SP = 28).
When 5 service entries has set to the stack a new value of BP is assigned (in our example 26), while the previous value (12) is saved for later restoration.

This table below shows the contents of the stack for our example

SP	Meaning	Value	Remark
12			BP before function call
…			
15	Place for actual output argument *y*	Junk	SP before function call
16	Place for actual output argument *x*	Junk	
17	Input argument	Value of *a*	
18	Input argument	Value of *b*	
19	Input argument	Value of *c*	
20	Input argument	Value of *d*	
21	Save BP current	12	USRSUBJ start
22	Return address	Value of return address	One of operands of USRSUBJ
23	Index of the function in the Function Symbol Table	Function index	One of operands of USRSUBJ
24	Number of input arguments	4	Data from the Function Symbol Table
25	Number of actual output arguments	2	One of operands of USRSUBJ
26	Place for output argument *x*	Junk	New value of BP
27	Place for output argument *y*	Junk	
28	Place for output argument *z*	Junk	
29			USRSUBJ end.
…			

After the opcode USRSUBJ ends SP is 29 and BP is 26.
During USRSUBJ the jump to the beginning of the function is executed (PC is assigned to the relevant location in the Code Segment). Further filling of the stack is fulfilled according to the executable code of the function.
Every function must end with return (the opcode USRSUBRT).
During USRSUBRT the stack returns to the initial state before function call, but not exactly.
USRSUBRT copies the **actual** output arguments to preliminary allocated place in the inverted order. In our example *z* finally won't be assigned, *x* will be copied from SP = 26 to SP = 16 and *y* will be copied from SP = 27 to SP = 15.
BP is restored to the previous value (12). Service entries, input and output arguments are removed and finally SP is 17.

7.4 Data types

The virtual machine supports the following data types:

- **INTEGER: 32 BIT SIGNED INTEGER**
- Float: 32 bit IEEE floating point number (24 bit mantissa, 8 bit sign)

Logical operators yield Boolean results – True or False.
True is equivalent to the integer 1.
False is equivalent to the integer 0.

7.5 Op code structure and addressing modes

Op code it's a bit filed (16 bits) which give us information about the VAC (Virtual Assembly Code) and it's operands, more details in table below.
Each operand has an addressing mode indicates the location of the operand (stack, data segment, immediate).

Op code bit field:

Bits	Description	Meaning
0	ChangeStack	Flag shows whether SP should change (increase/decrease) after an execution of a VAC. This flag is ignored by some of the commands. 1 – change stack, 0 – don't change.
1-7	CodeIndex	Index of VAC in table
8	OperType1	Data type of first operand . 1 – Float, 0 – Integer
9	OperType2	Data type of second operand
10-12	AddrMode1	Addressing mode of first operand (destination)
12-15	AddrMode2	Addressing mode of second operand (source)

Addressing Modes:

Addressing Mode	**MEANING**
ABSENT	Lack operand after command code. Usually Operand is at the top of stack.
IMMEDIATE1	Operand is char integer and stated immediately after command code
IMMEDIATE2	Operand is short integer and stated immediately after command code
IMMEDIATE4	Operand is long integer or float and stated immediately after command code
MEM_DIRECT	Operand is at the memory address (ram) which stated after command code.
STACK_IMMEDIATELY	Operand stated at stack member which placed after command
BP_RELATIVE	Operand is placed in the stack relatively to the base pointer. Exact location in the stack is base pointer + operand stated after command code.

7.6 Short reference

A concise reference is given in the table below.

OP code	Meaning
MLT	Multiply the top of stack with the number below.
SUB	Subtract the top of stack from the number below.
ADD	Add the top of stack to the number below.
DIV	Divide the number below top of stack by the top of stack.

REM	Reminder from division of the number below top of stack by the top of stack.
RSLTAND	Check if number at top of stack **and** number below are non zero than condition is true otherwise condition is false
RSLTOR	Check if number at top of stack **or** number below are non zero than condition is true otherwise condition is false
XOR	Bitwise XOR operator – between the number at top of stack and the number below
NOT	Bitwise NOT operator – on the number at the top of stack
SHR	Logical right shift the number at the top of stack by the number below top of stack
SHL	Logical left shift the number at the top of stack by the number below top of stack
AND	Bitwise AND operator – between the number at top of stack and the number below
F_OR	Bitwise OR operator – between the number at top of stack and the number below
UNARY_NOT	Check if number at top of stack is non zero than condition is false otherwise condition is true
RSLTE	Compare the number at top of stack with the number below, if they are equal condition is true
RSLTA	Compare the number at top of stack with the number below, if the first number is bigger condition is true
RSLTAE	Compare the number at top of stack with the number below, if the first number is bigger or they are equal condition is true
RSLTB	Compare the number at top of stack with the number below, if the first number is smaller condition is true
RSLTBE	Compare the number at top of stack with the number below, if the first number is smaller or equal condition is true
RSLTNE	Compare the number at top of stack with the number below, if they are not equal condition is true
MOV	Assignment operator (=) between global/local variables, assign value can also be constant
CMP	Compare two values (they can be global/local variables, constants or from stack), if they are equal condition is true
JMP	Jump to different location in program
JMP_EOL	Jump to different location in program, and force end of line (stop execution of current program line until next cycle).
JMP_LABEL	Jump to a label
JNZ	Jump if value is not zero (tested value can be local/global variable or constant)
JNZ_EOL	Jump if value is not zero (tested value can be local/global variable or constant) and force end of line (stop execution of current program line until next cycle).
JZ	Jump if value is zero (tested value can be local/global variable, constant or from stack)
JZ_EOL	Jump if value is zero (tested value can be local/global variable, constant or from stack) and force end of line (stop execution of current program line until next cycle).
SPADD	Increase or decrease stack pointer with the value given as an argument
GETINDX	Get value from array variable defined in data segment (array member index is at top of stack)
SETINDX	Set value to array variable defined in data segment (array member index is at top of stack)
SET_COMM	Call to function service of a ‘set’ command whose index in the

	function table is given as an argument
GET_COMM	Call to function service of a 'get' command whose index in the function table is given as an argument
SYSSUBJ	Call to system function whose index in function table is given as an argument
USRSUBJ	Call to user subroutine whose index in the symbol function table is given as an argument (push relevant function data to stack)
USRSUBRT	Return from user subroutine – restore stack to its state before function call, save output arguments in stack and jump to return address
FORITR	Handle FOR loop – check condition, if true iterate and execute loop body. Otherwise, break loop and continue program
EOL	Indicate this is end of line
FREEVAC	Stop program execution
LINK	Increase stack pointer with the value given as an argument and zeroes the new entries to stack
JMP_LABEL	Unconditional jump to the label (entry point)

Table 7-1 – Harmonica op codes

7.7 Alphabetic reference

This section details the Harmonica virtual assembly commands.

7.7.1 ADD - ADDITION

Purpose:
Addition of two numbers (top two entries at the stack) .
Algorithm:
(SP-1) + (SP) → (SP-1)
SP → SP-1
Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types are set according to stack members flags
Addressing modes	Absent	
Promotion	Int + Float Executes as (float) Int + Float	
Output type	Int + int → int Int + Float → Float Float + Float → Float	
Imposes end of line	No	

7.7.2 AND – Bitwise AND Operator

Purpose:
Bitwise AND operator – top two entries at the stack.
Algorithm:
(SP-1) & (SP) → (SP-1)
SP → SP - 1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types must be integer (set according to stack members flags)
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int & Int → Int	
Imposes end of line	No	

7.7.3 CMP – Compare

Purpose:
Compare two values (they can be global/local variables, constants or from stack), if they are equal condition is true otherwise it is false.
Algorithm:
(Op1 == Op2) → (SP)
SP → SP + 1
Attributes:

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	2	
Argument type1	Int or Float	If argument is from stack, type is set according to stack member flag
Argument type2	Int or Float	
Addressing mode1	Absent , Immediate, Mem_Direct , Stack Immediate, BP_Relative	
Addressing mode2	Absent , Immediate, Mem_Direct , Stack Immediate, BP_Relative	
Promotion	N.A.	
Output type	Int	Compare result is always integer
Imposes end of line	No	

7.7.4 DIV – Divide

Purpose:
Division of two numbers (top two entries at the stack).
Algorithm:
(SP-1) / (SP) → (SP-1)
SP → SP-1
Attributes:

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types are set according to stack members

		flags
Addressing modes	Absent	
Promotion	Int / Float Executes as (float) Int / Float	
Output type	Int / Int → Int Int / Float → Float Float / Float → Float	
Imposes end of line	No	

7.7.5 EOL – End Of Line

Purpose:
Indicate this is end of line to the Harmonica environment.
Algorithm:
Zeroes line execution flag.
Attributes:

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	
Addressing modes	N.A.	
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	No	

7.7.6 FORITR – FOR Loop Iteration

Purpose:
Handle FOR loop – check condition, if true update iterator and execute loop body otherwise break loop and continue program.
Algorithm:
itr – (iterator)
(SP-2) → itr (set iterator with start value)
(SP-1) → step
(SP) → target (final value, to which itr is tested)
if (step > 0)
 if (itr > target) free top three entries at the stack and jump end of loop
else if (step < 0)
 if (itr < target) free top three entries at the stack and jump end of loop
else
 error (step = 0 , meaning infinite loop)
(SP-2) + step → (SP-2) (increment iterator by step)
(PC is incremented and loop body is executed)

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	2	
Argument Type1	Signed short integer	
Argument Type2	Signed short integer	
Addressing mode1	Immediate2	Jump address to the end of

		loop
Addressing mode2	Mem_Direct , BP_Relative	Iterator variable address, depend on the addressing mode.
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	Only if jump address is out of the present executing line.	

7.7.7 FREE_VAC - Free Virtual Machine

Purpose:
Stop program execution.
Algorithm:
Zeroes program execution flag.
Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	
Addressing modes	N.A.	
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	No	

7.7.8 F_OR – Bitwise OR Operator

Purpose:
Bitwise OR operator – top two entries at the stack.
Algorithm:
(SP-1) | (SP) → (SP-1)
SP → SP-1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types must be integer (set according to stack members flags)
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int | Int → Int	
Imposes end of line	No	

7.7.9 GETINDEX

Purpose:
Get value from array variable defined in data segment.
Algorithm:

DataSeg + Op1 → Arr (array address)
(SP) → Ind (array index)
Arr[Ind]→ (SP) (return value)

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	1	
Argument type	Signed short integer	
Addressing modes	Mem_Direct	Location of variable (index) in the variable symbol table. Address of array variable is taken from the table. Array index is at top of stack.
Promotion	N.A.	
Output type	Argument Type1	Value from array is pushed at the top of stack instead of array index
Imposes end of line	No	

7.7.10 GET_COMM – Get Command

Purpose:
Call to function service of a 'get' command.
Algorithm:
FuncTable → (list of functions handlers)
FuncTable[Op1]() → (SP) (put return value at top of stack)
SP → SP +1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	1	
Argument types	Unsigned char integer	
Addressing modes	Immediate1	Index of service function in function table. If system parameter is an array , array index is at top of stack .
Promotion	N.A.	
Output type	N.A.	Output argument is at top of stack. Type set according to service function .
Imposes end of line	No	

7.7.11 JMP – Jump

Purpose:
Jump to another location in the program.
Algorithm:

PC→ Immediate value

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	1	
Argument type	Signed short integer	Jump Address
Addressing modes	Immediate2	
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	Only if jump address is out of the present executing line.	

7.7.12 JMP_EOL – Jump

Purpose:
Jump to another location in the program and force end of line . This is used in cases when we want to set breakpoints at specific address but we cant because its not at new line.
Algorithm:
PC→ Immediate value

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	1	
Argument type	Signed short integer	Jump Address
Addressing modes	Immediate2	
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	Only if jump address is out of the present executing line.	

7.7.13 JMP_LABEL – Jump to the label

Purpose:
Jump to another location in the program.
Algorithm:
PC→ Immediate value
Remarks:
The JMP_LABEL is similar to the opcode JMP. The only difference between them is in the argument. The argument of JMP_LABEL contains the absolute jump address in the Code Segment, while the argument of JMP is relative regards to the current location of the command pointer.

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	1	
Argument type	Unsigned short integer	Label Address in the Code Segment
Addressing modes	Immediate2	

Promotion	N.A.	
Output type	N.A.	
Imposes end of line	Only if jump address is out of the present executing line.	

7.7.14 JNZ – Jump Not Zero

Purpose:
Jump to if value is not zero (change PC to different location in program).
Algorithm:

PC → IMMEDIATE VALUE

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	2	
Argument type1	Unsigned short integer	Jump Address
Argument type2	Int or Float	Tested value
Addressing mode1	Immediate2	
Addressing mode2	Absent , Immediate, Stack_Immediate,Mem_Direct, BP_Relative	(tested value can be local/global variable or constant)
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	Only if jump address is out of the present executing line.	

7.7.15 JNZ_EOL – Jump Not Zero

Purpose:
Jump to if value is not zero (change PC to different location in program) and force end of line to allow breakpoints in after jump instruction.
Algorithm:

PC → IMMEDIATE VALUE

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	2	
Argument type1	Unsigned short integer	Jump Address
Argument type2	Int or Float	Tested value
Addressing mode1	Immediate2	
Addressing mode2	Absent , Immediate, Stack_Immediate,Mem_Direct, BP_Relative	(tested value can be local/global variable or constant)
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	Only if jump address is out of	

	the present executing line.	

7.7.16 JZ – Jump If Zero

Purpose:
Jump to if value is zero (change PC to different location in program).
Algorithm:

PC → IMMEDIATE VALUE

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	2	
Argument type1	Unsigned short integer	Jump Address
Argument type2	Int or Float	Tested value
Addressing mode1	Immediate2	
Addressing mode2	Absent, Immediate, Stack_Immediate,Mem_Direct, BP_Relative	(tested value can be local/global variable or constant)
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	Only if jump address is out of the present executing line.	

7.7.17 JZ_EOL – Jump If Zero

Purpose:
Jump to if value is zero (change PC to different location in program) and force end of line to allow breakpoints in after jump instruction.
Algorithm:

PC → IMMEDIATE VALUE

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	2	
Argument type1	Unsigned short integer	Jump Address
Argument type2	Int or Float	Tested value
Addressing mode1	Immediate2	
Addressing mode2	Absent, Immediate, Stack_Immediate,Mem_Direct, BP_Relative	(tested value can be local/global variable or constant)
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	Only if jump address is out of the present executing line.	

7.7.18 LINK

Purpose:
Increase stack pointer with the value given as an argument and zeroes the new entries to stack.

Algorithm:

SP → SP + OP1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	1	
Argument type	Unsigned char integer	
Addressing modes	Immediate1	Number of new entries to stack
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	No	

7.7.19 MLT – Multiply

Purpose:
Multiply two numbers (top two entries at the stack).
Algorithm:
(SP) * (SP-1) → (SP-1)
SP → SP-1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types are set according to stack members flags
Addressing modes	Absent	
Promotion	Int * Float Executes as (float) Int * Float	
Output type	Int * int → int Int * Float → Float Float * Float → Float	
Imposes end of line	No	

7.7.20 MOV – Assignment Operator (=)

Purpose:
Assignment operator (=) between global/local variables, assign value can also be constant or value from stack.
Algorithm:

OP1 → OP2

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	2	
Argument type1	Int or Float	If argument is from stack, type is set according to stack member flag
Argument type2	Int or Float	
Addressing mode1	Absent, Immediate, Mem_Direct , Stack_Immediate, BP_Relative	Assigned value (source)
Addressing mode2	Absent, Mem_Direct , Stack_Immediate , BP_Relative	Variable or stack member (Destination)
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	No	

7.7.21 NOT – Bitwise NOT Operator

Purpose:
Bitwise NOT operator – number is at top of stack .
Algorithm:
~(SP) → (SP)

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Argument type must be integer (set according to stack member flag)
Addressing modes	Absent	
Promotion	N.A.	
Output type	~Int → Int	
Imposes end of line	No	

7.7.22 REM – Reminder

Purpose:
Reminder from division of two numbers (top two entries at the stack) .
Algorithm:
(SP-1) % (SP) → (SP-1)
SP → SP-1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types must be integer (set according to

		stack members flags)
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int % Int → Int	
Imposes end of line	No	

7.7.23 RSLTA – Relational Operator (>)

Purpose:
Compare two numbers, if the first number is bigger condition is true otherwise it's false (numbers are at top two entries of the stack) .
Algorithm:
((SP-1) > (SP)) → (SP)
SP → SP - 1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types are set according to stack members flags
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int	Compare result is always integer
Imposes end of line	No	

7.7.24 RSLTAE – Relational Operator (>=)

Purpose:
Compare two numbers, if first number is bigger or they are equal condition is true otherwise it's false (numbers are at top two entries of the stack) .
Algorithm:
((SP-1) >= (SP)) → (SP)
SP → SP - 1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types are set according to stack members flags
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int	Compare result is always integer
Imposes end of line	No	

7.7.25 RSLTAND – Logical AND Operator (&&)

Purpose:
If both numbers are non-zero condition is true. Otherwise it is false (numbers are at top two entries of the stack).
Algorithm:
(SP-1) && (SP) → (SP-1)
SP → SP – 1
Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types must be integer (set according to stack members flags)
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int && Int → Int	
Imposes end of line	No	

7.7.26 RSLTB – Relational Operator (<)

Purpose:
Compare two numbers, if the first number is smaller condition is true otherwise it's false (numbers are at top two entries of the stack) .
Algorithm:
((SP-1) < (SP)) → (SP)
SP → SP - 1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types are set according to stack members flags
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int	Compare result is always integer
Imposes end of line	No	

7.7.27 RSLTBE – Relational Operator (<=)

Purpose:
Compare two numbers, if the first number is smaller or they are equal condition is true otherwise it's false (numbers are at top two entries of the stack) .
Algorithm:
((SP-1) <= (SP)) → (SP)
SP → SP - 1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types are set according to stack members flags
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int	Compare result is always integer
Imposes end of line	No	

7.7.28 RSLTE – Relational Operator (==)

Purpose:
Compare two numbers, if they are equal condition is true otherwise its false (numbers are at top two entries of the stack) .
Algorithm:
((SP-1) = = (SP)) → (SP)
SP → SP - 1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types are set according to stack members flags
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int	Compare result is always integer
Imposes end of line	No	

7.7.29 RSLTNE – Relational Operator (!=)

Purpose:
Compare two numbers, if they are not equal condition is true otherwise its false (numbers are at top two entries of the stack) .
Algorithm:
((SP-1) != (SP)) → (SP)
SP → SP - 1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types are set according to stack members flags

Addressing modes	Absent	
Promotion	N.A.	
Output type	Int	Compare result is always integer
Imposes end of line	No	

7.7.30 RSLTOR – Logical OR Operator (||)

Purpose:
If one of the numbers or both are non-zero condition is true otherwise it's false (numbers are at top two entries of the stack).
Algorithm:
(SP-1) || (SP) → (SP-1)
SP → SP – 1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types must be integer (set according to stack members flags)
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int || Int → Int	
Imposes end of line	No	

7.7.31 SET_COMM – Set Command

Purpose:
Call to function service of a 'set' command.
Algorithm:
FuncTable → (list of functions handlers)
(SP) → ArrayIndex

(SP-1) → VAL
FuncTable[Op1](val , ArrayIndex) (call to 'set' command)

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	1	
Argument types	Unsigned char integer	
Addressing modes	Immediate1	Index of service function in function table. Array index is at top of stack . Set value is at one entry below top of stack.
Promotion	N.A.	
Output type	N.A.	

Imposes end of line	No	

7.7.32 SETINDEX

Purpose:
Set value to array variable defined in data segment.
Algorithm:
DataSeg + Op1 →Arr (array address)
(SP) → Ind (array index)
(SP –1) → Val (assign value)
Arr[Ind] = val ; (enter value to array)

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	1	
Argument type	Signed short integer	
Addressing modes	Mem_Direct	Address of array in data segment (ram) . Array index is at top of stack . Set value is at one entry below top of stack .
Promotion	N.A.	
Output type	N.A.	Type of assign value from stack
Imposes end of line	No	

7.7.33 SHR – Shift Right

Purpose:
Shift right the number below top of stack by the number at the top of stack.
Algorithm:
(SP-1) >> (SP) → (SP-1)
SP → SP – 1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types must be integer (set according to stack members flags)
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int >> Int → Int	
Imposes end of line	No	

7.7.34 SHL – Shift Left

Purpose:

Shift left the number below top of stack by the number at the top of stack.
Algorithm:
(SP-1) << (SP) → (SP-1)
SP → SP – 1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types must be integer (set according to stack members flags)
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int << Int → Int	
Imposes end of line	No	

7.7.35 SPADD

Purpose:
Increase or decrease stack pointer with the value given as an argument.
Algorithm:

SP → SP + OP1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	1	
Argument type	Signed char integer	
Addressing modes	Immediate1	
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	No	

7.7.36 SUB - SUBTRACT

Purpose:
Subtraction of two numbers (top two entries at the stack) .
Algorithm:
(SP-1) - (SP) → (SP-1)
SP → SP-1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types are set according to stack members flags

Addressing modes	Absent	
Promotion	Int - Float Executes as (float) Int - Float	
Output type	Int - int → int Int - Float → Float Float - Float → Float	
Imposes end of line	No	

7.7.37 SYSSUBJ – Jump To System Subroutine

Purpose:
Call to system function.
Algorithm:
FuncTable → (list of functions handlers)
FuncTable[Op1]() → (SP) (put return value at top of stack)
SP → SP +1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	1	
Argument types	Unsigned char integer	
Addressing modes	Immediate1	Index of system function in function table
Promotion	N.A.	
Output type	N.A.	Output argument is at top of stack. Type set according to service function .
Imposes end of line	No	

7.7.38 UNARY_NOT - Logical NOT Operator (!)

Purpose:
If number is non-zero condition is false otherwise condition is true.
Algorithm:
! (SP) → (SP)

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Argument type must be integer (set according to stack member flag)
Addressing modes	Absent	
Promotion	N.A.	
Output type	!Int → Int	
Imposes end of line	No	

7.7.39 USRSUBJ – jump To User Subroutine

Purpose:
Call to user subroutine (push relevant function data to stack).

SEE 8.2
Algorithm:
BP → (SP) (save base pointer)
SP → SP + 1
OP2 → (SP) (save return address)
SP → SP + 1
FuncIndex → (SP) (save function index)
SP → SP + 1
NArgIn → (SP) (save number of input arguments)

7.7.39.1.1.1.1.1 SP → SP + 1

NActArgOut → (SP) (save number of actual output arguments)
SP → SP + 1
SP → BP (update base pointer)
PC → FuncAddr (jump to user subroutine)

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	2	
Argument type1	Unsigned short integer	Lower 8 bits : Actual ouput arguments Upper 8 bits : Subroutine Index
Argument type1	Unsigned short integer	Return Address
Addressing mode1	Immediate2	
Addressing mode1	Immediate2	
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	Yes	jump address is out of the present executing line.

7.7.40 USRSUBRT – Return from user subroutine

Purpose:
Return from user subroutine - restore stack to its state before function call, save output arguments in stack and jump to return address.
See 8.2
Algorithm:

7.7.40.1.1.1.1.1 (SP-1) → nActArgOut (number of actual output arguments)

(SP-2) → nArgIn (num of input arguments)
(SP-4) → RtAddr (return address)
SP → BP – 5 - nArgIn
(save output arguments from subroutine)
(SP-5) → BP (base pointer)
PC → RtAddr (return from user subroutine)

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	
Addressing modes	N.A.	
Promotion	N.A.	
Output type	N.A.	
Imposes end of line	Yes	Jump address is out of the present executing line.

7.7.41 XOR – Bitwise XOR Operator

Purpose:
Bitwise XOR operator – top two entries at the stack.
Algorithm:
(SP-1) ^ (SP) → (SP-1)
SP → SP-1

Attributes

Attribute	Value	Comment
Op code		May change in future versions
Number of arguments	None	
Argument types	N.A.	Both arguments types must be integer (set according to stack members flags)
Addressing modes	Absent	
Promotion	N.A.	
Output type	Int ^ Int → Int	
Imposes end of line	No	

8 The Recorder

The recorder mechanism enables the user to record various signals that exist in the Harmonica, up to 8 signals simultaneously. The recorded signals can be uploaded to the host by communication, for the purpose of presentation and analysis.
The following section details how to define the recorder parameters to the Harmonica, how to launch and trigger the recorder, and how to fetch the recorded data.
The average user does not have to know all this detail, since the Composer program normally operate the recorder with a user-friendly interface.
The list of record-able signals supported by a Harmonica is stored internally and can be retrieved by the host. Please refer to LS and LP commands in the Software Manual.

The following commands are relevant for the recording process.

Command	Description
BG,BT, IL[N]	Begin motion by software or hardware command. Motion beginning may be used to trigger the recorder.
BH	Upload recorder results.
LS	Load a record from the serial flash. Use in order to retrieve the list of recorder signals
RC	Defines which of the mapped signals shall be recorded.
RG	Recorder gap. Specify the sampling rate of the recorder.
RL	Recorder length.
RP[N]	Recorder parameters. This command defines what event will trigger the recorder, and the trigger position. It also defines: - The basic time quantum for the recorder is TS or four times TS. - The data to be uploaded to the host by the next BH command.
RR	Launch the recorder, and read back its status
RV	Signal mapping. This command maps the ID's of the record able signals to logical ID's that the recorder can refer.
SR	Status register – tells the status of the recorder – idle, armed, triggered and recording, or ready with data.
TS	Sampling time the basic resolution of the recorder.
WI[21]	The actual amount of recorded data.

Table 8-1: Commands relevant to the recorder

8.1 Recorder sequencing: Programming, launching, and uploading data.

In order to activate the recorder, it must be programmed. Programming the recorder means telling the recorder what signals to record, at what resolution, and what shall be the trigger event.
Few limitations apply:

- The recorder cannot be programmed while it is armed, or recording. It has to be killed first (RR=-1).
- Changing the recorder programming invalidates any previously recorded data. Be sure to upload all the data you need before programming the recorder.

After programming, the recorder can be launched by:

- RR=1 will arm the recorder to trigger at the next motion begin (**obsolete, please use RR=3)**
- RR=2 will start recording immediately.
- RR=3 will arm the recorder to start recording at the next trigger event.

The status of the recorder may be polled using the RR and SR commands.
After the recorder data is ready, use the BH command to upload the data.

8.2 Signal mapping

The recorder can record many different signals.
The first 16 signals that may be recorded are compatible with older amplifiers. They are listed is in the table below.

Signal ID	Signal Name (Command)	Length – Type	Description
1	Main Speed (VX)	Long – Float	Speed of main feedback sensor in counts/sec
2	Main Position (PX)	Long – Integer	Position accumulation of the main feedback sensor in counts
3	Position Command (DV[3])	Long – Integer	Position reference in counts. When external reference mode is set (RM=1) this signals includes also the analog reference command.
4	Digital input pins status	Short integer	The IP variable (refer IP in the command reference)
5	Position Error (PE)	Long - Integer	Position tracking error the difference between the main Position Reference and the main Position Feedback in counts.
6	Current Command (DV[1])	Short – Float	The Current reference to the current controller in amperes.
7	DC Bus Voltage	Short – Integer	Bus voltage in volts.
8	Auxiliary Position (PY)	Long – Integer	Position accumulation of the auxiliary feedback sensor in counts
9	Auxiliary Speed (VY)	Long – Float	Speed of auxiliary feedback sensor in counts/sec
10	Active current command (IQ)	Short – Float	The active part of the vectored current reference in amperes.
11	Reactive current command (ID)	Short – Float	The reactive part of the vectored current reference in amperes.
12	Analog Input 1 (AN[1])	Short – Float	Analog input 1 after offset compensation of AS[1] in volts.
13	Reserved		
14	Motor current phase A (AN[3])	Short – Float	Phase A current in amperes.
15	Motor current phase B (AN[4])	Short – Float	Phase B current in amperes.
16	Speed Command (DV[2])	Long – Float	Speed reference to the speed controller in counts/sec.

Table 8-2: Default mapping of recorded signals

After power-on, the recorder can access the first 16 signals tabulated above. In order to access other signals, the recorder must make them available in a process called "mapping". In the mapping process, the ID's of the desired signals is mapped to the recorder "cells". On power-on, the signals of Table 8-2 are mapped to the recorder "cells"[2].
The table below lists some more signals available to the recorder. The full list of recorded signals may differ for Harmonica grades and releases. It can be retrieved from the personality partition of the serial flash memory.

Signal ID	Signal Name (Command)	Length – Type	Description
17	Field angle	Short	Stator field angle, 1024 counts per electrical revolution
18	Commutation sensor	Long –	The position counter, counted modulo a mechanical

[2] The word "cell" is used for a logical ID that can be directly referred by the recorder.

		Integer	revolution, with origin at the electrical angle of zero.
21	Filtered torque command	Short	The command to the Q current controller, at the output of the command filter.
45	Motor DC supply voltage	Short	Sample Motor DC supply voltage.
64	External position reference	Long - Integer	The part of the position reference generated by external inputs.

Table 8-3: Some additional recorded signals

Before recording a signal from Table 8-3, this signal must be "mapped to the recorder". In the mapping process, the signal is given a logical ID that can be referred directly by the recorder.
Up to sixteen signals may be mapped to the recorder at any time. Up to 8 signals can be recorded at the same time.
The command RV[N]=x maps a signal with the ID of x to the logical ID of N, N=1..16.
RC is a bit field parameter (bit 0 to 15) that signals the recorder the actual required signal. In our case the bit N-1 of RC points to signal x.
For example, RV[2]=1 maps the signal with the ID of 1 (the main encoder speed) to the logical ID of 2. It means that in order to record the main encoder speed bit 1 of RC should be set (RC=2).
The default mapping, restored at boot, maps the signals of ID's 1 to 16 to the corresponding logical ID's 1..16. Thus the signals with the ID's 1..16 can be recorded by using the logical ID's 1..16 respectively, without further programming.
If the user desires to record only signals with ID's in the range 1..16, he need not be aware to the difference between signal ID's and logical ID's, and he need not program any mapping.

8.3 Defining the set of recorded signals

The command RC defines which of the mapped signals to record.
The RC command is a 16bit bit field. Each bit of RC specifies a logical signal ID to record.
For example, consider RC=5. The value 5 has the 16bit binary value of 0000000000000101.
The first and the third bits in the binary value of 5 are on, and the rest of the bits are zero.
RC=5 thus specifies that the signals with the logical ID's one and three shall be recorded, and all the other signals will not.
Example:
The commands
RV[1]=5;RV[2]=1;RC=3;
defines that when the recorder will be launched, it will record the main speed and the position error.
RC may define 8-recorded signals at most. In another words, the binary representation of RC may not include more than eight one's.

8.4 Programming the length and the resolution

The length and the resolution of the recorded signals is programmed by the RP[0],RL, and RG command.
RP[0] defines what is the basic time quantum of the recorder.

RP[0]=0	Synchronize the recorder to the speed or position control cycles. The time quantum of the recorder will be $4 \cdot TS$
RP[0]=1	Synchronize the recorder to the torque cycles. The time quantum of the recorder will be TS.

RP[0] is set automatically by the UM (Unit mode) command.
The torque mode UM=1 sets RP[0] to 0.
All the other unit modes set RP[0] to 1.

RG defines the sampling rate of the recorder, in terms of the time quantum.

RG=1 means that a new sample shall be taken once per time quantum. If TS=60, and RP[0]=0, the recorder will sample once per 240usec.
RG=2 means that a new sample shall be taken once per two time quanta. If TS=60, and RP[0]=1, the recorder will sample once per 120usec.
Similarly, RG=N means that a new sample shall be taken once per N time quanta.
Note that RG specifies only the recorder sampling-rate, not the trigger sampling-rate. A trigger event will cause the recorder to start within one time quantum.
RL defines the number of maximal data items to collect.
For example, if RL=11, TS=60,RG=1, RP[0]=0, then the 11 samples at the rate of 240usec shall be taken, lasting $(11-1)\times 240\text{u sec} = 2400$usec.
The recorder has a limited memory – total of 8kb. When more signals are recorded, less memory is available for each recorded signal.
If $RL > (\text{recorder memory})/(\text{number of signals})$, the recorder will fail to record the specified RL samples – instead it will record to fill its data volume.
The actual amount of recorded data can be polled using WI[21].

8.5 Trigger events and timing

The recorder is started by a trigger event.
The trigger event may be one of the following:

- Immediate: The recorder starts immediately after the recording request has been issued.
- Motion begin: The recorder is triggered by the execution of a software BG command, or by timed BG command, or by a hardware commanded hardware begin.
- Triggered by an analog signal: The recorder starts upon the following event:
    - Positive slope – The signal crosses a prescribed level with positive slope
    - Negative slope – The signal crosses a prescribed level with negative slope
    - Window – The signals exits a window of two prescribed signal levels.

The picture below depicts the positive slope, negative slope, and window trigger types.

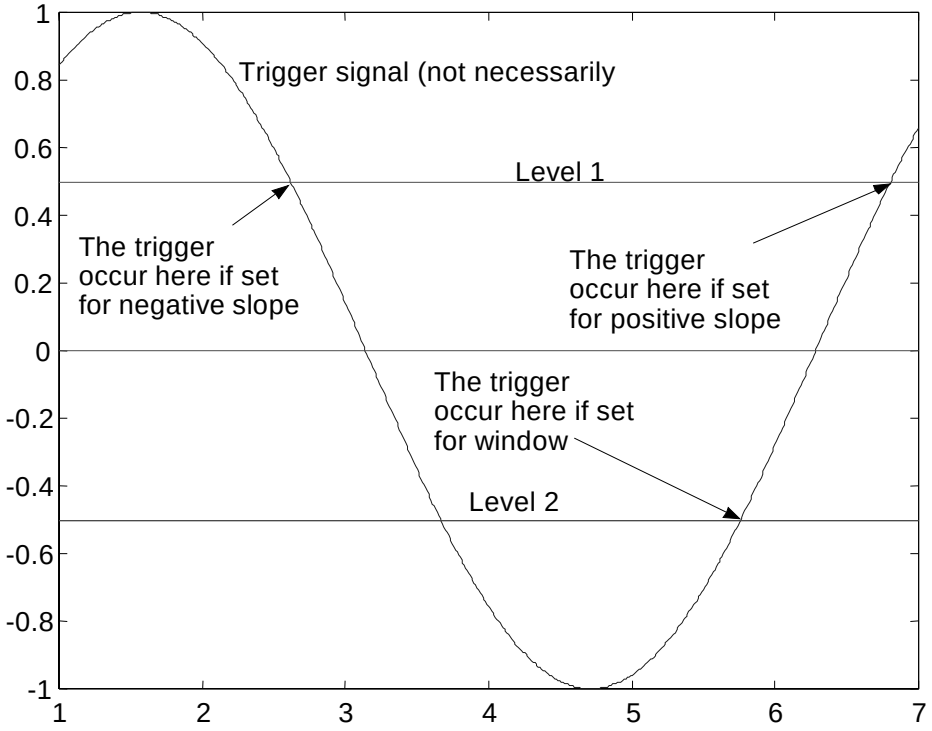


Figure 1 - Slope and window trigger types

- Triggered by a digital signal: The Harmonica will support this only in the future, TBD.

Trigger delay

The trigger defines when the recorder is to start. The recorder can be told to start before the trigger event, so that the trigger event can be caught in "the middle of the picture". This is possible since the recorder starts to record at the instance it is launched by the RR command, so that when the trigger event occurs, the pre-trigger information is already recorded.
The following picture examples this. The recorded signal is the speed. The trigger is set on BG. After launching the recorder with RR=3, the command sequence JV=5000;BG is entered. The results are shown in the picture below.

Note that in the above example, the recorder works for 10sec. Therefore, a pre-trigger delay of 20% requires 2 seconds to acquire the pre-trigger data. A BG command that is set less than 2 seconds after the RR=3 will be missed.

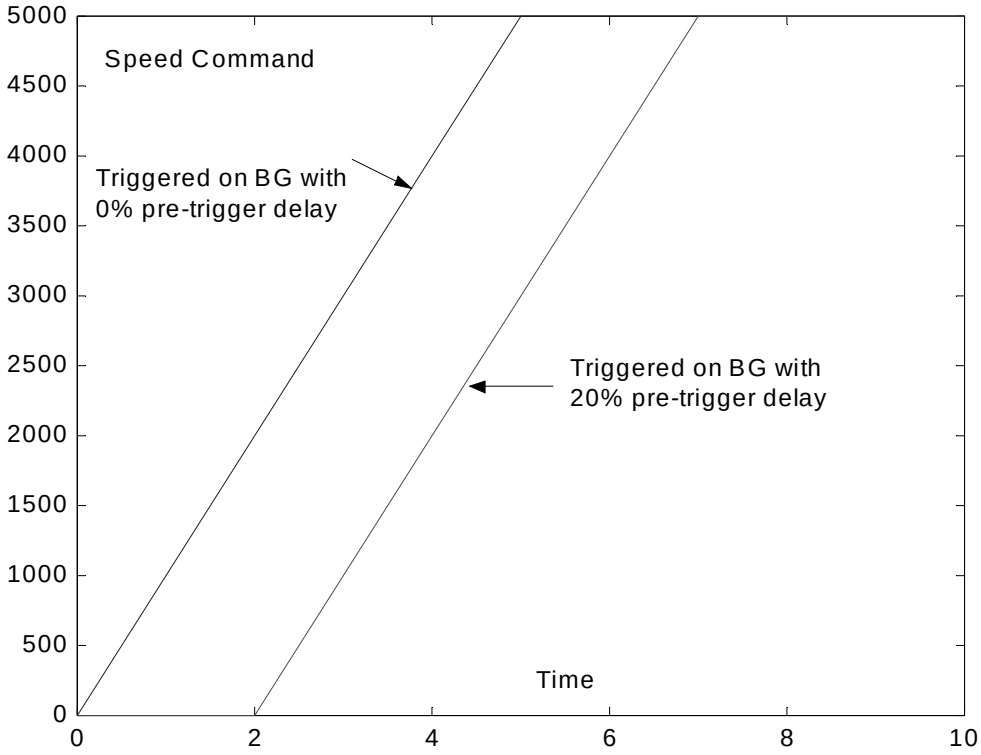


Figure 2 – Pre trigger delay

The trigger parameters are listed in the table below. .

RP[N]	Definition
RP[0]: Recorder time quanta	0: The recorder time quantum is the speed controller sampling time ($4 \cdot TS$ usec) 1: The recorder time quantum is the torque controller sampling time TS usec.
RP[1]: Trigger variable	Defined similarly to RC, but only 1 bit may be non-zero. The trigger variable does not need to be one of the recorded variables. For example, RV[1]=17, RP[1]=1 defines that the trigger will be made on signal #17 (the stator's field angle).
RP[2]: Pre trigger storage in percents.	0 to 100 (percent)
RP[3]: Trigger type	0 for immediate
	1 for a BG
	2 for positive slope
	3 for negative slope
	4 for window
	5 reserved for future digital input trigger option.
RP[4]: Level 1	Level for positive slope trigger, or high side for window trigger
RP[5]: Level 2	Level for negative slope trigger, or low side for window trigger

The trigger levels RP[4] and RP[5] can be entered either as integer or floating point

numbers.

8.6 Launching the recorder

The recorder is launched (or killed) using the RR command. RR also reports the recorder status.

The RR command has the following options:

RR Value	Meaning
-1	Kill the recorder if active, and invalidate any recorded data.
0	Kill the recorder (Do nothing if the recorder is not active)
1	Launch the recorder, triggered on the next BG.
2	Launch the recorder with immediate trigger.
3	Launch the recorder with the trigger defined by the RP parameters.

Table 8-4: RR command options

Note that the actions of RR=1 and RR=2 can be obtained with RR=3 and the appropriate RP[N] definitions. The commands RR=1 and RR=2 are simply meant to allow easy interface to the most common recorder actions.

As a status report, RR may return the following values:

RR Report	Meaning
-1	No valid data in the recorder.
0	The recorder action is complete, and it is loaded with valid data.
1,2,3	Waiting for the completion of RR=1, RR=2, or RR=3 respectively. The report value of 1,2, and 3 does not tell if the recorder is already recording or is just waiting for a trigger. If this differentiation is required, use the SR command.

Table 8-5: RR reports

The SR (Status Register) command details the status of the recorder. SR returns a bit field. Bits 16 and 17 of SR may have the following values:

Bit 16	Bit 17	Recorder status
0	0	Recorder inactive, no valid recorded data
0	1	Recorder armed, waiting for a trigger event
1	0	Recorder finished; valid data is ready for use.
1	1	Recording now.

Table 8-6: SR recorder status reports

8.7 Uploading recorded data

This section summarizes how to upload recorded data from the amplifier to a host.
The relevant commands for recorder data uploading are:

Parameter	Description
RR	If zero, tells that the recorder is ready for data upload.
WI[21]	Tell how much data is actually recorded.
RR[8],RR[9]	Define the part of the signal to be next uploaded
BH	Upload recorded data command.

Table 8-7:Parameters for uploading recorded data

The BH command is used to upload the values recorder by the recorder to a host. The BH command is designed to optimize the data transfer from the Harmonica to the host, assuming that the host has the computing power to analyze the Harmonica message.

The basic condition for executing a BH command is that valid data is stored in the recorder for uploading. In that case the fields of the RC variable define the variables that had been recorded.
The command BH=n will upload the recorded variable defined by RC&n , where & is the bit-wise AND operator. For example, if RC=7, the command BH=2 will bring the variable that would be recorded by setting RC=2, since 7&2=2. BH=3 is illegal, since the binary representation of BH includes more than one '1' and the uploaded variable is not uniquely defined. BH=16 will return an error since BH&RC=0.
It is convenient to use hexadecimal notation for the BH command.
BH=0x4000 may look more understandable then BH=32768.

The BH command may load an entire recorded signal, or a part thereof. If RP[8]=0 and RP[9]=0, then BH=n will upload an entire uploaded signal.
Otherwise, BH will upload the recorded signal starting at the index RP[8], and until the index of RP[9]. RP[9] must always equal or be less than the length of the recorded signal.

The data is uploaded in hexadecimal form. This is done in order to minimize the transmission time (relative to ASCII formatted text), while adhering to the ASCII nature of the transmissions.

Each data byte is parted to two nibbles, and the ASCII code of the nibbles is sent from the controller to the Host. For example, the short integer number 43794 has the hexadecimal representation AB12. It will be transmitted as 'A','B','1','2', with the most significant nibble first and the least significant nibble last. The long integer number 1 will be sent as '0' '0' '0' '0' '0' '0' '0' '1'.

In order to analyze the BH record, one have to understand that the internal representation of quantities inside the controller is not in user units. For example, the user desires the recorded motor current in Amperes, but the controller represents currents internally by the bits of the A/D that measures the current. The BH record uploads recorded currents (and other variables as well) in their internal representation units-and it also provides the scaling multiplier to bring it into user units.
That way, no multiplication inaccuracies are introduced to the BH records, and CPU load is minimized.

The record transmitted by the controller in response to a BH=n command is described below. The record includes 20 bytes of overhead, and numerical records data.
The values in the table are translated to ASCII as explained above.

Byte Number	Value	Type
0-1: Variable type for user. This field does not have any practical significance.	0 for integer 1 for real	Byte
2-3: Data width – number of hex character of a single transmitted data item.	4 for short integer, 8 for long integer.	Byte
4-7: Data length – The actual number of transmitted data items.		Word
8-11: Variable time multiplier. This is the number in which TS must be multiplied to obtain the basic period of the recorder.	Depends in the RP[0] value.	Word
12-19: Floating number factor. Multiply every uploaded data item by this number in order to convert it to user units, such as Amperes, or Count/sec.		32 bit Float number in the IEEE format.
20 – 20 + (Data Length)*(Data width) –1 : Data items. The oldest record is transmitted first, and the most recent record is transmitted last.		Words or long integers, according to the Data Width

Table 8-8: BH – record structure

The transmission of a BH record can be quite long. A record of 2000 long numbers is about 8000 bytes, which take at least 4 seconds to transmit in RS232 in the baud rate of 19200. In this time,
The user program continues to run normally.
CAN commands are accepted and processed normally.
RS232 commands are accepted and executed normally, but the transmission of the response to them is deferred until the BH upload completion.

Example
A BH command may return the string
0008000100013f80000000010000
This string is decomposed to field as follows:
00 (int) 08 (8 bytes per data item in the message)
0001 (Only one data item in message, after the 20 bytes overhead)
0001 (Record taken every TS)
3f800000 (To scale, multiply result by 1.0. The floating point IEEE representation of 1.0 is 0x3f800000)
00010000 (1st Data item: 0x10000 = 65536)

9 Commutation

9.1 General

The harmonica drives fixed magnet motors.
The principle of all the fixed magnet motors is the same:
A winding creates a magnetic field. If the magnet is directed along the field lines of the winding, the magnet is in its steady state, and the winding exerts no power on the magnet.
If the magnet is not along the winding field lines, the magnet will try to align with the field lines.
Mathematically,

$$T = K_e I \cdot \cos(\theta)$$

θ	Angle between the magnet and the field of the winding
K_e	A parameter, proportional to the strength of the magnet
T	Motor winding torque
I	Motor current

If the magnet is allowed to move, it will rotate until aligned with the winding field, and remain stationary there.
A motor is composed of a fixed magnet and several windings. The windings are arranged so that each winding generates a field of different direction.
By powering the windings alternately, the direction of the windings field move, and so does the direction to which the rotor is attracted.
The process of alternating the powered winding is called **commutation**.
Brush DC motors are equipped with a mechanical arrangement that select which winding to power. In brushless motors, the windings to power are selected electronically.
The winding powering policy, or the **commutation policy**, has two major variants.
These variants are called **Stepper** and **BLDC** (Brushless DC).

The setting of the commutation parameters is normally done using the Composer interface routine. If you want to tune commutation manually, read the CA[] documentation in the Command Reference Manual thoroughly.

9.1.1 Brush DC motors

The Harmonica has three motor output connections. A DC motor has only two wires. In order to drive a DC motor by the amplifier, connect the motor to the B and C motor phase outputs, and leave the phase A connection open.
We next have to set the following parameters:

CA[28]	Set to 1 for a DC motor
CA[18]	Set encoder resolution just for reference. The value of CA[18] will not affect anything in later activities
CA[16]	Encoder direction: Set 0 or 1 so that the encoder will count forward in the desired movement direction.
CA[25]	Motor direction: Set 0 or 1 so that the motor will rotate to the desired direction for positive torque commands.

The values of CA[16] and CA[25] **MUST** be correlated, otherwise the feedback direction is wrong - the encoder will count negative displacement for positive torques.
If the feedback direction is wrong, the motor will "run away" immediately upon attempting speed or position control.
If you intend to use the Harmonica with DC motor, you may skip the rest of this chapter.

9.1.2 The Stepper Commutation Policy

In the stepper commutation policy, the windings field is set to point at the desired rotor position. The commutating device doesn't have to know where the rotor is – it just assumes that the rotor will come to rest at the field position.
The stepper commutation has the advantages of simplicity and reliability. The main drawback is that normally $|\theta| << 90^\circ$, thus to generate a given torque large currents are required.
At the steady state, the motor torque is zero, and indifferent to the motor current. The sensitivity of the motor torque to deviation of the rotor angle is maximal.
The large sensitivity of the torque to the rotor angle generates a fast, but oscillatory position feedback.

9.1.3 The BLDC commutation policy

In the stepper commutation policy, the windings field is set to point 90° away from the rotor position. The commutating device has to know where the rotor is in order to keep the field direction 90° away.
The BLDC commutation has the advantage of maximum torque per given motor current, and of smooth, controllable torque. The BLDC policy involves much real time calculation and it requires a rotor position sensor. The commutation dependence in the sensor decreases motor reliability.
The torque is not sensitive to the rotor angle.
The BLDC commutation is ideal for servo applications.

9.2 Mechanical and electrical motion Figures are missing

Most of the brushless motors has two or three phases (windings). The figure below shows a two phased linear motor and a three phased linear motor.

Figure 3: Two phased linear motor

Figure 4: Three phased linear motor

The Harmonica suits only three phased motors.
When the rotor travels, the coils of the 3-phased motor are powered in the sequence A-B-C-A-B-C... and so on. Although the moving part (rotor) travels continuously, it sees the windings in a repetitive, cyclical pattern. When the rotor passes from a location over the A coil to the next A coil, it covers an electrical cycle.

"Rolling" the linear motor of Figure 4 so that its right hand C phase rolls towards the left hand A phase makes a rotary three-phase motor.

Figure 5: Three phase rotary motor

The motor of Figure 5 has two coil sets (and also two magnetic pole pairs in the rotor). The rotor follows two electrical cycles in each mechanical revolution.
For rotary motors, normally the number of coil sets equals the number of magnet pole pairs, so the term "number of pole pairs" is commonly used as substitute to "number of electrical cycles in a shaft revolution".

You can read the electrical and the mechanical angle of the motor using the following commands:

Command	Description

WS[20]	Stator field angle, in 1024 counts/rev units. Stator field angle (deg) = WS[20]×(360/1024)
WS[21]	Commutation counter. WS[21] counts the main high-resolution position sensor, modulo CA[18].

9.3 Commutation sensors

For BLDC commutation, rotor position sensors are required.
The commutation sensors divide into two main groups.
- Direct field sensors that sense the magnetic field of the motor.
- Shaft position sensors.

9.3.1 Rotor Magnetic field sensors

The first group consists mainly of digital Hall sensors. (Analog Hall sensors are less common and the Harmonica doesn't know to use them). Digital Hall sensors may be used for commutation with minimum calculation, since they reflect the direction of the rotor with respect to the windings directly. The digital Hall sensors yield, however, only crude information. Standard digital Hall sensors divide the electrical period of motor to six – see Figure 6.
The reading of the digital Hall sensors is according to Table 9-1.
The BLDC field angle is the field angle that produces maximum torque for this Hall sensor reading.

Hall A	Hall B	Hall C	Electrical rotor position (deg)	BLDC Field angle (deg)
0	0	0	Illegal	
1	0	0	330-30	90
1	1	0	30-90	150
0	1	0	90-150	210
0	1	1	150-210	270
0	0	1	210-270	330
1	0	1	270-330	30
1	1	1	Illegal	

Table 9-1: Hall sensor values

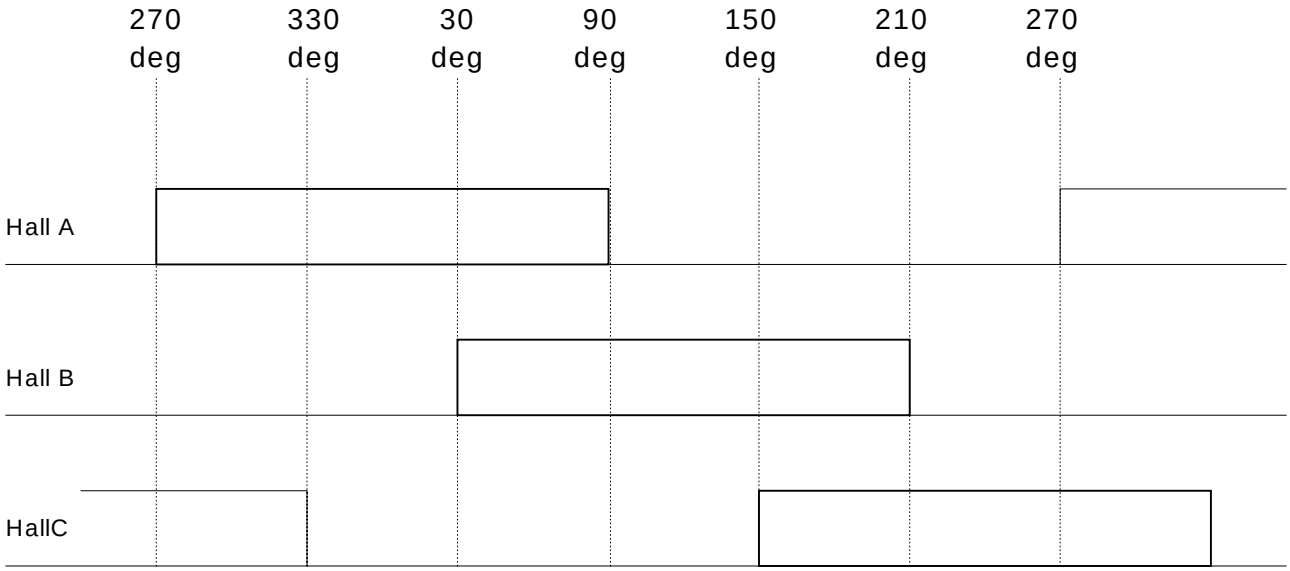


Figure 6: Digital Hall sensors readout

The crude division to six draws rough torque from the motor, and requires hard switching of the motor winding currents.
The digital Hall sensors are used, in many applications, together with higher resolution position sensors. Combined with high-resolution sensors, the Hall sensors serve to initialize the field direction, and add redundant position sensing for increased reliability.

9.3.2 Shaft Angle Sensors

The second group consists of shaft angle sensors like encoders of all types (Incremental digital and analog, absolute digital and analog), resolvers, capacitive sensors, and more.
The Harmonica can interface directly only digital and analog incremental encoders.
The shaft angle sensors normally have good resolution, but they need to be homed (referenced absolutely) with respect to the rotor electrical angle.
From the above discussion it is clear that for the sake of commutation, it is important to know how many bits the shaft sensor counts per one electrical cycle.
The number of counts per electrical may be not an integer number. If the number of counts per electrical cycle is not an integer, then after enough movement the calculated commutation angle may accumulate numerical errors and the motor shall lose torque.
The numerical commutation error is not a big issue with linear motors, since the limited travel limits also the commutation calculation error.
With rotary motors, we observe that each mechanical shaft rotation involves an integer number of encoder counts, per an integer number of electrical cycles. This means that the commutation can be kept accurate as
p1 = mod(encoder, counts per shafts revolution)
and
Electrical rotor angle =

$$360^{\circ} \cdot \frac{\text{number of pole pairs}}{\text{counts per shaft revolution}} \cdot \text{mod}(p1, \frac{\text{counts per shaft revolution}}{\text{number of pole pairs}})$$

9.3.3 Combination of direct magnetic field sensors and shaft angle sensor

9.3.3.1 Initializing encoder based commutation

When starting a motor, a rough estimate of the electrical angle can be taken from the digital Hall sensors. At most of the time, the digital Hall sensors read the electrical angle to +/-30 electrical degrees. The exception is the instance at which the Hall sensor reading switches.

At that very instance the Hall sensors read the electrical angle accurately.
After the first Hall sensor switch, the commutation is kept accurate by updating the commutation counter incrementally using the shaft position sensor.

9.3.3.2 Detecting commutation errors (loss of feedback)

After commutation by the encoder starts, there exist two sources for the electrical angle measurement. There is the high-resolution measurement by the encoder, and also the low resolution, but reliable digital Hall sensor readout.
The digital Hall sensor readout defines a range in which the high resolution calculated angle should reside. Deviating from this range by more than few degrees results in declaring a commutation error and automatic motor shut down.
The allowed deviation is increased in higher speeds, since there the sensor and the calculation delays contribute a significant matching error.
The most common causes for commutation loss are:

- A slit in the encoder disk is clogged with dirt. When this happens, an encoder of 4000 counts/rev may count for example 3999 counts/rev. After enough revolutions, the cumulative error is large.
- Speed to fast relative to the defined encoder filer, and encoder pulses are lost.
- An interpolator derives the encoder A/B pulses. Some interpolators send, from time to time, pulse batches of much higher frequency then the motor speed. Avoid using the encoder filter with interpolators even if you think that the motor speed shall be slow enough.

9.3.4 Parameterization of the commutation and the commutation sensors

9.3.4.1 Winding order

The Harmonica has three motor connection pins, named A, B, and C.
The pin names are not rigidly tied to their actual role. The harmonica can define internally which motor phase is connected to which output pin. The parameter CA[25] controls the connection:

CA[25]	Phase A connected to pin	Phase B connected to pin	Phase C connected to pin
0	A	B	C
1	A	C	B

As seen in the table, changing CA[25] switches the B and C phases.
Actually, changing CA[25] will reverse the direction to which the motor moves for a given torque command.

9.3.4.2 Hall sensors parameterization

Figure 6 presents an idealized picture of the digital Hall sensor reading. All the waveforms are in their precise phase and precise polarity.
In practice, the results of the figure may not equal what we see immediately after connecting the motor and its sensors to the Harmonica.

- The Hall sensors must be matched to the motor coils. Possibly the order of connecting the Hall sensors to their respective connector pins is incorrect, or the motor phase connections has been switched as describe above to modify the motor direction.
- The Hall sensors may be active high or active low. For some Hall sensor arrangements (known as 30° arrangements) two sensors may be active high, and the other one is active low, or vice versa.

In addition, the switching lines in the figure are set in 30,90,150,210,270, and 330 degrees.
The even spacing of 60° is true for most motors, but many motors exhibit a significant origin deviation: for example, with 10 degrees deviation, the digital Hall sensor switching points may be at 40,100,160,220,280,and 340 degrees respectively. This error, although

hardly noticeable in low speed, significantly looses motor torque at high speed.

If Hall sensors are not present, and if the commutation is performed using an incremental encoder, then upon motor start the Harmonica must first find the electrical direction of the motor.
If digital Hall sensors are not present (CA[20]=0), then at motor on, a commutation search is made. The commutation search is described in the section "Commutation search".

The following seven parameters describe the Hall sensors:

CA[1]	The polarity of the A digital Hall sensor. 1 for active high, 0 for active low.
CA[2]	The polarity of the B digital Hall sensor. 1 for active high, 0 for active low.
CA[3]	The polarity of the C digital Hall sensor. 1 for active high, 0 for active low.
CA[4]	The actual Hall sensor connected to the A Hall connector pin 1 for A, 2 for B and 3 for C
CA[5]	The actual Hall sensor connected to the B Hall connector pin 1 for A, 2 for B, and 3 for C
CA[6]	The actual Hall sensor connected to the C Hall connector pin 1 for A, 2 for B, and 3 for C
CA[7]	The offset of the digital Hall Sensors in encoder units. The range for this parameter is [0..CA[18]-1]. This parameter compensate for deviations in the hall sensor switching point. If no encoder is present, set this parameter to zero.
CA[20]	Digital Hall sensors present. 0: No digital Hall sensors are connected 1: Digital Hall sensors are connected.

9.3.4.3 Encoder parameterization

Accurate commutation requires high a resolution sensor. Many types of high-resolution sensors exist. The selections available for Harmonica are:

CA[17]	Commutation sensor type. 1: Main Encoder
CA[21]	Position sensor present 0: No high-resolution commutation sensor. Commutation will be done based on the digital Hall sensors only. 1: The main position sensor shall be used for commutation

The encoder is normally used both for motion feedback and for commutation. As a motion feedback counter, it must count up when the motor go forward in the application sense. As a commutation counter, it must count up when the commutation angle increases.
The above two requirements are not necessarily same, so we need the following two parameters:

CA[16]	Encoder direction: Set 0 or 1 so that the encoder will count forward in the desired movement direction.
CA[25]	Motor direction: Set 0 or 1 so that with positive torque, the motor will rotate in the direction for which the encoder counts up.

The encoder measures the shaft angle. In order to commutate, we must know the encoder count per an electrical revolution. Normally, the number of encoder counts per motor revolution is an integer (If not, we may not be able to commutate correctly with an encoder...). The number of pole pairs per revolution is always integer. By knowing the encoder counts per mechanical revolution, and by knowing how many pole pairs are within a revolution, the commutation counter can infinitely update without accumulating errors.
For linear motors, it is best to set the number of pole pairs CA[19] as the largest number of

full electrical cycles possible for that motor. Enter the "encoder counts in revolution" CA[18] parameter as the number of encoder counts for CA[19] electrical cycles.

The encoder parameters are listed in the following table.

CA[18]	Encoder bits per revolution, after resolution multiplication by 4, in the range [24..10,000,000]. For an incremental encoder with 1000 lines, CA[18] will be 4000
CA[19]	Number of motor pole pairs [1..50].
CA[23]	Counts per meter (any positive integer) 0: Rotary motor 1: Counts per meter in a linear motor. This parameter is not used directly by the amplifier – it is just stored there for the convenience of a host

9.4 Commutation search

9.4.1 General

When starting the motor, the rotor can be anywhere.
The torque of a brushless DC motor is given by the equation

$$T = K_T \cdot I \cdot \sin(\theta) \quad (1)$$

$$\theta = \theta_s - \theta_r \quad (2)$$

where

T	Motor torque
K_T	Motor constant
I	Motor current
θ	The electrical angle between the rotor and the field at the stator
θ_s	The electrical angle of the stator field
θ_r	The electrical angle of the rotor

The angle θ_s is known, since the amplifier controls it directly.

The angle θ_r is unknown.

If we rotate θ_s, i.e. $\theta_s - 2\pi f \cdot t$ where **f** is some frequency and **t** is the time, we get the sinusoidal torque

$$T = K_T \cdot I \cdot \sin(2\pi f \cdot t - \theta_r) \quad (3)$$

For that torque, the motor shaft will move according to

$P(t) = A(f) \cdot K_T \cdot I \cdot \sin(2\pi f \cdot t - \theta_r - \phi(f))$, where

P(t)	Motor shaft position
$A(f)e^{j\phi(f)}$	Transfer function of the motor and its load at the frequency **f**.

By applying the torque of (3) and measuring the position, we can identify both **A(f)** and θ_r.

9.4.2 Selecting the parameters

The parameters I and f can be selected from the range specified below:

I	The torque is selected by the parameter CA[26]. CA[26] defines I as a percentage of the continuous current rating CL[1]. For example, if CA[26]=50, then I=0.5 CL[1].
f	CA[15] controls the frequency of the sinusoidal motor torque . The basic frequency (with CA[15]=0) is so that a cycle is completed in 128 * TS microseconds. For a Harmonica with TS=50, this will be a cycle of 6400

	microseconds, which is about 160Hz. The frequency is $2^{CA[15]} \cdot$ basic frequency for CA[15] in the range [-4..4]. For example, with CA[15]=2 the frequency is about 40 Hz, whereas with CA[15]=-2 the frequency is about 640Hz

The selection rules for the parameters **I** and f are few and simple.

- The torque **I** must be as large as possible, so as to reduce the relative effect of disturbance torques (like cogging and friction) on the resulting waveform. Normally **I** is taken about 50% of the continuous motor current.
- The frequency f must be selected so that the amplitude of the position sine (See figure 1) will be 6 to 8 bits. The motor position can be analyzed accurately enough when the position sine amplitude is 4 encoder bits or more. Slightly higher amplitude is set to prevent minor load changes from disturbing the analysis. Larger position amplitudes will work, but the shaft oscillation at the motor starting process will be unnecessarily large.
- The frequency f must be so that in that frequency the load behaves inertially. This means that in that frequency the phase angle $\phi(f)$ is in the range of –140 to –220 degrees, and also that in this frequency the amplitude function A(f) foes not have the high gradient of near resonance regions. The algorithm will not function near to a resonant frequency, or in a low frequency where a large viscous friction is present.
- Specifying very high oscillation frequency (CA[15] equals –3 or –4) is not recommended, since then not enough position samples are available for each sine cycle.

The composer program normally selects the parameters **I** and f at the "Establishing commutation" stage.

9.4.3 Method limitation

The algorithm presented in this paper is quite robust, and it is expected to work at most of the motor systems, including systems with moderate backlash. The users of the algorithm must, however, be aware of its limitations, which are as follows:

- The encoder must have sufficient resolution – at least 256 counts per pole pair. For example, if a motor has 3 pole pairs, a 256 lines (1000 counts/rev) will suffice, an encoder with 128 lines will not.
- The algorithm will not work with highly unbalanced systems. It is essential that the motor will not accelerate significantly if no current is applied.
- The algorithm will work only if the motor is not free to oscillate to both directions
- The algorithm will not work too well if the motor static load is changed significantly. If the static load is subject to significant changes, tune the algorithm with the highest possible load. Namely, the inertia of the load must be no less than 40% from its value while the parameters of the algorithm were tuned, and no more than 40% more than its value while the parameters of the algorithm were tuned.
  If the load inertia is to high, the preset torque level will not suffice to oscillate the motor shaft. The algorithm will fail and the motor won't start.
  This has the consequence that the algorithm must be tuned with the load present
- The algorithm assumes that in the in the excitation frequency, the motor and the load behaves like an inertia. This assumption fails in one of the following cases:
  The excitation frequency is a resonant frequency of the system
  There is a large viscous (speed dependent) friction and the excitation frequency is so low, that the phase of the transfer function between the torque and the acceleration is not in the range [-30 … 30] degrees.
  If the found phase is out of range, the algorithm fails and the motor won't start.

9.4.4 Protections

The method described here can work in reliably many practical situations, although it does not fit every application. If the parameters of the method are not tuned properly, or the method is not good for the application, the motor starting process will fail.
After setting MO=1, the algorithm will try to oscillate the motor and find the commutation angle. If the results are not reliable, since the oscillation amplitude is too small or since the load behavior is not inertial, then MO will reset automatically to zero and the motor won't start.

9.4.5 Parameters related to starting the motor with no digital Hall sensors

The following parameters are relevant:

Parameter	Comment
CA[18],CA[19]	The encoder resolution and the number of motor pole pairs. CA[18] must be greater than CA[19]*256.
CA[20]	Must be set to 0 to indicate that Hall sensors are not present
CA[21]	Must be set to 1 to indicate that an encoder is present.
CA[15]	Set the frequency of the oscillation, as explained above.
CA[24]	Set the minimum acceptable oscillation amplitude – set to 4
CA[26]	Set the excitation amplitude, as a percentage of the continuous current.
MO	MO=1 sets the motor on. If the motor starting fails, MO resets to 0
WS[15]	Reads the actual oscillation amplitude.
WS[16]	Flag if a non-inertial behavior has been observed.

9.5 Continuous Vs. Six-Steps commutation

The basic commutation equation for a sinusoidal motor (repeated here for convenience) is

$$T = K_T \cdot I \cdot \sin(\theta_F - \theta_r) \quad (1)$$

Above, θ_F, θ_r are the field electrical angle and the rotor electrical angle, respectively.

For optimizing the torque, we have to keep $\theta_F - \theta_r \approx 90^\circ$.

Let us define the commutation error

$\varepsilon_\theta = 90^\circ - (\theta_F - \theta_r)$, Ideally $\varepsilon_\theta = 0$.

The torque production is not very sensitive to ε_θ. Writing equation (1) as

$T = K_T \cdot I \cdot \cos(\varepsilon_\theta)$ we see that for a miss of 5°, we lose about 0.4% of the torque. With a miss of 30°, we lose 13.4% of the torque – see figure below.

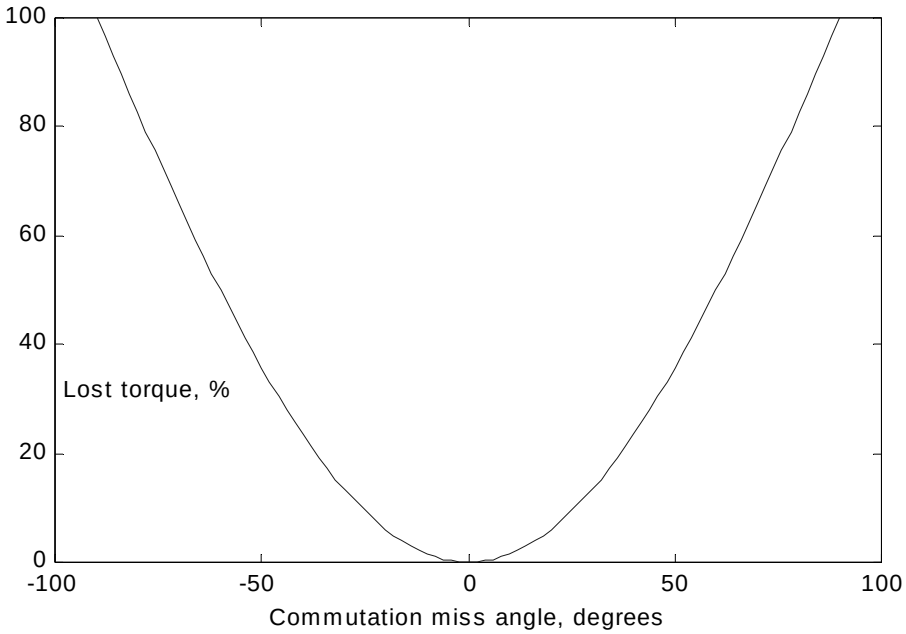


Figure 7: Loss of torque due to commutation miss

Two principal methods are used to keep ε_θ near zero. The first is called "Six-Steps" commutation, and the other is the continuous commutation.

9.5.1 Six-step commutation

With Six-Steps commutation, only two motor terminals are energized at each time instance. The third motor phase is open-circuited.
Six field angles are possible:

Current flows:	Field direction, degrees
C→A, B open	30
C→B, A open	90
A→B, C open	150
A→C, B open	210
B→C, A open	270
B→A, C open	330

Table 9-2: Six-Steps commutation

The digital Hall sensors have evolved to support Six-Steps commutation.
The crude Six-Steps produces about 13% ripple torque when used with sinusoidal motors, and much less ripple torque when used with trapezoidal motors – see the section "Winding shapes" below. The main drawback of the Six-Step commutation is the need to abruptly switch the phase currents. This need imposes an extreme bandwidth demand for the current controller. If the bandwidth of the current controller is less than satisfactory, there will be noticeable "knocks" at commutation switching points.
The Harmonica uses six-step commutation if no commutation encoder is present (CA[21]=0).
In that case, the Hall effect sensors will be also used for speed and position control as the position sensors.
The Harmonica also uses six-step commutation immediately after motor on, and before a first Hall sensor transition is encountered. When the first Hall transition is encountered, the high-resolution commutation sensor (encoder) can be homed and commutation may proceed in the continuous mode.

9.5.2 Continuous commutation

With continuous commutation all the three motor coils are powered simultaneously to yield a magnetic field exactly at the direction of the rotor. This brings ε_θ near zero continuously, with minimal torque losses and ripple torques.
The continuous commutation mode is native for the Harmonica and is used most of the time.
The continuous commutation is much more complex to implement then six-steps commutation. In fact it requires two independent current controllers to control both the amplitude and the direction of the windings magnetic field.
Continuous commutation reduces the dynamic demands from the current controller, since current demands are almost never switched abruptly.

9.6 Winding shapes

The implementation of continuous commutation implies that we know how to direct the magnetic field of the windings.
For a general motor, we have

$$T = K \cdot (I_a h(\theta) + I_b h(\theta - 120^\circ) + I_b h(\theta - 240^\circ))$$

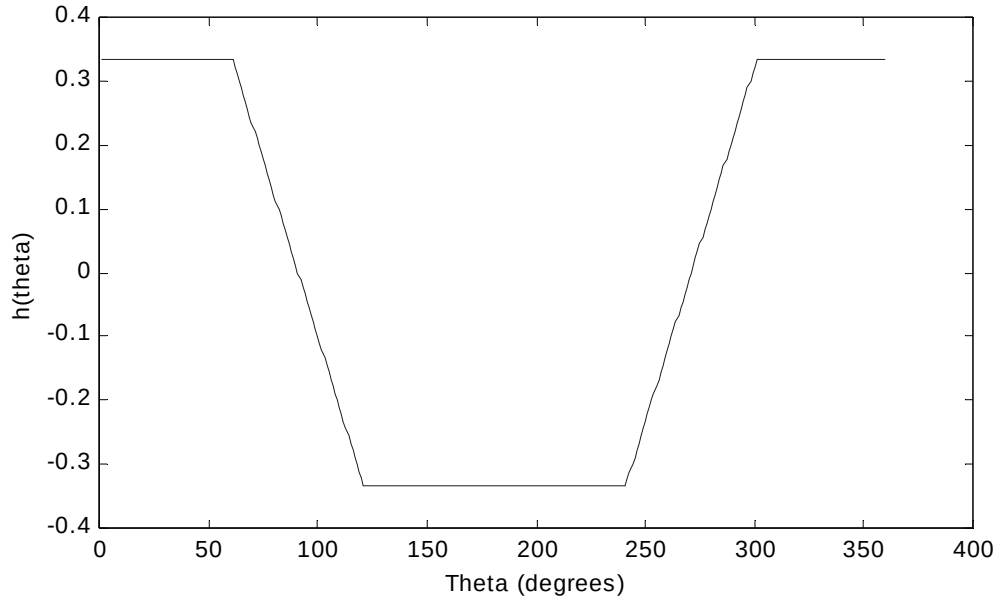
With

T	Torque
K	Constant
h	Windings shape function.
I_a, I_b, I_c	The A,B, and C phase currents respectively.

For optimal efficiency, it is easy to show that the phase currents must be

$$I_a = I_0 h(\theta), I_b = I_0 h(\theta - 120^\circ), I_c = I_0 h(\theta - 240^\circ) \text{ for some value } I_0 . \quad (1)$$

In another words, the phase currents must be proportional to the corresponding commutation function values. If (1) is satisfied, the magnetic field produced by the winding currents is perpendicular to the rotor magnet.
Most motor are wound to sinusoidal or Trapeze winding forms, but no motor can be exactly sinusoidal or trapeze. Trapeze motors are normally chosen for six-step commutation,



whereas sinusoidal motors are normally chosen for continuous commutation.

Figure 8: Winding shape function for a Trapeze motor

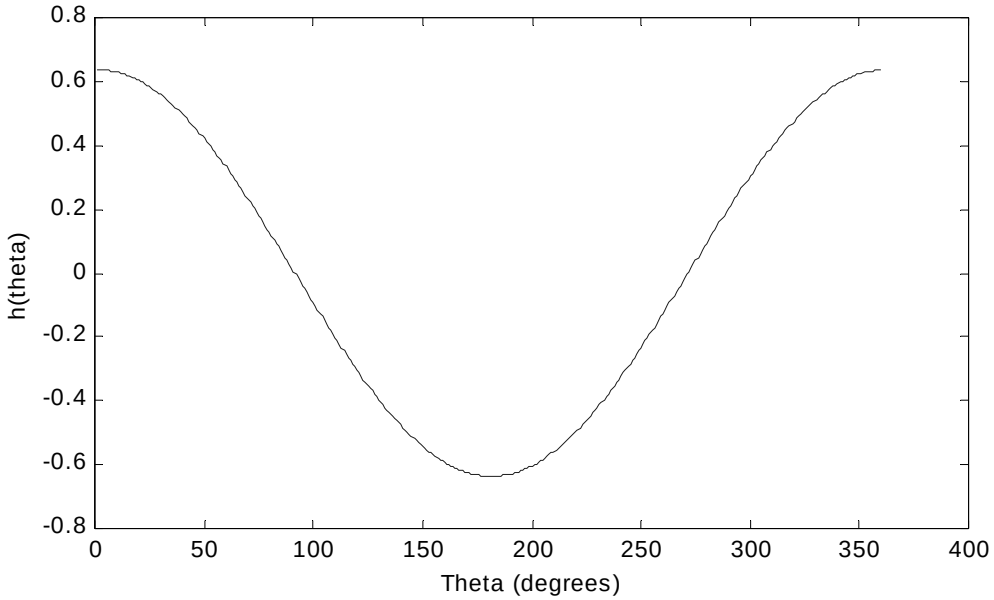


Figure 9: Winding shape function for sinusoidal motor

In order to optimize the Harmonica to as many motors as possible, the Harmonica can be programmed to any winding shape function. The Harmonica comes from the factory programmed by default for sinusoidal motors, but using a tool in the Composer program the motor waveform may be changed.

9.6.1 Loading the commutation table

The commutation table can be programmed using the HV command.
The syntax of the HV command is not standard. For a detailed description of commutation table loading, refer HV in the Command Reference Manual.

10 The current controller

This section describes the current controller and its parameterization. In addition, it describes the current limiting process and the amplifier protections.
The commands relevant to the current controller are:

Command	Description
TS	Current controller sampling time
KP[1]	Proportional gain of current controller (nominal voltage)
KI[1]	Integral gain of the current controller (nominal voltage)
XP[4]	Time constant for DC power bus filter
XP[5]	Step limiter for torque command filter
XP[6]	Time constant for torque command filter
MC	Largest available amplifier current
PL[1]	Application peak current limit
PL[2]	Duration for the peak current limit
CL[1]	Continuous application current limit

Please read about the above instruction in the Command Reference Manual to complement the material of this chapter.

> This Chapter deals with currents in Amperes. The definition we use for an "Ampere" of three-phased motor current is explained in the section 2.3: "Related Software

The Harmonica energizes all the 3 motor terminals simultaneously. This means it has to control two current components simultaneously. It must control only two current components, not three, since the phase currents are linearly dependent – the sum of all the phase currents is identically zero. The Harmonica has a vector current controller. This means that the two directly controlled current components are:

- IQ: The current component that produce magnetic field in the desired direction (normally perpendicular to the rotor fixed magnet field)
- ID: The current components that produce magnetic field orthogonal to the desired direction (normally parallel to the rotor fixed magnet field)

The structure of the current controller is as follows:

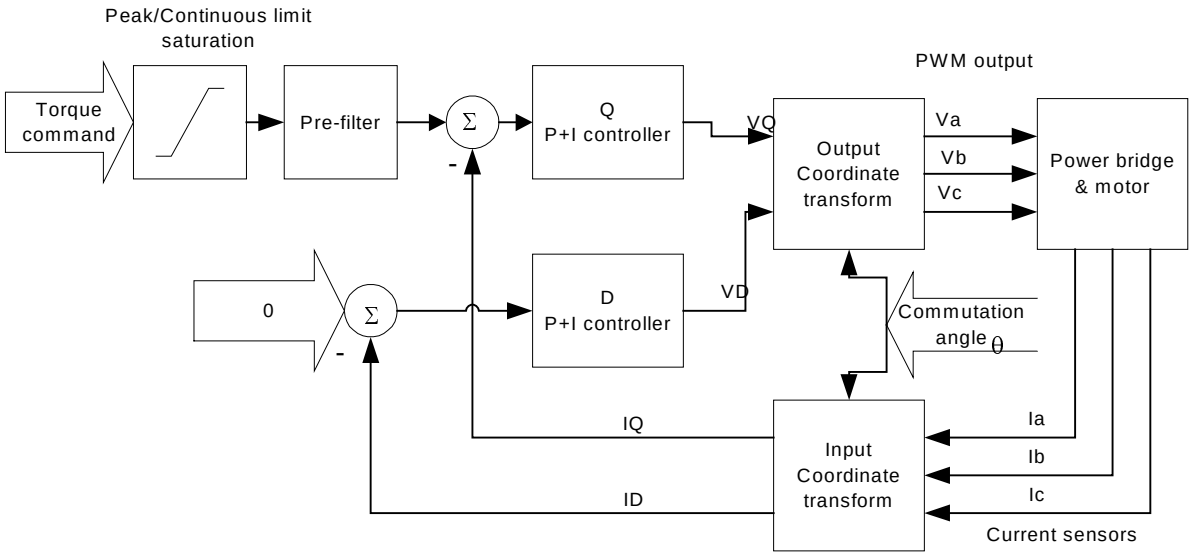


Figure 10: Current controller structure

The input coordinate transform retrieves the IQ and ID (Active and Reactive) components of the motor currents.

$$IQ = I_a h(\theta) + I_b h(\theta + 120^\circ) + I_c h(\theta + 240^\circ)$$

and

$$ID = I_a h(\theta + 90^\circ) + I_b h(\theta + 210^\circ) + I_c h(\theta + 330^\circ)$$

Above, θ is the commutation angle and $h(\theta)$ is the input winding function.

The output coordinate transform predicts the variation of the phase voltages during the motor rotation:

$$VA = V_Q g(\theta) + V_D g(\theta + 90^\circ)$$

$$VB = V_Q g(\theta + 120^\circ) + V_D g(\theta + 210^\circ)$$

$$VC = V_Q g(\theta + 240^\circ) + V_D g(\theta + 330^\circ)$$

Above, $g(\theta)$ is the output winding function.

For sinusoidal motors, we have $h(\theta) \equiv \cos(\theta)$ and also $g(\theta) \equiv \cos(\theta)$.
The harmonica tabulates both $h(\theta)$ and $g(\theta)$. The tables for these commutation functions can be programmed using the HV command.

10.1.1 Current limiting

10.2 The maximum phase current of the Harmonica is given by the parameter MC. The parameter MC describes the hardware of the power stage and cannot be changed. The peak current limit for a given application is programmed to the parameter PL[1] Amperes. Please refer the definition of motor Amperage at Section 2.3: **"Related Software**

The Harmonica requires supporting software for setup, tuning, programming, and performance assessment. The support software is called “Composer”. It runs on a PC computer, under Windows.
The Composer includes the following functions (among many others):

- Terminal for direct user interface by RS232 or by CAN
- A recorder with advanced scope controls. You can observe up to 8 signals simultaneously, triggered by a selection of events.
- Setup and tuning tools:
  - Menus for entering basic application data and limits
  - Tools for associating functions to I/O connector pins
  - Automatic tuning of current controller
  - Automatic tuning for commutation
  - Manual and automatic Speed controller tuning
  - Manual and automatic Position controller tuning
- Application database maintenance
  - Save and load application database
  - Edit application parameters, with help
- Advanced IDE for user program development:
  - Editor
  - Compiler
  - Downloading and uploading of user programs
  - Debugger, with:
    - Various break point and stepping options
    - Watches for local and global variables
    - Call stack watch

The Composer software reads the personality data from the Harmonica, and can therefore

adapt to the specific amplifier model you bought.
Units".
Program PL[1] smaller then MC if you don't want to utilize the full power of the amplifier – because the amplifier is oversized with respect to the application, or because the line voltage is not enough to drive MC amperes into the motor. Do not specify $PL[1] > V_B / R_M$, where V_B is the DC motor supply voltage and where R_M is the motor resistance. We advise to choose PL[1] small enough so that at peak current there is enough voltage to drive current changes. Otherwise, at large currents the amplifier speed of response will be limited by voltage saturation.
The continuous current limit for your application is programmed by CL[1]. The parameter CL[1] cannot exceed MC/2, since greater continuous currents will risk amplifier overheating.
You may set CL[1] < MC/2 to avoid motor overheating.
We advice to select PL[1] and CL[1] as large as the amplifier and the application permit. This will extend the linear range of the amplifier to maximum and will thus optimize the servo performance.

> The parameters PL[1] and CL[1] specify limits for the current demand. At transients, the motor current may exceed PL[1] and CL[1] due to overshooting.

The decision if to limit the motor current demand to CL[1] or to PL[1] is made by the amplifier in real-time. The LC flag reports the current limiting status. The decision mechanism is depicted below.

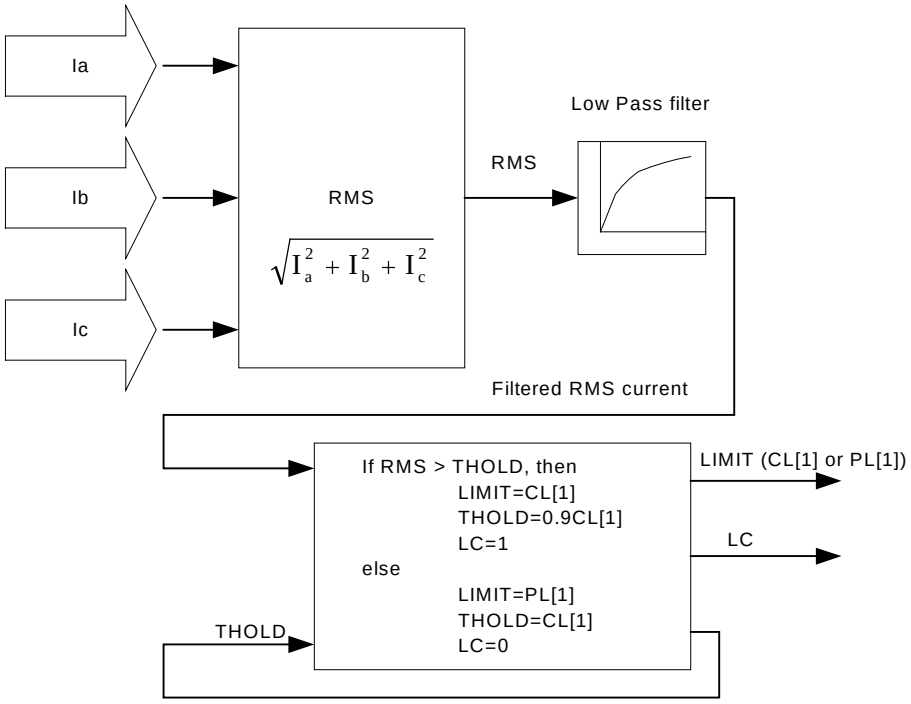


Figure 11: Peak/Continuous current limit selection

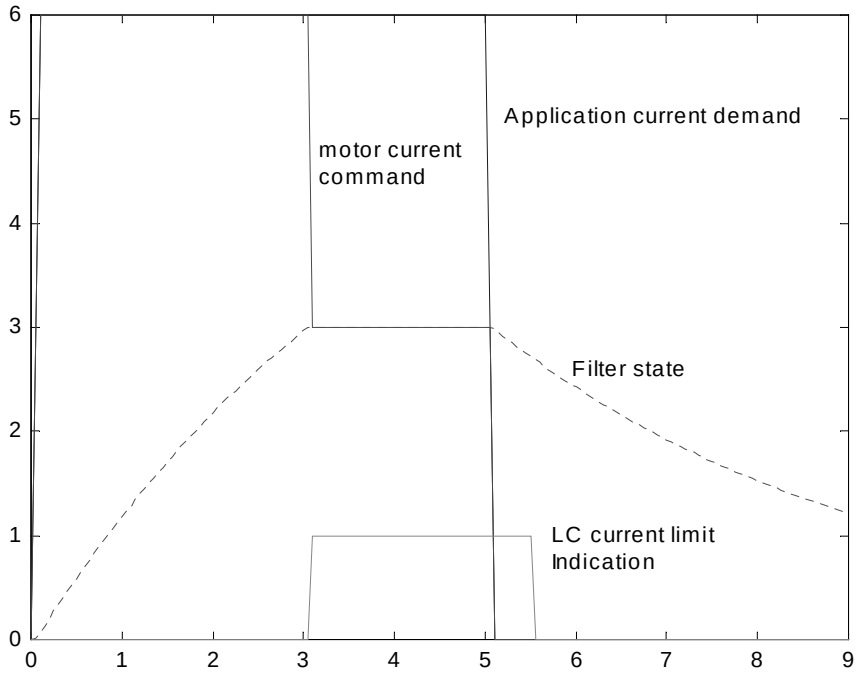
When the current limit switches, the comparison threshold (named THOLD in the figure above) is slightly changed. This hysteresis prevents to frequent switching of the current limit.
Slower time constants in the low pass filter will permit a peak current demand for longer times, but will also take more time to recover from a limiting to CL[1]. The time constant τ of the low pass filter is selected by PL[2] as follows:

$$\tau = \frac{-PL[2]}{\log\left[1-\frac{CL[1]}{MC}\right]}$$

With this selection, when PL[2] is set to MC, and after the current demand has been zero for a long time, the amplifier will permit a maximum of PL[2] seconds of peak current, and then switch to continuous current limiting.
For other settings of PL[1], the maximum time, for which the peak current can be maintained, after the current demand has been zero for a long time, is $-\log\left[1-\frac{CL[1]}{PL[1]}\right]\cdot\tau$ seconds.

The programming range for PL[2] is very limited. In most applications we recommend to leave PL[2] in its 3 seconds factory default.

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 motor current demand 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 low-pass 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.



10.2.1 The torque command filter

The torque command filter prevents that abrupt torque commands shall generate large current overshoots.
The filter is very simple, having a single time constant and a rate limiter.
The parameter XP[6] defines the time constant:

Torque command filter bandwidth = $\frac{-10^6}{2\pi \cdot TS}\log\frac{XP[6]}{32768}$ Hz where TS is the sampling time in usec.

The output of the filter cannot change in steps greater than XP[5]. The units of XP[5] are internal torque command units, not Amperes. The ratio between the internal torque commands and Amperes is given by WS[22]. The role of XP[6] for small torque commands is plotted in the figure below.

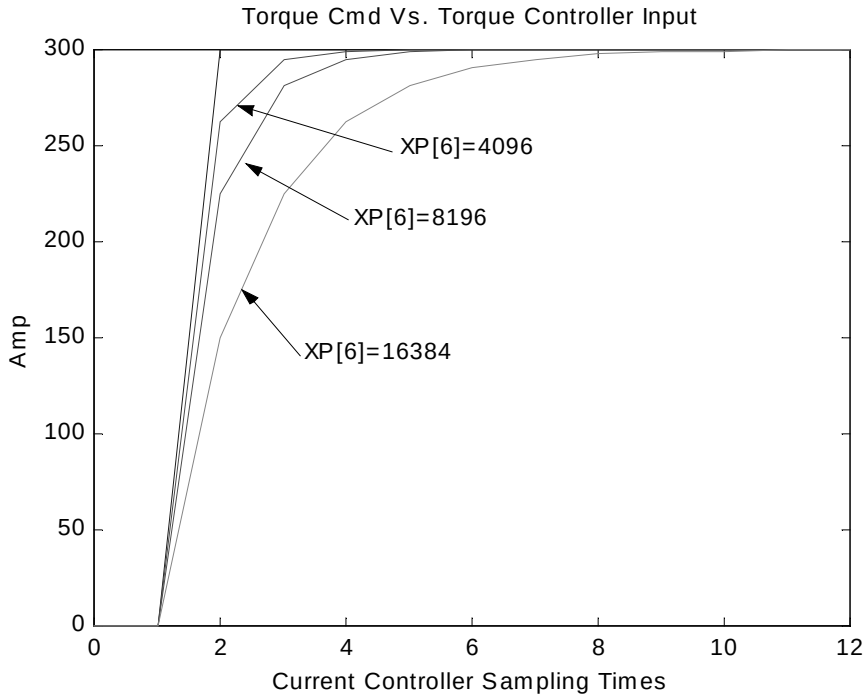


Figure 12: Time constant selection (Small signal) for the torque input filter

The automatic current tuner of the Composer program optimizes the gains of the current controller for XP[6]=16000.

The parameter XP[6] is set to larger values for very high inductance motors, for which the current cannot rise rapidly, or if additional low-passing is required.

The role of XP[5] is shown in the figure below. In the example, the torque command is 5000 internal units (5000/WS[22] Amperes). XP[6] is set to its default of 16000.

XP[5] is factory-set to 1000.

Use lower values of XP[5] for very high inductance motors, for which the current cannot rise rapidly.

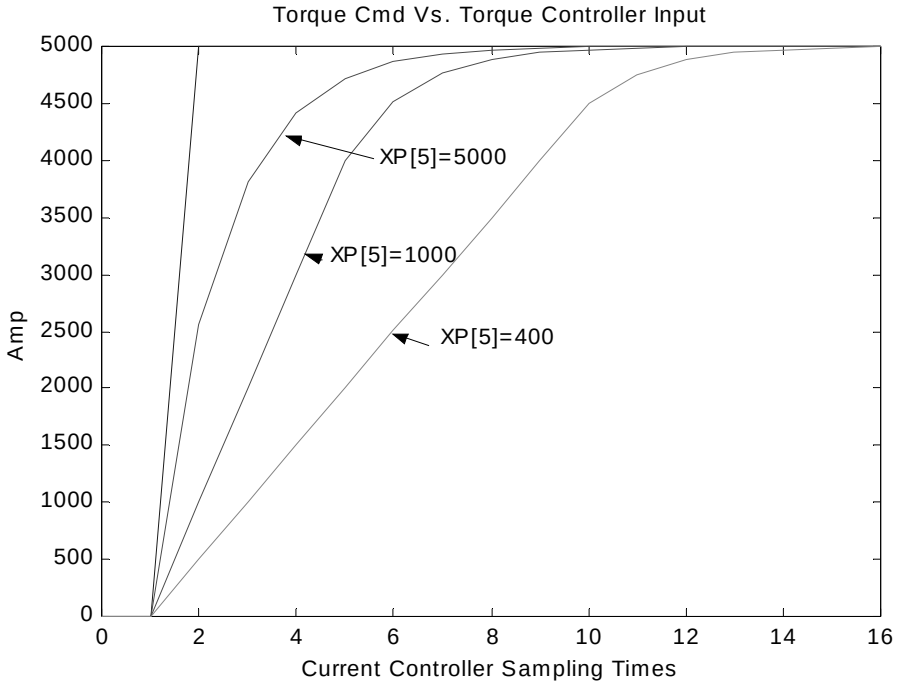


Figure 13: Rate limiting for the torque input filter

We recommend not to change the factory setting of XP[5] and XP[6] in standard applications.

10.2.2 The PI current controller

The controllers for the IQ and the ID components are similar. Their block diagram is given below.

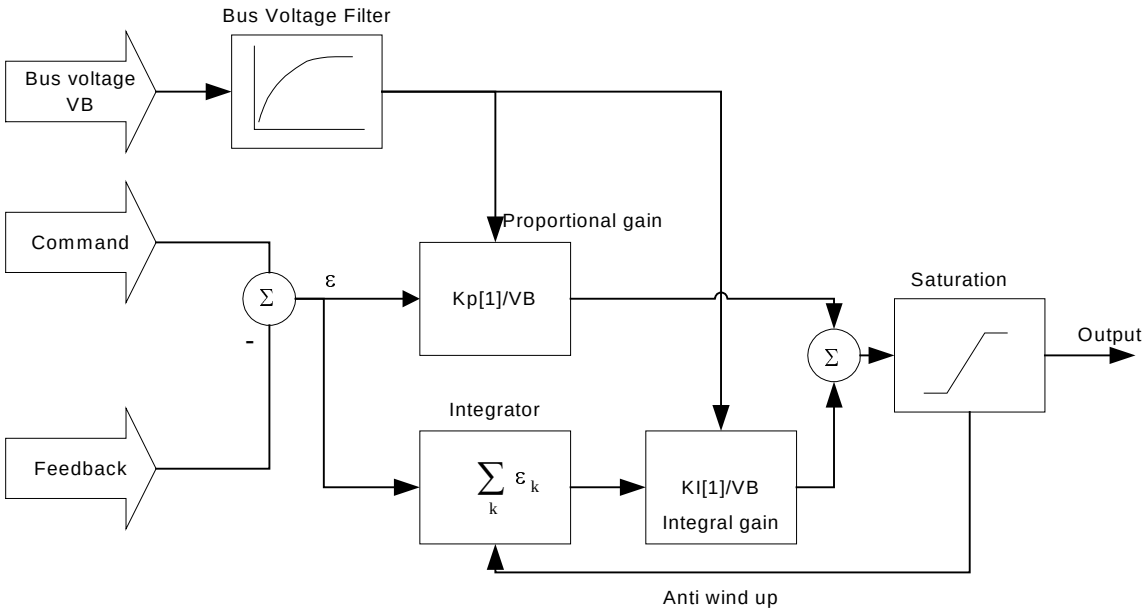


Figure 14: Current PI controller

For the IQ and the ID controllers, the Inputs/Outputs have the following roles:

Parameter	Q controller	D controller
Bus voltage	Measured & filtered motor power DC voltage	
Command	Torque command	0

Feedback	IQ	ID
Output	VQ	VD

The saturation is given by $0.5TS/25\cdot10^{-3}$, where TS is the current controller sampling time in usec and $25\cdot10^{-3}$ is the period of the 40mHz PWM generator clock in usec.
The division of the proportional and the integral gains by the DC voltage is since the output of the current controller is PWM duty cycle.

The PWM duty cycle sets the corresponding motor terminal voltage to
Motor terminal voltage = (PWM duty cycle)$\times$(DC power voltage)
It is seen from the equation above that the uncertainty in the DC power voltage acts as a gain uncertainty for the current controller.
The DC power voltage varies a lot – it goes down in high current because of the power supply output impedance. It increases when upon braking the motor acts as a generator.
The division in the bus voltage makes the output of the controller proportional to the physical motor voltage, eliminating the uncertainty.
The bus voltage is filtered, to avoid too rapid changes of the current loop PI parameters. The bus voltage filter is a simple low-pass, with a bandwidth of $\frac{-10^6}{8\pi\cdot TS}\log\frac{XP[4]}{32768}$ Hz.
The relation between XP[4] and the time constant of the filter is calculated in the table below for several values of XP[4], and TS=50.

XP[4]	Bus voltage filter bandwidth, Hz
31750	25Hz (High impedance supply)
30720 (factory set)	50Hz
28900	100Hz (Low impedance high ripple supply)

Table 10-1: Bandwidth selections for bus voltage filter

XP[4] is factory set to 30720.
Set higher XP[4] if the output impedance of your power supply is too high, causing interplay between the filter gains and the DC motor voltage.
Set lower XP[4] if you have very low impedance power supply, but high ripple voltage.
In normal applications, we recommend not to change the factory setting of XP[4].

10.3 Current amplifier protections

The current amplifier and the power stage have several protections, detailed in this section. When one of the protections is activated, the amplifier shuts immediately. If brake action is defined (Refer the OL[] and BP[] commands), the brake is immediately activated. In the next 10msec the amplifier will not set the motor on, even if instructed to do so.
You can know that an amplifier has been shut down by a protection by:

- Observing the Motor-On and the Fault bit in the SR report, or in the CAN special status object.
- Mapping the Motor Fault event to an event-driven CAN PDO.
- Polling the MO variable (drops to 0 on exception)
- Polling the MF variable (Reports a non zero value after an exception has been trapped).
- Mapping a digital output to AOK – refer the OL[] command.

You may install in your user program an AUTO_ERR routine to automatically respond an exception shutdown.

The protections are:

Protection	MF reports	Reason
Over voltage	0x5000	The voltage of the power supply is too high, or the servo drive did not succeed in absorbing the kinetic energy while breaking a load. A shunt resistor may be required. The over voltage threshold differs with the power stage

		model – please refer product user manual.
Under Voltage	0x3000	The power supply is shut down, or it has too high output impedance. The under voltage threshold differs with the power stage model – please refer product user manual.
Short circuit	0xb000	A large, fast current pulse has been detected. The motor or its wiring may be defective, or a faulty amplifier.
Over-Temperature	0xd000	The power stage of the amplifier is too hot. Additional heat sink or forced cooling is required.
Over current	0x8	Unexpected large current – $1.15 \cdot MC$ has been exceeded. The current loop or the winding shape function HV[] may need better tuning.

11 Unit Modes

The feedback structure of the amplifier can be arranged in several options. Those options are called "unit modes" and programmed by the parameter UM. Switching the unit mode is possible only with the motor off, since the feedback structure need be re-arranged.
The following unit modes are available:

Value	Description (Related commands)
1	Torque control mode.
2	Speed control mode.
3	Micro Stepper mode.
4	Dual feedback position control.
5	Single feedback position control.

11.1 Torque control: Unit mode 1

In this mode the amplifier controls the motor torque only.
In this mode the amplifier may serve as a torque driver, controlled by an external controller.
Commutation is performed to get the maximum torque for the commanded motor current.
If position sensors are connected, the position and the speed are evaluated, and the amplifier will abort upon over-speed, as specified by the parameters LL[2] and HL[2]
The torque command may be set either by a software command, an analog input, or a combination thereof, according to the figure below:

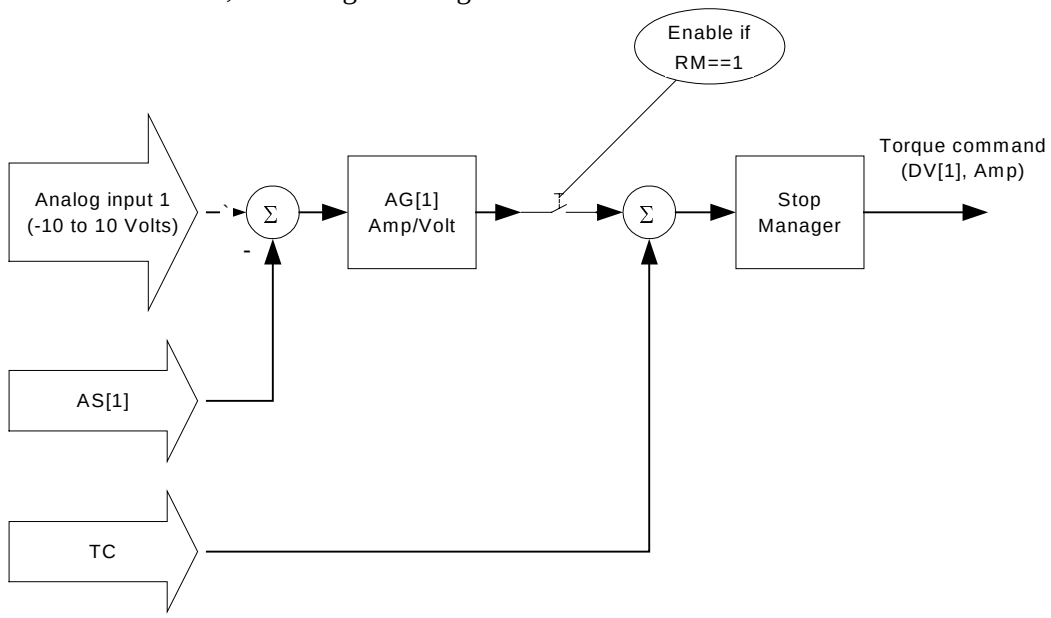


Figure 15: Unit mode 1 (Torque) structure

The AS[1] parameter compensates for possible offsets in the driving equipment and internally in the Harmonica.
If you are not using the analog input for torque command, set AG[1] =0 or RM=0, to avoid that noises and offset will affect the amplifiers torque command.
The combined (software and analog input) current demand is reported by DV[1].
The Stop Manager does as follows:

- If hard-stop is active, the torque command is set to zero
- If RLS is active, negative torque commands are clipped to zero
- If FLS is active, positive torque commands are clipped to zero.

The Stop-Manager does not affect the reference generator, and when the relevant switch is released, the torque command restores to the value set by the reference generator.

11.2 Speed mode: Unit mode 2

In this mode the amplifier controls the motor speed by feedback. The speed controller demands torque from the current controller.
The reference to the speed controller is summed from a software commands, and an auxiliary speed command. The auxiliary speed command is derived using the analog input, the auxiliary encoder input, and the ECAM table – details are given below.
The Stop digital input and the limit switches (RLS,FLS) can be used to stop or to limit the direction of the motor – see the paragraph on stop management below.
The amplifier will abort upon over-speed, as specified by the parameters LL[2] and HL[2].
The speed control scheme is depicted below:

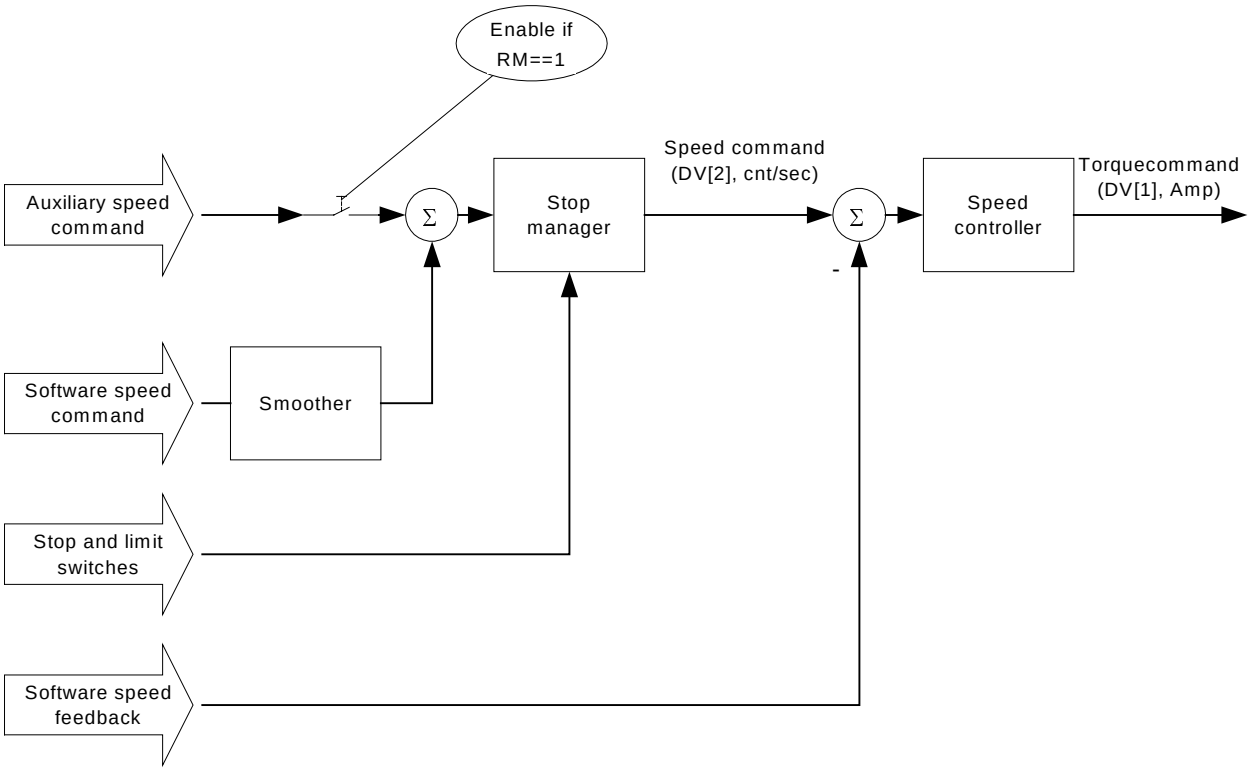


Figure 16: Unit mode 2 (Speed) structure

If you are not using the analog input for torque command, set RM=0, to avoid that noises and offset will affect the amplifiers speed command.
The DV[2] command reports the combined (software and analog input) speed demand.

11.2.1 The software speed command

The software speed command generator generates speed commands, subject to acceleration, deceleration, and to speed limits. The following commands are relevant to the generation of the software speed command:

Command	Description
AC	Profiler acceleration limit. Units counts/sec^2.
BG	Begin command
DC	Profiler deceleration limit. Units counts/sec^2.
EM	EM[1]=1 use ECAM table, EM[1]=0 do not use
JV	Profiler final speed command. Units counts/sec.
PM	Set PM=0 to remove the acceleration limits from the software speed command. PM=0 is used by the Composer program in the process of tuning the speed controller, but in normal operation we advise not to program PM.

Command	Description
SF	Smooth Factor: The time in milliseconds required to develop the full acceleration of AC and deceleration of DC.
ST	Stop command, activate profiler stop mode with Deceleration=SD
VH[2]	Maximum speed command. Units counts/sec.
VL[2]	Minimum speed command (bound on speed command from below, negative value). Units counts/sec.
IL[]	Map functions to digital inputs: Digital inputs may function as hardware ST or BG instructions.

The algorithm of the acceleration-limited software speed command is listed below:

1. Any new BG command accepted by software or by hardware? If yes, update the speed target to the value to JV, and also update the permitted acceleration and deceleration to the values of AC and DC.
2. If the speed target and the speed command are positive, and the speed target is greater than the speed command, select AC for the acceleration limit. If the speed target and the speed command are negative, and the speed target is less than the speed command, select also AC. Otherwise select DC for the acceleration limit.
3. Advance the speed command towards the speed target as much as the selected acceleration/deceleration permits. In the trivial case where the speed command already equals the speed target, do nothing.

The acceleration limited software speed command is fed to a smoothing filter.
The smoothing filter limits the rate of developing the full acceleration or deceleration.

Example:
This example demonstrates the concepts of target speed and speed command. It also demonstrates the AC and DC acceleration limits.
Let MO=1; JV=4000; AC=100000; DC=200000; SD=10^6; PM=1; RM=0; SF=0; BG;
The figure below depicts how the speed command to the controller tracks the target speed specified by changing the JV parameter, followed by a BG.

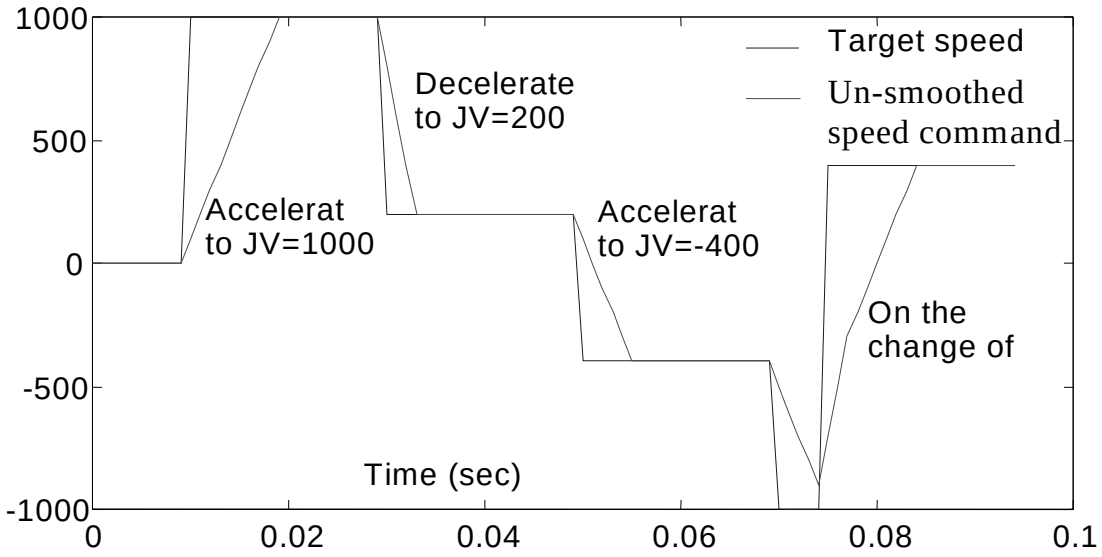


Figure 17 – Speed Profiling using JV, AC and DC

Note: The speed reference to the controller may be changed any time, regardless of the state of the profiler. In Figure 17 the required speed was changed at time=0.075 from –1000 to +300 before reaching the speed output 1000.

Example:
This example demonstrates the smoothing filter and the smoothing factor SF.
Let us MO=1; JV=4000; AC=100000; DC=100000; SD=10^6; PM=1; RM=0; BG; with three different values of SF.

1. The SF=0 graph displays sharp corners, since smoothing is ignored, therefore non-continuity of acceleration is allowed.
2. The SF=10 graph takes 10 milliseconds more to stabilize the speed software command, however the speed reference profile is much smoother.
3. For the SF=50 graph, the smoothing is so strong with respect to the total acceleration time that the AC acceleration is never reached.

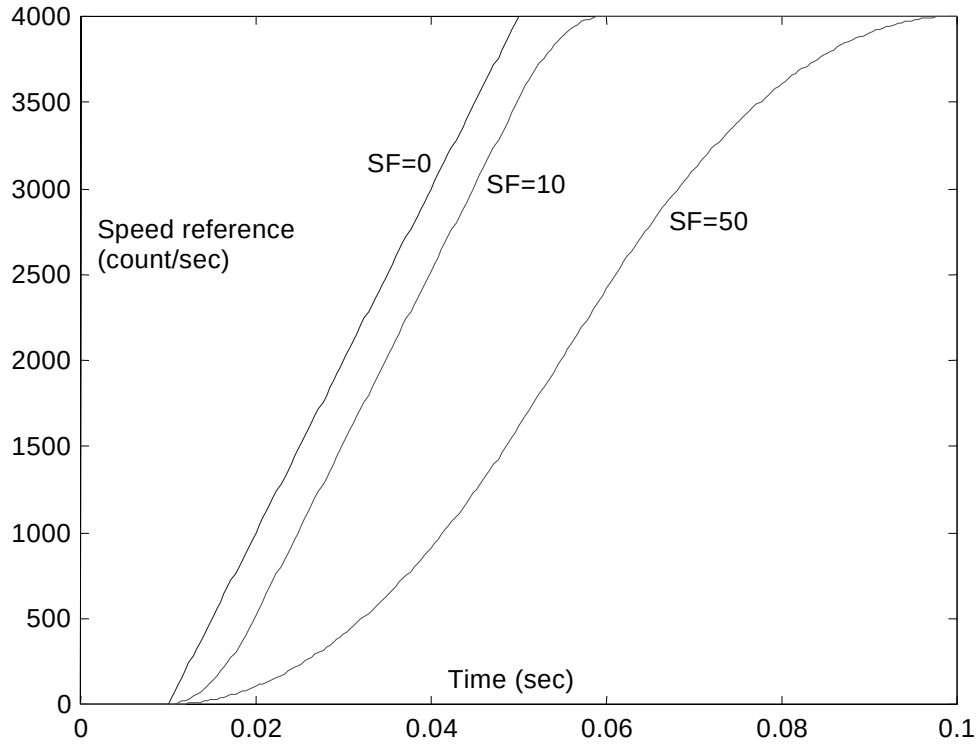


Figure 18 – Speed command for different smooth Factor

Note: The profiler smoothes the motion and decreases (may also prevent) overshoots, while introducing a delay in the controller response**.**

11.2.2 The auxiliary speed command

The auxiliary speed reference is generated according to the block diagram below. The parameters relevant to the auxiliary speed command generation are:

Command	Description
AG[2]	Analog input gain, counts/sec/Volt.
AS[1]	Analog offset
FR[2]	Follower gain
RM	Reference mode – 1 to use the auxiliary speed command, 0 to null the auxiliary speed command.

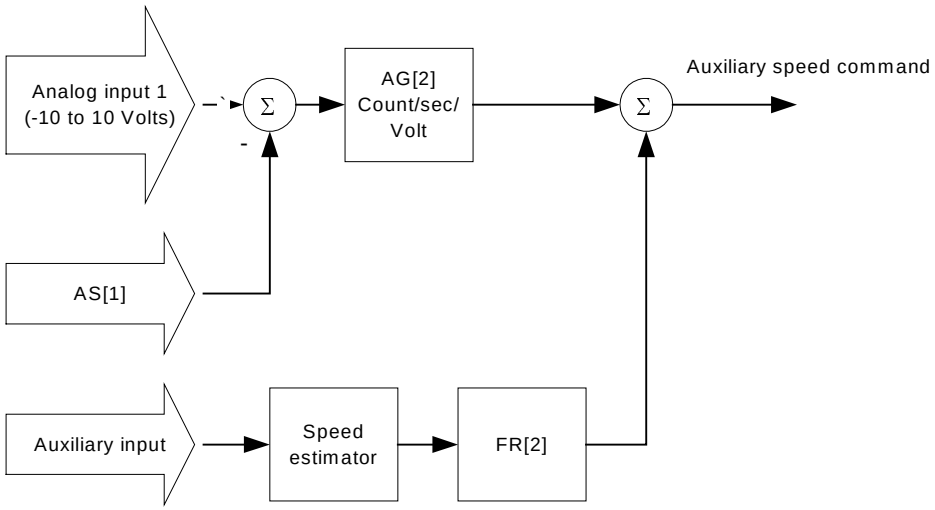


Figure 19: Auxiliary speed command generation

The analog input is most useful when the Harmonica serves as an inner controller, embedded in an external control loop.
The auxiliary encoder speed input enables to issue speed commands relative to a conveyor or other moving object.

11.2.3 Stop management

The stop manager does the following functions:

- Bring the motor to stop upon software or hardware ST command.
- Bring the motor to stop when the speed demand is positive and FLS (Forward Limit Switch) is active.
- Bring the motor to stop when the speed demand is negative and RLS (Reverse Limit Switch) is active.
- Prevent acceleration or deceleration beyond the motor torque limits.

The parameters relevant to the stop manager are:

Command	Description
SD	Maximum motor acceleration/deceleration, counts/sec^2
LV[2],HV[2]	Speed command limit
IL[]	Input logic – define digital inputs as hard-stop, or as directional limit switches (RLS,FLS).

A block diagram of the stop manager is given below.

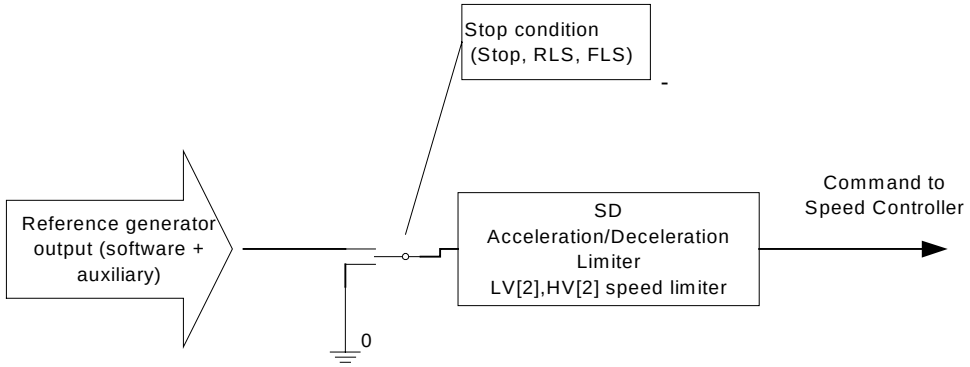


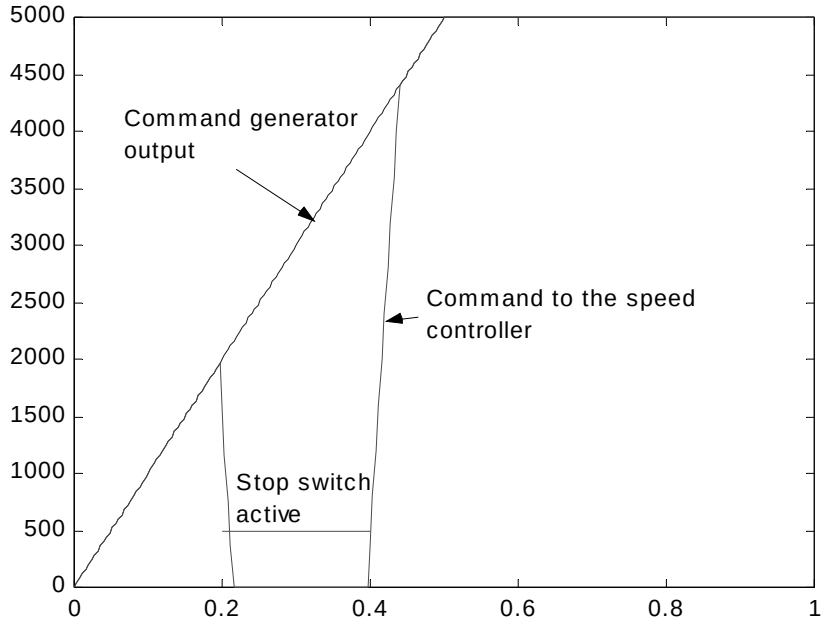
Figure 20: Speed mode Stop Manager

The Stop-Manager prevents the speed-controller command from changing abruptly by

limiting the rate of reference change to SD counts/sec^2.
When the Stop Manager stops the motor due to a switch action, the reference generator is replaced by a zero command at the input to the Stop Manager. The Stop Manager uses the SD parameter to decelerate the motor command from the output of the reference generator to a complete stop.
When the switch action terminates (for example a "Stop" switch is released) the Stop-Manager uses the SD acceleration to recover the speed command to the output of the reference generator.
Note the following:

1. The Stop-Manager must limit the speed command to the range [LV[2]..HV[2]]. Although both the software speed command and the external speed command are each limited to the legal range, their sum may be out of range.
2. When the stop manager stops the command to the speed controller, it does not affect the reference generator. The reference generator continues to behave normally. When the switch action is over, the Stop-Manager will try to recover the speed controller command to the output of the reference generator.
3. The Stop-Manager limits the decelerations and the accelerations due to abrupt changes of the auxiliary reference. The SD parameter is the only acceleration limiter for the auxiliary input. The AC and the DC parameters are only relevant to software commands.
4. When AC and DC are greater than SD, they become irrelevant, since SD will limit the acceleration lower.
5. The IL[N] command can program a variety of "Stop" options to an input pin. A pulse at the pin can be directed to the stop manager as in the example below, stop the software reference generator, or both.

Example:
Let, SD=100000,MO=1,AC=10000,DC=10000,JV=5000,BG;
At the time of 0.2sec, a switch programmed as hard-stop is activated, and released at the time of 0.4 sec.
The following waveforms result:



When the stop switch is applied, the speed command is brought to complete stop, decelerating with 100000 counts/sec^2. When the switch is released, the acceleration of 100000 counts/sec^2 is used to recover the output of the reference generator.

11.3 The stepper mode: Unit mode 3

The stepper unit mode enables the rotation of a motor without feedback control. The motor field is rotated to the desired direction, and the rotor magnet is believed to follow. The user must not rotate the field too abruptly in order that the rotor will be able to track its desired direction. If the rotor misses a full electrical revolution, it will be attracted to a wrong electrical equilibrium, with no feedback to correct that.
Moreover, if the field rotation is stopped abruptly, the motor will not properly brake. If the rotor misses half an electrical revolution, it will start accelerating – and the net braking torque sum over an electrical revolution is zero.
Specifying higher motor current enables larger accelerations and decelerations, but also lose more power at the steady state.
In the stepper mode, most of the time the stator field and the rotor magnet are near equal, and the motor efficiency is very low.
The Harmonica uses the stepper mode mainly for safe testing and controller tuning before any feedback control is tuned. The Harmonica can serve as an advanced micro-stepper driver, with 1024 micro-steps per electrical revolution. The main limitation is that the Harmonica is a 3-phase driver, while most the stepper motors in the market are two-phased.
The block diagram of the micro-stepper mode UM=3 is depicted below:

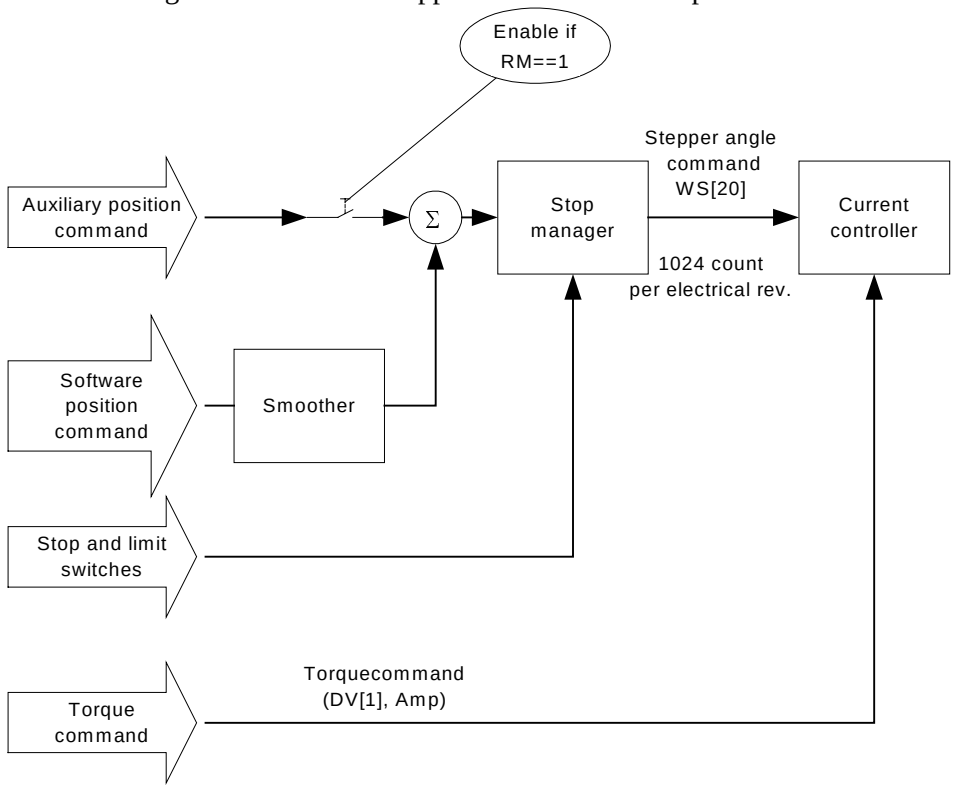


Figure 21: Stepper mode (UM=3)

The stepper angle command is generated by:

- The position software command generator
- The position auxiliary command generator
- The position Stop Manager

The generation of the stepper angle command is similar to the position command generation in UM=5, and will be described in "Stop management" below.
The torque command is simply given by the command TC.

11.4 The Dual feedback mode: UM=4

The dual feedback mode is used when different sensors are used for speed/commutation and for position. This is a common situation if the motor drives the load through a reduction gear. The controlled position is that of the load. The load position, however, may not be good for commutation or speed feedback, since:

- The commutation accuracy will be limited by backlash and gear compliance
- The motor speed can be measured with better resolution and less delay because the motor rotates much faster than the load. In addition, the speed sensor is not subject to dead zones caused by backlash.

The block diagram of the dual feedback mode UM=4 is depicted below:

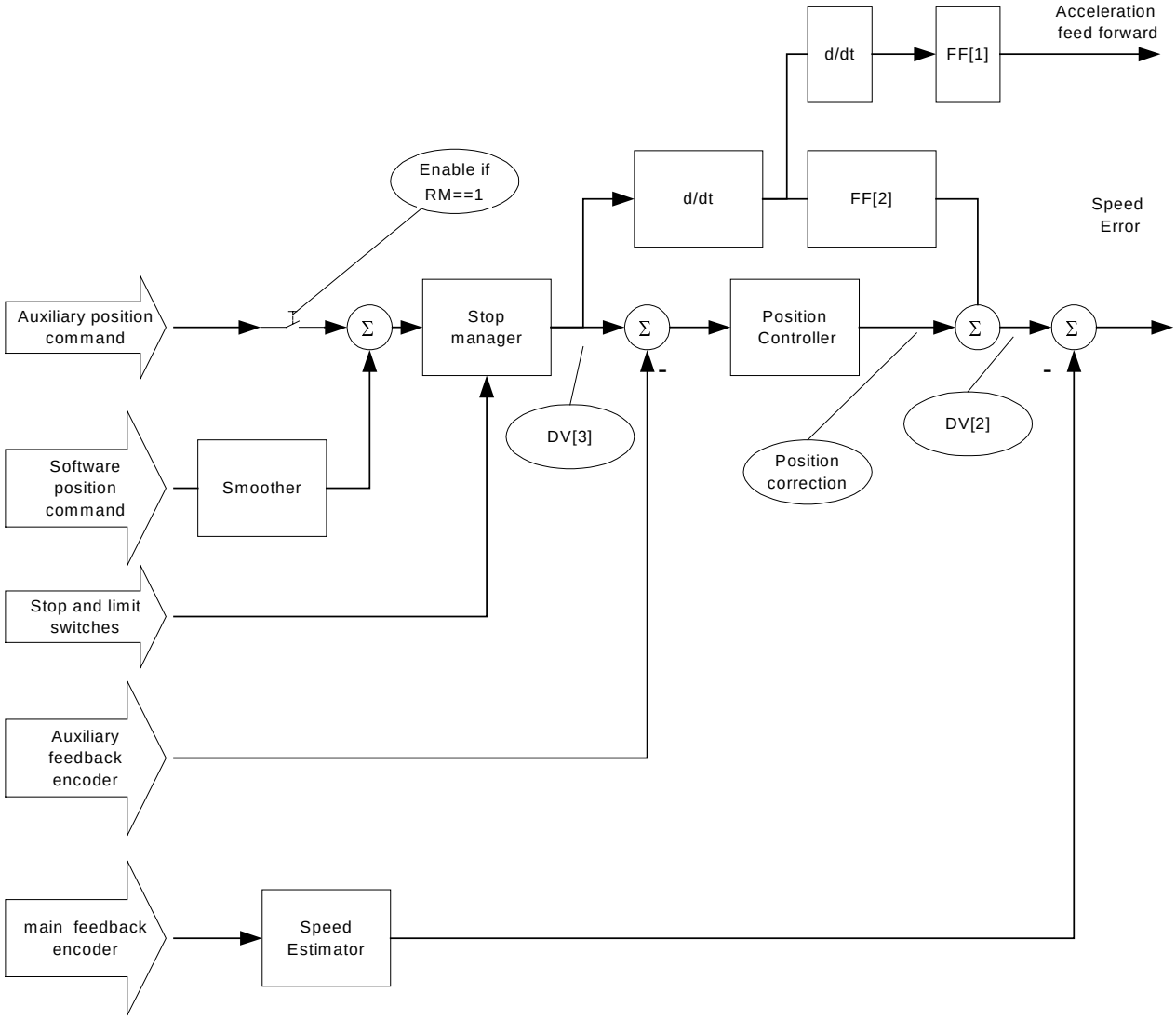


Figure 22: Dual feedback mode (UM=4)

The position and the speed commands are generated by:

- The position software command generator
- The position auxiliary command generator
- The position Stop Manager

The generation of the position and speed commands is similar to the position and speed commands generation in UM=4, and will be described later the chapter on "The position reference generator".

The speed command, multiplied by the gain FF[2], is fed as reference to the speed controller in addition to the position correction. When FF[2] is set exactly to the gear ratio between the position sensor and the speed sensor, there will be no steady state constant-speed tracking error.

The acceleration of the position command, multiplied by FF[1], can be injected directly as torque command. By default, FF[1]=0.

The reference values to the position, speed, and torque controllers can be inquired by DV[3],DV[2], and DV[1] respectively.

The details of the speed and the position controllers are given in the chapter "**Error! Reference source not found.**".

11.5 The single feedback mode: UM=5

The single feedback mode is used when the same sensor is used for speed, commutation and for position. This is a common situation where only one position sensor is installed in the system.

The block diagram of the single feedback mode UM=5 is depicted below:

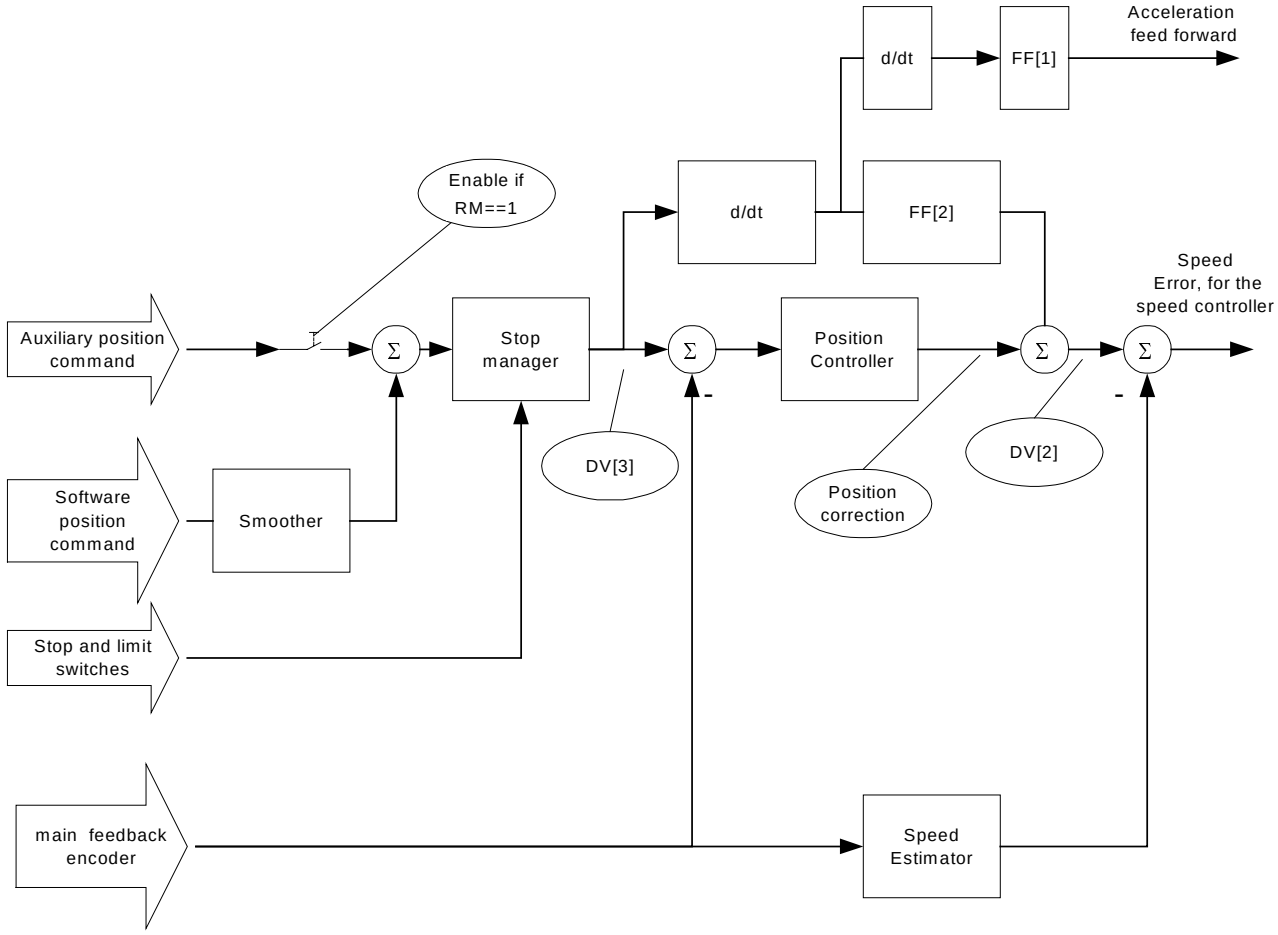


Figure 23: Dual feedback mode (UM=5)

The position and the speed commands are generated by:

- The position software command generator
- The position auxiliary command generator
- The position Stop Manager

The generation of the position and speed commands is described in the chapter on "The position reference generator".

The speed command, multiplied by the gain FF[2], is fed as reference to the speed controller in addition to the position correction. When FF[2] is set exactly to unity, there will be no steady state constant-speed tracking error.

The acceleration of the position command, multiplied by FF[1], can be injected directly as torque command. By default, FF[1]=0.

The reference values to the position, speed, and torque controllers can be inquired by DV[3],DV[2], and DV[1] respectively.

The details of the speed and the position controllers are given in the chapter "**Error! Reference source not found.**".

12 The position reference generator

The position reference signal is generated by the following components:

- Software command generator
- Auxiliary command generator
- Stop manager.

This section describes the components of the position reference generator in detail.

12.1 The software reference generator

The Harmonica supports five modes of software references.
The reference modes are:

- Idle – The motor just stands at place. This is the default mode after setting motor on, and after an interpolated motion is complete.
- PTP – Point to Point: The user specifies the position in which the motor is to stand, and the limits subject to which the trajectory to the target is to be designed. The Harmonica designs automatically a minimum-time trajectory and executes it.
- Jogging: The user specifies the speed in which the motor should steadily move, and the acceleration limits towards that speed. The Harmonica designs automatically a minimum-time trajectory and executes it.
- PT – Position/Time: The user specifies a set of point to be visited with fixed time differences. The Harmonica interpolates a 3rd order polynomial between the user's points. The speeds at the user points are selected to form a smooth motion trajectory. Using the PT, complex motions are easily designed. PT data may be transferred to the amplifier on-line using an efficient CAN PDO protocol, so that infinite PT motions are possible.
- PVT – Position/Velocity/Time: The user specifies a set of motion points. Each point includes a position, and the speed and the time at which the position is to be visited. The user points are interpolated using third order polynomials. The PVT mode allows absolute time specification, thus several amplifiers may compose a fully synchronized motion using the PVT mode. PVT data may be transferred to the amplifier on-line using an efficient CAN PDO protocol, so that infinite PVT motions are possible.

In order to initiate a software motion, you must initialize its parameters and issue a mode command. The next BG (software, hardware, or timed) will start the motion.
The mode command are listed in the following table:

Command	Mode	
ST	Idle.	ST stops any motion.
PA	PTP.	PA=n specifies a PTP motion, to the absolute position n counts
JV	Jogging.	JV=n specifies a Jog motion, with the speed n counts/sec
PT	PT.	PT=n specifies a PT motion, beginning from the n'th item in the PT data table
PV	PVT.	PV=n specifies a PVT motion, beginning from the n'th item in the PVT data tables

Table 12-1: Software motion mode commands

12.1.1 Switching Between Motion Modes

Stopping by the ST command is available anytime. The amplifier will decelerate the position reference until a complete stop.
PTP and jog motions can be commanded to anywhere, any time. The Amplifier will calculate the path to be followed so that the desired speed or position is reached, subject to the acceleration limits. PTP and jog motions can even be initiated while PT or PVT motion

is on. Care must be taken if the smooth factor (SF) is nonzero. Upon switching from interpolated motion to PTP or jogging, the PTP or the Jogging will start un-smoothed. Smoothing will gradually build up. After SF milliseconds, the motion shall be fully smoothed.

Care is required, however, when switching to the tabulated (PT and PVT) motion modes. The PV and the PT modes should be started only when the motor is in complete stop (MS=0 or MS=1), at the exact position specified as the first point for the PT or for the PVT. If the motor is not in complete stop at the correct starting position, the software reference shall jump. The command to the position controller will try to catch up with the PT or the PVT motion, using the SD acceleration – refer the section on "Stop management".

12.1.2 Comparison of the PT and the PVT interpolated modes

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	Specified 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.	Specified 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 12-2 – Tabulated Motion Difference

Feature	Preferred
Long pre-programmed motions	PT
Ease of use	PT
Variable command sampling time	PVT
Motion design independent of controller sampling time	PVT
Synchronized, multiple axis motions	PVT

Table 12-3 – Tabulated Feature Preference

12.1.3 The Idle Mode and Motion Status

After motor on, the position profiler is Idle. This means that the software position command is fixed. The idle mode is revisited whenever a mode terminates (A PTP or a tabulated motion is completed), or by a stop (ST) command.
The Idle mode is the only mode in which it is possible to change the parameters of the external reference generator.
The Idle mode of the position reference generator may be identified by the MS (Motion Status) variable. The reading of the MS variable are summarized in the following table:

MS Value	Description
0	The position reference generator is idle. Moreover, the Motor position stabilized within the target radius for long enough time.
1	The position reference generator is idle, or the motor is off (MO=0).
2	The position reference generator is active in one of the optional motion profilers – PTP, Jog, PT, or PVT.

Table 12-4: Motion status indications

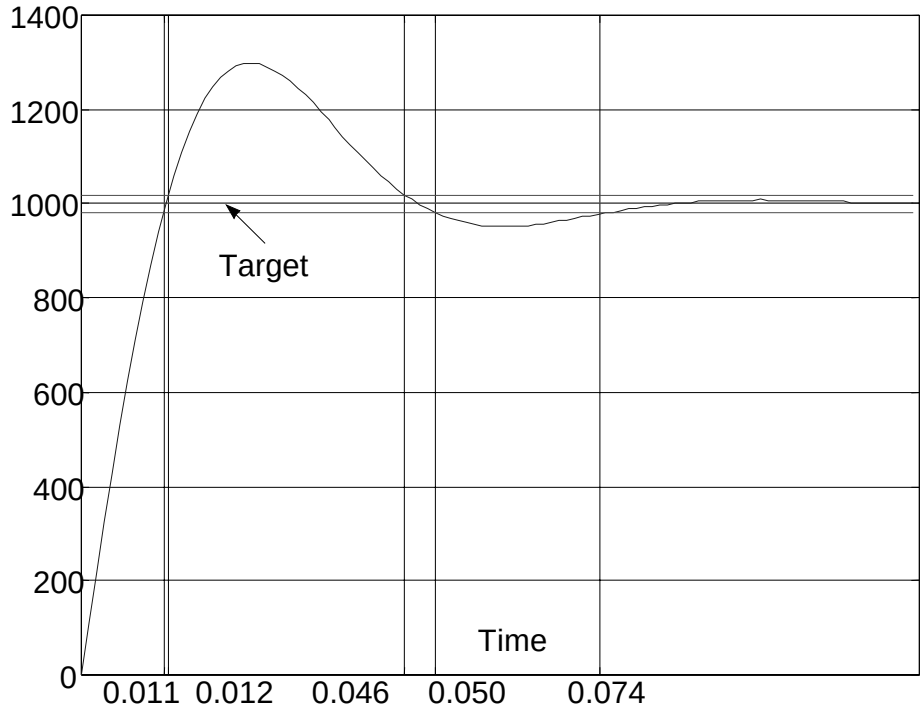
When MS is 1 or less, it is possible to switch the parameters of the external reference generator.

The condition MS=0 is stronger – it implies that the motor position is indeed stabilized.

The motor position is considered stable if the motor position is within the target radius, for the target time at least. The target radius is given by TR[1] counts, and the target time is TR[2] msec.

Example

The target time and the target radius concepts are demonstrated in the drawing below.



In this drawing we see an overshooting settling of the motor position on a target at 1000 counts. The target radius is 20. The motor position is within target radius is the range of [980-1020] counts.

For the dynamics of the example, it is reasonable to select the target time TR[2]=30. If TR[2] is selected too low, false "On Target" decisions may result. For example, if the target time is set as 3msec, a final stabilization will be concluded at the time of 0.049 sec, since during the time interval [0.046, 0.049] the motor was within the allowed position error [980-1020] counts.

12.1.4 Point-To-Point (PTP)

12.1.4.1 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 with modulo calculations, 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
PR	Relative position in counts
PA	Specifies that the next motion shall be a PTP, and the absolute target position.

Table 12-5 – Parameter of PTP Motion

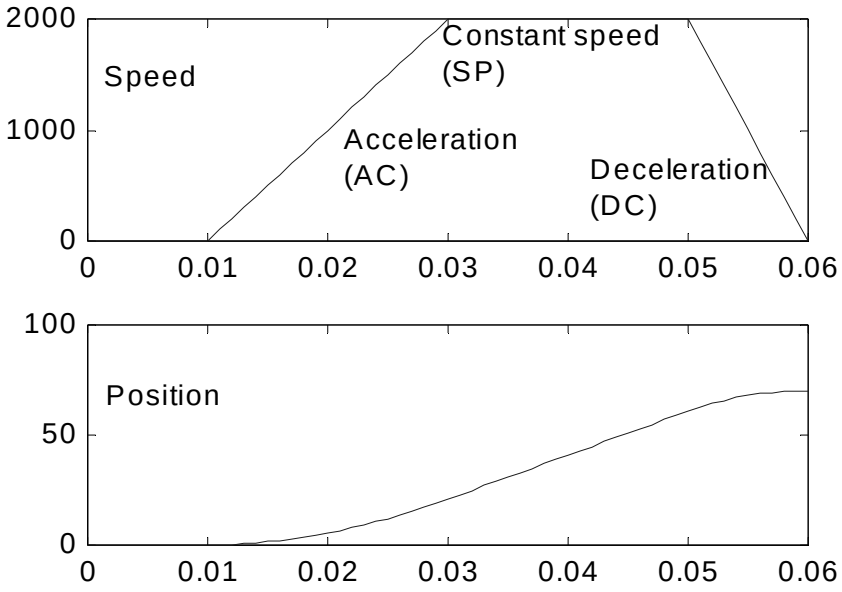
The PA command has more than one role. PA=n specifies that the next BG will start a PTP motion. In addition, it specifies the target of the next PTP move.
When in the PTP mode, a BG command performs the following algorithm:

- PA=PA+PR
- Go to PA

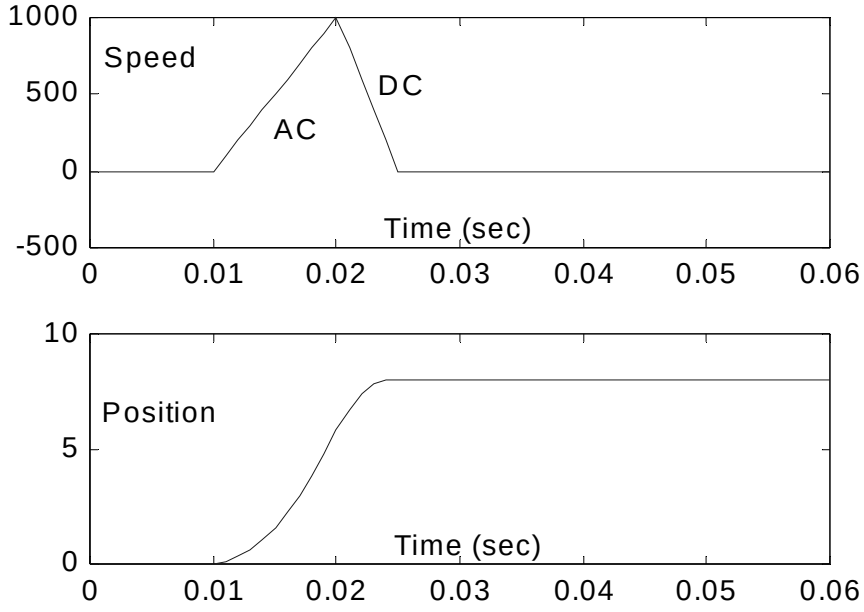
PA is the absolute position target.
PR is the relative position target. The PR parameter enables the specification of an incremental move, or a series of incremental moves. PR is reset to zero by the PA=n command. Therefore, PA=**n**; BG initiates a motion towards **n**.
The value of PA is incremented automatically with PR every time a new motion is initiated.
For example, the sequence
PA=0;BG;while(MS==0);PR=1000;BG; while(MS==0);BG; while(MS==0);BG
Initiates four consecutive PTP motions, targeted to 0,1000,2000,and 3000 counts respectively.

12.1.4.2 Example

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



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



12.1.4.3 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.

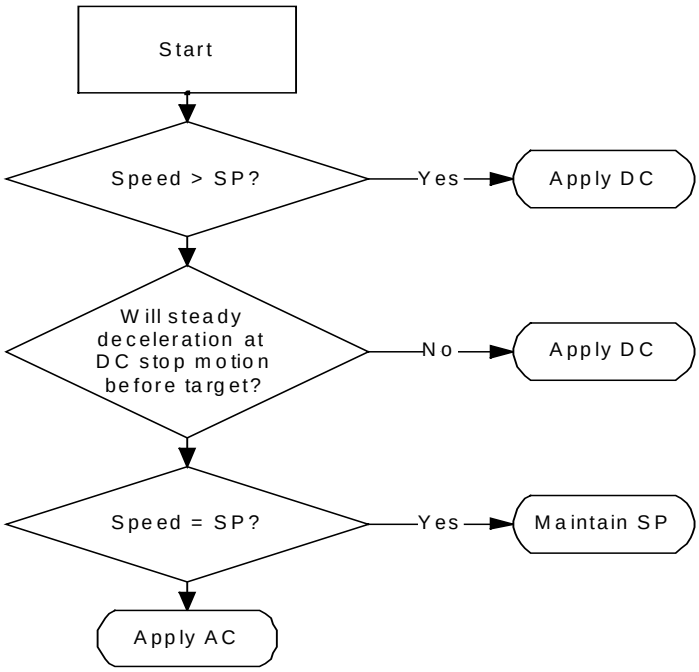
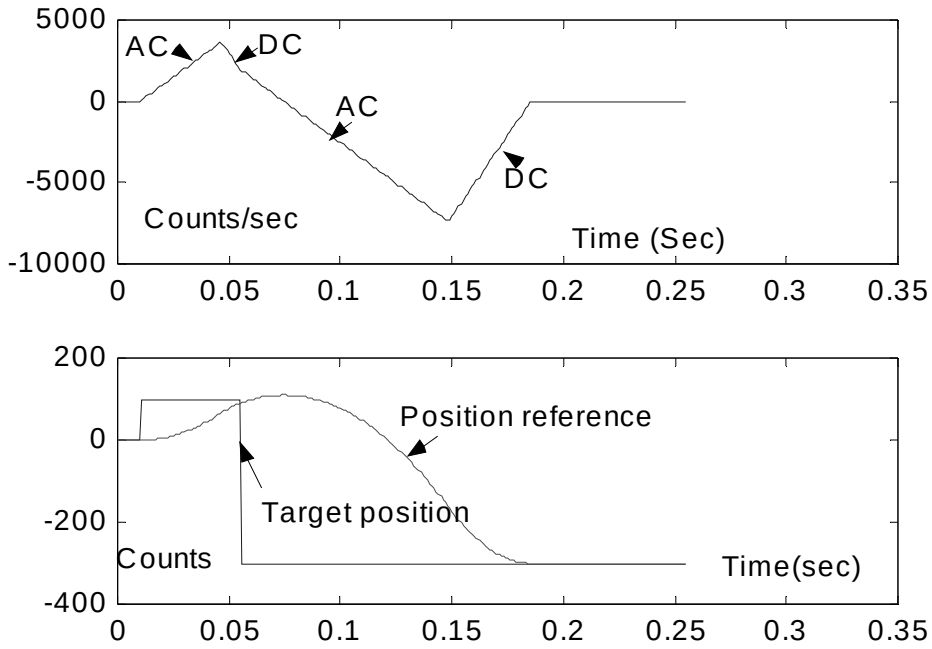


Figure 24 – PTP Decisions Flow Chart

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 equivalent.

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

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 Amplifier calculates the position reference to reach the new target.
Note that in this example the position reference overshoots the original target position of 100. This is since at the time the target position was changed, the Amplifier was approaching the target using DC. When a new target is set, the Amplifier exits the state of final approach. It therefore uses the AC acceleration, which in this example is smaller, than DC. With the reduced acceleration, it cannot avoid the overshoot.



12.1.5 Jogging

In a jogging motion, the motor is commanded to move in a fixed speed. The acceleration (or deceleration) to the desired speed are 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.

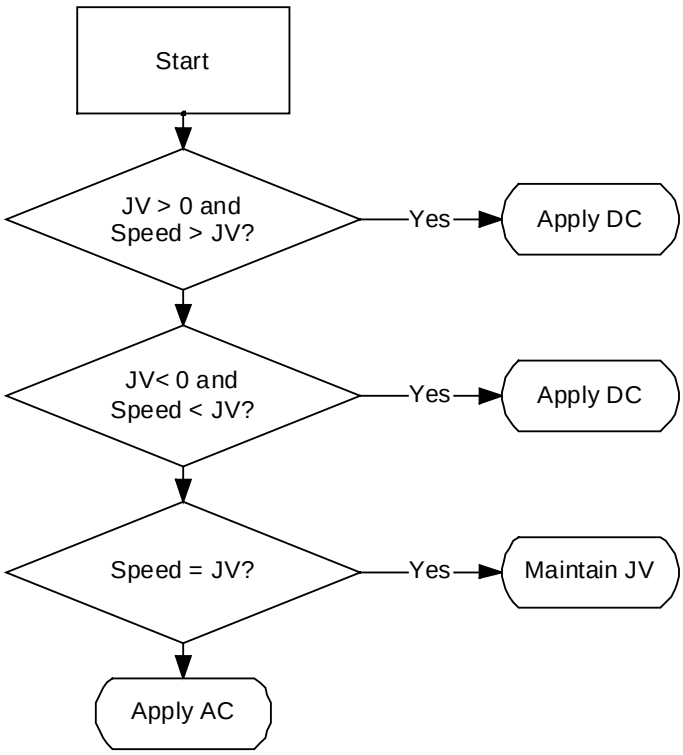


Figure 25 – Jog Decisions Flow Chart

All the parameters of the above flow chart, including AC,DC, JV, and the position target are updated by a BG command, or by its hardware equivalent (refer the IL[N] command).

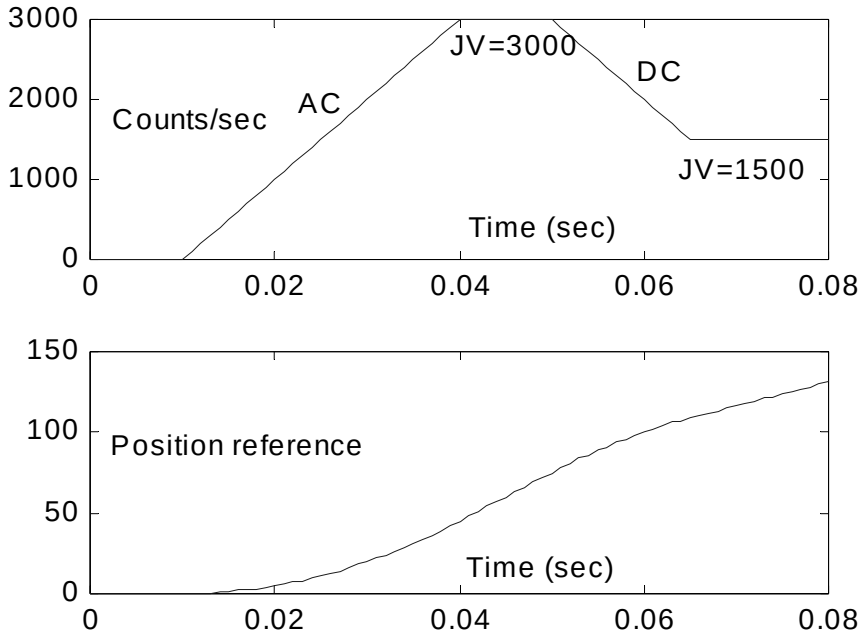
Jog motions can continue forever. The position reference jumps when it arrives the modulo boundary (+/-2^{30} for XM=0 or +/- XM/2) but the speed is kept constant
The parameters of Jog motion are summarized in the table below.

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

The JV command has double role. It specifies that the next motion will be a Jogging, and also to what speed the jog will target.

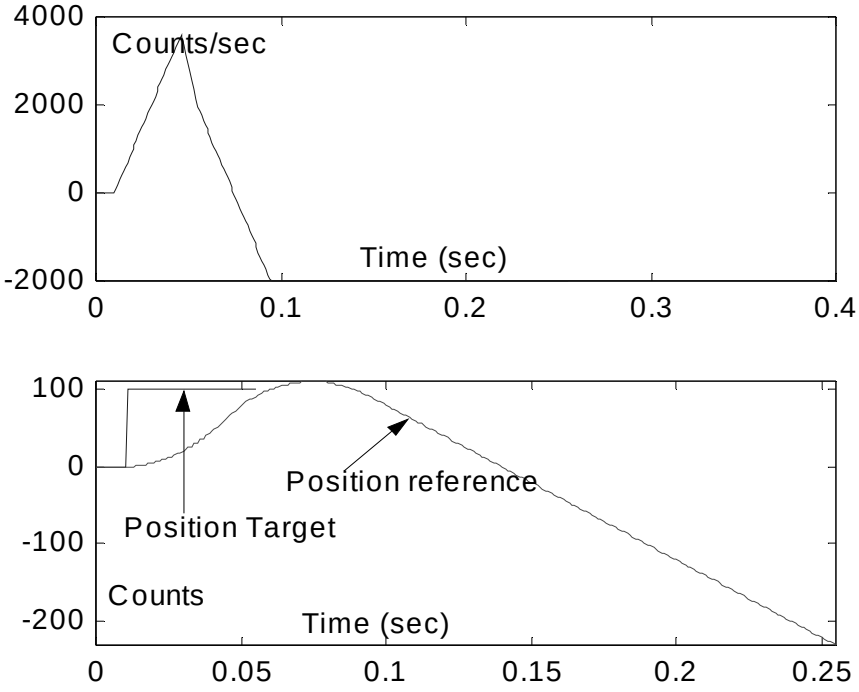
12.1.5.1 Example – Simple jogging

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.



12.1.5.2 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 Amplifier 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 terminates, the command JV=-2000;BG; is entered. The Amplifier uses the AC acceleration to arrive at the desired speed.
If, at the time of the BG, the speed of the position reference were more negative than –2000, the Amplifier would use the DC parameter instead of AC.



12.1.6 PVT: Position-Velocity-Time interpolated motion

12.1.6.1 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.
A PVT motion can be referred to absolute time by requiring the PVT motion to start at a specified start time. Use the BT (Begin on Time) command to start a PVT motion exactly at a given time, with microsecond resolution.
The ability to refer PVT motions to absolute time makes them ideal for multiply axis tightly synchronized motions.
For an explanation how to synchronize the absolute time counters of several amplifiers to the precision of few microseconds, refer the CAN manual.

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 1620 and the speed of 110000 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 = 1620,
Speed = 110000,
Time = 6msec

The time is defined with msec units, not in amplifier sampling times. The Amplifier interpolates the motion specification to calculate the desired position and speed at the sampling instances, when it needs the information.

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

The user provides, for each motion interval, the boundary positions and speeds. Mathematically, the user provides the following data:

- The starting position and speed, denoted by P0 and V0, respectively
- The end position and speed, denoted by PT and VT, respectively.

Let t_0 denote the starting time, and let **T** denote the length of the time interval.

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 given by:

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

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

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

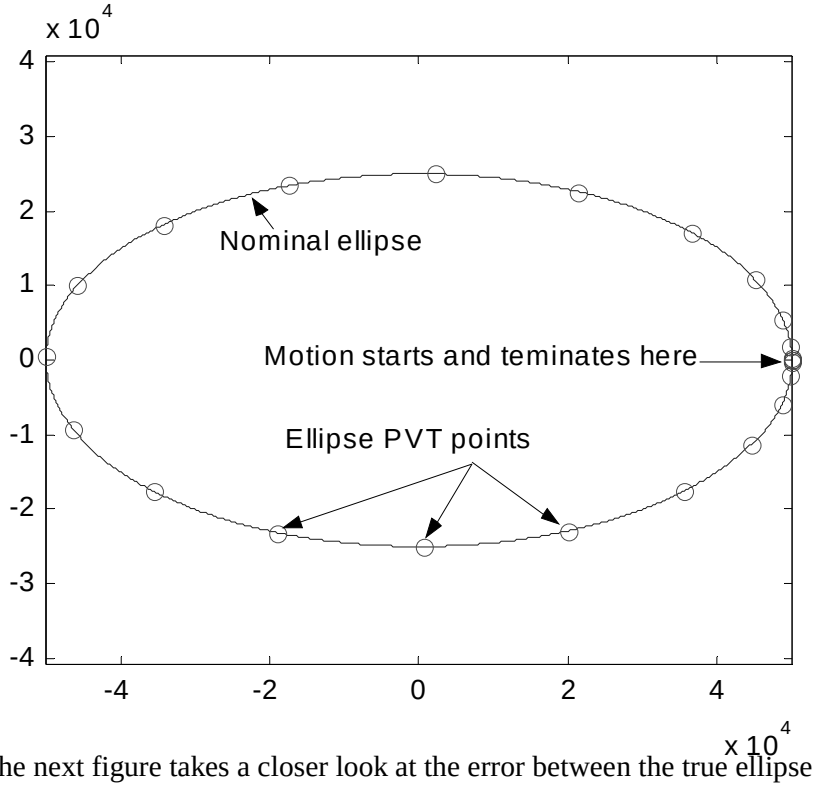
$V(t_0) = V0$, Namely $c = V0$

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

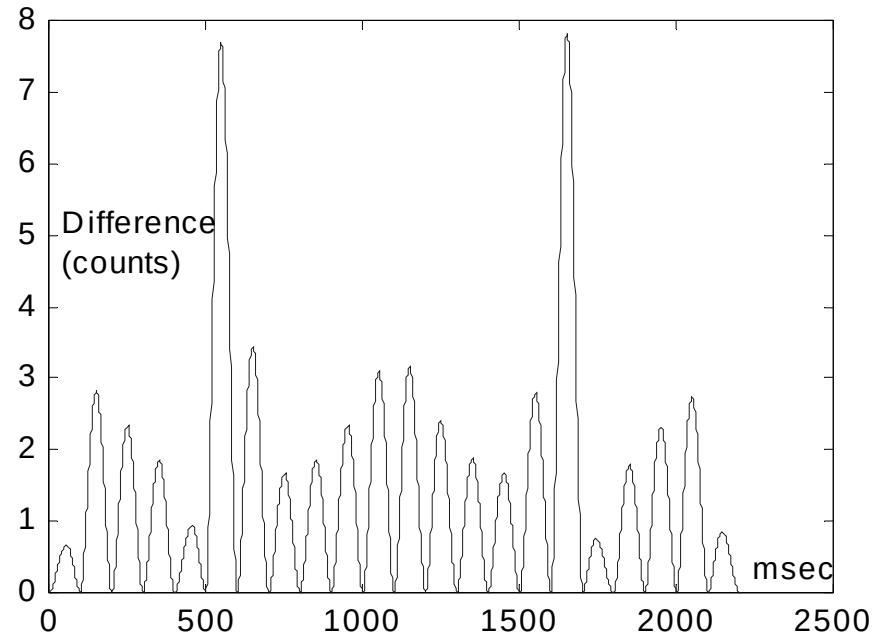
$V(t_0 + T) = VT$, Namely $VT = 3aT^2 + 2bT + c$

Example

In this example, we demonstrate how very few points can accurately describe a smooth and long motion path.
Consider two Amplifiers, driven synchronously do draw an ellipse. One Amplifier drives the x-axis, and the other drives the y-axis.
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 a fixed inter-point time of 100millisec. 23 points are enough for 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 Amplifier by interpolating the PVT points.
The original ellipse and the Amplifier interpolation of the PVT points are so close, that they can't be resolved on the plot.

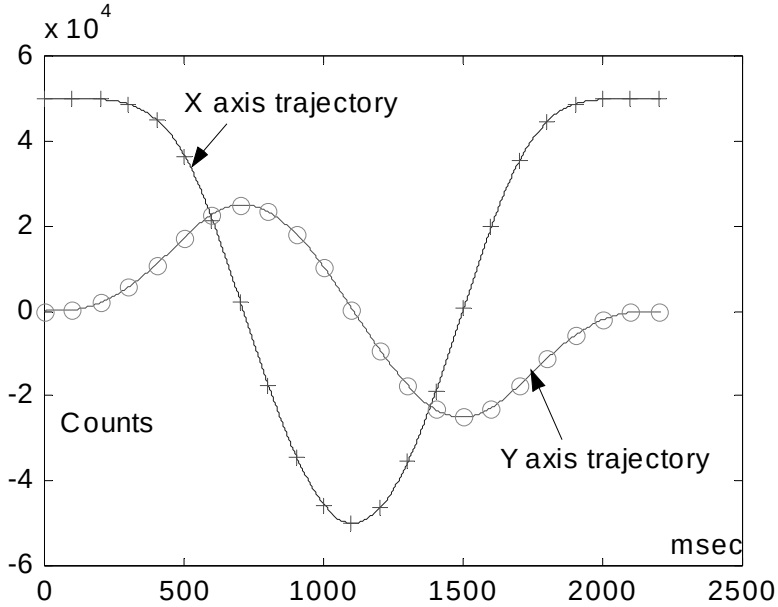


The next figure takes a closer look at the error between the true ellipse and the Amplifier interpolated path.



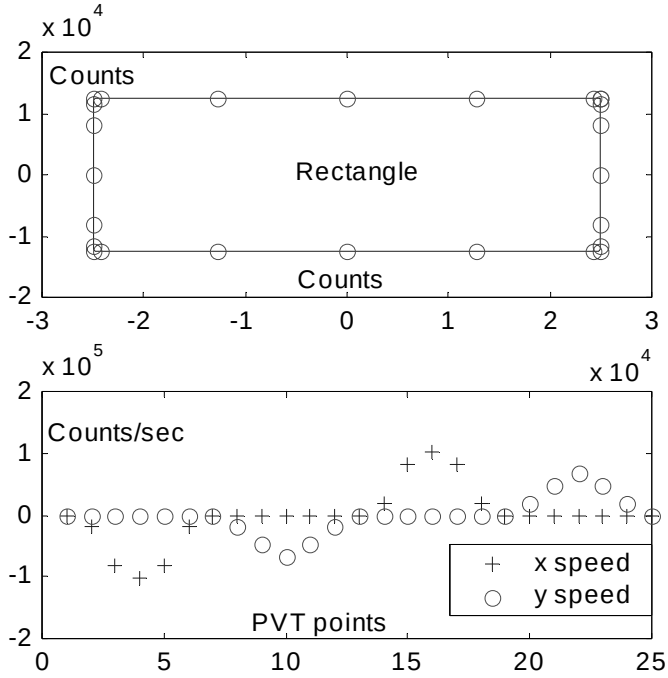
We observe that with 23 PVT points only, the interpolated path never differs from the original ellipse by more than 8 counts (remember that the ellipse axes are 100000 and 50000 counts respectively)!
At the PVT points, the interpolation error is zero.

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



Example

The Amplifier 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, non-smooth interpolation. 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.

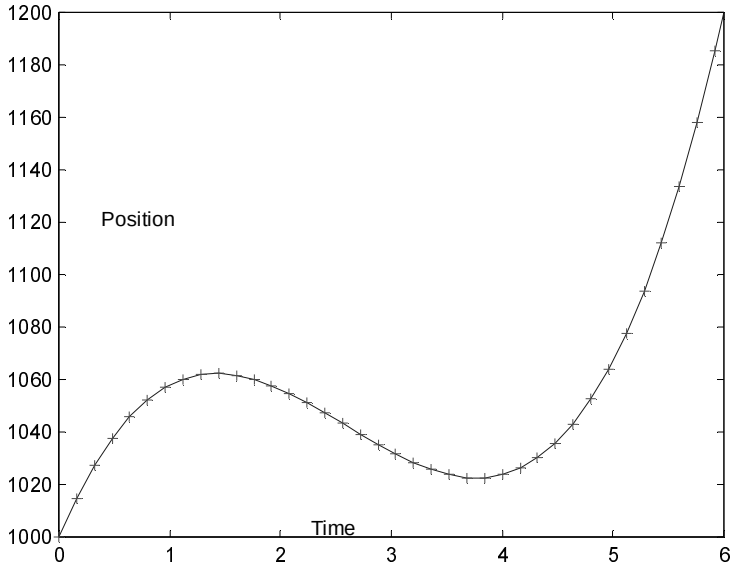
Example

Consider the following PVT interpolation interval:

Parameter	Value
Starting position	1000cnt

Starting speed	100000cnt/sec
End position	1200cnt
End speed	190000cnt/sec
Time	6msec

The interpolated path for the data of the table 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 Amplifier evaluates the interpolated motion path.



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. This average conflicts with the boundary point speed specifications of 100000cnt/sec and 190000cnt/sec respectively.

12.1.6.2 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	QP[1]	QV[1]	QT[1]
2	QP[2]	QV[2]	QT[2]
…	QP…	QV…	QT…
64	QP[64]	QV[64]	QT[64]

Table 12-6 – PVT Table

The table has 64 rows, and can thus specify up to 63 consecutive PVT motion segments[3].
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.

[3] 64 segment if the table is used cyclically.

The speeds in the table are limited to +/- 2^{23} counts/second.
The times in the table are in the range [1,255] msec.
If the position feedback sensor counts modulo, the PVT data must be so that the range of the interpolated data is [-XM...XM-1] (Set YM instead of XM here and in the next sentences if UM=4). Interpolated results in the ranges [-XM...-XM/2-1] or [XM/2...XM-1] will be folded modulo XM to the sensor range of [-XM/2...XM/2-1]. Interpolated points out of the range [-XM...XM-1] will be clipped by the Stop-Manager, refer the section on the "Stop management".

12.1.6.3 Motion Management

In the PVT mode, the Amplifier 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[4].
When the time period specified by QT[N] elapses, the N segment is done, the Amplifier 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.
The use of the PVT table is defined by the following parameters:

Parameter	Use
MP[1]	The lowest valid row of the PVT table
MP[2]	The highest valid row of the PVT table
MP[3]	A bit field. Bit 0 is: 0: Motion is to stop if the read pointer reaches MP[2] 1: Motion is to continue when the read pointer reaches MP[2]. The next row of the table is MP[1]. Bit 1 is: 0: PVT motion not expected to terminate. Issue an exception if PVT motion terminates. 1: PVT motion expected to terminate. When all the data in the PVT table has been used, exit the PVT mode to the Idle mode without issuing an Emergency object.

Table 12-7 – PVT Motion Parameters

The flow chart of the basic PVT mode is depicted below. The flow chart assumes that the commands PV=N;BG; were entered, with $1 \le N \le 64$.

[4] 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.

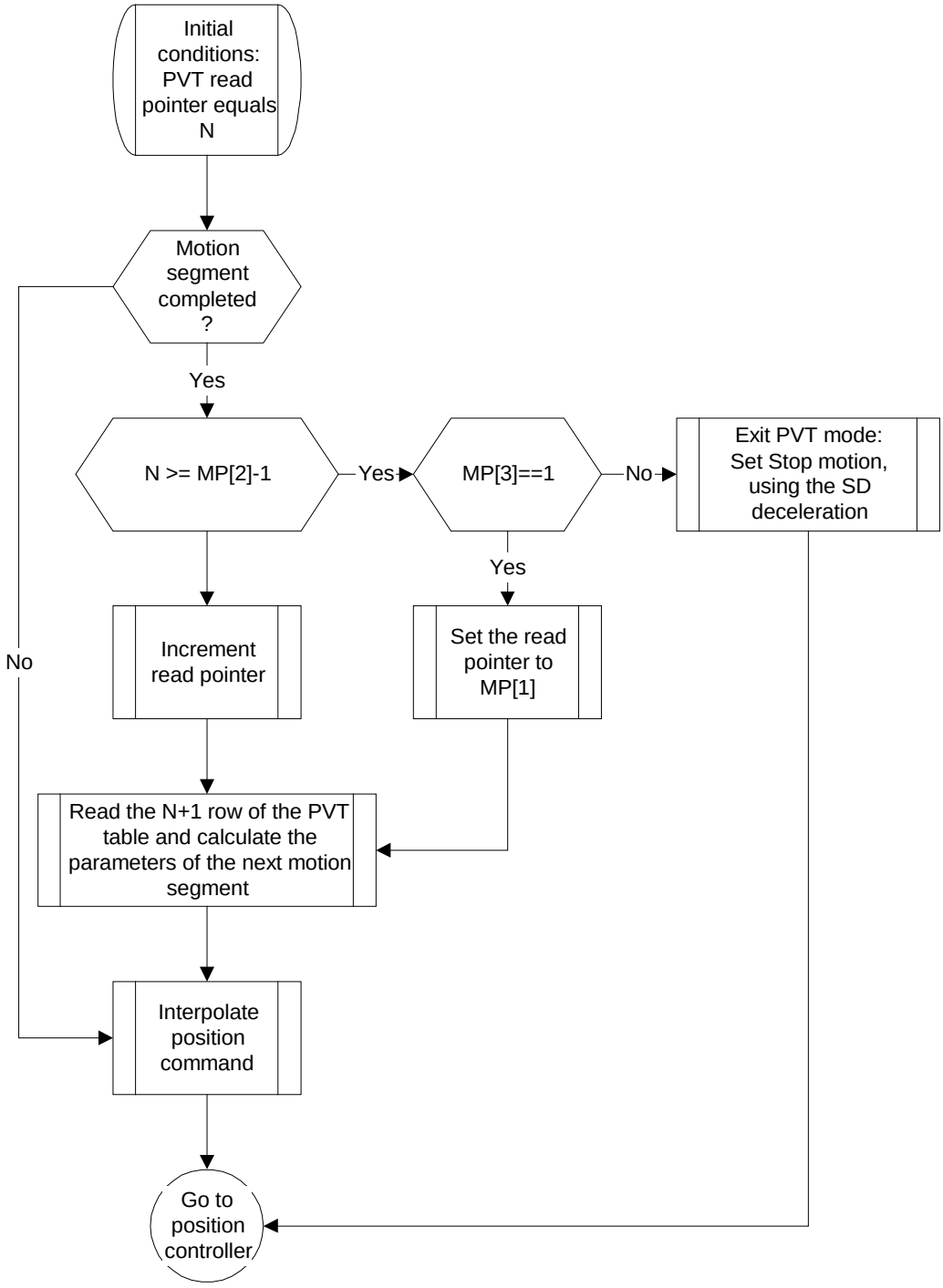


Figure 26 – PVT Decisions Flow Chart

A PVT motion is initiated by stating
PV=N with $1 \leq N \leq 64$, 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. An infinite-time non-periodic motion can be generated in the cyclical mode (MP[3]=1), by programming the PVT table on the fly.

The host must know how much free place there is in the PVT table, in order to keep programming an executing PVT motion. The best way is to keep tracking the table read and write pointers. The host knows the write table, since it control the writing to the table. If the host is in doubt, it can inquire MP[6]. To inquire the read pointer use the command PV.
The read pointer and the write pointer can be mapped to a synchronous PDO, so a CAN master can efficiently and continuously get informed about the status of many amplifiers running PVT in parallel in a network.
If the host doesn't want to poll the status of the PVT motion continuously, it can use the queue underflow CAN emergency object as a request to refill the PVT table – see details in the sequel.

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.

12.1.6.4 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.

12.1.6.5 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. For on-line motion design, you can write new entries to 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. At the communication rate of 115200baud, this may take 4msec.

The CAN communication option allows much faster PVT table programming, by packing an entire PVT table row into one PDO communication packet. For easy synchronization with the host, the Amplifier may be programmed to send the PVT read and write pointers continuously to the host as a synchronous PDO, or to send an emergency object whenever the number of yet unexecuted motion segments falls below a given threshold.

12.1.6.5.1 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 Amplifier.
Note that before using PDO 0x300+ID it must be mapped to the object 0x2001.
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 PVT table row 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) writes the data to the table row indicated by MP[6], and then automatically increments MP[6].
The CANopen auto increment mode has the following flow diagram:

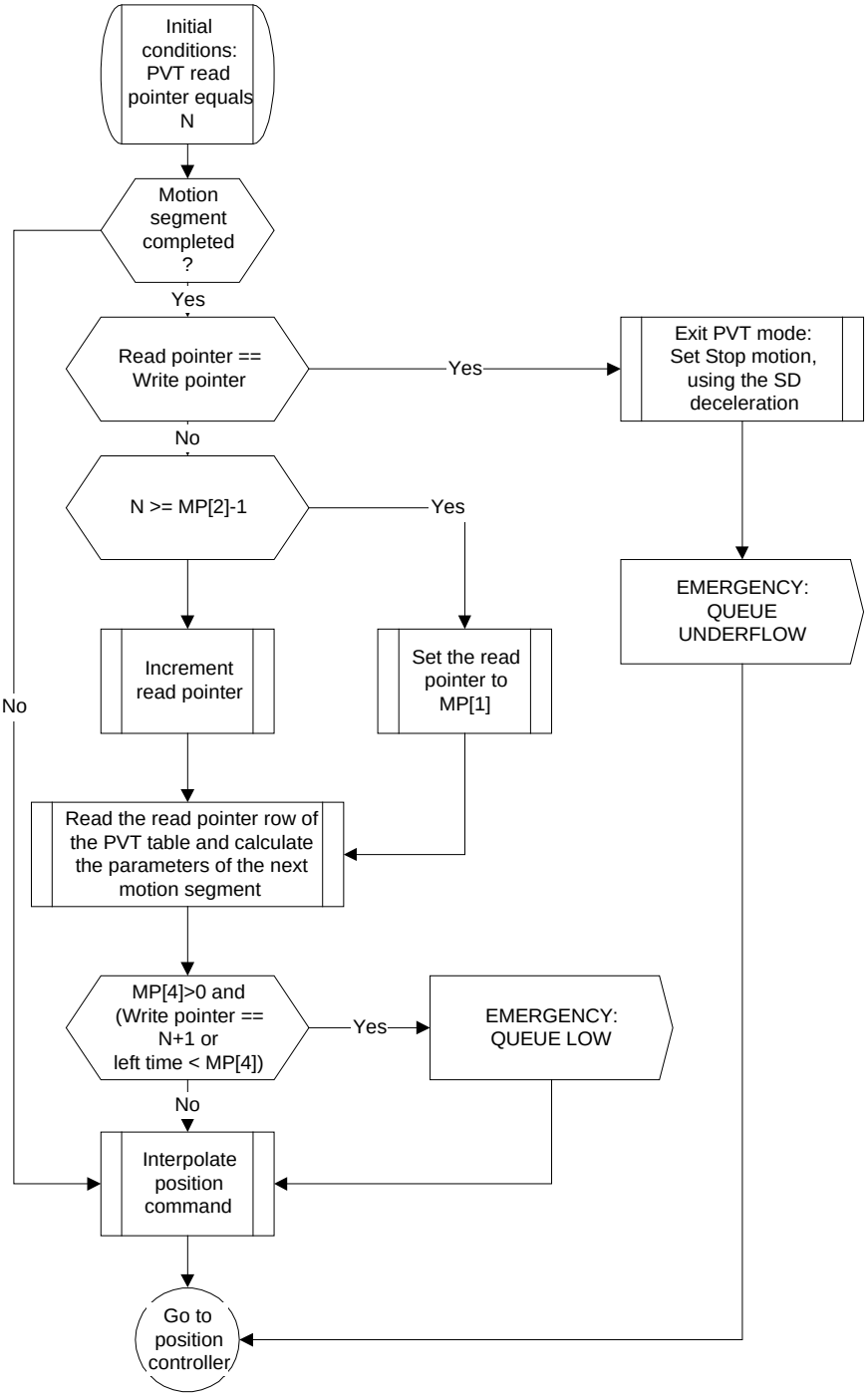


Figure 27 – PVT Auto Increment Mode Flow Chart

The above flow diagram differs from the basic mode in the following:

- 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.

12.1.6.5.2 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" Emergency object. 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, assuming QP[N] ,QV[N] ,QT[N] and QP[N+1] ,QV[N+1] ,QT[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.

Keep informed how the PVT motion advances, either by receiving the read and the write PVT table pointers continuously, mapped to a synchronous PDO, or by using the queue low emergency signal to signal the need for more data. 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 Amplifier 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.

Do not set MP[5] (Number of rows left for queue low emergency) too high. For example, consider a slow-responding host managing a 64 rows long PVT queue in the Amplifier. 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. Suppose that 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 PVT table rows have been executed, and there are only 54 valid rows in the PVT table. As 54 is lower than MP[5], there will be no more queue-low emergency messages until the PVT table is exhausted, and the PVT mode is terminated. This situation may be remedied if the host asks the Amplifier 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 Manual.

12.1.6.6 The Parameters of The PVT Motion Mode

The following parameters apply to PVT motion

What	How	Comment
Unit Mode (UM)	Unit modes 3, 4 and 5 select the position modes.	

Stop Deceleration (SD)	The rate of deceleration in the case where motion is killed by queue underflow or by an exception. The rate off acceleration to catch up if PVT is started with bad initial conditions.	
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]: Bit0 = Cyclical motion (0 non-cyclical, 1 cyclical) Bit1 = Expected stop (0 – Issue Emergency on stop, 1 – Expect stop)	
	MP[5] = Number of yet unexecuted table rows for queue low alarm.	
	MP[6] = Initial value for the write pointer	

Table 12-8 – PVT Related Parameters

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 12-9 – PVT CAN Emergency Messages

12.1.7 PT Motion

12.1.7.1 What Is PT

PT stands for Position-Time.
In a PT motion, the user specifies a sequence of absolute positions to be visited by the Amplifier with equal time spaces. The time space must be an integer multiple of the Amplifier sampling time. Between the user specified positions, the Amplifier interpolates 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.

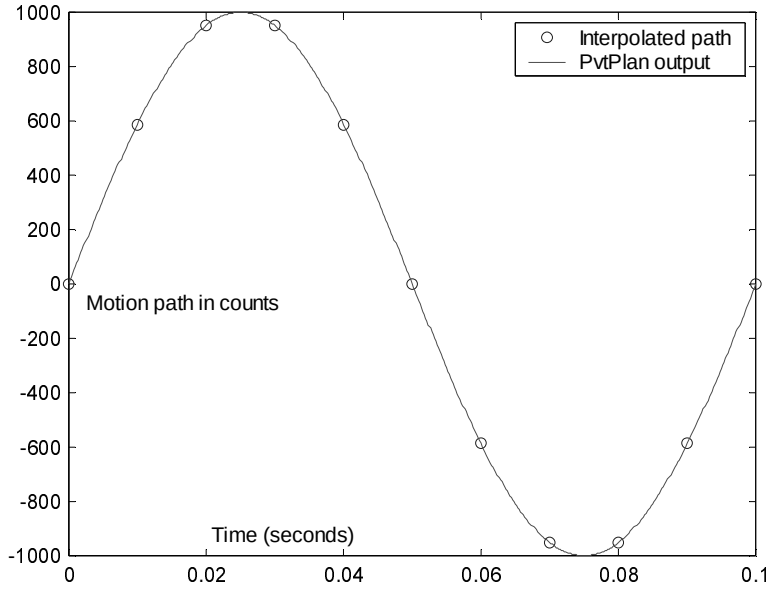
The user provides the position points $P(k), k = 1...N$.
The Amplifier calculates the speeds $V(k), k = 1...N$ for the points $P(k), k = 1...N$ as follows:
If k is an ordinary point inside the path: $V(k) = \frac{P(k+1) - P(k-1)}{2T}$
If k is the first programmed point in the path: $V(k) = \frac{P(k+1) - P(k)}{T}$
If k is the last programmed point in the path: $V(k) = \frac{P(k) - P(k-1)}{T}$

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.

12.1.7.2 Example

Consider the position controller reference signal $P(t) = \sin(2\pi \cdot 10t)$ where t is the time in seconds.
The controller has a position sampling time of 200 microseconds.
The path is programmed with a data point once per MP[4]=50 controller sampling-times (10msec).
The set of PT reference points are below depicted in circles.
The path interpolated by the Amplifier is shown as a solid line.



12.1.7.3 PT Motion Programming – The Basic Mode

12.1.7.3.1 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.

If the position feedback sensor counts modulo, the PT data must be so that the range of the interpolated data is [-XM...XM-1] (Set YM instead of XM here and in the next sentences if UM=4). Interpolated results in the ranges [-XM...-XM/2-1] or [XM/2...XM-1] will be folded modulo XM to the sensor range of [-XM/2...XM/2-1]. Interpolated results out of the range [-XM...XM-1] will be clipped by the Stop-Manager, refer the section on the "Stop management".

12.1.7.4 Motion Management

In the PT mode, the Amplifier 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)[5]. After MP[4] control sampling times, the Amplifier 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 parameters of a PT motion are summarized in the following table:

Parameter	Use	Comment
MP[1]	The lowest valid element of the QP vector	
MP[2]	The highest valid element of the QP vector	

[5] 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.

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 12-10 – PT Motion Parameters

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

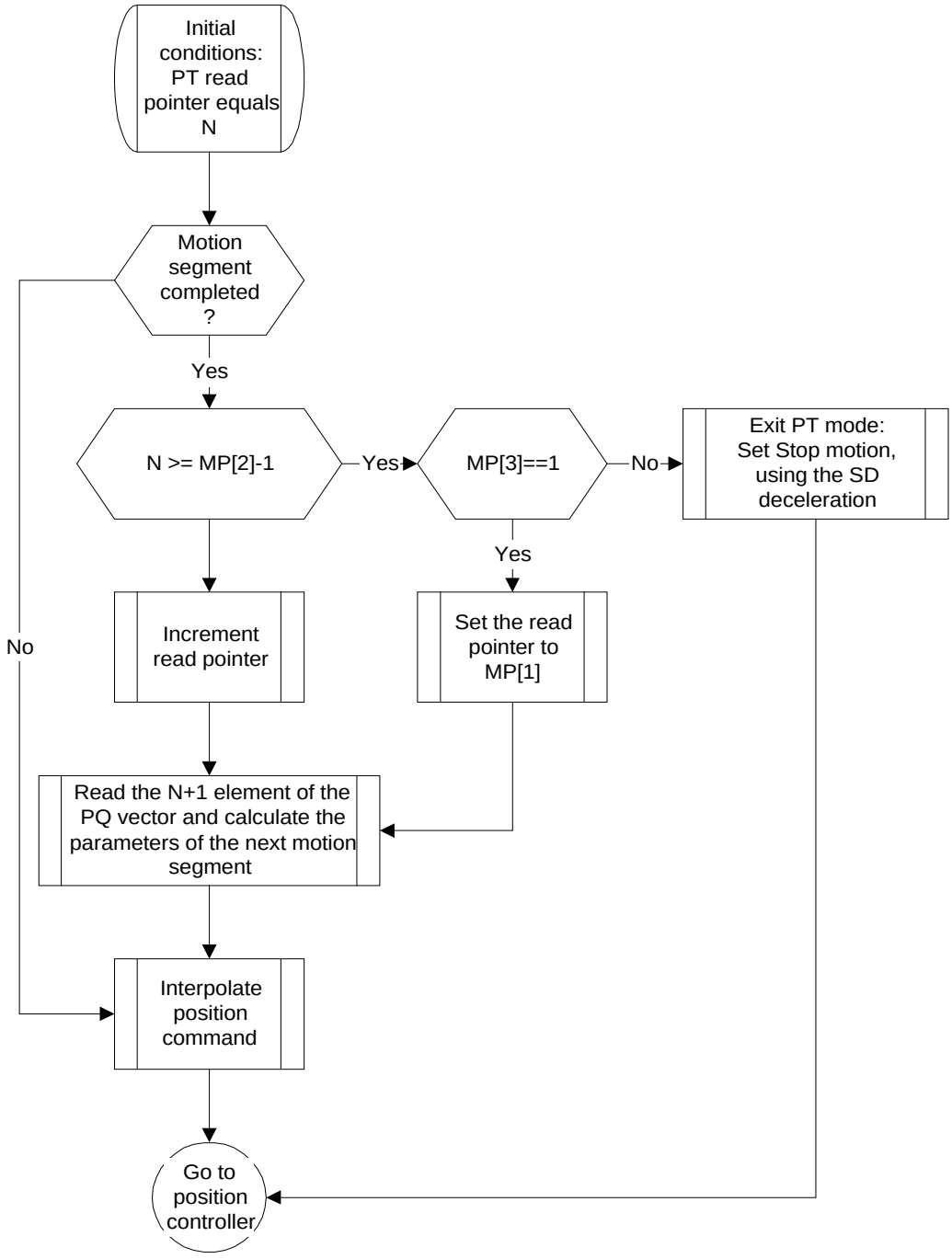


Figure 28 – PT Decisions Flow Chart

A PT motion starts 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, as long as you don't program presently executing PT elements.

12.1.7.5 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.

12.1.7.6 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, and 1.8msec at the baud rate of 115200.

The CAN communication option allows much faster PT programming, by packing two position points into one PDO communication packet. For easy synchronization with the host, the Amplifier may be programmed to send the PT read and write pointers continuously to the host as a synchronous PDO, or to send an emergency object whenever the number of yet unexecuted motion segments falls below a given threshold.

12.1.7.7 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 Amplifier. 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 the amplifier receives a new PT motion-programming message.
The CAN auto increment mode has the following flow diagram:

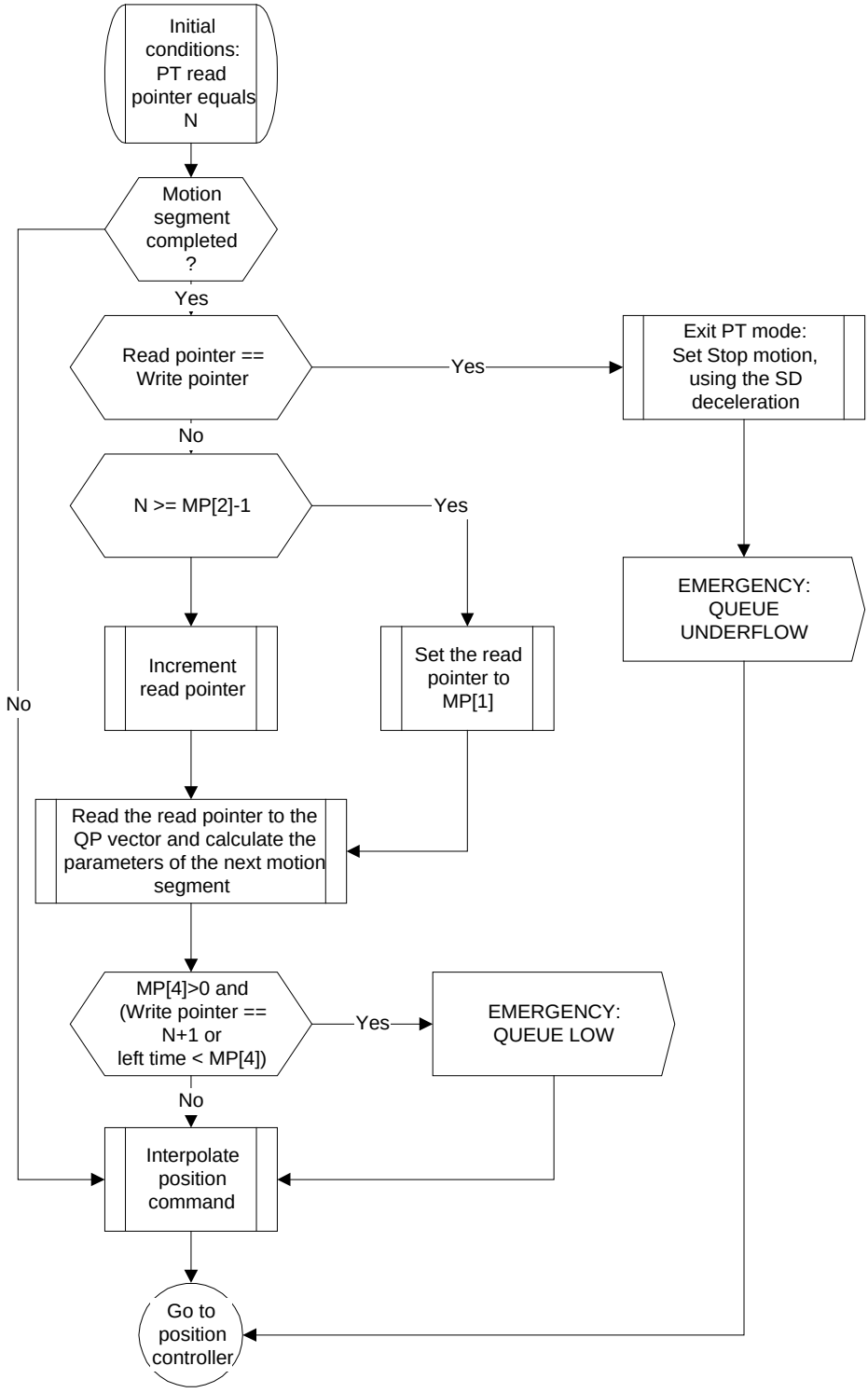


Figure 29 – PT Auto Increment Mode Flow Chart

The above flow diagram differs from the flow diagram of the basic mode in the following:

- The read pointer reaching the write pointer identifies motion queue underflow.
- Emergency objects are issued for the queue low and for the queue underflow events.

12.1.7.8 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.
Keep informed how the PT motion advances, either by receiving the read and the write PT table pointers continuously, mapped to a synchronous PDO, or by using the queue low emergency signal to signal the need for more data. 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 Amplifier 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.

12.1.7.9 The Parameters of The PT Motion Mode

The following parameters apply to PT motion

What	How	Comment
Unit Mode (UM)	Unit modes 3,4 or 5 select a position mode.	
Stop Deceleration (SD)	The rate of deceleration in the case where motion is killed by queue underflow or by an exception. SD is also the acceleration to catch up with a PT motion started with bad initial conditions.	
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 12-11 – 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 12-12 – PT CAN Emergency Messages

12.2 The External Position Reference Generator

This section summarizes how the Amplifier generates the external motion command for the position controller.
The external position reference is useful for:

- Positioning a manipulator on a moving object. The desired position of the manipulator with respect to the object on the conveyer can be programmed as a software motion, like

PTP or PVT. The position of the conveyer is not known in advance, and it must be measured on line, for example by using the auxiliary encoder input. The reading of the auxiliary input is scaled by the follower ratio (FR[3]) parameter, and added to the software command.

- Driving the amplifier as a slave in a larger arrangement. For example, when the amplifier moves a valve in a pressure control system, its position command may be an analog output of a pressure controller.
- Synchronizing several amplifiers. Several amplifiers may be driven by an auxiliary encoder signal. Each of the amplifiers uses its ECAM table to derive its own motion path from the auxiliary signal.

The external position reference is generated by the following scheme:

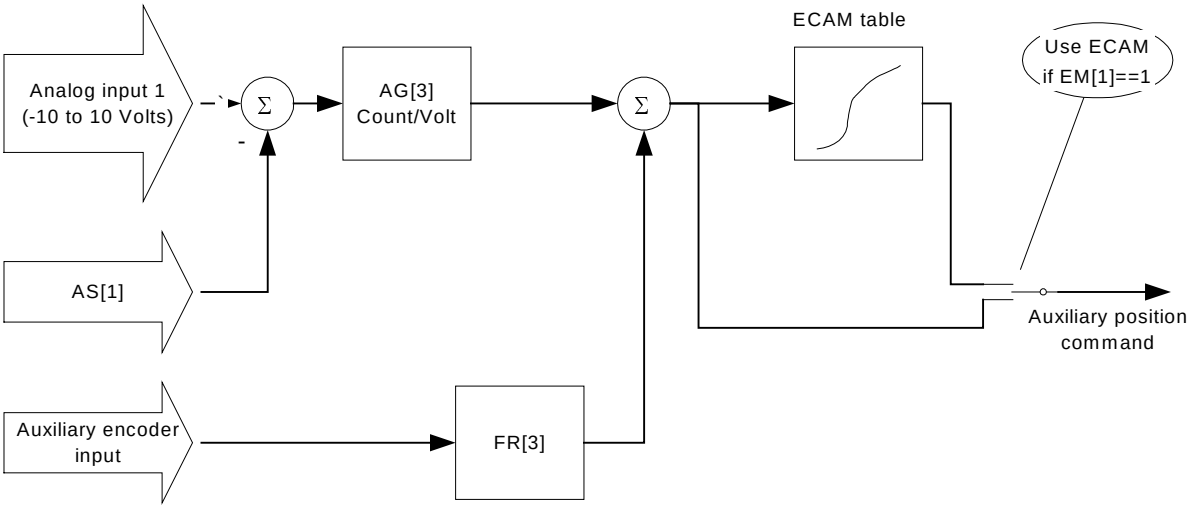
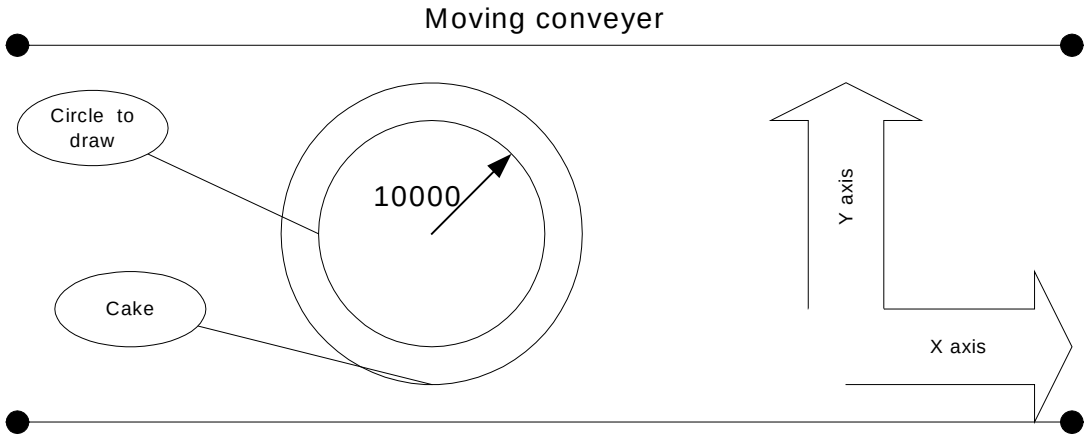


Figure 30: External position reference generator

The following parameters determine how the position reference is composed:

Parameter	Action
AG[3]	Scale the analog input. The units of AG[3] are counts/Volt.
AS[1]	Offset the analog input. The units of AS[1] are Volts.
FR[3]	Scale the auxiliary encoder input. FR[3] 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

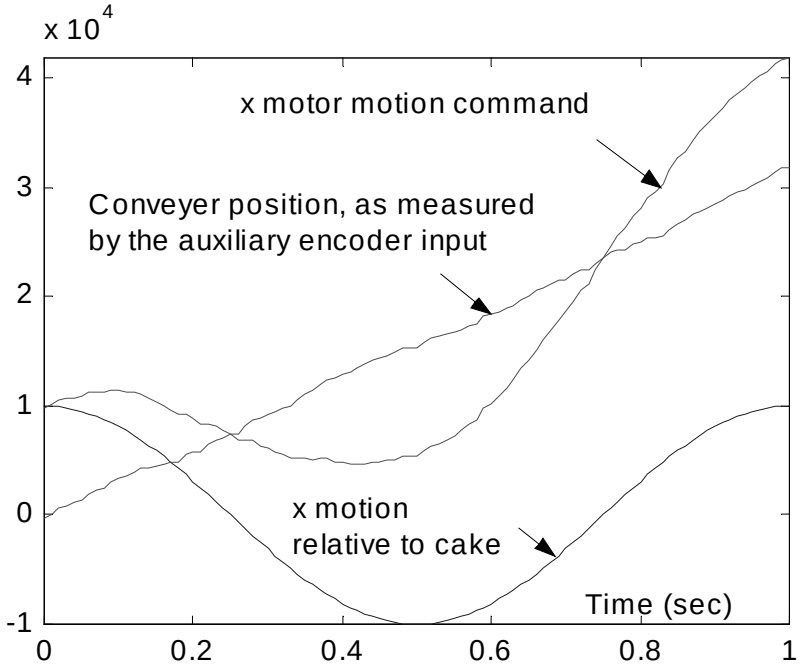
Example:
This example illustrates working on a moving object. Consider the application depicted below:



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

$$x(t) = 10000 \cdot \cos(2\pi t) + c(t)$$

Where c(t) is the position of the conveyer. If the conveyer has an encoder, we can use the conveyer encoder to compensate its motion. Suppose that the resolution of the conveyer encoder is similar to the resolution of the x-axis encoder.
For drawing an exact circle on the moving cake, we program the motion $10000 \cdot \cos(2\pi t)$ as PVT, and set RM=1, FR[3]=1.



12.2.1 ECAM

ECAM is an acronym for "Electronic Cam". It means that the position reference to the Amplifier 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 whenever 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[3]+EM[4], the ECAM function will output ET[2].When the input to the ECAM table will be EM[3]+2*EM[4], the ECAM function will output ET[3], etc.
EM[5]	The first valid index of the ECAM table.

Table 12-13 – ECAM Related Command

The input to the ECAM Table (IET) is composed from:
IET = ((AG[3] * (Analog input 1 – AS[1])) + (FR[3] * Auxiliary Encoder)).

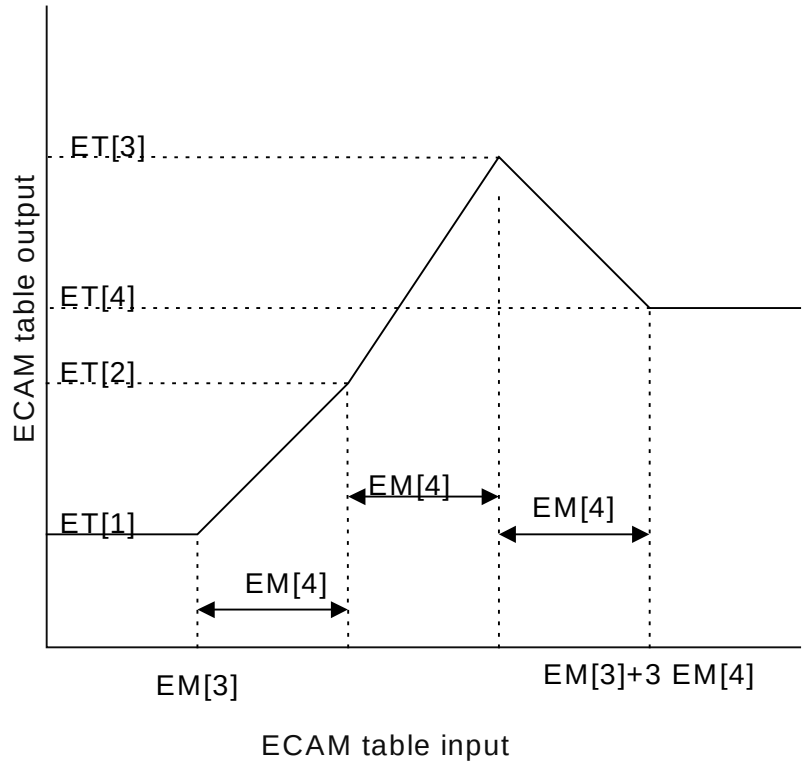
As a result to IET the response may be:

- If IET < EM[3], the external position command (output of the ECAM table) will be ET[EM[5]].

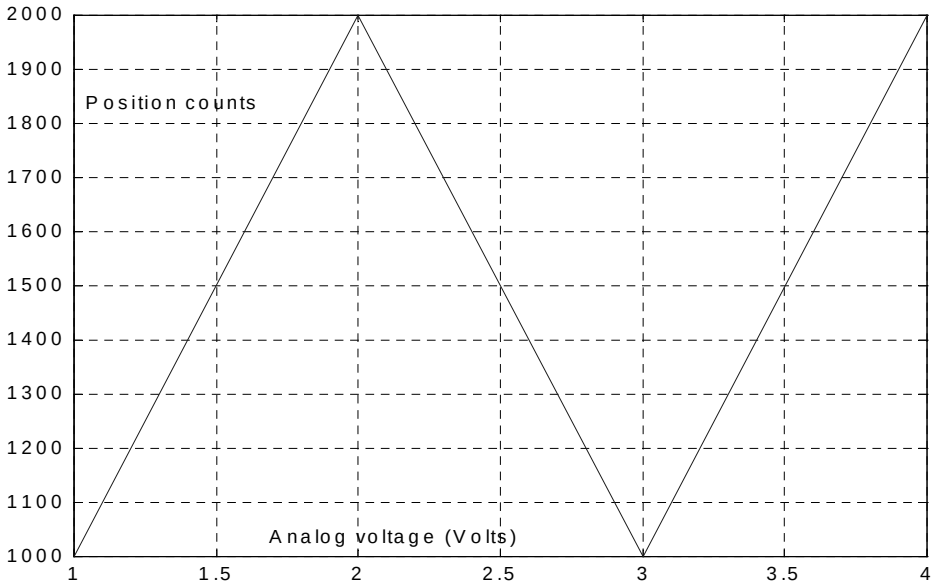
- If IET > (EM[3]+(EM[2]-1)*EM[4]), the external position command will be ET[EM[2]].

- 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:
The parameters of the ECAM table are illustrated graphically in the figure below, for the case EM[5]=1 and EM[2]=4.



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



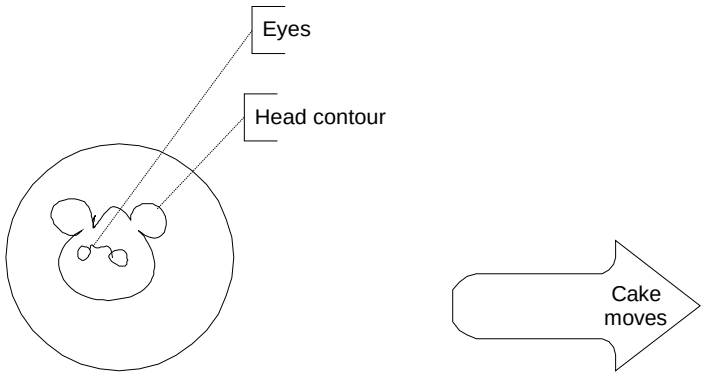
We set:

AG[3]=1000	Set 1000 counts at the input of the ECAM function for 1v of input
FR[3]=0	Don't use the auxiliary encoder.
EM[2]=4	Last ECAM points has the index 4.
EM[3]=1000	The first point is at 1v, which by AG[3] is equivalent to 1000 counts
EM[4]=1000	Each two position break points are separated by 1v that by AG[3]

	is equivalent to 1000 counts
EM[5]=1	First index of the table
ET[1]=1000	ECAM table values
ET[2]=2000	
ET[3]=1000	
ET[4]=2000	
EM[1]=1	Activate ECAM.

- For every input voltage less than 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 than 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 amplifier. 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 Amplifiers 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 Amplifiers controls, by a digital output, the flow of the chocolate out of the drawing nozzle.

The initial programming of the Amplifier 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]=…;ET[100]=…;	Program the numeric data of the ECAM table
UM=5	Set single sensor position mode
RM=1	Enable external referencing
AG[3]=0;FR[3]=0;	Kill the external input
MO=1	Start motor
PA=1000;BG	Go to waiting position
HY[2]=0;HY[3]=5;HY[1]=1	Null the auxiliary encoder count upon cake arrival (RLS high)

Each Amplifier has the following AUTO_RLS routine:

```
function AUTO_RLS
/*
The RLS has operated an already programmed auxiliary homing process to synchronize
the follower input
*/
OB[1]=1;FR[3]=1;          **Activate chocolate nozzle
                          ** Enable the auxiliary encoder input with the
                          follower ratio of 1.
until (PY >= 2000)        **Wait until end of the head contour
OB[1]=0;                  **Stop the chocolate
until (PY>=3000)          **Go to start of eyes
OB[1]=1;                  **Restart the chocolate
until (PY>=4000)          **Draw the eyes
OB[1]=0                   **Stop the chocolate
FR[3]=0;PA=1000;BG;       **Return to starting point
HY[1]=1                   **Program the auxiliary encoder to reset again at the
                          next cake
return                    **End of auto subroutine
```

12.2.2 Dividing ECAM table into several logical portions

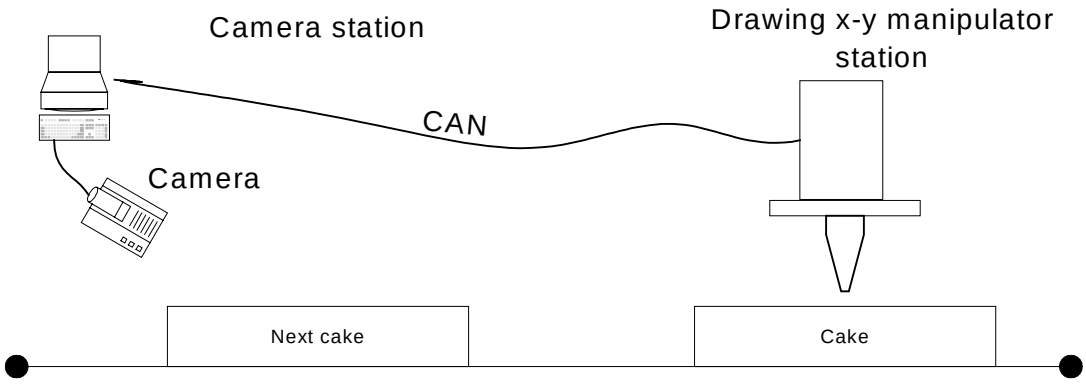
The ECAM table can store several distinct movements. A portion of the ECAM table is used for each movement. That way a future movement can be programmed into the amplifier while the present movement is executing.

Example

In the previous "Chocolate Teddy Bear" application example, we assumed that the Teddy Bear could be programmed once.

In many food applications, the products to be worked on (the cake in the example) are not placed exactly on the conveyer, or their shape may not be regular. A camera images the next coming product, and the image is analyzed to form the next motion path.

The motion path for the analyzed image is communicated to the amplifier while the amplifier works a previously programmed path – see figure below.



Assume that the amplifiers at the x-y manipulator station runs the ECAM table entries ET[1]...ET[100]. In the meantime, the system controller programs the next shape to run into ET[101] ...ET[200]. The preferred way to do that is by using the fast CAN ECAM points protocol. When the x-y manipulator will work on the ECAM table entries ET[101]...ET[200], the system controller will program again ET[1]...ET[100] and so on. The AUTO_RLS program of the previous example is now slightly modified to

```
function AUTO_RLS
/*
The RLS has operated an already programmed auxiliary homing process to synchronize
the follower input
*/
if( ET[201]==1)EM[5]=1;EM[2]=100;else,EM[5]=101;EM[2]=200;end;EM[1]=1;...
OB[1]=1;FR[3]=1;
                                  /* The host uses ET[201] to signal which part of the
                                  table to use next. Select the ECAM table portion to
                                  use by ET[201], and the set EM[1]=1 to activate the
                                  new used portion, activate follower gain and the
                                  chocolate nozzle*/
until (PY >= 2000)                **Wait until end of the head contour
...                               ** Continue as in the previous example.
return
```

12.2.3 On the fly ECAM programming using CAN

ECAM table points can be programmed via fast, auto increment PDO service.
Two positions of the ET table 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 Amplifier. 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	0x2003
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].

12.2.4 Initializing the external reference parameters.

This section details what happens when the external reference generator is initialized.
The external reference generator is initialized at MO=1, and each time a relevant parameter (FR[3], AG[3], EM[1]) is changed. Note that changing EM[2],EM[3],EM[4],and EM[5] does nothing immediately. Setting EM[1] activates the entire set of ECAM parameters.

12.2.4.1 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, the motion mode is Idle, so that commanding BG without the prior specification of another motion mode 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, in the Idle mode. 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[3]=1, AG[3]=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[3]*PY= 1000 - 1*3000 = -2000.
If the auxiliary encoder input is stable, the command sequence PA=0;BG will launch a PTP motion from the position of 1000 counts to the position of 3000.

12.2.4.2 Switching the parameters of the external reference on the fly

The parameters of the external reference generator can be changed on the fly only if the software reference generator is idle. This can be verified by the motion status – MS should return 0 or 1.
When the parameters of the external reference generator are changed, the software reference is set so that the motor will not jump at the instance the new parameters are set. The setting of the software reference is similar to the setting described in the section "Jump-Free Motor Starting Policy" above.

The external reference parameters that can be changed on the fly are:

Parameter	Description
AG[N]	Analog gain
FR[N]	Follower ratio
EM[N]	Ecam parameters
AS[1]	Offset for analog input
HY[N]	Reading of auxiliary encoder

Example:
Consider a manipulator that works a conveyer. Whenever a box arrives, the roller prints a label on the box. When a new box arrives, it first homes the auxiliary encoder to read zero. The auxiliary encoder references the roller in the follower mode, with FR[3]=CA[18]/(Conveyer encoder counts/mm*2*π*r(mm)).

The software command to the roller is set as point to point, to correct the roller position for the next arriving box.
That way the printing is synchronized to the box.

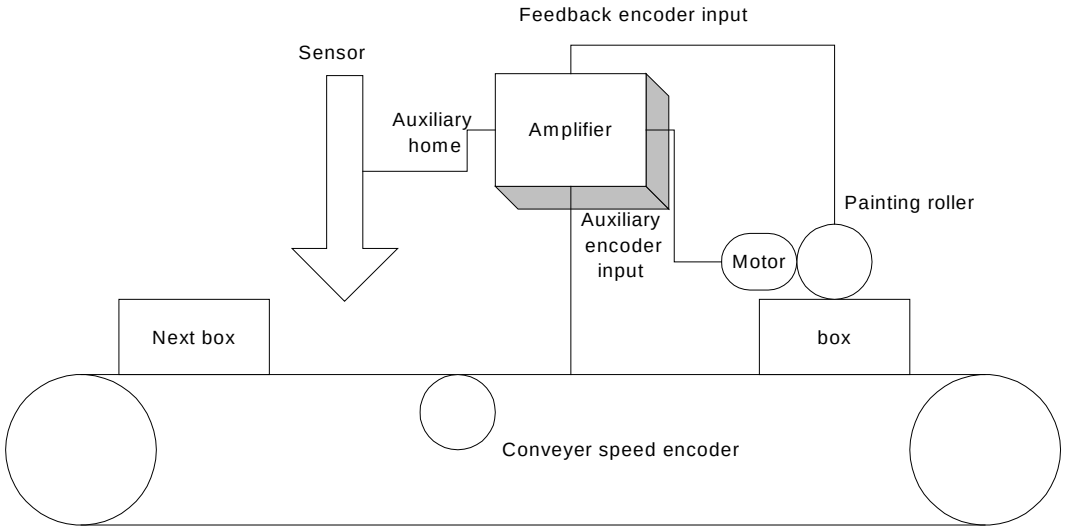


Figure 31: Print On A Moving Box Application

At the time of auxiliary homing (when a new box arrives at the sensor), the external reference jumps by PY*FR[3]. The software command to the roller jumps in parallel by –PY*FR[3], so that the roller continues to move normally.
Programming the ##AUTO_HM routine as

```
function AUTO_HM()
PA=0;BG;
return
```

Will position the roller phased correctly with respect to the next coming box.

12.3 The Stop management

12.3.1 General description

The "Stop Manager" block has the following functions:

- Stop the motion upon sensing a "Stop" switch, or upon sensing a limit switch (RLS or FLS).
- Protect against discontinuity it the controller's command. A discontinuity can occur due to:
  - A switch stops the motion abruptly.
  - An application error (An absolute motion mode like PVT is started with bad initial conditions, or PVT generates position command out of the modulo range of the feedback sensor)
- Limit the magnitude of the controller's command to the maximum allowed range. This is necessary since even if the software command is generated within the permitted limits, and the external command is within the permitted limits also, their some may be out of the permitted limits.

The Stop Manager prevents the position reference generator from driving the motor to undesired places. The stop manager does not affect the reference generator.
While this feature enable

The commands relevant to the stop manager are:

Command	Description
SD	The maximum rate that the motor can accelerate/decelerate.
IL[N]	Input logic – define the functions associated to digital inputs.
VH[N],VL[N]	The maximum allowed controller command
XM,YM	Modulo count for the main and the auxiliary sensors

12.3.2 Stop Manager Internals

The Stop Manager is depicted in the block diagram below:

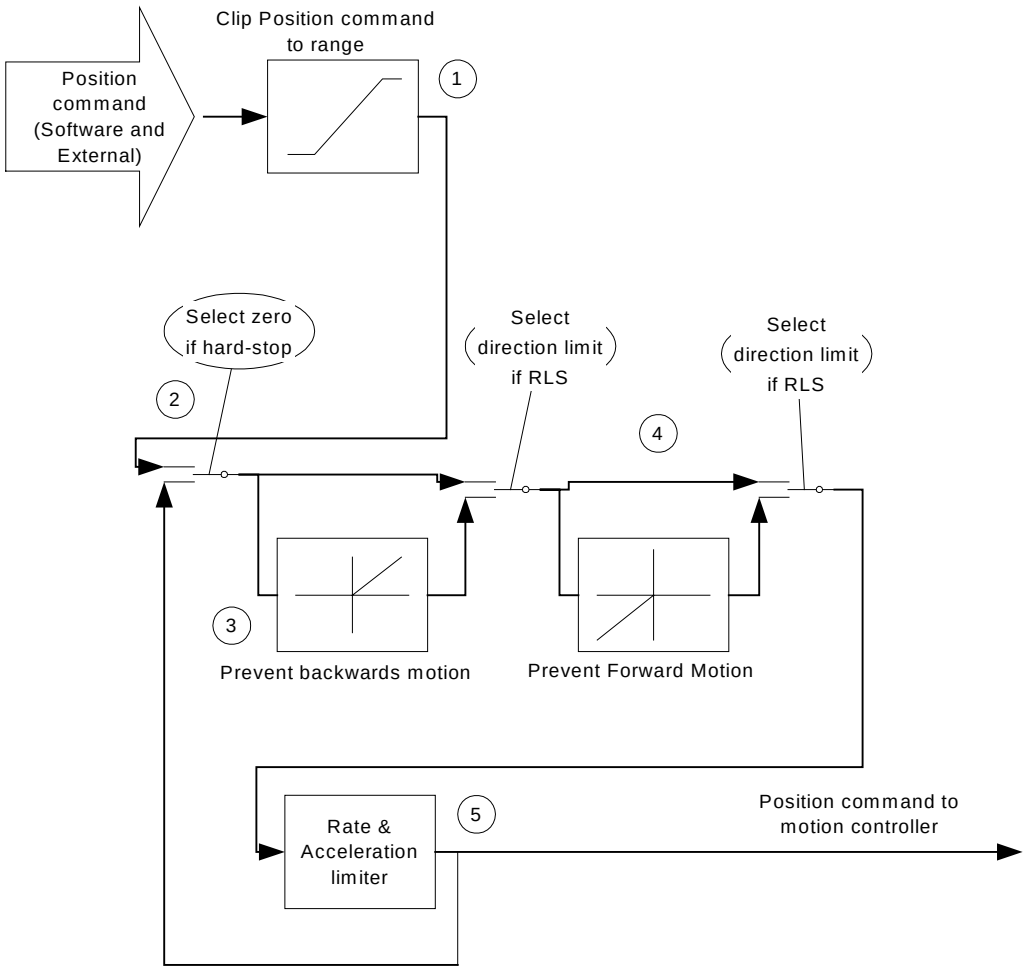


Figure 32: Stop Manager Block Manager

In the next paragraphs, we shall explain the blocks in Figure 32.

Position command clipping (Marked as 1 in the Figure)
The position command is clipped to the following values:

- VH[3] above and VL[3] below, if the position feedback sensor counts linearly.
- XM/2-1 above, and -XM/2 below in UM=4, if PX is counted modulo.
- YM/2-1 above, and -YM/2 below in UM=5, if PY is counted modulo.

The clipping is necessary since, in some circumstances explained above, the sum of the software command and the external command, or even the software command alone, may exceed the command limits.

Hard Stop (Marked as 2 in the Figure)

This block stops the desired position reference to its present position if a hard-stop switch is sensed.

Right Limit Switch (Marked as 3 in the Figure)
This block stops the desired position reference to its present position if an RLS switch is sensed, and if the output of the position reference generator is less than the controller's position command.

Forward Limit Switch (Marked as 4 in the Figure)
This block stops the desired position reference to its present position if an FLS switch is sensed, and if the output of the position reference generator is greater than the controller's position command.

Rate and Acceleration (Marked as 5 in the Figure)
This block limits the speed and the acceleration of the controller's position command. The block limits both the acceleration and the deceleration to the parameter SD, and the speed command (the derivative of the position command) is limited to VL[2] from below, and VL[3] from above.
The rate and acceleration limiter block intervenes in the following situations:

- The position command to the controller experiences an abrupt change – for example when a hard-stop switch is sensed, or when a hard-stop switch is released
- The reference generator tries to command the motor in the permitted position range, but with too great speed.
- The position command moves towards its permitted boundary (VL[3] or VH[3]) with speed greater then can be braked until the boundary with SD acceleration. For example, near VH[3], the upper speed limit may be much smaller then VH[2]. When the position command equals VH[3], the maximum allowed speed command is zero.

Example
In this example, the software reference generator generates a sine, using PVT.
The low position limit is VL[3]=-5000. The input to the Stop Manager and its output (the position command to the controller) are depicted in Figure 33 below. The input of the Stop Manager (The output of the position reference generator) is in red, and the output of the Stop manager is in dashed green. In At the time of 0, the PVT starts at the position of 17000. The Stop-Manager catches up with the sinusoidal command, with the acceleration limit of SD.
At the time of 1sec, a hard-stop switch becomes active, and remains active until the time of 1.2 sec. The Stop manager decelerates the position command to zero speed, using the deceleration SD. At the time of 1.2sec, the Stop Manager begins to catch up again with the PVT reference, with the SD acceleration.
At the time of approximately 1.7sec, the PVT reference nears the limit of VL[3]=-5000. Before the reference waveform actually reaches –5000, the Stop Manager finds that speed is too large for stopping at VL[3] with the deceleration of SD. It therefore reduces the speed, coming to a complete stop at VL[3] with very small or no overshoot.
At 2.75sec, approximately, the Stop Manager finds that although the position reference is not yet in range, it will in short time be in. The Stop-Manager begins to accelerate the motor into the permitted range, so that catching up when the reference returns to range will be immediate, without the delay of acceleration.

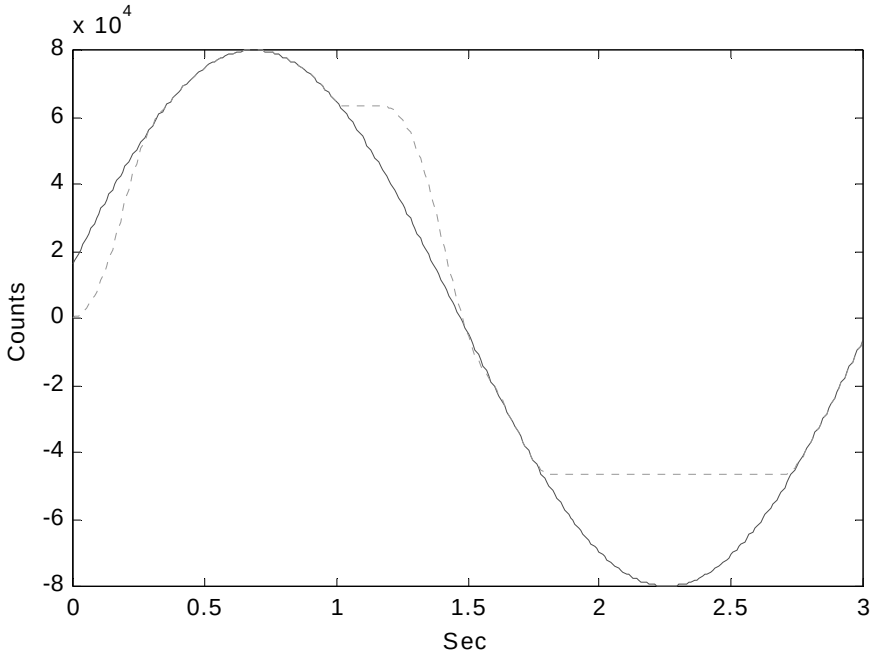


Figure 33: Position output of the stop manager

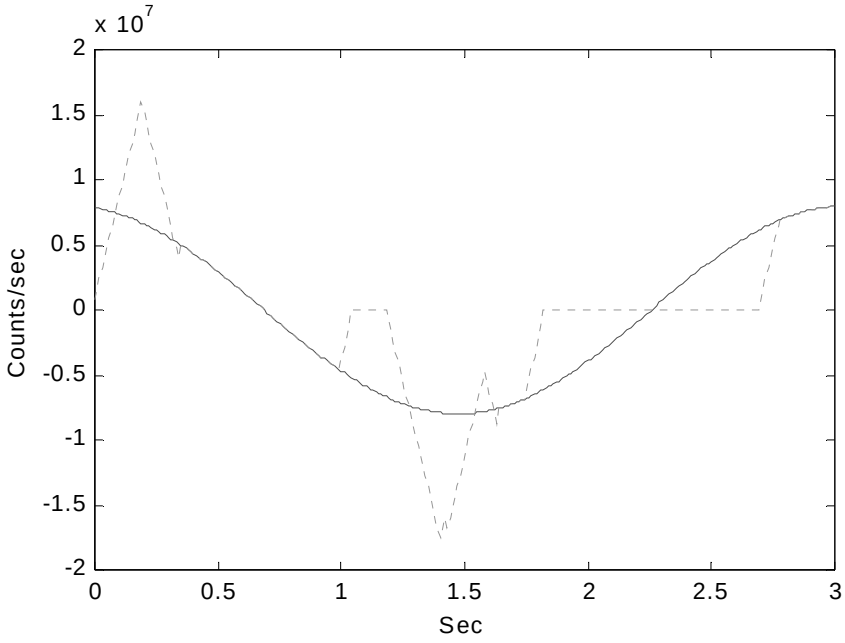


Figure 34: Speed output of the Stop Manager

13 Sensors, I/O, and Events

The Harmonica has two encoder inputs for feedback, commutation, and auxiliary reference generation.

In addition, it has an analog input, Hall sensor inputs, and digital I/O.

The digital inputs and the encoder index signals can generate events that may register the motor position, or reload the position counters, or flag a digital output, or call a special user program.

The digital outputs responds also to software events, like affirmation of the conditions to start the motor, or brakes activation upon starting or shutting the motor.

13.1 Modulo counting

13.1.1 Modulo Counting

The variable PX counts the main position sensor. The PX variable cannot increase indefinitely. In fact, PX is limited to the range [-2^{30}...2^{30}].
For limited motions, that may be good enough.
Some applications, however, require that the motor position be counted cyclically. When a certain top position is arrived, the position counter "rolls off" and counting continues from the bottom position.
The most common examples for modulo counting are rotary pointing equipment like radar pedestals, camera pointers, or rotary robot axes. In this type of equipment, the position is normally counted modulo the mechanical rotation of the load. When the load points to a given direction, the encoder readout will be always the same, no matter how many full rotations have been made.
The Harmonica can handle modulo counting: It can run all the types of software motions with modulo position counting. Point-to-Point position motions in modulo systems are ambiguous, since any point can be reached from both directions. The Harmonica will always go the short way.
The commands relevant to modulo counting are:

Command	Description
XM	Modulo value for the main position counter. When XM is nonzero, the main position sensor counts cyclically in the range [-XM/2...XM/2-1]. When XM=0, the main position sensor count linearly, limited to the range [-2^{30}...2^{30}].
YM	Modulo value for the auxiliary position counter. When YM=0, the auxiliary When YM is nonzero, the auxiliary position sensor counts cyclically in the range [-YM/2...YM/2-1]. position sensor count linearly, limited to the range [-2^{30}...2^{30}].
RM	Reference mode. Must be 0 (Do not use the auxiliary reference generator) if modulo counting is used. For further details refer the chapter "The position reference generator"

Table 13-1: Commands relevant to modulo counting

If XM is nonzero with UM=5, or if YM is nonzero with UM=4, the controlled motor position will count cyclically.
When the position feedback counts cyclically:

- You cant use external position referencing (RM=1).
- The RLS and the FLS (Reverse and Forward limit switches) cannot behave directionally, since any position can be approached from both directions.

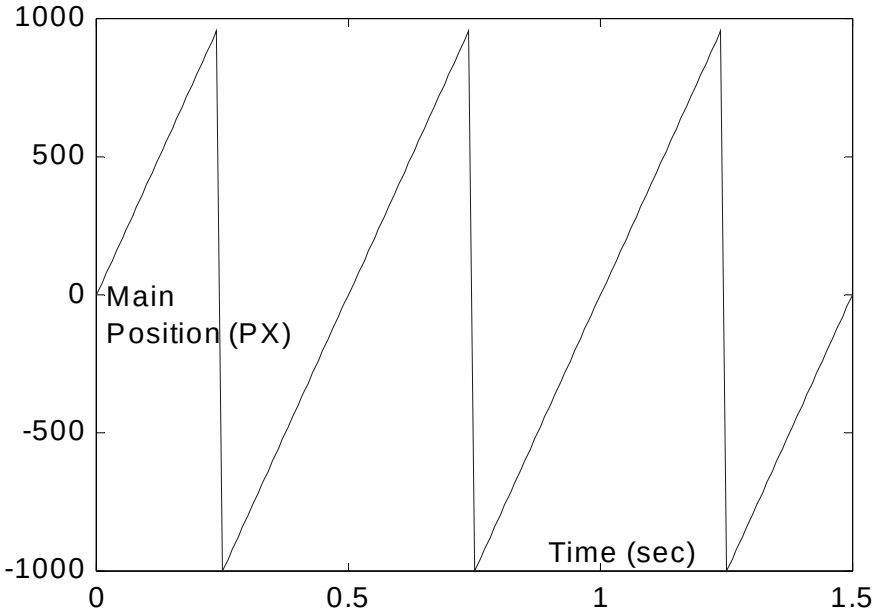
RLS and FLS can be read by the user program and activate their corresponding automatic routines, but they do not affect the motion immediately.

- The position referencing limits VH[3] and VL[3] become ineffective.

! If the modulo value is selected low, and the sensor speed is high, more than one full revolution of the position counter may elapse within a single sampling time.
This will cause the position counter to behave in an unpredictable way.

Example:

Consider a motor rotating in the 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 $[-2^{30}..2^{30}-1]$.

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.

13.2 Digital Inputs

The Harmonica has 6 digital input pins. The digital input pins must be associated to functions before being used. All the digital input pins can be associated to the Enable, Stop, RLS, FLS, and Begin functions. Two of the digital inputs are connected to high-speed hardware counters. These inputs can also associate to the Home function.
The association of input pins to functions is described in detail in the Command Reference Manual, the IL[] command.

13.3 Digital Outputs

The harmonica has two digital output pins. The digital output pins must be associated to a function before being used. The functions may be general purpose output, Amplifier Ready

(AOK) indication, or brakes output.
The association of output pins to functions is described in detail in the Command Reference Manual, the OL[] command.

13.4 Events, and response methods

The Harmonica identifies the following events[6]:

- Change of a GPI (General Purpose digital Input)
- Change of a limit switch
- Change of a "Home" switch
- Change of an "Enable" switch
- Change of a "Stop" switch
- Change of a "Begin" switch
- Encoder Index pulse.

In order to identify an event, the event needs to be defined. From the above list, only the Index is defined permanently. All the other events are programmable. The Harmonica has 6 digital input connector pins. The connector pins can be routed, with direct or inverted polarity, to functions like general-purpose digital input or RLS. To learn how to route connector pins to functions, refer the IL[] command in the Command Reference Manual.
Events can be handled in three speed levels.

- Manual inquiry
- Periodical inquiry
- Automatic routines
- Real time (Motion management, Homing, Capture, and Flag)

13.4.1 Manual inquiry

The manual inquiry is the simplest method. By using the IP/IB[] commands, one can inquire the state of the input pins. The user program logics, or a host, can consider the inquiry results.
The table below summarizes some of the properties off manual inquiry.

Topic	Comment
Capture probability:	Low. Even if a digital input is continuously polled (with great communication or user program loading), an input pulse may slip away unnoticed between consecutive polls.
Deterministic delay:	No. Depends in the non-deterministic delays of the interpreter or the user program management.
Use:	Non time-critical inquiries. Simple operations.

13.4.2 Periodical Inquiry

The periodical inquiry is possible only in CAN networks. The user can map the digital input word to a synchronous PDO. The host can collect the inputs state of many slaves as a response to single Sync.
The table below summarizes some of the properties off periodical inquiry.

Topic	Comment
Capture probability:	Low. An input pulse may slip away unnoticed between consecutive Syncs.
Deterministic delay:	Yes, 0.5msec maximum + CAN network time.
Use:	CAN networks only.

[6] The Harmonica also identifies emergency events such as over-voltage, short-circuit, or over-current. The emergency events are treated in the chapter on "Limits, Protections, Faults, and Diagnosis"

	Host logics only.

13.4.3 Automatic routines

Many events can be tied to automatic handler routines in the user program. To learn more about automatic user program routines, refer the section "Automatic subroutines".
The table below summarizes some of the properties off automatic routines.

Topic	Comment
Capture probability:	High. An input pulse may slip away unnoticed only if the user program is not running, the input is masked, or the pulse is so short it does not pass the input filters.
Deterministic delay:	No, although in most cases the automatic routine will start within 2msec.
Use:	User program logic.

13.4.4 Real time – Motion management, Homing, Capture, and Flag

Events can do several things in real-time response.
These are:

- Stopping the motor, or prevention move in one direction (refer the section on "The Stop management" and the chapter "Unit Modes")
- Begin a new motion (refer the BG command in the Command Reference Manual and the chapter on "Unit Modes")
- Disable the power amplifier
- Log the position counters
- Modify the value of a position counter, and optionally stop.
- Flag to an output connector pin.

The table below summarizes some of the properties off automatic routines.

Topic	Comment
Capture probability:	High. An input pulse may slip away unnoticed only if the user program is not running, the input is masked, or the pulse is so short it does not pass the input filters.
Deterministic delay:	Yes. Position logging and value setting are made immediately and yield accurate results regardless of the speed. Position flagging is made within 100usec. Motion management functions are responded at the next controller sampling time.
Use:	Homing (Setting an absolute origin to the incremental position sensors) Motion management. High speed signaling.

13.5 Homing and Capture

13.5.1 What Is Homing?

The Amplifier uses incremental position sensors. Therefore, the Amplifier can only tell the distance the motor traveled since power on, but not where the position counting started.
The process of teaching the Amplifier 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 12.2.1.

In the homing process, a trap is set for the event of reaching a desired position. 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 Amplifier 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. The capture accuracy depends, however, much in the homing event selection. If a HOME or an INDEX signal is selected as the captured event, then the captured position will not miss. If another digital input is selected, the sensing delay for the switch is d=(0.004*TS+IF[N]) msec, and VX*d/1000 counts may be missed.
Example
Suppose:

Parameter	Symbol	Value
Torque controller sampling time, usec	TS	50
Input filter width for digital input #1, msec	IF[1]	2
Motor main speed, count/sec	VX	20000

When homing on Digital Input #1, we expect to miss

$$\text{miss} = \frac{\text{VX} \cdot \left(0.004 \cdot \text{TS} + \text{IF}[1]\right)}{1000} = \frac{20000 \cdot \left(0.004 \cdot 50 + 2\right)}{1000} = 44\text{count}$$

To capture an event, the event must be first defined by a proper IL[N] setting. For example, if no digital input is associated to the FLS function, a homing process on the FLS will never terminate. Moreover, it is legal to un-define the FLS function while a home search on the FLS is on. The home search will not succeed until FLS is re-defined.

13.5.2 Homing Programming

The HM[] parameters control The main encoder homing process.

- HM[3] defines the trigger event – immediately, or the change of some digital input.
- HM[4] is what to do after the event, in addition to position registration. You can command the amplifier to stop immediately, or to flag a digital output.
- HM[5] defines if and how to update the main encoder counter – do nothing, set a new value, or shift by a known amount.
- HM[1]=1 arms the homing process (sets the trap for the homing event).

Refer the HM[] command in the Command Reference Manual for details.

13.5.3 Homing the auxiliary encoder

The HY homing function gives an additional homing and capturing function.

HY is very similar to HM, except that HY refers the auxiliary Home switch and Index signals instead of the main Home switch and Index, and that HY can update the value

of PY, not PX.

13.5.4 On the fly position counter updates

The updating of a position sensor during homing process has no effect in UM=1,2,3 since these modes do not use position feedback.
The implications of position sensor (PY in UM=4, PX in UM=5) update by a homing process depend in the mode.

- In PTP motions, the remaining motion to target becomes longer or shorter – refer example below. If the home correction is made in a constant speed region of the PTP motion, the redesign of the motion path may be hardly visible. This mechanism enables registration and final motion corrections on the fly.
- In Jog motions, the position command is jumped according to the position feedback, so that the motion is unaffected by the position counter update.
- If the software position reference generator is stopping or stopped, the software position command is corrected according to the position feedback. The motion is unaffected by the position counter update.
- In PVT or PT motions, an on-the-fly position counter update may lead to immediate high position error, and the motor may abort with an excessive position tracking error exception.

If with UM=5 the auxiliary encoder counter is modified, and the following is true:

- The motor tracks the auxiliary encoder with no ECAM table (RM=1, FR[3] nonzero, EM[1]=0).
- The software position generator is Idle, Jogging, or running PTP

The software position reference is modified so that the motion shall not be affected. In the PTP mode, the parameter PA will change automatically to reflect the modification.

Example:
A PTP motion starts with PA=PX=0. We set PA=1000;BG;, and expect an 1000 counts long motion. If, at PX=500, the position has been reset to zero by homing, then immediately after the homing we still have 1000 counts to go. The total length of the motion becomes 1500.

13.5.5 A homing with home switch and index example

Consider the very common switch arrangement of Figure 35.
A homing algorithm may be:
Start the motor
Jog back until RLS.
Jog forward speed until home.
Look for the next index and set the position there to 0.

The position setting is taken by the index, since in many applications the index is much more accurate then the home switch. The home switch is needed to resolve index ambiguity-many index pulses may occur along the travel.

In normal operation time, the FLS and the RLS serve as emergency indicators, and the motor is not supposed to reach there.
RLS is visited in the homing process, since then we don't know initially in which direction to look for the home switch.

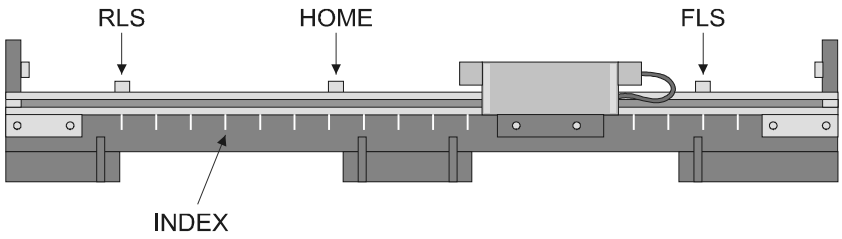


Figure 35 – Switches location

The following user program does the algorithm:

```
function [int status]  = homing1(int TimeOut)
/*
Homing routine.
Input:
TimeOut: Timeout for failure
Output:
status=1 if o.k., otherwise a negative error code
Assumptions:
RLS,FLS,HOME already programmed by IL[1],IL[2],IL[3] commands
*/
int OldMi;
/* Go reverse until limit switch */
OldMi=MI;MI=MI|0x16; /* Prevent operational AUTO_RLS routine while in homing process*/
/* Arm homing for ↑RLS, stop after homing, don't initialize counter*/
HM[3]=7;HM[4]=0;HM[5]=2;HM[1]=1;
/* Go to the reverse */
JV=-10000;BG;WaitArrive(2000);if(status<=0)goto LastLine; end
/* Go until home switch*/
HM[3]=1;HM[4]=2;HM[5]=2;HM[1]=1;
JV=-JV;BG;WaitHome(2000);if(status<=0)goto LastLine; end
/* Finallyset position by index */
HM[3]=3;HM[4]=0;HM[5]=0;HM[1]=1;
WaitHome(2000);if(status<=0)goto LastLine; end
status=1;/*Success return */
##LastLine
MI=OldMI; ** Restore AUTO_RLS routine status
return
```

Note that the function above uses MI to prevent RLS from activating the AUTO_RLS routine. Another possible approach would be to use IL[] to change the functionality of the switch to GPI[7], and then home on the GPI. With the latter approach, however, the routine programmer need to know which connector pin is programmed as RLS.
The algorithm used the following helper functions:

[7] General Purpose Input

```
function [int status]=WaitArrive(int TimeOut)
/*
Wait until MS=0, or until too much time elapses
*/
int StartTime ;
status = -1;
StartTime = TM;
while (MS)
        if ( tdif(handle) >= TimeOut) return ; end
end
status = 1 ;
return

function [int status]=WaitHome(int TimeOut)
/*
Wait until HM=0, or until too much time elapses
*/
int StartTime ;
status = -1;
StartTime = TM;
while (HM)
        if ( tdif(handle) >= TimeOut) return ; end
end
status = 1 ;
return

##ErrorOut
/* Error handler – just exit*/
return
```

13.5.5.1 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. The reason may be:

- Motor or gear repairs should not require the tuning of the index position or the Amplifier.
- 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 routine that does that is listed below. The routine uses the helper functions of the previous example.

```
function [int status]  = homing1(int TimeOut)
/*
Homing routine.
Input:
TimeOut: Timeout for failure
Output:
status=1 if o.k., otherwise a negative error code
Assumptions:
RLS,FLS,HOME already programmed by IL[1],IL[2],IL[3] commands
*/
int OldMi,Pos1;
/* Go reverse until limit switch */
OldMi=MI;MI=MI|0x16; /* Prevent operational AUTO_RLS routine while in homing process*/
/* Arm homing for ↑RLS, stop after homing, don't initialize counter*/
HM[3]=7;HM[4]=0;HM[5]=2;HM[1]=1;
/* Go to the reverse */
JV=-10000;BG;WaitArrive(2000);if(status<=0)goto LastLine; end
/* Go until falling home switch, stop, and capture position*/
HM[3]=2;HM[4]=0;HM[5]=2;HM[1]=1;
JV=-JV;BG;WaitHome(2000);if(status<=0)goto LastLine; end
Pos1=HM(7)
/* Sample home switch from other side */
HM[1]=1; JV=-JV;BG;WaitHome(2000);if(status<=0)goto LastLine; end
/* Final calculation – set immediate difference correction to PX*/
HM[2]=10000-(HM[7]+Pos1)/2;HM[3]=0;HM[4]=2;HM[5]=1;HM[1]=1;
status=1;/*Success return */
##LastLine
MI=OldMI; ** Restore AUTO_RLS routine status
return
```

Program Example 1 - Double Sided Homing.

13.5.6 Capturing

Capturing is a special case of homing, in which we program that the event will only register the counters value, but not affect motion and not update the position counters.
Capturing is efficient in synchronizing the motion origin to objects in the working space.
The Amplifier can capture two event defined by the HX and the HY commands, and register up to 4 consecutive occurrences of PX and PY of each event.
The capture function captures both the main and the auxiliary position counters simultaneously. This is useful for synchronizing the main and the auxiliary positions.

14 Limits, Protections, Faults, and Diagnosis

This chapter discusses the limits and the protections implemented by the Harmonica.
Limits are software restrictions that prevent the Harmonica from running into dangerous situations. Examples for limits are:

- Limiting the torque command prevents the burning of the motor or the Harmonica itself
- Reaction to limit switches stops the motor before it accidentally hits something out of its expected motion range.

Protections prevent motor starting, or shut an active motor, because of abnormal situations. Examples for protections are:

- Bad or inconsistent setup data prevents motor starting.
- An unexpectedly high motor current endangers the Harmonica and the motor. This high current is not normal, since the command to the current (torque) amplifier is limited. The suspect that the current controller does not function well implies immediate amplifier shut down.
- The amplifier is too hot.

When the amplifier shuts down by a protection, the motor will continue to run by its own inertia unless brakes are used (refer the section Connecting an external brake).
In order to avoid spurious motor shut downs, always:

- Leave enough space between the limits and the protection. For example, HL[2] specifies the over-speed limit. A protection is activated when VX > HL[2]. VH[2] specifies the maximum legal command to the speed controller. Speed commands over VH[2] are clipped to VH[2]. The protection HL[2] must be bigger then the limit VL[2]. Moreover, the distance HL[2]-VL[2] must leave enough space for the expected speed overshooting.
- When safety is concern, always use brakes. The Harmonica can program one of its digital outputs to activate a brake immediately upon motor shutdown.

If an exception stops the motor, or prevents motor starting, the Harmonica will in most cases be able to tell the user exactly what happened.

If a real-time exception reaction is desired, you can add an AUTO_ER routine in the user program – refer "Automatic Routines" in this manual.

The commands relevant to limits and protections are:

Command	Description
BP[N]	Brake parameters.
CD	Tell the health of the Harmonica, and report database inconsistencies.
CL[N]	Continuous current limit, and motor not moving protection.
EC	Error code – find why a previous command returned with error.
ER[N]	Tracking error exception limits for speed and position.
HL[N]	Protection high limits for position and speed
IL[N]	Input logic. Defines digital inputs as stop and limit switches.
LL[N]	Protection low limits for position and speed
MF	State the reason for a motor fault
OL[N]	Output logic – programs digital outputs as brake activation, or as amplifier ready indication.
PL[N]	Peak current limit
PS	Program status. If a program aborted, tell why.
SR	Status register. Tells if motor is shut by an exception, or if the user program is running.
VH[N]	Command high limits for position and speed

VL[N]	Command low limits for position and speed

Table 14-1: Commands relevant to limits, protection, and diagnosis

CAN users have an added protection level. The Harmonica issues Emergency objects in the case of errors or applied protections – refer the CAN manual.

14.1 Current limiting

The Amplifier 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 low-pass 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.

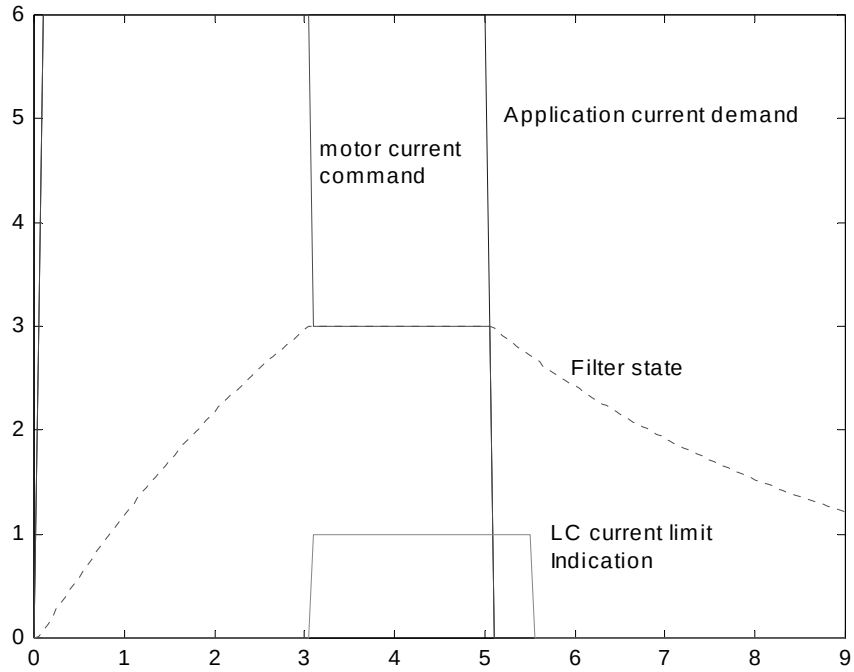
The time constant of the low-pass filter is $\tau = \frac{-PL[2]}{\log\left[1 - \frac{CL[1]}{MC}\right]}$, where MC is the maximum servo drive current.
The maximum time, for which the peak current can be maintained, after the current demand has been zero for a long time, is $-\log\left[1 - \frac{CL[1]}{PL[1]}\right] \cdot \tau$ 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 $-\log\left[1 - \frac{CL[1] - I1}{PL[1]}\right] \cdot \tau$ 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.



14.2 Speed Protection

The reference to the speed controller is limited to be in the range [VL[2]...VH[2]].
The limiting (applied on the sum of the software reference and the external reference) is made by the Stop-Manager.

In addition to the speed limiting, the Harmonica provides two forms of speed protections.

- Over error protection.
- Over speed protection.

The first protection detects a too big speed error.
The parameter ER[2] specifies the limit for the position error.
If the absolute value of the speed tracking error VE exceeds ER[2], The Harmonica issues the exception MF=0x80 and the power amplifier is shut down.

The second protection detects over speeding. If VX is greater than HL[3], or less than LL[3], The Harmonica issues the exception MF=0x200000 and the power amplifier is shut down.

The over-speed protection is active in all unit modes.
The speed for the over-speed detection is always derived from the main position sensor. This means that in UM=4, the motor speed is over-speed protected, not the load speed.

The speed command and feedback limits are illustrated in the figure below. The amplifier normally operates in the area marked "Within command and feedback limits". It may, however, visit instantaneously the areas marked "Overshoot area" due to overshoot, and this is Ok. If the speed feedback reaches the "Abort area", that amplifier will issue an over-speed exception and automatically shutdown.

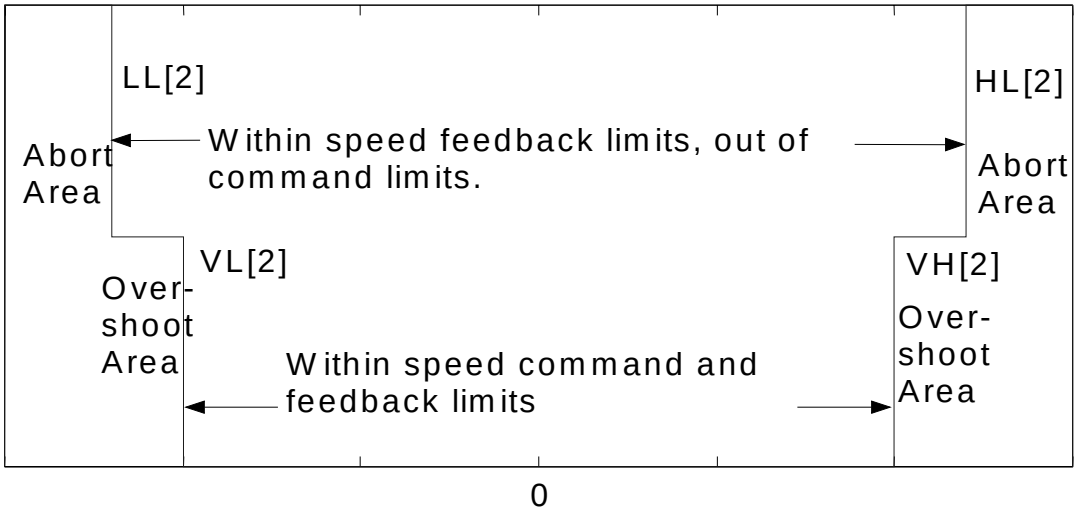


Figure 36: Speed command and feedback limits

! Remember that when the amplifier shuts down by exception, the motor will continue to run by its own inertia unless brakes are used (refer the section Connecting an external brake).

In order to avoid spurious motor shut downs, always:

- Specify the largest ER[2] you can tolerate
- Leave enough space between VH[2] and HL[2], to allow speed overshoots
- Leave enough space between VL[2] and LL[2], to allow speed overshoots.

14.3 Position Protection

The Harmonica provides two forms of position protections.

- Over error protection.
- Out of position range protection.

The first protection detects a too big position error.
The parameter ER[3] specifies the limit for the position error.
If the absolute value of the tracking error PE exceeds ER[3], The Harmonica issues the exception MF=0x100 and the power amplifier is shut down.

The second protection detects that the motor position is out of range. If the position feedback (PX in UM=4, PY in UM=5) is greater than HL[3], or less than LL[3], The Harmonica issues the exception MF=0x400000 and the power amplifier is shut down.

The position reference and feedback limits are inactive if the position counter counts modulo, since then the relations "Greater Than" and "Smaller Than" are not well defined.
The table below details the particular cases:

Case	Description
UM=1, nonzero XM	Torque control, limits taken by the main modulo feedback
UM=2, nonzero XM	Speed control by main modulo sensor.
UM=3	No feedback limits since there is no feedback
UM=4, nonzero YM	Position control by an auxiliary modulo sensor.
UM=5, nonzero XM	Position control by an main modulo sensor.

Table 14-2: Cases where feedback limits do not apply

The position command and feedback limits are illustrated in the figure below. The amplifier normally operates in the area marked "Within command and feedback limits". It may, however, visit instantaneously the areas marked "Overshoot area" due to an overshoot, and this is Ok. If the position feedback reaches the "Abort area", that amplifier will issue an out-of-position-range exception and automatically shutdown.

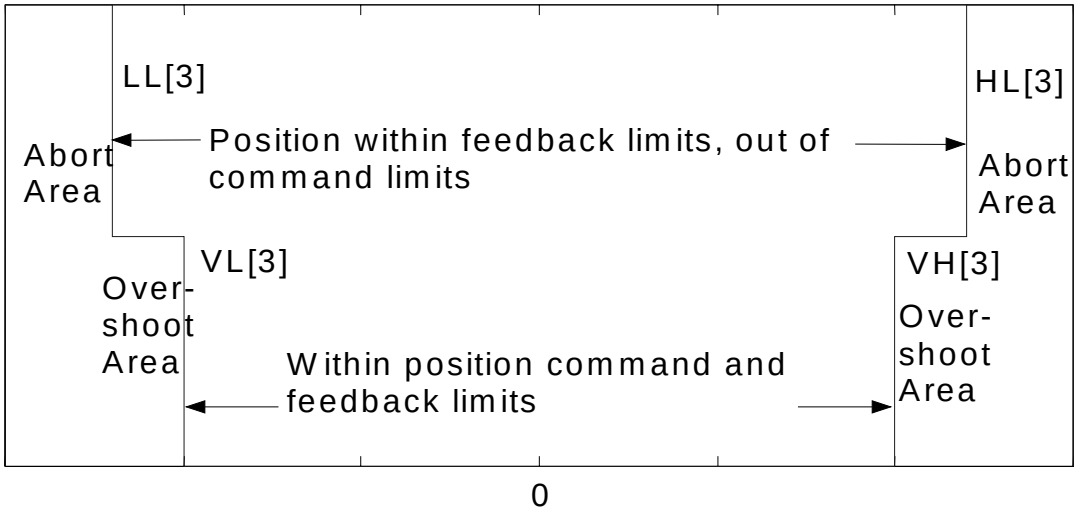


Figure 37: Position command and feedback limits

! Remember that when the amplifier shuts down by exception, the motor will continue to run by its own inertia unless brakes are used (refer the section Connecting an external brake).
In order to avoid spurious motor shut downs, always:

- Specify the largest ER[3] you can tolerate
- Leave enough space between VH[3] and HL[3], to allow positioning overshoots
- Leave enough space between VL[3] and LL[3], to allow positioning overshoots.

! If you want the motor to run infinitely in the current or in the speed (UM=1 or UM=2) modes, either set XM to nonzero value, or set LL[3]<-2^{30} and HL[3]>2^{30}.

14.4 Enable switch

You may program one of the digital inputs as an "Inhibit" or "Inhibit with Automatic Restart".
Digital input programming is detailed in the Command Reference Manual, the IL command.
When an Inhibit input is active:

- If the motor is off, MO=1 will not start the motor
- If the motor is on, MO=0 is set immediately. If the motor is rotating with high speed, the inhibit function may be unsafe, as the motor continues to run uncontrolled by its own inertia.

When an Inhibit input is inactive:

- If the motor is off, MO=1 will start the motor

When an Inhibit input with automatic restart is active, it acts similar to Inhibit.
When an Inhibit input with automatic restart is inactive:

- If the motor is off, and RM=1, the Harmonica shall try to set MO=1. No restart

attempts are made 10msec after the last motor shutdown, or 10msec after the last failure of MO=1.

For safety reasons, we recommend to program the Inhibit function as "active low". That prevents incidental motor starts when the input pin is disconnected, or its driving source is powered-down.

14.5 Limit switches

The Harmonica has 6 digital input pins. Each of the pins may be associated to a different function. A pin may function as a general-purpose input, or it can function as a motion limiter.
As a motion limiter, a digital input can stop the motion via the Stop-Manager, stop the Reference Generator, or limit the motion to a single direction.
In addition to stopping the motion, the digital input can activate an automatic routine in the user program.
Refer IL[N] in the Command Reference Manual to learn about function association to input pins.

14.6 Connecting an external brake

Connecting an electrical brake to a motor helps safety where:

- The load is unbalanced, and moves by its own weight when the motor is off.
- Motor freewheeling can't be tolerated in malfunction cases.
- Extreme motor deceleration is required in response to emergency event.

The brake activation must be synchronized to the motor-on/motor-off process.

- If the brake is engaged too much time after MO=0, the motor may "run away".
- If the brake is released only after the motor is on, then if a position error existed, the position controller integrated it. When the brake completely releases, the motor may jump.

The Harmonica can program a digital output as a "brake" output – refer the OL[N] command.
The brake output must be connected as follows:

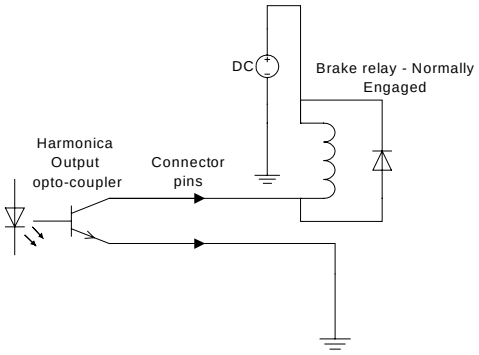


Figure 38: Brake output connection

- The brake must be engaged when there is no current in the brake coil.
- The brake relay coil must be equipped with a freewheeling diode.
- For the current and voltage rating of the Harmonica connector pins, please consult the Harmonica Manual.

The normal waveforms for brake activation are depicted in Figure 39.

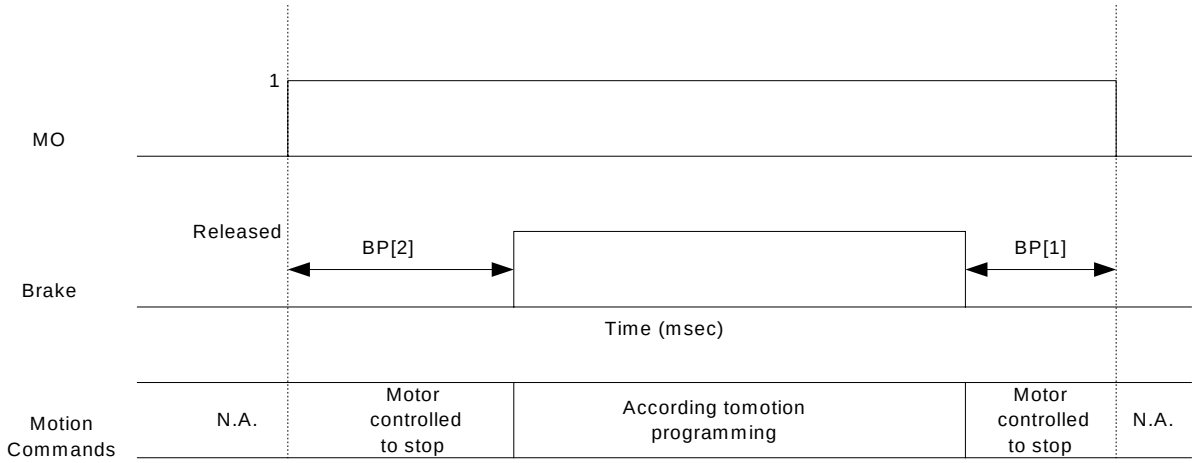


Figure 39: Normal Brake Activation Timing

At the brake activation times (BP[2] msec to disengage, BP[1] msec to engage) the motor is controlled to complete stop.
If MO=0 is set automatically by an exception, the brake is activated immediately, without any delay.

14.7 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 (refer the section "Inconsistent setup data" below)

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 not excessive, the Amplifier will try to start the motor.
For more details, refer the section "Current amplifier protections".

14.8 Motion faults

Error conditions may cause the Amplifier 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.

When the amplifier shuts down by exception, the motor will continue to run by its own inertia unless brakes are used (refer the section Connecting an external brake).
We recommend the use of automatic brake activation upon motor shut down (refer the OL command) for all safety critical application.

An amplifier exception shut down can be captured and reacted by placing an AUTO_ER routine in the user program.

For a full list of the exceptions that can cause automatic shutdown, refer the MF command in the Command Reference Manual.

14.9 Diagnosis

14.9.1 Monitoring motion faults

You can monitor motion faults by:

- Continuously polling the amplifier status
- Observing an "AOK" digital outputs.
- Trapping CAN Emergency objects. The CAN Emergency objects are covered in the CAN manual.

14.9.1.1 Polling the amplifier status

The amplifier can be polled using the SR command. The SR command reports a bit-field that draws an entire picture of the amplifier activity. One of the bits of SR (refer the Command Reference Manual) reports the existence of motion fault.
Another bit of SR reports the existence of a user program fault.
When a motor fault is detected, the MF command will report the exact reason for that fault – refer the Command Reference Manual.
When a program fault is detected, the PS command will report the fault reason.

14.9.1.2 Observing AOK

One of the digital outputs may be programmed to reflect that the amplifier is ready – no physical reason such as over-temperature, over voltage, or under voltage prevents the amplifier from working. The AOK output reports that physical operation conditions exist, but it tells nothing about the motion status.

14.9.2 Inconsistent setup data

The setup data is checked when loading the setup from the parameters non-volatile flash storage, and before starting the motor.
When the setup parameters are retrieved from flash storage (At power-on, or by an LD command), they are thoroughly checked for legality and consistency. If the parameters are found illegal or inconsistent, the amplifiers resets to its factory defaults. What was wrong can be found using the CD command (see below), and the contents of the flash storage can be corrected using the application editor.
Loading the setup parameters from the flash will rarely fail, since the parameters are checked before allowing non-volatile storage. The almost only failure reason is a major firmware revision upgrade.

Before enabling the motor, the Amplifier 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] equals CA[5], then two Hall sensors are assigned to the same position, which is an absurd (refer the chapter "Commutation").
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 (refer "Device failures, and the CPU dump" below).
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.
Other exceptions that may be caught at motor on are:

- RM=1 with a modulo-counting position feedback.

14.9.3 Device failures, and the CPU dump

Humans have programmed the Amplifier. 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 Amplifier has traps for CPU exceptions and for stack overflow.
When the Amplifier 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 Amplifier can be asked what had happened. Them motor, however, cannot be set again to work until the Amplifier 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 14-3 - CD Reported Elements

The CPU dump also returns the database status, as described above.

14.10 Sensor faults

14.10.1 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 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 Amplifier user is asked to define whether a high torque stopped motor is a fault or not. If the parameter CL[2] is less than 2, a high torque that does not lead to motion 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.
Do not set CL[3] to a very small number. When a motor is stuck, a vibration may develop that will induce a speed-reading. When an encoder wire is damaged, the motor will run-away with the encoder readout vibrating +/- bit. This also creates speed-reading.

Example
If CL[2]=50 and CL[3]=500, the Amplifier 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.

14.11 Commutation is lost

14.11.1 General

The Amplifier 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.

14.11.2 Reasons and effect of incorrect commutation

Commutation errors may be 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.

14.11.3 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 than 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.

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.

15 The Controller

15.1 General

This Section details the speed and position control algorithms. 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 the Composer program. This chapter is aimed for the advanced user who wants to tune the servo drive manually, or to understand thoroughly what goes on when the Composer tunes the servo drive. Familiarity with the basic ideas of Digital Control Theory is mandatory. This section does not cover the digital current loop.

The type of the controller used depends on the Harmonica usage mode- refer the chapter on "Unit Modes".

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 15-1 – Unit Mode Values and Definitions

In the basic level of the speed controller, the control algorithm is the traditional PI (Proportional-Integral). In the basic level of the position controller, the control algorithm is an internal PI speed loop and an external position loop using a simple gain for the controller, this structure is known as cascaded loop.

Few hours of playing will suffice for a technician to percept how the modification of the P,I and gain controller parameters affect the performance of the motor speed loop and position loop.

Life with the above control structures becomes hard as the control requirements become tough.
For this reason, the algorithms are extended to include

1. An high-order filter operating in series with the PI controller, with blocks of the following types:
   - Notch filters for notching resonance.
   - Low pass filters for attenuating very high frequency resonance and decreasing sensor's noise.
   - Lead-lag element, an extra element that helps tradeoff margins versus bandwidth.
2. A continuous scheduling algorithm, which modifies the PI algorithm and some of the advanced filter parameters as a function of the closed loop operating point.

An advanced user may tune the various filter blocks, but this is not as simple as tuning the PI and the simple gain. 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, dual-position, or open loop
KP[N]	N=2: Inner speed loop proportional gain.

	N=3: Outer loop gain (UM=4 and UM=5)
KI[N]	N=2: Inner speed loop Integral gain, I.
KV[N]	Coefficients for the high-order filter. The parameter KV[0] asserts if these filters are used at all – if KV[0] is zero, then the advanced filter is not used.
GS[N]	Gain scheduling parameters. More details see chapter "**Error! Reference source not found.**"
MC	Maximal motor phase current
WS[28]	Sampling time of the controller in microseconds.

Table 15-2 – List of all control parameters

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 $\int x dt$, 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

$$zx(kT_s) = x[(k+1)T_s] \text{ or simply } zx(k) = x(k+1).$$

15.2 Speed Control

15.2.1 Block diagram

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 high order filter (composed of a series of blocks of the types explained above) and the gain scheduler. A block diagram of the speed controller is given in Figure 40.
The gain scheduler can be used, or the gains of the controller may be fixed.
The high order filter can be used, or not.
To fix the controller gains, set GS[2]=0.
To avoid the use of the high order filter, set KV[0]=0.

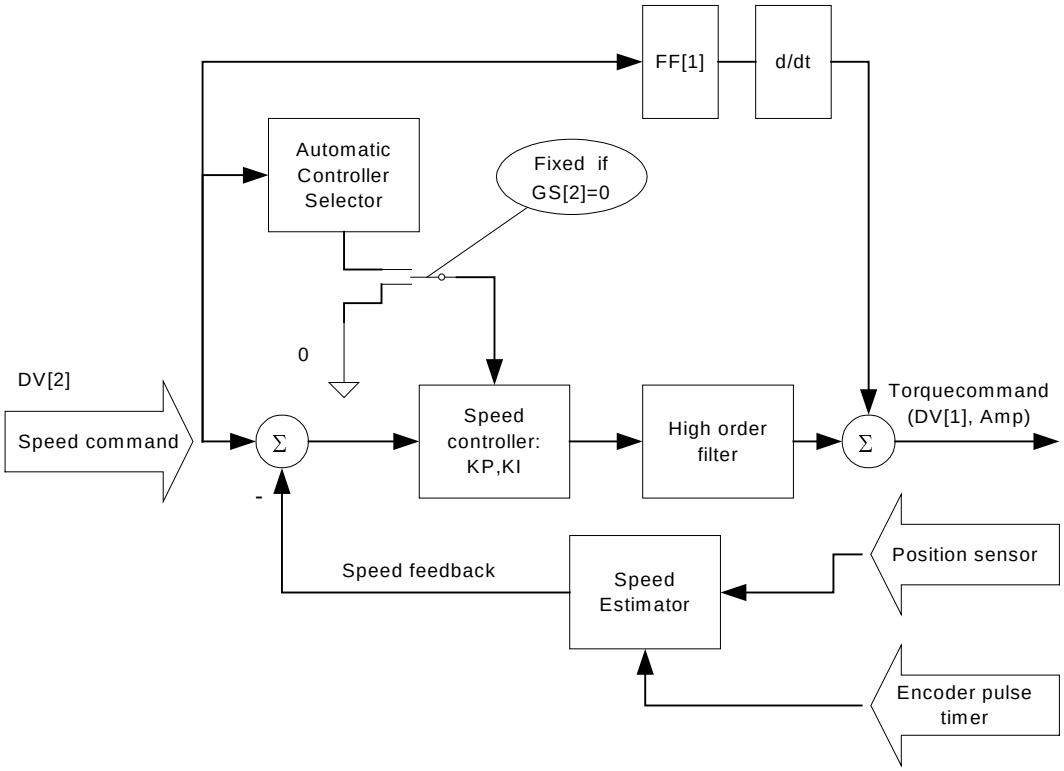


Figure 40 – A Block Diagram of the Speed Controller

15.2.2 The Parameters of the Speed Controller

The basic continuous time PI controller is

$$\frac{K_I + K_P s}{s},$$

where:

K_I is the integral parameter, and

K_P is the proportional parameter

When using the scheduler, K_P and K_I are functions of time. When not using the gain scheduler, they are fixed.

The input to the PI element is the speed error, $e_{SPEED}(t)$ [Internal speed units]

$$e_{SPEED}(t) = SpeedCommand(t) - SpeedFeedback(t)$$

The output is the current command $I(t)$ in Ampere units:

$$I(t) = K_{Speed}\left(\int_0^t K_I \cdot e_{SPEED}(\tau)d\tau + K_P \cdot e_{SPEED}(t) \cdot p(t)\right)$$

K_{Speed} is the conversion factor from D/A scale to current in [Amp]

$$K_{Speed} = \frac{MC}{MC\_VALUE\_BITS},\ MC\_VALUE\_BITS = 14000$$

$$p(t) = \begin{cases} 0 & if\ \ no\ encoder\ counts\ \ for\ GS[0]\ samples\ and\ \left|\frac{1}{32768}\int e_{SPEED}dt\right| < GS[14]count \\ 1 & otherwise \end{cases}$$

For the non-scheduled case

$$K_P = KP[2]$$

$$K_I = KI[2]$$

The scheduled case is explained in the section on “Gain Scheduling”.

The parameter GS[0] helps stabilizing the motion in very slow speeds. It cuts the proportional gain of the speed controller after enough controller sampling times elapsed without a change in the encoder readout. Consider for example a 200-count/sec, speed reference. A new encoder count is available once per 5 msec – which are about 25 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. Setting GS[0]=12, the proportional gain of the controller is applied only for about half of the time, leading to a practical reduction of the proportional gain to half at that speed. The reduced gain implies reduced bandwidth and increased stability. The price is a slow acceleration from a complete rest. To avoid the cutting of the speed proportional gain, set GS[0] to its maximum value – refer the GS command in the Command Reference Manual.

The parameters of the non-scheduled speed controller are:

Parameter	Description
KP[2]	Proportional gain
KI[2]	Integral gain
GS[0]	Proportional gain duration
GS[2]	Controller gain selection
GS[14]	Speed reference integral threshold
FF[1]	Torque feed forward
CL[1]	Continuous torque limit
PL[1]	Peak torque limit
KV[N]	High order filter parameters

15.3 The Position Controller

15.3.1 Block Diagram

The position controller is made of a proportional gain, cascaded over the speed controller. The block diagram of the position controller is given in Figure 41. The reference to the speed controller is composed of the derivative of the position command (speed) and of the output of the auxiliary position controller. The derivative of the position command is multiplied by the FF[2] factor, in order to eliminate the tracking errors when the speed is steady.

The gain scheduler can be used, or the gains of the controller may be fixed.
The high order filter can be used, or not.
To fix the controller gains, set GS[2]=0.
To avoid the use of the high order filter, set KV[0]=0.

The same controller is used for the single feedback mode (UM=4) and for the dual feedback mode (UM=5). Use the single feedback mode when there is only one sensor in the system, for commutation, speed, and position sensing. Use the dual feedback mode when the load is driven through a gear, where a separate load sensor enables precise load positioning in spite of some backlash and gear compliance.

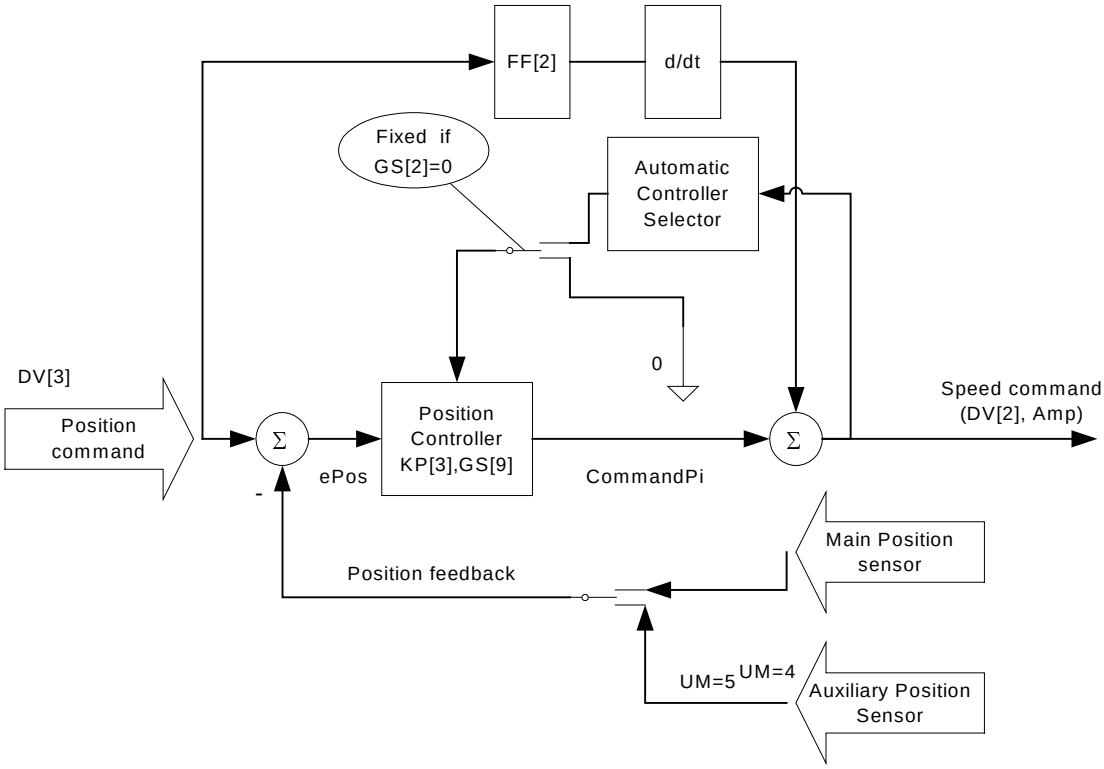


Figure 41 – Block Diagram of The Dual Loop Controller

15.3.2 The Parameters of the Position Controller

The position controller is implemented as a cascaded loop, the inner loop is a speed controller and the outer loop is a simple gain: The simple gain operates on the position error, $e_{POS}(t)$:

$$e_{POS}(t) = PosCommand(t) - PosFeedback(t) \quad \text{[counts]}$$

to give [Internal speed units, see "Units" section]

$$CommandPI(t) = K_P^{Out} \cdot e_{POS}(t).$$

However when the position error, $e_{POS}(t)$, is too large the gain is modified to avoid instability, K_P^{Out} is replaced by

$$\min(K_P^{Out} \cdot abs(e_{POS}(t)), \sqrt{2 \cdot \alpha \cdot abs(e_{POS}(t))}) \cdot sign(e_{POS}(t)).$$

where

$$\sqrt{2 \cdot \alpha} = GS[9]$$

The above formula assures that the speed controller will not be demanded to decelerate beyond α count/sec^2. GS[9] must be calculated with α as the 80% of the highest deceleration the motor can produce with the current of PL[1].

The inner loop is a speed controller, described previously in this chapter.

The parameters of the non-scheduled position controller are:

Parameter	Description
KP[3]	Proportional gain , K_P^{Out}
GS[2]	Controller gain selector
GS[9]	Acceleration limit
VL[2],VH[2]	Maximum speed command
FF[2]	Speed Feed-forward
UM	For UM=4 the position feedback is taken from the auxiliary encoder

	For UM=5 the position feedback is taken from the main encoder.
…	All the parameters of the speed controller

15.4 The High Order Filter

15.4.1 Block Types

Several blocks chained in series make the high order filter. The block types are listed in the table below.

Block type	**Block structure**	**Number of parameters**	**Comment**
12	Double lead-lag block	2	
14	1-st order block, lead-lag or a pole	2	
15	2-nd order, notch or complex pole	4	

15.4.1.1 Double lead block (Block type=12):

The basic continuous time double lead-lag element is

$$\left(\frac{b}{a}\cdot\frac{s+a}{s+b}\right)^2.$$

The frequency $a/(2\pi)$ [Hz] is the lead corner frequency. The frequency $b/(2\pi)$ [Hz] is the lag corner frequency.
The discrete equivalent form is

$$\left(G+(1-G)\left(\frac{z(1-\beta)}{z-\beta}\right)\right)^2$$

Order	**Parameter**	**Description**	**Comment**
1	$1-\beta$	$0<\beta<1$	Float, represented by 2 long values
2	G	0<G<1	Float, represented by 2 long values

The DC gain of this block is unity.
β is selected as $\beta = e^{-bT}$.
G is selected by $e^{-aT} = \frac{G\beta}{1-G\beta-\beta}$.

15.4.1.2 Fist order block (Block type=14)

The basic continuous time single lead-lag element is

$$\frac{b}{a}\cdot\frac{s+a}{s+b}.$$

The discrete equivalent form is

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

where $p = \frac{\beta - \alpha}{1 - \alpha}$, $q = 1 - \beta$, i.e. $\beta = 1 - q$, $\alpha = 1 - \frac{q}{1 - p}$.

Order	Parameter	Description	Comment
1	p		Float, represented by 2 long values
2	q		Float, represented by 2 long values

15.4.1.3 Second order block (Block Type=15)

The basic continuous time second order element is filter

$$\frac{D}{B} \cdot \frac{Es^2 + As + B}{Es^2 + Cs + D}$$

Note that it can be used as a notch filter or as a low pass complex pole filter.
The equivalent discrete form is

$$\frac{b_0 z^2 + b_1 z + b_2}{z^2 + (b_0 + b_1 + b_2 - a_2 - 1)z + a_2}$$

Order	Parameter	Description	Comment
1	k_1	$b_0 + b_1 + b_2$	Float, represented by 2 long values
2	k_2	$-(b_1 + b_2)$	Float, represented by 2 long values
3	k_3	$-b_2$	Float, represented by 2 long values
4	k_4	a_2	Float, represented by 2 long values

$b_0 = k_1 + k_2$, $b_1 = k_3 - k_2$, $b_2 = -k_3$, $a_2 = k_4$.

15.4.1.4 Scheduled Double lead block (Block type=22):

The basic continuous time double lead-lag element is

$$\left(\frac{b(k)}{a(k)} \cdot \frac{s + a(k)}{s + b(k)} \right)^2 .$$

The index k is the gain scheduler selector: it selects one set of (a,b) from the 63 possible selections (a(1),b(1)) , (a(2),b(2))....(a(63),b(63)).
The frequency $a(k)/(2\pi)$ [Hz] is the lead corner frequency. The frequency $b(k)/(2\pi)$ [Hz] is the lag corner frequency.
The discrete equivalent form is

$$\left(G(k) + (1 - G(k)) \left(\frac{z(1 - \beta(k))}{z - \beta(k)} \right) \right)^2$$

Order	Parameter	Description	Comment
1	$\beta(k)$	$0<\beta(k)<1$	Floating point
2	G(k)	$0<G(k)<1$	Floating point

The DC gain of this block is unity.

For more details about gain scheduling, refer the section " The Gain-Scheduling Algorithm".

15.4.2 User Interface

The parameters defined by an array of long integer values set by the KV command.
If KV[0]=0, high order filter is not used and the vector is ignored.
If KV[0] is not zero, the KV[2]..KV[n] defines the filter structure, where KV[1]=n, is the last index used.
Anyway n <= 100.
The filter consists of a number of blocks. It is possible to configure a filter flexible by user interface command KV, i.e. to define the list of blocks of given types.
KV[0]=100
KV[1]=Filter_Length (last index of KV used)
Filter_structure
{
Block1{
KV[2] =Block_type
KV[3]… =Block_parameter_list
}
Block2{
}
…
Block_Last
{
KV[n]=Block_type=0
}
}
KV[last-2]=0 (reserved)
KV[last-1]=0 (reserved)
KV[last]=-1 – terminator of list

Each block is represented as a list of parameters. The first parameter in each block is the block type, represented by one KV array element; the block type defines the length of the block parameter list and the meaning of every parameter.
Other parameters of a block are float values. Every float value x is represented by a pair of KV elements (Man, Exp) so that
X = Man*exp(Exp),
`-2^31 <= Man <= 2^31-1,`
-1000 <= Exp <= 32
The following formulae give the appropriate values:
If X=0 the Exp =-1000
If X>0 then
Exp = `floor(log2(X))-30;`
`If X <0`
`Exp = ceil(log2(-X))-31;`
Man = floor(`X / 2^Exp`)

15.4.2.1 An Example

Consisted a filter made of one 2nd-order block and one 1st-order block.
The second order block is a notch filter with 300Hz notch frequency and damping of 0.14. The sample time is
400 us. The discrete equivalent of the filter is parameterized by k1 = 0.4038, k2=0.3447, k3= -0.7414, k4= 0.4900
The second block is simple pole with frequency 400 Hz. It may be represented by
p=0.3659, q = 1-p =0.6341

The KV parameters vector is be programmed as follow:

KV [0]	100		
KV[1]	19	Last index used	
KV[2]	15	1st block, 2-order	
KV[3]	1734476372	Parameter 1 (k1), man	
KV[4]	-32	Parameter 1 (k1), exp	
KV[5]	1480626866	Parameter 2 (k2), man	
KV[6]	-32	Parameter 2 (k2), exp	
KV[7]	-1592218865	Parameter 3 (k3), man	
KV[8]	-31	Parameter 3 (k3), exp	
KV[9]	2104573671	Parameter 4 (k4), man	
KV[10]	-32	Parameter 4 (k4), exp	
KV[11]	14	2nd block, 1-order	
KV[12]	1571662995	Parameter 1 (p), man	
KV[13]	-32	Parameter 1 (p), exp	
KV[14]	1361652150	Parameter 2 (q), man	
KV[15]	-31	Parameter 2 (q), exp	
KV[16]	0	End of the filter blocks	
KV[17]	0	Reserved	
KV[18]	0	Reserved	
KV[19]	-1	End of filters	

The value of KV[N] parameters for N>19 is not important.

15.5 The Gain-Scheduling Algorithm

The gain scheduling (GS) is implemented for speed and position controllers. This means, that the controller parameters (Speed controller KI, KP, Position controller gain, and some links of the high order filter) are changed automatically with the working point of the servo drive.
Gain scheduling solves the following problems:
Low Resolution Position Sensor: When the position sensor has low resolution, the speed is derived from the position data with large time delay, typically half of the time between consecutinve encoder pulses. The speed measurement delay risks the controller stability. In order to maintain stability in low speeds, the controller gains are reduced. For high speed, the controller achieves its full possible bandwidth.
The Harmonica can schedule of the controller gains automatically, as a function of the command to the speed controller. In the speed unit mode, the command to the speed controller is a function of the reference generator only. For the position control modes (UM=4,UM=5) the reference to the speed controller is the sum of the position command derivative, and the output of the position controller.

Load and Posture Variations When controlling an articulated manipulator, the inertia that some of the manipulator motors see varies much with the posture of the manipulator. Resonance frequency may also vary considerably with varying posture or load conditions. By gain scheduling, one can switch the active controller parameters, as a function of posture and load.

The following table describes the programming of 63 controller parameters sets:

Parameters	Name	Description
KG[1]...KG[63]	*SpeedKiTable*	*SpeedKiTable[k]* is the speed KI gain for the k'th controller
KG[64]...KG[126]	*SpeedKpTable*	*SpeedKpTable[k]* is the speed KP gain for the k'th controller
KG[127]...KG[189]	*PositionKpTable*	*PositionKpTable[k]* is the position KP gain for the k'th controller
KG[190]...KG[252]	*DoubleLeadBeta*	*DoubleLeadBeta[k]* is the β parameter for the double lead element of the k'th controller
KG[253]...KG[315]	*DoubleLeadG*	*DoubleLeadG[k]* is the G parameter for the double lead element of the k'th controller

Table 15-3: Programming sets of controller parameters for gain schedulilng

Only a subset of the controller parameters can be schedualed.
The KG[] parameters can program sets of KP,KI gains, as well as one scheduled double-lead block at the high order filter.
The other elements of the high order filter cannot be gain scheduled.

The scheduling is automatic for GS[2]=64.
GS[2]=0 selects the controller parameters of KP[2],KI[2],and KP[3].
For other values of GS[2], the value of GS[2] selects directly which of the available sets of controller parameters is used.
Example:
If GS[2]=1, the proportional gain of the speed loop will be *SpeedKpTable[1]*.By Table 15-3, *SpeedKpTable[1]*=KG[64].

15.6 Automatic Controller Gain-Scheduling

The speed reference controls the controller gain scheduling. The goal of the algorithm is to decrease the gains when the speed is low, since when speed is low, large delay is added to the encoder measurement, forcing low bandwidth.

The gains are scheduled according to the reference speed. If the desired speed is high, then the encoder data is updated at high rate, so that the speed sensing delay will be low. 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 an average of 2.5msec delay that may kill the stability of the control algorithm. For that reason, if the speed is 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 scheduled parameters, KP and KI and some of the advanced filter parameters, 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 of the selection sets of speed controller parameters is stable controller for its goal speed)

The speed range where the scheduling is effective is about 0..20000 count/sec.

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(0:63).*

The actual parameter indexing process goes as follows:

Saturate speed command	$X = \max(\|\text{speed reference}\|, 20000 \text{ count/sec})$
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	$Y(m+1) = \beta \cdot Y(m) + (1-\beta)X$ where $\beta = \begin{cases} GsFilterUpGain, \text{ if } X \geq Y(m) \\ GsFilterDnGain, \text{ if } X < Y(m) \end{cases}$

commands are accepted slowly to guarantee stability.	
Limit the schedulink below	Y = max (Y , *MinSpeed*)
Get the index to the gains table by interpolating the indexing function	$n = [Y / 4096]$ $q = (Y / 16 \bmod GS[15]) / GS[15]$ $k = SpeedIndexTable[n] +$ $(SpeedIndexTable[n+1] - SpeedIndexTable[n]) \cdot q$
Get the gains	$KP = SpeedKpTable[k]$ $KI = SpeedKiTable[k]$ *Kpos = PositionKpTable[k]* (UM=4 and UM=5 only) $\beta(k)$ =*DoubleLeadBeta[k]* G(k)=*DoubleLeadG[k]* (if a block of type #22 used in high order filter)

Table 15-4 – Automatic Gain Scheduling Process

GS[N] and KG[N] programs the parameters of the automatic gain scheduling process, as follows:

Parameter	Description	Range
GS[1]	*MinSpeed:* minimal speed command for speed and dual gains scheduling [cnt/sec]	0.. 16*62*GS[15] internal speed units (1 Speed unit = Cnt/Ts / 2^16)
GS[2]	Use automatically speed-scheduled gains: 64 – yes	0,64
GS[4]	*GsFilterUpGain* 32768* Upwards gain of the gain scheduling filter	0.. 32767
GS[5]	*GsFilterDnGain* 32768* Downwards gain of the gain scheduling filter	0.. 32767

Table 15-5 – Gain scheduling automatic indexing parameter

16 Appendix A: The Harmonica Flash Memory Organization

This Appendix describes the internal partitioning of the internal serial Flash memory of the Harmonica.
For normal use of the Harmonica, you don't need to read this appendix, since the Composer IDE will deal with it for you.
To learn how to upload and download flash data, refer the DL,LS, and LP commands in the Command Reference Manual.

16.1 Main partitions

The flash has four main partitions at the following order.

- The firmware partition
- The parameters partition
- The factory code partition
- The user code partition

The firmware partition always starts at address zero of the serial flash.
The location of the other partitions is defined in the table of contents of the firmware partition.

16.2 The firmware partition

The firmware partition includes identity information for the driver.
It includes the following:

- Table of contents
- Text1 - description of the contents of the parameters 1 segment
- Text2 - unused memory space for possible later use
- Text3 - unused memory space for possible later use
- Text4:7 – description of the commands supported by the Harmonica
- Text8 – description of the available recorded signals
- Text9 – system limits
- Text10 – error codes descriptions
- Binary1 – record of default values for parameters1 segment
- Binary2 – unused memory space for possible later use
- Binary3 – unused memory space for possible later use

The firmware partition can be accessed by DL command only when we make download firmware when running from boot memory.
The firmware partition can be read by the LS command at any time.

16.2.1 Table of Contents (TOC)

This table of contents gives the starting address and the length for each of the text files, relative to the start of the firmware partition. Each text file starts in a page boundary. The table below specifies how the TOC is built.
This TOC is located at physical address 0 in the serial flash.
For example, the third line of the table is:

Length of Text1	unsigned short	2	

It means that the parameter "Length of Text1" can be read as
Length_of_Text1 = * ((unsigned short *)(Firmware_Partition+2))

Name	Size	Byte location in the firmware partition	Comment

Start of Text1	unsigned short	0	
Length of Text1	unsigned short	2	
Start of Text2	unsigned short	4	
Length of Text2	unsigned short	6	
Start of Text3	unsigned short	8	
Length of Text3	unsigned short	10	
Start of Text4	unsigned short	12	
Length of Text4	unsigned short	14	
Start of Text5	unsigned short	16	
Length of Text5	unsigned short	18	
Start of Text6	unsigned short	20	
Length of Text6	unsigned short	22	
Start of Text7	unsigned short	24	
Length of Text7	unsigned short	26	
Start of Text8	unsigned short	28	
Length of Text8	unsigned short	30	
Start of Text9	unsigned short	32	
Length of Text9	unsigned short	34	
Start of Text10	unsigned short	36	
Length of Text10	unsigned short	38	
Start of Binary1	unsigned short	40	
Length of Binary1	unsigned short	42	
Start of Binary2	unsigned short	44	
Length of Binary2	unsigned short	46	
Start of Binary3	unsigned short	48	
Length of Binary3	unsigned short	50	
Start of factory code partition	unsigned long	80	Length of factory code partition is constant: 5 kwords or 10 kbytes
Start of parameters partition	unsigned long	84	Length of parameters partition is constant: 5 kwords or 10 kbytes
Start of user program partition	unsigned long	88	Length of user code partition is calculated based on flash size.

All the unused page entries are set to zero.

16.2.2 Contents of Text1

This text duplicates the definitions of CUserParameters of commonvars.h

16.2.3 Contents of Text2

Unused memory space for possible later use

16.2.4 Contents of Text3

Unused memory space for possible later use

16.2.5 Contents of Text4-Text7

This text specifies which op-codes a specific the Harmonica supports, and also the limitations: Size of available flash space (may vary with Harmonica grades), size of data segment in RAM, maximal number of variables, maximum number of functions, and Maximum stack depth.
The **text4** file specifies the supported mnemonics. By "mnemonic" we mean a case insensitive combination of two letters, which refer to a system service. It contains comma-delimited list of 676[8] numbers. Each number corresponds to a combination of two English letters. Each letter is given a value – A is 0, B is 1, C is 2, and so on until Z=25.
The first of the 676 numbers corresponds to the two letters combination with the value of 0 – that is AA = 0*26 + 0. The second number corresponds to value of 1 which matches the combination AB = 0*26+1 until the last number 675 which matches the combination ZZ=26*25+25.
Each number of the list specifies the index of the service function for a specific mnemonic. The number –1 specifies that the mnemonic is not connected to any service function.
The list –1,34,67,-1,-1,... specifies that from the mnemonics AA,AB,AC,AD, and AE, only AB and AC are meaningful – AB is connected to the service function of index 34 and AC is connected to the service function of index 67.

The **text5** file describes the virtual assembly commands that the Harmonica can service, and specify the index of their service functions. Text5 contains a list with the following format:
MLT , 0
SUB , 1
ADD , 2
DIV , 3
REM , 4
RSLTAND , 5
RSLTOR , 6
...
From the first line of the list we understand that there is a virtual assembly command named MLT. MLT is implemented by the service function of index 0. The next lines are interpreted similarly.
The list of HARMONICA op codes is given in Section **Error! Reference source not found.**:

The **text6** file lists all the system functions. It has the format
rnd , 34 , 1 , 1
sqrt , 35 , 1 , 1
...
The first line specifies that the rnd (round to nearest integer) function is serviced by the service function of index 34, and that it has 1 input argument and 1 output argument.
Next lines are interpreted similarly.

The **text7** file specifies the type of service given by each of the service functions. It has the format
34562
8232

Each number is a bit field which its composed according to the table below.
Each number describes the nature of a specific command, system function or virtual assembly code.

Bit	Description	Meaning
0x1	FLAG_Float	System parameter type is float
0x2	FLAG_Long	System parameter type is long
0x4	FLAG_PreProcess	Return an answer based on internal data processing
0x8	FLAG_PostProcess	Process the data for internal use
0x10	FLAG_BitField	System parameter is a bit field

[8] There are 676 possible combinations of two English letters. There are 26 letters and 676 = 26 * 26.

0x20		Not used
0x40	FLAG_Assign	System parameter can be assigned a value
0x80		Not used
0x100	FLAG_Operator	Virtual Assemby Code or operator
0x200	FLAG_SysFunc	System function
0x400	FLAG_Mnemonic	System command - Mnemonic
0x800	FLAG_String	Indicate this is a string command
0x1000	FLAG_Array	Command is not executed automatically upon reset
0x2000	FLAG_ResetIgnore	Indicate this is a string command
0x4000	FLAG_NotProgram	Command can't be used from a user program
0x8000	FLAG_NotPdo	CAN binary interpreter can't use command.

Example:
The value of 4100 is interpreted as 0x1000+0x4. It means that the related command is a report that returns a long integer.
The value of 4164 is interpreted as 0x1000+0x40+0x4. It means that the related command refers to a parameter that can be either set or reported.

16.2.6 Contents of Text8

This text duplicates the definitions of the struct RecVarType within the file "ComVars.h". The first line of the file is:
Recorded Signals = xxxx
Where xxxx is the number of recorded signals.
Next come xxxx text lines with the format
{ RV index, Float_Flag , Length , Text name}
RV index specifies the signal index for use with the Harmonica RV command.
Float_Flag = 1 for floating point quantity, 0 for integer
Length = 4 for short, 8 for long
Text name is a delimited text signal name like "Main Speed"
Last comes
End Recorded Signals

16.2.7 Contents of Text9

This file specifies the setup of the Harmonica.
It has the .ini format:

[Program Limits]
TextSpace=xxx
CodeSpace=xxx
MaxUserFuncs=xxx
MaxUserVars=xxx
NameLength=xxx
StackDepth=xxx
DataSpace=xxx
UserPartMaxSize=xxx

[Automatic Functions]
AUTO_ERR=1
AUTO_DIN1=2
...

The [Program Limits] entries are:

Entry	Meaning	Remarks
TextSpace	The space (in characters) to store user program backup. This numbers specifies the entire space – including overhead, such as text-opcode matching data. The net text length is smaller that this number.	
CodeSpace	The space (in characters) to store compiled op-code.	
NameLength	The maximum length (in characters) of a symbolic function or variable name.	
StackDepth	The maximum depth of the stack of a specified VAC machine	
DataSpace	The space for global user variables, in words.	
MaxUserFuncs	The maximum number of user functions and labels.	
MaxUserVars	The maximum number of user variable names (either local or global)	
UserPartMaxSize	The maximal size of user code partition in the flash in bytes	This parameter is absent at the flash firmware partition

The parameter “UserPartMaxSize” is calculated according to the formulae:
UserPartMaxSize = TotalFlashSize – UserPartStart,
where TotalFlashSize is a maximal physical memory space of the flash and UserPartStart is a start address of the user code partition (see TOC at the address 88).

The [Automatic Functions] entries detail which events can be responded by dedicated automatic routines, and what are the priorities. The list of events and priorities is hard coded into the Harmonica.
The list of Harmonica automatic routines is:

Entry	Meaning
AUTOEXEC	Priority for AUTOEXEC – the routine that executes automatically upon power-up
AUTO_ER	Priority for AUTO_ER – the routine that executes automatically upon motor fault event
AUTO_STOP	
AUTO_BG	
AUTO_RLS	Priority for AUTO_RLS – the routine that executes automatically upon change on Logic Level of RLS (Reverse Limit Switch)
AUTO_FLS	Priority for AUTO_FLS – the routine that executes automatically upon change on Logic Level of FLS (Forward Limit Switch)
AUTO_ENA	
AUTO_I1	Priority for AUTO_I1 – the routine that executes automatically when Digital Input #1 goes high
AUTO_I2	Priority for AUTO_I2 – the routine that executes automatically when Digital Input #2 goes high
AUTO_I3	Priority for AUTO_I3 – the routine that executes automatically when Digital Input #3 goes high
AUTO_I4	Priority for AUTO_I4 – the routine that executes automatically when Digital Input #4 goes high
AUTO_I5	Priority for AUTO_I5– the routine that executes automatically when Digital Input #5 goes high
AUTO_I6	Priority for AUTO_I6 – the routine that executes automatically when Digital Input #6 goes high

If the priority is –1 if the Harmonica can't service this function.

16.2.8 Contents of Text10

This file lists all the possible run-time errors of a specific Harmonica, including their codes. Text10 has the following format:
1 , motor must be off
2 , out of range

In each line the number is the error code and the string follow it is the meaning of the error. For example if error code is 1 we know that motor is on and it must be shut down.
Another usage of text10 presence in the serial flash is by having the ability to display the error description for the user instead of just showing him the error code.

16.2.9 Contents of Binary1

To be read by the reset procedure to UserPars

16.2.10 Contents of Binary2

Unused memory space for possible later use

16.2.11 Contents of Binary3

Unused memory space for possible later use

16.3 Parameters Partition

The parameters partition includes the UserPars parameters.
The parameters are saved to flash by SV command and loaded to ram by LD command.
The parameters are saved in the serial flash due to space limitation in the flash of the DSP56F800 processor.
The parameters partition occupies the fixed 10kbyte space of the flash.

16.4 Factory Code Partition

The factory code partition includes factory programs. Factory programs are:

- Routines used for the auto-tuning process
- Functions given for various other purposes.

The firmware partition can be accessed by DL command only in download firmware.
The firmware partition can be accessed by LS command.
The factory code partition is defined only for future compatibility.
The Factory Code partition shall be loaded as a part of the firmware downloading process. The Factory Code shall be later accessible for running as any other user program compiled code.
The parameters partition occupies the fixed 10kbyte space of the flash.

16.5 User Code partition

The user code partition carries the information concerning the user program.
The user code partition can be accessed by DL command.

The user code partition can be read by the LS command.
The data in this partition is:

- A TOC (Table of Contents)
- A "Compilation done" flag
- A virtual machine code segment
- A text backup segment. The text backup includes virtual code counter references.
- A function symbol table
- A variable symbol table
- An automatic function table

Each partition starts at a page (264 bytes) boundary.

16.5.1 The TOC

The table of contents is located at the starting address of user code partition.
Each of the items below is relative to the start of the user code partition.
The length of the TOC is one flash page (264 byte).
The TOC includes:

Item	Address	Size	Comment
Compilation done	0	unsigned long	Points to the fixed address 264
Virtual machine code segment start address	4	unsigned long	Points to the fixed address 2*264. The length of this segment is less then 63k.
Virtual machine code segment length	8	unsigned long	
Text backup segment start address	12	unsigned long	
Text backup segment length	16	unsigned long	
Function symbol table start address	20	unsigned long	
Function symbol table length	24	unsigned long	
Variable symbol table start address	28	unsigned long	
Variable symbol table length	32	unsigned long	
Automatic function table start address	36	unsigned long	
Automatic function table length	40	unsigned long	
Last used address	44	unsigned long	The checksum for the user code partition is checked from Address 0 in the serial flash to that address. Must be in the range [2*264,100000]

All unused TOC items are set to zero.

16.5.2 The Compilation Done Flag

The Compilation Done Flag tells the Harmonica if a program is ready to run. This means that a program image has been successfully downloaded to the Harmonica and its checksum verified, and also that the symbol tables has been copied to the local data flash of the Harmonica.

If the compilation flag is zero, program is ready.
Other values indicate not ready program.
The command CP sets the compilation flag to –1.
The Compilation Flag is checked when:

- DL command – If the flag is zero, DL will not execute (CP must be issued before DL).
- XQ command – If the flag is nonzero, the command returns with PROGRAM_NOT_READY error
- CC inquiry – Returns 1 if the flag is zero, otherwise 0.

The compilation flag is set to zero at a CC=xxxx command, if xxxx matches the 32 bit program checksum, to the length stated at the TOC.

16.5.3 The Virtual Machine Code Segment

The virtual code segment holds consecutively the virtual machine code as compiled.

16.5.4 The Text Backup & Compiler data segment

The text backup segment holds the program text and the compiler data.
The first 80 characters are dedicated to the compiler version.
Characters [81.. End of segment] include the program text and line number information.
The compiler version is a zero terminated string, including:

- Compiler Version.sub.subsub, something like 15.2.1
- Op-codes version: same method as compiler version.
- Data of release, something like 17-Dec-2002
- Target: Release/Eng/Special
- Any general comment.

In each program line, the corresponding program counter value is the first word (unsigned short). If the first word is 0xFFFF, the text line contains no executable code. For example, the code
mo=1;
** This is a comment
mo=0;um=5;
will be written as
0x0246mo=1;<CR>0xFFFF**This is a comment<CR>024Cmo=0;um=5;<CR>....
supposing the VAC code for mo=1 is at PC=0246 and for mo=0 is at PC=0x24C, when PC is an address in the serial flash relatively to the start of the code segment of the user code partition.

16.5.5 The Function Symbol Table

The function symbol table states what functions and labels are accessible through the user program. The symbol table does not include any information on system functions. The first 34 bytes of the function symbol table is devoted to statistics:

- Number of functions in the table (signed short)

For each function, the following data 34 bytes record is set:

- Name (24 bytes long, every character takes two bytes - signed short, unused tail filled with zeros)
- Address in the Code Segment (2 bytes – unsigned short)
- Type (2 bytes, signed short – 0 for label, 1 for auto routine, 2 for user function)
- Number of input arguments (2 bytes, signed short, for functions only)
- Number of output arguments (2 bytes, signed short, for functions only)
- LabelProcRef (2 bytes, signed short). If this member is local label, this field contains index of the function inside which this label resides. For global label this field is equal to –1. For user function this field is not relevant and equal to -1.

The byte length of the function symbol table is 34 * (number of functions+1).

16.5.6 The Variable Symbol Table

The variable symbol table states what variables are accessible through the user program. Its first 34 bytes of the variables symbol table is devoted to statistics:

- Number of variables in the table (signed short)

For each variable, the following data 34 bytes record is set:

- Name (24 bytes long, every character takes two bytes - signed short, unused tail filled with zeros)
- Context: -1 for global variables, otherwise the index in the function symbol table defined above, to which this variable is local. (2 bytes, signed short)
- Number of array elements (1 for non-arrays) (2 bytes, signed short)
- Data address (2 bytes, signed short). This number specifies data segment relative address for global variables and base pointer relative address for local variables.
- Type (2 bytes, signed short – 0 for signed long integer, 1 for float)
- Spare (2 bytes, always zero)

The byte length of the variables symbol table is 34 * (number of variables+1).

16.5.7 The Automatic Routines Table

This table has 17 entries, one word long each.
The first entry is a bit field. It specifies for which priority levels a user defined a handler.
The next words specify the indices of auto routine handlers in the Function Symbol Table.

For example, if a user did not define a handler for priority 0, AUTO_ERR is specified for priority 1, and AUTO_DIN1 is specified for priority 2, then the bit field of value 0x2 specifies that the only implemented automatic routine is AUTO_ERR. The next words will be –1,12,-1,-1,... assuming 12 is an index of AUTO_ERR in the Function Symbol Table.

17 Appendix B: Harmonica Internals

This chapter summarises some of the Harmonica internals.
The chapter to give some deeper insights. It is not required for the standard Harmonica user.

17.1 Software Structure

The firmware is built from an initialisation block and from a run time block.
The run-time block is built by (event) interrupt handlers and by an idle loop.
Most of the motion control routines are performed by a periodic interrupt. The interrupt period is programmable.
The idle loop consumes all the remaining CPU time, polling if there is anything to do (like interpreting host commands or running a user program) and executing whatever found to do.

17.1.1 The Initialization block

The initialization block is called immediately after power-on.
The initialization is a linear set of routine, doing the following:

- Complete DSP register programming
- Initializing all the volatile data
- Attempt to read database from a serial flash, or reset data base to factory default if database reading failed
- Initialize the communication.
- Reset all CAN database – PDO mappings etc. and set CAN status to pre-operational.
- If there is an autoexec program defined, launch it (with thread 0).

17.1.2 The periodic Interrupt

This interrupt has the cycle time of 50-100usec programmable by the parameter TS. The increased TS are used for high inductance – high user program CPU demand combinations, or High loss motors, or large power applications.
This interrupt does, every cycle, as follows:

- Commutation
- Hazard handling
- Motor current control
- RS232 transmission and receiving
- Data recorder (may be activated only every fourth cycle)

After completing the “every cycle” jobs, the periodic interrupt forks to periodic jobs of lesser frequency. The following jobs are made once per four periods:

- Calculate speed
- Calculate motion reference
- Closed loop speed or position control
- Controller gains scheduling.
- Read digital input, with software de-bouncing
- Manage Homing post-processing (Catching the home event itself is done by hardware)
- Handle high priority CAN actions:
    - React to time stamps
    - React to SYNC message by logging time and sending synchronous PDO's
    - React to NMT
    - React to high-priority motion messages like PT and PVT.
    - Route lower priority CAN messages to handling by the idle loop.

The following jobs are made once per 32 sampling periods:

- Decide if to use the peak current limit or the continuous current limit.
- Match digital hall readout to encoder data, to detect loss of commutation
- Detect "Motor stuck" condition

17.1.3 The Idle Loop

The term "Idle loop" describes the repetitive task performed filling the entire time when the processor is free from interrupt handling.
Idle loop processing starts immediately after the initialization sequence is complete.
The Idle loop does:

- Run user programs
    - Run one line of user program for one virtual machine
    - Respond to events –interrupt routines, break points and single step debug mode.
- Interpret terminal commands
- Interpret CAN low priority messages:
    - CAN interpreter commands
    - SDO
    - Node guarding
- Calculate status and cleanup after motion fault exceptions.
- Route to the RS232 transmitter either data from the recorder or the output of the interpreter.

The sampling time parameter TS controls how much CPU power is reserved for idle-loop processing. Lower TS will deliver ultimate motion control performance, but will consume more CPU time for motion control algorithms. The CPU power remaining for idle loop processing decreases.

The following block diagram describes the idle loop.

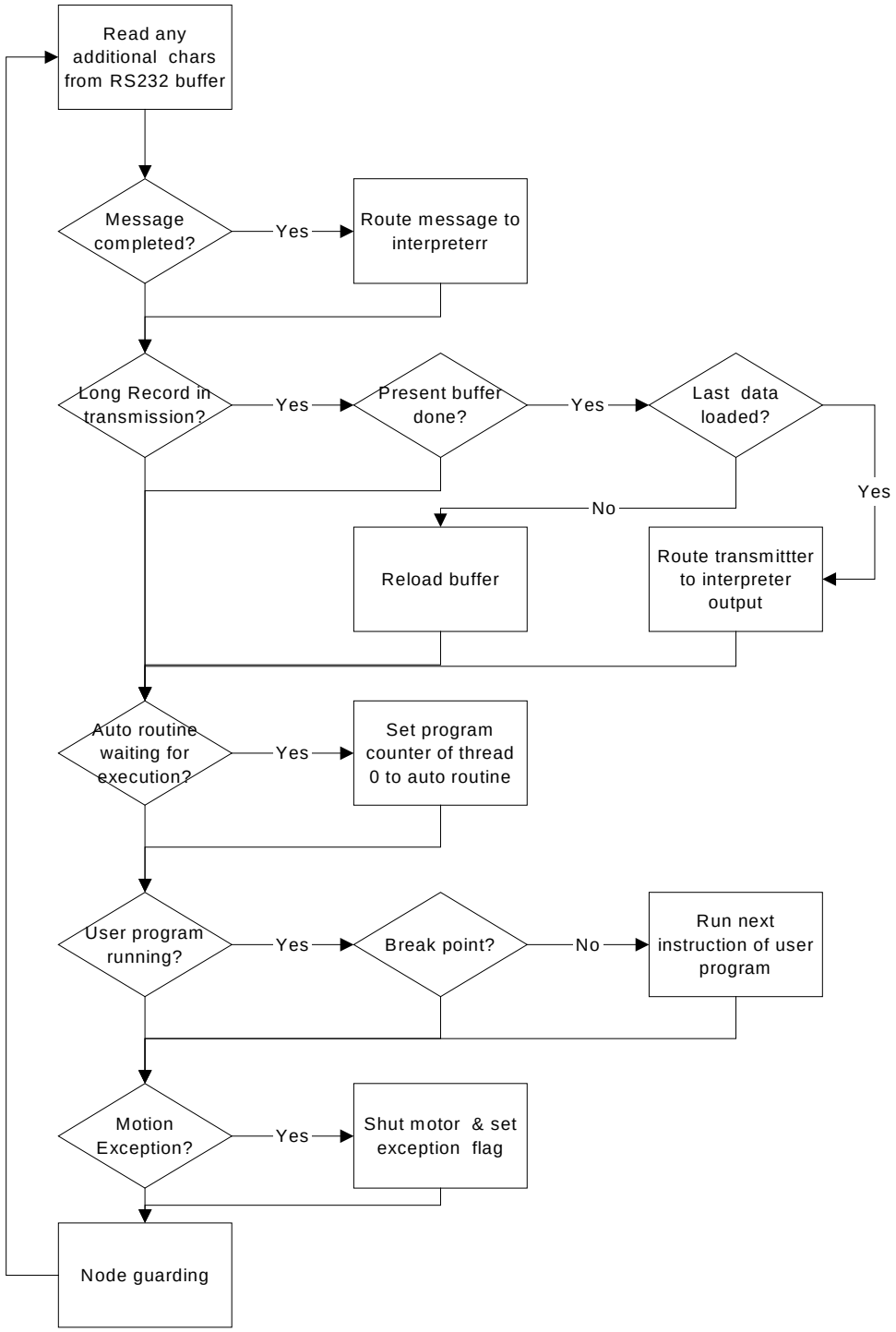


Figure 42: Idle loop

18 Appendix C: Converting Clarinet/Saxophone programs to the Harmonica language

18.1 The Converter

The user program syntax for the Saxophone or Clarinet is different from the Harmonica's syntax. The converter program provides an easy method to convert Saxophone/Clarinet style programs to the Harmonica syntax.
The converter program requires that the source Sax/Cla program is a simple text file.
If the Sax/Cla has been produced by the Composer program in .pgm (actually reach edit) format, be sure to convert it first to text.

18.2 The Converter Call

The converter is a console application. It gets their arguments from the command line. The arguments are the following:
1. The first argument is the name of the input file with a user program. The default is the file named "OldFormat.txt" from the current directory. If such file does not exist, the program exits doing nothing.
2. The second argument is the name of the output file that contains new text of the user program. The default is the file named "NewFormat.txt" in the current directory.
This file will be created (if it does not exist) or updated (if it exists) only if the conversion is successful.
3. The third argument is the name of the output file that contains errors of compilation. The default is the file named "Errs.txt" in the current directory.
This file will be created (if it does not exist) or updated (if it exists) only if the conversion or compilation is failed.
4. The fourth argument is SaveOldTextflag. A user may desire to save an old program text inside comment block for comparison with the text after conversion. The default is 0, i.e. an old text is not saved. If this command line argument is '1' – the format before conversion is saved inside C-style comment block (/* … */).

Example:
ConvSyn MySax.prg NewHar.txt 1
Converts the old program MySax.prg to the new program NewHar.txt, preserving the old Sax-style code, commented.

18.3 The Algorithm

The converter program does the following:

1. Load program text input file.
2. Try to compile old format text in order to verify that initial text is valid according to the Saxophone/Clarinet syntax. If the compilation failed - it writes error to the error output file and exits.
3. If SaveOldTextflag (the fourth command line argument) is '1', save it inside comment block as it is.
4. Convert to the new format.
5. Writes new program text to the output file.

18.4 The Conversion Process

Most the program flows instructions of the Sax/Cla need to be converted for an Harmonica application.
The table below presents a list of the changes with the brief explanation.

N	Description	Saxophone/Clarinet	Harmonica
1	AF command The keyword 'until' is used instead the	AF,cond	until cond

	command AF		
2	WT command The keyword ‘wait is used instead the command WT	WT=xxx	wait xxx
3	JP command without condition The keyword ‘goto’ is used for jump to the label	JP##LABEL	goto##LABEL
4	JP command with condition The ‘if’ statement is used for condition	JP##LABEL,cond	if cond goto##LABEL end
5	JS command without condition The subroutine call is used	JS##LABEL	LABEL
6	JS command with condition The ‘if’ statement is used for condition	JS##LABEL,cond	if cond LABEL end
7	JZ command without condition The keyword ‘reset’ is used for killing stack and jump to the subroutine	JZ##LABEL	reset LABEL
8	JZ command with condition The ‘if’ statement is used for condition	JZ##LABEL,cond	if cond reset LABEL end
9	EN command The keyword ‘exit’ is used instead the command EN	EN	exit
10	RT command The keyword ‘return’ is used instead the command RT	RT	return
11	The ‘is equal’ symbol inside the condition	=	==
12	The ‘not equal’ symbol inside the condition	<>	!=
13	Hexadecimal number representation	1234H	0x1234
14	Separation of expressions A single line may contain several expressions to evaluate separated with semicolon. In the Harmonica all expressions of the same line must be executed without return control to the Interpreter. It does not relevant for the Saxophone/Clarinet, so every expression will be set to the separate line.	UM=5;MO=1;BG	UM=5; MO=1; BG
15	IA[N], RA[N] commands The global variables IA and RA are defined in the beginning of every program. They are vectors of the length 100.		int IA[100]; float RA[100];

18.5 Examples

Here are some examples of the conversion:

Clarinet/Saxophone format	The Harmonica’s format
	int IA[100]; float RA[100];
##START	##START

PX=0;UM=2;AC=60000000;	PX=0; UM=2; AC=60000000;
IA[1]=2000;IA[2]=-2000 ;	IA[1]=2000; IA[2]=-2000 ;
MO=1;**Motor On	MO=1;**Motor On
BG;**Begin	BG;**Begin
##LOOP1	##LOOP1
JP##LOOP1,PX<IA[1] ;	if PX<IA[1] goto##LOOP1 end ;
##LOOP2	##LOOP2
JZ##LOOP2,PX>IA[2];	if PX>IA[2] reset LOOP2 end
#@AUTOEXEC	#@AUTOEXEC
WT=200	wait 200
##LOOP	##LOOP
JP##LOOP	goto##LOOP
RT	return
#@AUTO_I1	#@AUTO_I1
**THE BLOCKING MODE	**THE BLOCKING MODE
MO=0;UM=2;RM=0;MO=1;JV=0;BG	MO=0; UM=2; RM=0; MO=1; JV=0; BG
AF,IB[1]=0	until IB==0
MO=0;UM=1;RM=1;MO=1	MO=0; UM=1; RM=1; MO=1
SM=2AH	SM=0x2A
RT	return
EN	exit