Contents

2.9 ASCII Interface.................................... 60
2.9.1 Overview.................................... 61
2.9.2 Messages.................................... 62
2.9.3 GPIB Codes and Formats.................................... 63
2.10 New Trace and Triggering.................................... 68
2.11 Trigger View.................................... 69
2.12 Auto-range.................................... 70
2.14 Waveform Transfer.................................... 71

3.0 Common Issues.................................... 75
3.1 Startup Sequence.................................... 75
3.2 Error Handling.................................... 77
3.3 Other Common Commands.................................... 81

4.0 Self-test.................................... 85
4.0.1 Plug-In Communication Initialization.................................... 85
4.1 Startup Self-tests.................................... 86
4.1.1 Messages.................................... 87
4.1.2 Signal Path Tests.................................... 89
4.1.3 Auxiliary Trigger Line Test.................................... 89
4.1.4 Exiting Test Mode.................................... 90
4.2 Diagnostic Mode.................................... 91
4.2.1 Messages.................................... 91
4.2.2 Display and Trigger Signal Paths.................................... 95
4.2.3 Serial Data Interface Test.................................... 95
4.2.4 Auxiliary Trigger Line Diagnostics.................................... 96
4.3 Summary.................................... 96

5.0 Calibration.................................... 99
5.1 Configuration Determination.................................... 99
5.1.1 System Configuration.................................... 99
5.1.2 Probe Configuration.................................... 100
5.2 Calibration Messages.................................... 101
5.3 Calibration Modes.................................... 105
5.3.1 Entering Calibration Mode.................................... 105
5.3.2 Initial Calibration Mode.................................... 106
5.3.3 Hold Calibration Mode.................................... 106
5.3.4 Display Path Calibration - Without CVR.................................... 107
5.3.5 Display Path Calibration - With CVR.................................... 107
5.3.6 Display/Trigger Imbalance Determination Mode.................................... 108
5.3.7 New Configuration Calibration Mode.................................... 108
5.3.8 Other Calibration Mode.................................... 109
5.3.9 Final Calibration Mode.................................... 109
5.3.10 CVR Pass-Through Mode.................................... 110
5.4 Calibration Procedures.................................... 112
5.4.1 Power-Up Calibration.................................... 112
5.4.2 Warm Calibration.................................... 114
5.4.3 Full Calibration.................................... 116
5.4.4 Auxiliary Trigger Calibration.................................... 119
5.4.5 Probe Calibration.................................... 120
5.4.6 Trigger View Calibration.................................... 121
5.4.7 Special Plug-In Calibration.................................... 122
5.5 Calibrated Voltage Reference.................................... 124
5.6 Voltage Measurement.................................... 125

Change Log for Version 2.2

12. The **delay** and *rel_delay* parameters of the **disp_attr status** message should be shown as optional on page 235. The *gain_res* parameter should also be shown as type us not float.

13. On page 280, the response paragraph should show use of the **formatted** token instead of the **ready** token.

14. The value for the **init** token on page 335 should be 01_{16} instead of 04_{16}.

15. A plug-in may only send the **normal** *mode* token when notifying the mainframe of its cal status using the **cal acc_status** message. Affects page 168.

16. The *const_num* parameter of the **cal const_set** message should be shown as type ui on page 83. The *number* parameter for the **cal const_num** message should also be shown as type ui (same page).

17. The transaction list for the **diag status** message should contain the transaction **diag enter : diag status ready.** Affects page 226.

18. The transaction paragraph for the **external set_status** message should show **external set_query : external set_status** instead of **: external set_data**. Affects page 257.

19. The *data* parameter for the **function select** message should not be shown as optional on page 261.

20. The response: paragraph for the **menu def_generic** message should show *none* on page 296.

21. The transaction paragraph for the **probe status** message should include the transaction **probe query : probe status.** Affects page 347.

Table of Contents

Introduction.. 1

1.0 Generic Amplifier Plugins.. 5
1.1 Model.. 5
1.2 Handshaking.. 6
1.3 Gain... 7
1.4 Offset... 9
1.5 Coupling... 14
1.6 Input Impedance.. 15
1.7 Bandwidth Limit.. 16
1.8 Units.. 18
1.9 Knobs and Knob Displays.. 18
1.10 Generic Menus... 21
1.10.1 Coupling Menus.. 22
1.10.2 Input Impedance Menu.. 23
1.10.3 Bandwidth Limit Menus... 24
1.11 ID Function... 25
1.12 Display Updating.. 26
1.13 Stored Settings... 26
1.14 Interactions With Smart Plugins................................... 27
1.15 Error Handling.. 27
1.16 New Trace... 28
1.17 Auxiliary Triggers.. 31
1.18 ASCII Interface... 31
1.18.1 Input Commands.. 31
1.18.2 Waveform Commands... 33
1.18.3 Triggering Commands... 33
1.19 Generic Plug-In SRQ's... 33
1.20 Probes.. 34.1
1.20.1 Effects of Adding a Probe....................................... 34.1
1.20.2 Effects of Removing a Probe..................................... 34.3

2.0 Smart Plugins.. 35
2.1 General.. 35
2.2 Menu Interface... 35
2.2.1 Menu Messages.. 35
2.2.2 Operation.. 38
2.2.3 Menu Definition.. 40
2.2.4 Menu Examples.. 42
2.2.5 Status Message... 45
2.3 Knobs and Keypad... 46
2.3.1 Knob Messages.. 47
2.3.2 Knob Examples.. 49
2.4 Cursors 51
2.5 Front Panel Settings... 53
2.6 Interactions With Other Plugins.................................... 54
2.6.1 Generic Plugins.. 55
2.6.2 Smart Plugins.. 55
2.6.3 System... 56
2.7 Updates.. 59
2.8 Error Handling... 60

Contents

6.0 Interface Protocols.. 127
6.1 Application Protocols.................................... 127
6.1.1 Message Formats.. 128
6.1.2 Command/Status Protocol.................................. 128
6.1.3 Plug-In SRQ's.. 129
6.1.4 Long Messages.. 131
6.2 Transport Protocol....................................... 134
6.2.1 Scope.. 134
6.2.2 Features... 134
6.2.3 Format... 136
6.2.4 Operation.. 137
6.2.5 Errors... 139
6.2.6.1 Mainframe Transmit..................................... 141
6.2.6.2 Mainframe Receive...................................... 142
6.2.6.3 Plug-In Transmit....................................... 143
6.2.6.4 Plug-In Receive.. 144
6.3 Definitions.. 145

7.0 Message Definitions.. 149

Appendix A ASCII Interface.. A1
A.1 Introduction..
A.2 Translators...
A.2.1 Type E..
A.2.2 Type R..
A.3 Tables..
A.3.1 Type E..
A.3.2 Type R..
A.4 Scanners..
A.4.1 Type E..
A.4.2 Type R..
A.5 Parsers...
A.5.1 Type E..
A.5.2 Type R..

Appendix B...

Appendix C...

11000 Series Plug-In to Mainframe Interface Manual, Software

Introduction

The purpose of this document is to define the complete software interface between 11000 series mainframes and plug-ins. The first five sections show how the messages defined for plug-ins and mainframes are used to communicate status and to provide control and is intended to be a user's guide. The sixth section defines the transport system that is used to transfer messages between mainframes and plug-ins. The seventh section specifies each command in detail and is intended to be an implementer's guide.

A separate document titled *New Probe Interface Manual* defines the interface between new probes and new plug-ins. A document titled *11000 Series Accuracy System* specifies the operation of the calibration system. The *11000 Series Plug-In to Main Frame Interface Manual* specifies the hardware interface between plug-ins and mainframes. The *Command Reference Specifications 11K Series Family of Products* document defines the external interface for 11000 Series mainframes.

Message Formats and Conventions

Each message is uniquely identified by two tokens always shown in **boldface**. These are called the message tokens. The first is identified as the primary message token. The second as the secondary message token. All primary message tokens have distinct values. All secondary message tokens associated with a particular primary message token have distinct values. Secondary tokens associated with different primary tokens may have the same values. The token names used in this document symbolically represent the value of the token. Each name (whether used for a primary or secondary message token or for a parameter token) always represents the same value.

Each pair of message tokens identifies the type of message and the arguments that may follow. All message tokens defined for generic plug-ins are understood by all generic plug-ins. Smart plug-ins and mainframes will understand all applicable message tokens. Appropriate handling of messages that relate to functions that are not available in a particular plug-in or mainframe is required. Plug-Ins and mainframes should not issue an error for any applicable message defined in this document.

Primary message tokens have a range of values from 00_{16} to FF_{16}. Values in the range $E0_{16}$ to FE_{16} are reserved for plug-ins to define plug-in specific, non-supported messages for whatever use the plug-in may define. The mainframe is not required to support messages defined in this range. These messages are not included in the scope of this interface specification. The value FF_{16} is reserved.

Message tokens may be followed by a variable number of parameters. The number and types of parameters are defined in this document. The values of the parameters (except for token parameters) depend on instrument operation. All token parameter values are defined in this document. These predefined tokens (both message tokens and parameter tokens) are shown in **boldface**.

A message may be an action request or a status report. An action request, or command, always requires a response (usually a related status message). This is termed a command transaction. Each message has a defined set of command transactions in which it is involved. See section **7.0 Messages** for these transactions for each command. Other command messages will not be sent while waiting for a response from a previous command message. A response to a command is not returned until the action specified by the command or other appropriate action has been completed.

In general, the mainframe has transmit priority. This means that the mainframe may at any time send a message to the plug-in. The plug-in must send a plug-in SRQ to the mainframe if it has a message that is not in response to a mainframe message. The mainframe will send a message to the plug-in in response to the SRQ to request the plug-in's message. See section **6.1.3 Plug-In SRQ's** for details. Exceptions to this priority are allowed during diagnostics and calibration. See sections **4.0 Self Test** and **5.0 Calibration**.

Notational Conventions

Messages in the first five sections are defined in individual paragraphs. The first line of the paragraph shows the general form of the message. The following lines define the type of each parameter of the message and give a short description of the meaning of that parameter. The first element of each message is the message token pair. They are shown in **boldface** in the definition paragraph and everywhere else they are referenced. Other defined tokens are also shown in **boldface**. Message parameters are shown in *italics*. Optional parameters for messages are delimited by square brackets ([]). Parameters that may be repeated are followed by three dots (...). Curly braces ({}) are used for grouping and are generally used with (...) for repeated parameters.

Parameter Types

Parameter types and notation are defined as follows:

notation	meaning
short	short integer - this is a single byte value. 7 bit magnitude with a sign bit. Range is +127 to -128
us	unsigned short - this is a single byte value with no sign bit. 8 bit magnitude. Range is 0 to 255.
int	integer - this is a double byte value. 15 bit magnitude with a sign bit. Range is +32767 to -32768.

ui	unsigned integer - this is a double byte value with no sign. 16 bit magnitude. Range is 0 to 65535.
long	long integer - this is a four byte value. 31 bit magnitude with a sign bit. Range is $+2^{31}-1$ to -2^{31}.
ul	unsigned long - this is a four byte value with no sign bit. 32 bit magnitude. Range is 0 to $2^{32}-1$.
float	floating point - this is a four byte signed floating point value in IEEE format. 23 bit magnitude, 8 bit exponent, 1 sign bit.
double	floating point - this is an eight byte signed floating point value in IEEE format. 52 bit magnitude, 11 bit exponent, 1 sign bit.
string	string - this is a string of ASCII characters terminated by the NUL character (00_{16}) and includes the set of escape sequences defined in Appendix B of the Command Reference Specifications document. Note that where strings are shown limited to a specific number of characters, the limit applies to displayed characters only. Characters defined as a two character escape sequence occupy only one display location. The limit also does not include the NUL terminator character.
char	character - this is a single byte unsigned value that is typically (but not always) a printable ASCII value.
pb	packed binary - this is a single byte split into two nibbles of four bits each. Each nibble represents a value from 0 to 16.
token	when shown as a parameter in a message, this type is a specified list of byte values that are defined in this specification. The token name is a symbolic substitute for the absolute token value.
byte	this type is used to define absolute hex values for tokens.
special	this type, when used in a message, is not a general type but indicates that the parameter is defined in the text and does not have one of the meanings defined above.

All references to channel numbers as unsigned integers in all sections will use the value 1 to indicate channel 1, 2 for channel 2 etc. Thus, a four channel plug-in will accept values 1 - 4 for its channel selection parameter for all of its commands.

All parameter types of more than one byte will be transmitted least significant byte first.

The NaN floating point value used for terminating floating point lists is the value 0xffc00000. Plus infinity is defined as 0x7f800000. Minus infinity is defined as 0xff800000.

Parameters of special types will be defined as necessary for messages that need them.

1.0 Generic Amplifier Plug-Ins

All aspects of the interface defined in this section relating to the generic amplifier plug-in are activated when a plug-in identifies itself as **a generic** or **both** type plug-in in the **plugin_config status** message. See section **3.1 Startup Sequence** for the specifics of plug-in identification. A plug-in that uses the generic amplifier plug-in interface must provide controls for each function defined below.

This specification defines the interface to a generic amplifier plug-in. It does not define how a mainframe must use the interface but does include suggestions on how presently proposed mainframes might incorporate these functions into their GPIB and human interfaces.

1.1 Model

Generic amplifier plug-ins provide two multiplexed analog outputs for all inputs combined and one set of non-multiplexed outputs for each input. One of the multiplexed outputs is for display and one is for triggering. These are called the display and trigger outputs. The non-multiplexed outputs are for triggering plug-ins that operate in the right compartment. These are called the auxiliary trigger outputs. There may be no more than 4 inputs (channels) per plug-in compartment. Each channel has the following five functions each of which affects all three channel outputs (display, trigger and auxiliary trigger):

1. Gain - controls the gain of the signal path.

2. Offset - applies an offset signal to the input.

3. Input Coupling - selects the input coupling type.

4. Input Impedance - selects the input impedance.

5. Bandwidth Limit - controls the bandwidth of the signal path.

Menu entries for input coupling, input impedance and bandwidth limit specify the selections available for those functions. Status for each function will be reported when that function is used.

Display and trigger output multiplexing may be specified individually. See section **1.16 New Trace** for information on how the mainframe controls these plug-in functions. The non-multiplexed auxiliary trigger outputs provide one output for each input channel.

This generic amplifier model is for a single wide plug-in. Generic amplifier plug-ins that occupy more than one compartment will be treated as two separate one-wide plug-ins. They will communicate using two SDI's. A two-wide plug-in that only has one SDI communication port may only define channels that correspond to that port.

Generic plug-ins will handle all probe functions. See the *New Probe Interface Manual* for more information. As part of this interface, the plug-in will also provide the units of conversion of the probe for mainframe display. See section **1.8 Units**. Probe calibration will require user interaction with mainframe menus. See the document *11000 Series Accuracy System*.

Generic plug-ins will do their own calibration when requested by the mainframe. All functions provide calibrated conversion to a specified accuracy. Calibration will be done from the input connector to the mainframe internal pickoff point. Amplifier plug-ins will be able to switch their inputs to a calibrated voltage source supplied by the mainframe. Calibration is also provided to the probe tip using mainframe menus and requires user interaction.

Generic plug-ins will be instructed to perform self-tests to determine their availability for proper operation. They will notify the mainframe of failure in either of the calibration or diagnostic functions. See sections **4.0 Self-test** and **5.0 Calibration** for more information.

Some generic plug-ins are identified as differential plug-ins. These plug-ins have two input connectors for each channel. The output of a channel is the difference of the voltages at the two input connectors. The connectors are identified as the plus input and the minus input. These generic plug-ins have two sets of coupling controls one for each input. These plug-ins will send the **diff_offset** token in the plugin_config status message. See section **3.1 Start Up**. These plug-ins are limited to two channels. Mixing of differential and single-ended channels in a single plug-in is not allowed.

Differential plug-ins have additional offset capabilities. There are four offset values that are maintained internally to the plug-in: amplifier offset, comparison voltage, plus probe offset and minus probe offset. These values control different aspects of the signal path as defined in section **1.4 Offset**.

1.2 Handshaking

When a command is sent to a generic plug-in, the plug-in will attempt to execute the command. If it is successfully executed, the plug-in will return the new status of all affected functions. The new status of related functions will be reported prior to the status of the selected function so the mainframe will know when all status changes have been reported. Error messages will also precede the status of the selected function. See section **1.15 Error Handling** for more information. In all cases, the plug-in will return a message that indicates the completion of a command. The status message returned is identical to the one received in response to a generic plug-in function query command.

Generic plug-ins will only accept one command at a time for execution. Until the current command has been completed, the plug-in will not accept any further commands or any other messages.

Generic plug-ins will not accept messages longer than 32 bytes. See **Protocols** section **6.2.3 Format**.

1.3 Gain

The generic amplifier gain function is semantically understood by the mainframe to control the size of the input signal. This function will be connected to the mainframe's human interface size control. There are three methods of control that are also semantically understood by the mainframe and relate to specific mainframe functions: 1) absolute setting, 2) coarse change and 3) fine change. The absolute setting control will be used when the mainframe's keypad is assigned to the size function. The coarse and fine change controls will be used when the respective mainframe function is assigned to one of the control knobs. See section **1.17 ASCII Interface** for information on control of this function from the external interface.

If a request is made for a gain setting that is not compatible with the present offset setting, the offset will be changed to a range that is compatible with the requested gain setting. The previously requested offset value will be remembered according to the rules defined in section **1.4 Offset**.

The absolute setting control message causes the amplifier's gain for the specified channel to be set explicitly to the value of the associated parameter. The coarse and fine change control messages cause the gain to be changed by the specified number of steps from its present value. The coarse change steps are logrithmic in a 1-2-5 sequence. The fine change steps are a fixed percentage of the next more sensitive coarse setting. The fine change control will also change the value of the coarse setting if necessary to achieve the requested number of steps of change. When the coarse change control is used, the fine control will be set to zero (x1).

The **gain set** message with three different mode tokens is used to control these functions: **gain set abs**, **gain set coarse** and **gain set fine**. Each message has two parameters associated with it. One parameter for the **gain set abs** message is a floating point value that specifies the new absolute value of the gain setting. One parameter for the **gain set coarse** and **gain set fine** messages is a signed integer that specifies the number of coarse or fine steps to change the gain. Positive integers increase the volts per division. Negative integers decrease the volts per division. The other parameter for all three messages specifies which channel's gain the command is intended to change. These commands have the following format:

gain set abs channel value
gain set: message tokens
abs: token - selects absolute setting mode
channel: us - selects the channel
value: float - specifies the gain value

gain set coarse channel value
gain set: message tokens
coarse: token - selects coarse change mode
channel: us - selects the channel
value: short - specifies the number of coarse steps to change the gain

gain set fine channel value
gain set: message tokens
fine: token - selects fine change mode
channel: us - selects the channel
value: short - specifies the number of fine steps to change the gain

When any of the gain control commands are executed, the plug-in will return to the mainframe the new status of the gain setting in a floating point format along with the channel number. If the offset value of the specified channel is changed as a result of a change in gain, the plug-in will report the new offset status before reporting the new gain status. If the plug-in is a differential plug-in, the offset status for one of the internal offset values will also be reported if it has changed. The gain status message has the following format:

gain status channel value
gain status: message tokens
channel: us - specifies the channel
value: float - specifies the gain value

This status message will also be sent in response to an **SRQ query** message any time a probe change affects the gain setting. The mainframe must be able to take appropriate action when this message is received. The absolute minimum and maximum gain settings are reported by the **disp_attr status** message.

If the size function is assigned to a trace that is composed of more than one channel, it is the mainframe's responsibility to determine whether to send size control commands to all channels in the trace specification or to take other appropriate action.

gain query channel
gain query: message tokens
channel: us - specifies the channel

The mainframe may send this message to a generic plug-in to request its present gain setting without specifying a new setting. The plug-in will respond with the **gain status** message.

Error Handling

Generic plug-ins do no checking on channel numbers. If a channel number is received as part of a **gain set** command that a plug-in does not have, the plug-in will take unspecified action and return a **gain status** message indicating the action it took.

When the gain requested is outside the capability of the plug-in, the plug-in will take the following actions based on the command type. The nearest available setting is defined as either the maximum or the minimum gain the plug-in can produce.

For the **gain set abs** command, the plug-in will set to the maximum or minumum value and return an **error generic** message with an **exec_warning** *status* token and the *code* parameter set to 550 and the *index* parameter set to 1. It will return the new gain setting using the **gain status** message.

For the **gain set coarse** command, the plug-in will set its gain to the nearest available coarse setting, set the fine control to 0 (x1) and return the **gain status** message with the actual gain value. For the **gain set fine** command, the plug-in will set its gain to the nearest available setting and return the **gain status** message with the actual gain value. There will be no error reported in either of these cases.

If a negative value of gain is requested, the plug-in will take no action and return an **error generic** message with an **exec_error** *status* token and the *code* parameter set to 205 and the *index* parameter set to 1. The present gain setting will be reported using the **gain status** message.

Summary

Mainframe: When the keypad is assigned to the size function, the mainframe sends the **gain set abs** message with the channel specification and the value entered by the keypad to the plug-in. When the coarse size control is assigned to a control knob, the mainframe sends the **gain set coarse** message and the number of knob increments (with direction as sign) and the channel number to the plug-in. When the fine size control is assigned to a control knob, the mainframe sends the **gain set fine** message and the number of knob increments (with direction as sign) and the channel number to the plug-in.

Plug-In: When the plug-in receives the **gain set abs** message, it sets the gain of the specified channel to the specified value. When the plug-in receives the **gain set coarse** message, it changes the gain control of the specified channel by the specified number of coarse steps in the specified direction and sets the fine control to 0. When the plug-in receives the **gain set fine** message, it changes the gain control of the specified channel by the specified number of fine steps in the specified direction.

1.4 Offset

The generic amplifier offset function is semantically understood by the mainframe to control the displacement of the input signal. This function will be connected to the mainframe's human interface offset or position control. There are three methods of control that are also semantically understood by the mainframe and relate to specific mainframe functions: 1) absolute setting, 2) coarse change and 3) fine

offset status channel value
offset status: message tokens
channel: us - specifies the channel
value: float - specifies the offset value

This status message will also be sent in response to an SRQ query message any time a probe change affects the offset setting. The mainframe must be able to take appropriate action when this message is received. The absolute minimum and maximum offset settings are reported by the **disp_attr status** message.

If the offset function is assigned to a trace that is composed of more than one channel, it is the mainframe's responsibility to determine whether to send offset control commands to all channels in the trace specification or to take other appropriate action.

offset query channel
offset query: message tokens
channel: us - specifies the channel

The mainframe may send this message to a generic plug-in to request its present offset setting without specifying a new setting. The plug-in will respond with the **offset status** message.

Error Handling

Generic plug-ins do no checking on channel numbers. If a channel number is received as part of an **offset set** command that a plug-in does not have, the plug-in will take unspecified action and return an **offset status** message indicating the action it took.

When the offset requested is outside the capability of the plug-in, the plug-in will take the following actions based on the command type. The nearest available setting is defined as either the maximum or the minimum offset the plug-in can produce.

For the **offset set abs** command, the plug-in will set to the maximum or minimum value and return an **error generic** message with an **exec_warning** *status* token and the *code* parameter set to 550 and the *index* parameter set to 2. It will return the new actual offset setting using the **offset status** message.

For the **offset set coarse** and **offset set fine** commands, the plug-in will set its offset to the nearest available setting and return the **offset status** message with the actual offset value. There will be no error reported in either case.

Differential plug-ins maintain four offset values internally that may be controlled by the mainframe and accessed via the external bus. These values are identified as: amplifier offset, comparison voltage (VC), plus probe offset and minus probe offset.

The amplifier offset is identified as *amp_offset* and is equivalent to the offset function of a non-differential channel. The change. The absolute setting control will be used when the mainframe's keypad is assigned to the offset function. The coarse and fine change controls will be used when the respective mainframe function is assigned to one of the control knobs. See section **1.17 ASCII Interface** for information on control of this function from the external interface.

The absolute setting control message causes the amplifier's offset for the specified channel to be set explicitly to the value of the associated parameter. The coarse and fine change control messages cause the offset to be changed by a specified number of divisions from its present value. The coarse control steps are .25 divisions. The fine control steps are .025 divisions.

The **offset set** message with three different mode tokens is used to control these functions: **offset set abs**, **offset set coarse** and **offset set fine**. Each message has two parameters associated with it. One parameter for the **offset set abs** message is a floating point value that specifies the new absolute value of the offset setting. One parameter for the **offset set coarse** and **offset set fine** messages is a signed integer that specifies the number of coarse or fine steps to change the offset. Positive integers increase the offset. Negative integers decrease the offset. The other parameter for all three messages specifies which channel's offset the command is intended to change. These commands have the following format:

offset set abs channel value
offset set: message tokens
abs: token - selects absolute setting mode
channel: us - selects the channel
value: float - specifies the offset value

offset set coarse channel value
offset set: message tokens
coarse: token - selects coarse change mode
channel: us - selects the channel
value: short - specifies the number of coarse steps to change the offset

offset set fine channel value
offset set: message tokens
fine: token - selects fine change mode
channel: us - selects the channel
value: short - specifies the number of fine steps to change the offset

When any of the offset control commands are executed, the plug-in will return to the mainframe the new status of the offset setting in a floating point format along with the channel number. The offset status message has the following format:

vc_offset value is the comparison voltage and allows the user to subtract a specified voltage from the input signal. The *plus_offset* and *minus_offset* values provide the offset voltage to the probes on the plus and minus inputs, respectively.

When a differential plug-in is installed, the mainframe's offset function (via the **offset set** message) will control one of the above functions based on the selected input coupling and whether or not an offset probe is connected to each input. Which function is controlled and how each interacts with the coupling settings and offset probes is defined in the plug-in's EIS. When sent an **offset set** message, a differential plug-in will always send an **offset status** message that indicates the ground reference value. This value will be some combination of the four offset values in the plug-in as defined by the plug-in's EIS. The plug-in will also report the value of the differential offset function that was affected by the **offset set** message using the messages defined below.

The following messages are defined to support the operation and status reporting of the differential offset functions. These messages are only available for plug-ins that report the **diff_offset** token in the **plugin_config status** message.

diff_offset set control channel value
diff_offset set: message tokens
control: token - selects which control
channel: us - selects the channel
value: float - offset value for the probe

This message is sent by the mainframe to explicitly control the differential offset function of a differential plug-in. This command is an absolute setting command only. Incremental setting of the differential offset functions is only provided indirectly through the **offset set coarse** and **offset set fine** commands. The *control* parameter selects which offset function is to be changed. The **plus** *control* token selects the offset of the probe connected to the positive input. The **minus** *control* token selects the offset of the probe connected to the negative input. The **vc** *control* token selects the comparison voltage control. The **amp** *control* token selects the amplifier offset. The *channel* parameter specifies the channel for which the offset function is to be changed. The *value* parameter specifies the absolute offset value for the control.

diff_offset status control channel value
diff_offset status: message tokens
control: token - selects the control
channel: us - selects the channel
value: float - offset value of probe

This message is sent by the plug-in to report differential offset status mode. The *control* parameter specifies which function status is to be reported. The *control* tokens have the same meaning as for the **diff_offset set** message. The *channel* parameter specifies for which channel the offset function is being reported. The *value* parameter reports the value of the selected control. The plug-in will send this message in response to a **diff_offset set**, a **diff_offset query**, an **offset set** or an **offset query** message from the mainframe.

diff_offset query control channel
diff_offset query: message tokens
control: token - selects the control
channel: us - selects the channel

This message is sent by the mainframe to request the present status of the differential offset function. The *control* parameter specifies which offset function is to be reported. The *control* tokens have the same meaning as for the **diff_offset set** message. The *channel* parameter specifies the channel. The plug-in will send the **diff_offset status** message in response.

Error Handling

For the **diff_offset set** command, the plug-in will set to the maximum or minimum value for the specified offset function and return an **error generic** message with an **exec_error** *status* token and the *code* parameter set to 550 and the *index* parameter set to 2. It will return the new differential offset and actual offset settings using the **diff_offset status** and offset status messages.

Summary

Mainframe: When the keypad is assigned to the offset or position function, the mainframe sends the **offset set abs** message with the channel specification and the value entered by the keypad to the plug-in. When the coarse offset control is assigned to a control knob, the mainframe sends the **offset set coarse** message and the number of knob increments (with direction as sign) and the channel number to the plug-in. When the fine offset control is assigned to a control knob, the mainframe sends the **offset set fine** message and the number of knob increments (with direction as sign) and the channel number to the plug-in. When the mainframe receives an external bus command to control offsets in a differential plug-in, it will send the **diff_offset set** message.

Plug-In: When the plug-in receives the **offset set abs** message, it sets the offset of the specified channel to the specified value. When the plug-in receives the **offset set coarse** message, it changes the coarse offset control of the specified channel by the specified number of steps in the specified direction. When the plug-in receives the **offset set fine** message, it changes the fine offset control of the specified channel by the specified number of steps in the specified direction. When the plug-in receives the **diff_offset set** message it will set the specified offset to the specified value and report the new status using the **diff_offset status** message.

1.5 Coupling

This generic function is controlled by mainframe menus that provide selection capability. See section **1.10 Generic Menus** for more information. Also see section **1.17 ASCII Interface** for details on the control of this function from the external interface.

The **coupling set** message with two mode tokens is used to control the input coupling: **coupling set plus** and **coupling set minus**. These messages have two parameters associated with them. The first specifies the channel for which the change is intended. The second is a valid coupling token (defined in section **1.10.1 Coupling Menus**). These commands have the following format:

coupling set plus channel coupl
coupling set: message tokens
plus: token - selects the plus coupling function
channel: us - selects the channel
coupl: coupling token - specifies the coupling type

coupling set minus channel coupl
coupling set: message tokens
minus: token - selects the minus coupling function
channel: us - selects the channel
coupl: coupling token - specifies the coupling type

The **coupling set plus** command controls the coupling of either a single ended channel or the plus input of a differential channel. The **coupling set minus** command controls the minus input of a differential channel and is not legal for single ended channels.

When either of these commands is executed, the plug-in will return the status of the coupling control for the selected channel using the following messages:

coupling status plus channel coupl
coupling status: message tokens
plus: token - selects plus coupling mode
channel: us - specifies the channel
coupl: coupling token - specifies the coupling type

coupling status minus channel coupl
coupling status: message tokens
minus: token - selects minus coupling mode
channel: us - specifies the channel
coupl: coupling token - specifies the coupling type

These status messages will be sent in response to an SRQ query message to report coupling changes not requested by the mainframe. (eg. adding a probe may cause the input coupling to change.) The mainframe must be able to take appropriate action when these messages are received.

coupling query plus channel
coupling query: message tokens
plus: token - selects plus coupling mode
channel: us - specifies the channel

coupling query minus channel
coupling query: message tokens
minus: token - selects minus coupling mode
channel: us - specifies the channel

The mainframe may send either of these messages to a generic plug-in to request its present coupling setting without specifying a new setting. The plug-in will respond with the appropriate **coupling status** message.

If the plug-in does not send the **minus_coupl** token in the **plugin_config status** message, the **coupling set minus** message is not legal. See section **3.1 Startup Sequence** for more information.

For differential plugins, changing the coupling may affect the differential offset function. If this is the case, the plug-in will send **diff_offset status** and **offset status** messages to report the changes in offset prior to sending the **coupling status** message.

Error handling: If a coupling token is received that is a valid coupling token but is a coupling which the plug-in does not support the plug-in will take no action and return an **error generic** message with a **exec_error** *status* token and the *code* parameter set to 284. If the coupling token is not a valid coupling token, the plug-in will take no action and return an **error generic** message with a **command_error** *status* token and the *code* parameter set to 157. It will report the present coupling setting using either the **coupling status plus** or **coupling status minus** message. Generic plug-ins do no checking on channel numbers. If a channel number is received as part of a **coupling set** command that a plug-in does not have, the plug-in will take unspecified action and return a **coupling status** message indicating the action it took.

1.6 Input Impedance

This generic function is controlled by mainframe menus that provide selection capability. See section **1.10 Generic Menus** for more information. Also see section **1.17 ASCII Interface** for details on the control of this function from the external interface.

The **impedance set** message is used to control the input impedance. This message has two parameters associated with it. The first specifies the channel for which the change is intended. The second is a floating point value that specifies the requested input impedance. This command has the following format:

impedance set channel value
impedance set: message tokens
channel: us - selects the channel
value: float - specifies the input impedance value

If the requested value does not exactly match a legal plug-in impedance, the plug-in will set the input impedance to the nearest legal value.

When this command is executed, the plug-in will return the status of the impedance control for the selected channel using the **impedance status** message in the following format:

impedance status channel value
impedance status: message tokens
channel: us - specifies the channel
value: float - specifies the input impedance value

This status message will be sent in response to an **SRQ query** message to report input impedance changes not requested by the mainframe. (eg. adding a probe may cause the input impedance to change.) The mainframe must be able to take appropriate action when this message is received.

For plug-ins that have an electrometer mode, the plug-in will use the IEEE value for +∞ to report infinite input impedance. The mainframe will report this value as 9E19 over the external bus but will display the ∞ symbol on the screen. The plug-in will set an internal threshold at an appropriate value less than 9E19 such that any impedance value larger than the threshold value will cause the plug-in to enter electrometer mode.

Some smart probes will be able to notify the plug-in of their input impedance. In this case, the plug-in will report the input impedance of the connected probe, not the input impedance of the plug-in.

impedance query channel
impedance query: message tokens
channel: us - specifies the channel

The mainframe may send this message to a generic plug-in to request its present impedance setting without specifying a new setting. The plug-in will respond with the **impedance status** message.

Error handling: Generic plug-ins do no checking on channel numbers. If a channel number is received as part of an **impedance set** command that a plug-in does not have, the plug-in will take unspecified action and return an **impedance status** message indicating the action it took. If a low impedance input resistor is overheated, the plug-in will not change the input impedance to that setting if requested but will report an error using the **error generic** message with the **exec_error** ≤ΓβΓ]≤ token and the πΠΣσ set to 280.

1.7 Bandwidth Limit

This generic function is controlled by mainframe menus that provide selection capability. See section **1.10 Generic Menus** for more information. Also see section **1.17 ASCII Interface** for details on the control of this function from the external interface.

The **bandwidth set** message with two mode tokens is used to control the signal path bandwidth: **bandwidth set upper** and **bandwidth set lower**. These messages have two parameters associated with them. The first specifies the channel for which the change is intended. The second is a floating point value that specifies the requested bandwidth limit. These messages have the following format:

bandwidth set upper channel value
bandwidth set: message tokens
upper: token - selects upper bandwidth mode
channel: us - selects the channel
value: float - specifies the bandwidth value

bandwidth set lower channel value
bandwidth set: message tokens
lower: token - selects lower bandwidth mode
channel: us - selects the channel
value: float - specifies the bandwidth value

If a requested bandwidth value does not exactly match a legal plug-in value, the plug-in will set the bandwidth limit to the nearest legal value.

When either of these commands is executed, the plug-in will return the status of the bandwidth control for the selected channel using the following messages:

bandwidth status upper channel value
bandwidth status: message tokens
upper: token - selects upper bandwidth mode
channel: us - specifies the channel
value: float - specifies the bandwidth value

bandwidth status lower channel value
bandwidth status: message tokens
lower: token - selects lower bandwidth mode
channel: us - specifies the channel
value: float - specifies the bandwidth value

These status messages will be sent in response to an **SRQ query** message to report bandwidth changes not requested by the mainframe. The mainframe must be able to take appropriate action when these messages are received.

bandwidth query upper channel
bandwidth query: message tokens
upper: token - selects upper bandwidth mode
channel: us - specifies the channel

bandwidth query lower channel
bandwidth query: message tokens
lower: token - selects lower bandwidth mode
channel: us - specifies the channel

The mainframe may send either of these messages to a generic plug-in to request its present bandwidth setting without specifying a new setting. The plug-in will respond with the appropriate **bandwidth status** message.

If the plug-in does not **send** the **lower_bandw** token in the **plugin_config status** message, the **bandwidth set lower** message is not legal. See section **3.1 Startup Sequence** for more information.

Error handling: Generic plug-ins do no checking on channel numbers. If a channel number is received as part of a **bandwidth set** command that a plug-in does not have, the plug-in will take unspecified action and return **a bandwidth status** message indicating the action it took.

1.8 Units

Generic amplifier plug-ins accept the **units query** message to allow the mainframe to ask for the present units of conversion. It has the following format:

units query
units query: message tokens

The plug-in will respond with the **units status** message indicating the present units of conversion provided by the probe. The status message has the following format:

units status type
units status: message tokens
type: string - indicates the units of conversion

This status message will also be sent in response to an **SRQ query** message any time the probe is changed. The mainframe must be able to take appropriate action when this message is received. The units *type* will be that reported by the probe or, if the probe does not report its units or if there is no probe, the units reported will be "Volts".

1.9 Knobs and Knob Display

Generic amplifier plug-ins will provide the information necessary for control of the offset and gain functions by control knobs and for status display. The offset and gain functions provide incremental setting commands that are used by the knobs to control these functions.

If the operation of a knob causes it to send messages to the plug-in faster than the plug-in can respond, the plug-in will not attempt to queue those messages. The mainframe is responsible for accumulating knob changes.

The mainframe may at any time request the status of either the offset or gain functions for display (as when one of those functions is assigned to a control knob) using the query commands. Further updating of the displays will be done by monitoring the status messages returned by the plug-in.

When a differential plug-in is installed, the mainframe's offset function will control one of the four internal offset values dependant on the configuration of the plug-in. Which offset is to be controlled is defined in the plug-in's EIS.

The plug-in will use the following message to report display parameters to the mainframe:

disp_attr status lmpb (channel [**gain** gain_min gain_max gain_res] [**offset** offset_min offset_max offset_res] [**diff_offset** amp_res vc_res plus_res minus_res] [**bandwidth** bwl_res] [**impedance** imp_res] [**delay** rel_delay] **EOD**)...
disp_attr status: message tokens
lmpb: token - long message protocol
channel: us - selects the channel
gain: token - specifies the beginning of gain attributes
gain_min: float - specifies the minimum gain value
gain_max: float - specifies the maximum gain value
gain_res: us - specifies the gain resolution
offset: token - specifies the beginning of offset attributes
offset_min: float - specifies the minimum offset value
offset_max: float - specifies the maximum offset value
offset_res: float - specifies the offset resolution
diff_offset: token - specifies the beginning of differential offset attributes
amp_res: float - specifies the amplifier offset resolution
vc_res: float - specifies the comparison voltage resolution
plus_res: float - specifies the plus probe offset resolution
minus_res: float - specifies the minus probe offset resolution
bandwidth: token - specifies the bandwidth attribute
bwl_res: us - specifies the bandwidth limit resolution
impedance: token - specifies the impedance attribute
imp_res: us - specifies the impedance resolution
delay: token - specifies the delay attribute
rel_delay: float - specifies the relative delay
EOD: token - defines the end of a channel list

The plug-in will send this message in response to an **SRQ query** message when any of the parameters changes or when requested by the mainframe using the **disp_attr query** message. This message is also sent as part of a command transaction when one of the parameters has changed as the result of a mainframe command to the plug-in. When not initiated by the **disp_attr query** message, the plug-in will only send parameters that have changed. When initiated by the **disp_attr query** message, the plug-in will send all non-default parameters. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter.

The **gain** token signifies that the three parameters following it apply to the gain function. The *gain_min* and *gain_max* parameters specify the minimum and maximum settings (in units/division) available for the gain function. The *gain_res* parameter specifies the number of digits that are significant for display purposes.

The **offset** token signifies that the three parameters following it apply to the offset function. The *offset_min* and *offset_max* parameters specify the minimum and maximum settings (in units) available for the offset function. The *offset_res* parameter specifies the offset resolution in volts. The number of digits to be displayed is a function of the present offset setting and the offset resolution according to the following equation:

$$num\_digits = int(\log_{10}(\frac{present\ value}{resolution})) + 1$$

The int function truncates the value toward 0. The *offset_res* value is also the step size of the fine control.

The **diff_offset** token signifies that the following four parameters apply to the differential offset values. The *amp_res* parameter specifies the resolution of the amplifier offset control. The *vc_res* parameter specifies the resolution of the comparison voltage control. The *plus_res* parameter specifies the resolution of the offset to the probe connected to the plus input. The *minus_res* parameter specifies the resolution of the offset to the probe connected to the minus input. This information is provided to support formatting these values for reporting over the external interface. These values indicate the number of significant digits for conversion to ASCII. The calculation for the number of digits is the same as for the offset function (see above). The *offset_min* and *offset_max* parameters specify the minimum and maximum values of the presently controlled differential offset function.

The **bandwidth** token signifies that the parameter following it applies to the bandwidth function. The *bwl_res* parameter indicates the number of display digits of resolution that are meaningful for the bandwidth function.

The **impedance** token signifies that the parameter following it applies to the input impedance function. The *imp_res* parameter indicates the number of display digits of resolution that are meaningful for the input impedance function.

The **delay** token defines the following parameter to be the plug-in delay. The *rel_delay* value specifies the delay of the signal path relative to the value measured by the mainframe at some reference position. The *rel_delay* parameter reports the difference of delay in seconds.

The **EOD** token defines the end of a list of channel attributes. Following the **EOD** token is either the end of the message or the beginning of the next channel list.

disp_attr query
disp_attr query: message tokens

This message is sent by the mainframe to request information about generic function display parameters. The plug-in will send the **disp_attr status** message to report those parameters.

1.10 Generic Menus

There are three basic menus that are provided by the mainframe to control the coupling, input impedance and bandwidth limit generic amplifier plug-in functions. The coupling and bandwidth limit menus will be split into two parts depending on the configuration of the plug-in resulting in a total of up to five menus.

The types of entries for each of these menus are predefined and are understood by both the mainframe and the plug-in. The plug-in will upload a list of entries for each menu and each channel (there might be a different list of entries for each channel). The menus are titled by the mainframe and the entries are formatted for display by the mainframe according to the predefined types. This information is uploaded using the following format:

> **menu def_generic** lmpb [{menu_type channel item_list}... **EOM**]
> **menu def_generic**: message tokens
> lmpb: token - long message protocol
> menu_type: menu definition token
> channel: us - specifies the channel
> item_list: tokens or floating point numbers - menu entries
> **EOM**: token - terminates the definition

The *lmpb* parameter is the long message protocol byte. See section 6 for details on long message transfers. For the **menu def_generic** message, messages may be split for transfer only at the end of an *item_list*.

Valid *menu_type* tokens are **plus_coupl**, **minus_coupl**, **impedance**, **upper_bandw** and **lower_bandw**. The menu specification message may contain the specification for only one menu or for all menus. When the *item_list* consists of tokens (for the coupling menus), the list is terminated by the **EOC** token. If the *item_list* consists of floating point values, the list is terminated by the NaN (Not a Number) floating point value*. The **EOM** token specifies the end of all definitions. The plug-in may send definitions for several menus in a single **menu def_generic** message.

For the coupling menus, the mainframe will display the text associated with the token sent by the plug-in. For the input impedance and bandwidth limit menus, the mainframe will convert the floating point values sent by the plug-in to a text display format. These values will be formatted according the the information in the **disp_attr status** message. The entries for these menus will always be uploaded in increasing numerical order (first menu item is smallest,

---

* See the **Introduction** section for the definition of this parameter value.

last is largest). There is a limit of 4 entries for the coupling menus, 4 entries for the input impedance menu and 16 entries for the bandwidth menus. If a lower bandwidth limit of 0Hz is specified, the mainframe may optionally display this value as 'DC' instead of '0Hz'. If the floating point representation for infinity* is specified, the mainframe will display the infinity symbol (∞) for the impedance menu or "Full" for the bandwidth limit menu.

The plug-in may, in response to an **SRQ query** message, upload a new list of items for any menu. This action might be triggered, for example, by the user changing the probe. The plug-in may not supply an empty list for an already defined menu or supply a list for a menu that was not defined at power up. Changes to the list of active menus may occur only at system reset (power up).

When a menu item is selected (touched), the mainframe will send the message associated with the menu to the plug-in with either the floating point value of the menu selection or the token value (for coupling menus) and the appropriate channel number. These messages are the standard generic plug-in commands defined earlier in this document.

By definition, only one item in each menu may be selected at a time. The mainframe will handle selection indication by changing the font or status display. When an item is touched, the mainframe will indicate de-selection of that item then indicate the newly selected item when the plug-in returns its status message. If there is only one item in a menu, the item is not selectable and the menu is a status display only.

menu request generic
menu request: message tokens
generic: token - requests generic menus

This message is sent by the mainframe during the power up sequence to request generic plug-in menus. The plug-in will respond with the **menu def_generic** message to define the contents of all its menus for all channels. The plug-in will include all entries for all menus in response to the **menu request generic** message. Note that the **menu request** message is also used by smart plug-ins with the **min**, **mix** or **max** tokens to request extended menus. See section **2.1 Extended Menus**.

1.10.1 Coupling Menus

The coupling menus accept four predefined tokens as entries. These tokens have semantic significance to both the mainframe and the plug-in. The four tokens are **DC**, **OFF**, **AC** and **VC**. The semantic significance for each item is defined as follows:

DC: All components of the input signal are passed to the mainframe.

OFF: The input signal is disconnected. The amplifier input is connected to ground.

AC: The DC component of the input signal is blocked. Absolute ground reference information on this channel is not available for display by the mainframe (for axes, cursors etc.)

VC: The input is connected to the comparison voltage (used for differential amplifiers only). The offset function now controls the comparison voltage instead of the offset voltage.

These tokens are also used in the **coupling** messages. See section **1.3 Coupling.**

The entries for the coupling menus are uploaded to the mainframe during the initialization sequence and in response to an **SRQ query** message anytime thereafter a change is required in one of the menus. The plug-in will use the following format:

menu def_generic lmpb [{**plus_coupl** channel coupl... **EOC**}... **EOM**]
menu def_generic: message tokens
lmpb: token - long message protocol
plus_coupl: menu token
channel: us - specifies the channel
coupl: token - specifies the entry for that menu slot
EOC: token - terminates the coupling list
EOM: token - terminates the menu definition

menu def_generic lmpb [{**minus_coupl** channel coupl... **EOC**}... **EOM**]
menu def_generic: message tokens
lmpb: token - long message protocol
minus_coupl: menu token
channel: us - specifies the channel
coupl: token - specifies the entry for that menu slot
EOC: token - terminates the coupling list
EOM: token - terminates the menu definition

The coupling menu has two parts, plus and minus coupling. Single-ended plug-ins will use the plus coupling menu only defined by the **plus_coupl** message. If a plug-in does not have minus coupling capability, it will not send the **minus_coupl** token. When the minus coupling menu is empty, the mainframe may label the plus coupling menu simply COUPLING.

When an item in a coupling menu is selected, the mainframe will send either the **coupling set plus** or **coupling set minus** message to the plug-in with the appropriate channel and coupling parameters. The mainframe will update the menu display with the response from the plug-in.

1.10.2 Input Impedance Menu

The input impedance menu accepts floating point values as entries. These values define the resistive load on the input. This menu will contain the impedance of the presently connected probe for probes that provide that information. The plug-in will use the following format to upload these values:

menu def_generic lmpb [{**impedance** channel value... NaN}... **EOM**]
menu def_generic: message tokens
lmpb: token - long message protocol
impedance: menu token
channel: us - specifies the channel
value: float - menu entry
NaN: Not a Number - defines end of entry list
EOM: token - terminates the menu definition

When an item in an input impedance menu is touched, the mainframe will send the **impedance set** message to the plug-in with the appropriate channel and floating point parameters. The mainframe will update the menu display with the response from the plug-in.

1.10.3 Bandwidth Limit Menus

The bandwidth limit menus accept floating point values as entries. These values define the selections for control of the bandwidth of the signal path in the plug-in. The plug-in will use the following format to upload these values:

menu def_generic lmpb [{**upper_bandw** channel value... NaN}... **EOM**]
menu def_generic: message tokens
lmpb: token - long message protocol
upper_bandw: menu token
channel: us - specifies the channel
value: float - menu entry
NaN: Not a Number - defines end of entry list
EOM: token - terminates the menu definition

menu def_generic lmpb [{**lower_bandw** channel value... NaN}... **EOM**]
menu def_generic: message tokens
lmpb: token - long message protocol
lower_bandw: menu token
channel: us - specifies the channel
value: float - menu entry
NaN: Not a Number - defines end of entry list
EOM: token - terminates the menu definition

The bandwidth limit menu has two parts, upper and lower bandwidth limit. If a plug-in does not have lower bandwidth limit capability, it will not send the **lower_bandw** token. The mainframe will use this information for setting up the GPIB handler and the menu interface.

When an item in a bandwidth limit menu is touched, the mainframe will send either the **bandwidth set upper** or the **bandwidth set lower** message to the plug-in with the appropriate channel and floating point parameters. The mainframe will update the menu display with the response from the plug-in.

The bandwidth limit menu controls the bandwidth function of the signal path in the plug-in. It does not control the system bandwidth. If the maximum bandwidth of a plug-in is greater than the bandwidth of the mainframe in which it is installed, the mainframe will not attempt to modify the bandwidth values used for the bandwidth menus to conform to the mainframe capability. Lower bandwidth values will not be modified when the user selects **AC** coupling.

1.11 ID Function

Generic amplifier plug-ins provide a front panel push button that can be used to identify specific channels. This button is called the display ON/OFF button. When this button is pressed by the user the plug-in will send a status message to the mainframe. The format of this message is as follows:

channel_id status channel
channel_id status: message tokens
channel: us - specifies the channel

The *channel* parameter specifies which channel's button was pressed. The plug-in will send an SRQ to request service from the mainframe. The **channel_id status** message will be sent in response to the subsequent **SRQ query** message from the mainframe.

Generic amplifier plug-ins also provide a means for monitoring the ID button on probes that have such a facility. The plug-in will send a status message to the mainframe when the probe ID button is pressed. The format of this message is as follows:

probe_id status channel
probe_id status: message tokens
channel: us - specifies the channel

The *channel* parameter specifies which channel's probe id button was pressed. The plug-in will send an SRQ to request service from the mainframe. The **probe_id status** message will be sent in response to the subsequent **SRQ query** message from the mainframe.

Generic plug-ins also provide one front panel LED to indicate display status for each channel. The mainframe is given explicit control of the LED with the following messages:

led set channel state
led set: message tokens
channel: us - specifies the channel
state: token - specifies the LED state

The plug-in will set the condition of the LED for the specified *channel* according to the *state* parameter. There are two defined state tokens: **on** and **off**.

The plug-in will report the status of the LED with the following message:

led status channel state
led status: message tokens
channel: us - specifies the channel
state: token - specifies the LED state

This message is sent by the plug-in whenever the mainframe changes the status of an LED or when the **led query** message is sent. The *channel* parameter specifies which channel's LED status is being reported. The *state* token indicates the LED state: **on** or **off**.

led query channel
led query: message tokens
channel: us - specifies the channel

This message is sent by the mainframe to request the status of the plug-in front panel LED's. The *channel* parameter specifies the channel for which LED status is requested. The plug-in will respond with the **led status** message.

1.12 Display Updating

The mainframe will handle updating of all display items related to generic amplifier plug-ins using the status messages defined in this document. For gain and offset, the mainframe will update axes, numerical displays and knob status using the information supplied by the **gain status**, **offset status** and **disp_attr status** messages. For coupling, input impedance and bandwidth limit, the mainframe will update the control menus using the status messages returned by those commands. See section **2.7 Updates** for smart plug-ins for details of how smart plug-ins will be notified of changes to generic plug-ins.

1.13 Stored Settings

The mainframe will manage the stored settings function for generic amplifier plug-ins including the present settings. Generic plug-ins will not store any present settings for restoration at power up. For selectable front panel settings recall and saving, the mainframe will be responsible for creating the stored settings buffer, updating it when requested and sending setting information to the plug-in when required.

There are five functions for each plug-in channel that need to be saved: gain, offset, coupling, input impedance and bandwidth limit. Gain and offset status are floating point values, coupling status is a token and input impedance and bandwidth limit status are floating point values. The format used by the mainframe to store this information is determined by the mainframe. The mainframe will use the standard commands and status messages defined in this document to save and recall front panel settings.

For differential plug-ins, there are four floating point offset values that the mainframe must maintain. These are identified as the *amp_offset*, *vc_offset*, *plus_offset* and *minus_offset*. These values will be reported by the plug-in anytime the offset function is changed using the **diff_offset status** message. The mainframe will restore these values using the **diff_offset set** message.

The display and trigger selection functions of generic amplifier plug-ins will be handled separately. The mainframe will use the trace description information that is saved in the mainframe's stored settings buffer to initialize the plug-in using the standard commands defined in this document in section **1.16 New Trace**. Generic plug-ins will intialize the display and trigger output sequencers with all channels off in all sequence slots. All front panel LED's will be off after power up.

1.14 Interactions With Smart Plug-Ins

Smart plug-in interaction with generic plug-ins will be invisible to the generic plug-ins. The smart plug-ins will access the five basic functions of generic amplifier plug-ins by using the interface provided by the mainframe. The display and trigger selection functions will be accessed through mainframe functions provided specifically for smart plug-ins. See section **2.6.1 Generic Plug-Ins**. Information on interactions during calibration and self-tests is found in sections **4.0 Self-tests** and **5.0 Self-Cal**.

1.15 Error Handling

See section **3.2 Error Handling** for general information on the different error message types and how they will be handled. Generic plug-ins will not supply ASCII messages with error messages. The status byte associated with a generic error message will identify the type of error. The mainframe will handle display and reporting of these errors. Error handling for specific commands are defined in the section describing that command.

There are two types of message errors that are detected by generic amplifier plug-ins. An unknown message is a message token pair that a generic plug-in has no knowledge of (ie. it may be a smart plug-in message or just an erroneous value). In this case, the plug-in will take no action and return an **error generic** message with a **command_error** *status* byte and the *code* value set to 157. The other error type is an error for specific command. These are noted where the command is defined.

Parameter Resolution

This situation occurs when the parameter for a command specifies more resolution than the capability of the function. The plug-in will round the value to the nearest legal setting, set the function to that setting and return the new status to the mainframe. There will be no errors or warnings generated.

Input Overload

This error occurs when the user overloads the low impedance input termination resistor. The plug-in will change the input resistance as specified in the plug-in's EIS. The plug-in will send an **error generic** message with an **internal_warning** *status* token with the *code* value set to 651. The plug-in will return the new status of the input impedance using the standard **impedance status** message.

Input Overdrive

This error occurs when the input to an amplifier is overdriven in a way that might distort the displayed signal. The plug-in will not make any change to its input. This is not a catastrophic error condition. The plug-in will send an **error generic** message with an **internal_warning** *status* token with the *code* value set to 652.

For the input overload and overdrive cases, the plug-in will send an SRQ then wait for the **SRQ query** message from the mainframe. The defined error messages will be sent in response to the **SRQ query**.

Illegal Coupling

There are two types of coupling errors. If the plug-in receives a **coupling set** message with an invalid coupling token, the plug-in will not change to its coupling setting and will send the **error generic** message with a **command_error** *status* token with the *code* value set to 157.

If the plug-in receives a coupling token that is normally supported but is not presently allowed because a particular probe is connected, the plug-in will not change the coupling setting and will send the **error generic** message with an **exec_error** *status* token and the *code* value set to 284.

Illegal Input Impedance

If a plug-in receives an input impedance value that is normally supported but is not presently allowed because a particular probe is connected, the plug-in will not change the input impedance setting and will send the **error generic** message with an **exec_error** *status* token and the *code* value set to 285.

1.16 New Trace

Generic amplifier plug-ins provide a sequencer that controls the trigger and display multiplexed outputs. The operation of this sequencer is controlled by mainframe hardware lines. The sequencer has 12 slots. Each slot contains a combination specification. A combination specifies how each channel is combined for that slot: off, normal or inverted. Each combination is programmed by the mainframe using the **display set** and **trigger set** messages.

New Trace operations for generic amplifier plug-ins will be built into the mainframe. These operations will be enabled when a plug-in identifies itself as a generic plug-in at initialization. For new trace operations, these plug-ins can add or subtract any combination of channels. The limitations are: a channel may appear only once in a combination specification; no more than 12 combinations may be specified.

The plug-in sends two pieces of information to the new trace function at initialization: generic amplifier plug-in identification and channel information. The plug-in will specify how many channels are available for operation. Each channel must be able to be added or subtracted from all other channels. Channels are not allowed to have individual combination characteristics (eg. ch1 can only be added to ch2 not ch3 or ch4). The plug-in will send channel information for the display, trigger and auxiliary trigger outputs. See section **3.1 Startup Sequence** for more information.

The **display set** message with a list of combinations is used to control the display output of generic plug-ins. There may be up to 12 combinations in the list. A complete list of all active combinations must be supplied each time a change is made to any combination. If there are fewer than 12 combinations specified, the remaining slots in the sequencer are cleared by the plug-in (all channels set to off). Each combination specifies how each channel is to be combined for that slot: off, normal or inverted.

The **trigger set** message with a list of combinations is used to control the trigger output of generic plug-ins. The operation and interface for this command is identical to the display command.

The format for these commands is as follows:

display set num comb...
display set: message tokens
num: us - specifies the number of combinations
comb: special format - specifies the channel combination

trigger set num comb...
trigger set: message tokens
num: us - specifies the number of combinations
comb: special format - specifies the channel combination

The number of combinations sent by the mainframe defines the number of combinations used in the channel switching sequence. The mainframe must add all channels off combinations as necessary to fulfill this requirement. The *num* parameter specifies the number of combinations in the message. The comb parameter is defined below.

The plug-in will return the new display or trigger status after one of these commands is received using the following status message format:

display status num comb...
display status: message tokens
num: us - specifies the number of combinations
comb: special format - specifies the channel combination

trigger status num comb...
trigger status: message tokens
num: us - specifies the number of combinations
comb: special format - specifies the channel combination

These status messages always return all defined *combinations* even if some are all channels off. The number of *combinations* sent will be the number defined by the mainframe in the last **display set** or **trigger set** command. The *num* parameter specifies the number of combinations in the message.

Channel combinations (the *comb* parameter in the above messages) specify how each channel is combined for each sequence slot. The channel combination must specify two things: whether a channel is on or off and whether a channel is inverted or normal. Channel combination bytes are split into fields. Each field describes the method of combination for a single channel. Each field consists of two bits. One bit specifies whether the channel is on or off. The other bit specifies whether the channel is inverted or normal. There are as many fields in the combination as there are channels. The fields are ordered from channel one in the least significant bits to the last channel in the most significant bit field. Here are examples for one, two and four channel plug-ins:

one channel:	bits	7	6	5	4	3	2	1	0
	meaning	X	X	X	X	X	X	P1	E1
two channel:	bits	7	6	5	4	3	2	1	0
	meaning	X	X	X	X	P2	E2	P1	E1
four channel:	bits	7	6	5	4	3	2	1	0
	meaning	P4	E4	P3	E3	P2	E2	P1	E1

Pn: selects polarity for channel n: 0 = +up, 1 = inverted
En: enables output for channel n: 0 = off, 1 = on
X: don't care

When a channel is not enabled, the polarity bit has no effect. The polarity and enable status of each channel may be set independently.

display query
display query: message tokens

trigger query
trigger query: message tokens

These messages are used by the mainframe to request display and trigger output status. The plug-in will respond with either the **display status** or the **trigger status** message as appropriate.

If a plug-in sends the **no_invert** token in the **plugin_config status** message, the channel polarity bits for the *comb* parameters defined above are ignored. Channels are always non-inverted (+up).

1.17 Auxiliary Triggers

Each generic plug-in that provides auxiliary trigger outputs will be able to turn those outputs on or off on command. These outputs should be left off except when the auxiliary lines are terminated by a smart plug-in in the right compartment. This reduces the aberrations that would occur if the auxiliary lines are unterminated and cause reflections. There is no external or human interface access to this function.

aux_trig set mode
aux_trig set: message tokens
mode: token - selects mode

This message causes a generic plug-in to enable or disable its auxiliary trigger outputs. The **on** *mode* token causes all auxiliary channel outputs to be enabled. The **off** *mode* token causes all auxiliary channel outputs to be disabled. All channels are controlled simultaneously; channels may not be controlled individually.

aux_trig status mode
aux_trig status: message tokens
mode: token - specifies mode

This message is used by generic plug-ins to report the status of the auxiliary trigger outputs in response to an **aux_trig set** or **aux_trig query** message. The **on** *mode* token specifies that the outputs are enabled. The **off** *mode* token specifies that the outputs are disabled.

aux_trig query
aux_trig query: message tokens

This message is sent to a generic plug-in to request the status of its auxiliary trigger outputs. It will respond with the **aux_trig status** message.

1.18 ASCII Interface

The mainframe will handle the GPIB interface for generic amplifier plug-ins. The syntax for the GPIB commands that control amplifier plug-in functions is built into the mainframe. The mainframe will use the information supplied by the plug-in for menus and the new trace function to complete the parse tables for the GPIB interface. In this section are listed relevant GPIB commands and their associated plug-in command messages which are defined elsewhere in this document. Refer to the *Command Reference Specifications 11K Series Family of Products* document for more details on these GPIB commands.

1.18.1 Input Commands

The input commands control the generic amplifier plug-in functions of gain, offset, coupling, input impedance and bandwidth limit. These GPIB commands have a header that specifies the channel, a link argument that specifies the function followed by an argument that specifies the setting of the function. When the mainframe has parsed one of these commands it will send a command to the plug-in consisting of a plug-in command message based on the GPIB link argument, a channel argument based on the GPIB header and a function setting value based on the GPIB command argument. Generic amplifier plug-ins do not support the TERMINATION or SCALE link arguments.

The <alpha> field in the header specifies the plug-in compartment. This is used by the mainframe to direct the command to the proper plug-in. The <ui> field specifies the channel in the plug-in. This value corresponds to the *channel* parameter in all plug-in commands.

The mainframe will use the **gain set abs** command message when the SENSITIVITY link argument is received. The mainframe will convert the numeric argument received from the GPIB to a binary format for the plug-in command argument.

The mainframe will use the **offset set abs** command message when the OFFSET link argument is received. The mainframe will convert the numeric argument received from the GPIB to a binary format for the plug-in command argument.

The mainframe will use the **diff_offset set** command message when any of the AMPOFFSET, VC, PLSOFFSET or MNSOFFSET link arguments are received. The mainframe will send the *control* token defined in the following table based on the link argument that it receives:

link argument	*control* token
AMPOFFSET	**amp**
VC	**vc**
PLSOFFSET	**plus**
MNSOFFSET	**minus**

The mainframe will convert the numeric argument received from the GPIB to a binary format for the plug-in command argument.

The COUPLING, PLSCOUPLING and MNSCOUPLING link arguments correspond to the coupling commands for the plug-in. When the COUPLING or the PLSCOUPLING links are received, the mainframe will use the **coupling set plus** command message. When the MNSCOUPLING link is received, it will use the **coupling set minus** command message. The arguments for these links are ASCII text indicating the setting for the coupling function. They correspond directly to the coupling tokens defined in section **1.10.1 Coupling Menus**. The mainframe will convert the text arguments to the associated coupling token value for the message. PLSCOUPLING and MNSCOUPLING link arguments are not valid for single ended channels. The COUPLING link argument will set the coupling for both inputs for a differential channel. The mainframe will send both a **coupling set plus** and a **coupling set minus** message to a differential plug-in when the COUPLING link is received.

When the IMPEDANCE link argument is received, the mainframe will use the **impedance set** plug-in command message. The GPIB argument for this command is a numeric value. The mainframe will convert this argument to a floating point value and send it as the argument to the plug-in.

The BWHI and BWLO link arguments correspond to the bandwidth commands for the plug-in. The mainframe will use the **bandwidth set upper** command message for the BWHI link and the **bandwidth set lower** command message when the BWLO link is received. The GPIB argument for these commands is a numeric value. The mainframe will convert it to a binary value and send it as the argument to the plug-in. The BWLO link argument is not valid for plug-ins that do not have a lower bandwidth limit function.

1.18.2 Waveform Commands

The TRACE<ui> command corresponds indirectly to the plug-in **display set** command. As part of the operation of creating a trace based on the GPIB description, the mainframe will send a channel combination command to the plug-in using the **display set** command message. See the generic plug-in section **1.16 New Trace**.

1.18.3 Triggering Commands

The TRMAIN, TR1DELAY and TR2DELAY commands correspond indirectly to the plug-in **trigger set** command. As part of the operation of creating a trigger based on the GPIB description, the mainframe will send a channel combination command to the plug-in using the **trigger set** command message. See the generic plug-in section **1.16 New Trace**.

1.19 Generic Plug-In SRQ's

This section defines how generic plug-ins will use SRQ's to indicate status. See section **6.0 Protocols** for the definition and operation of the transport system SRQ.

Generic plug-ins will send an SRQ under the following conditions:

1. The user overloads or overdrives the input.

2. The user presses a front panel button.

3. The user adds or removes a probe.

4. The user presses the probe id button.

5. Power up.

When a generic plug-in detects an overloaded or overdriven input, the plug-in will send an SRQ to the mainframe. When the mainframe sends the **SRQ query** message to the plug-in, the plug-in will respond with the **error generic** message with the **command_error** *status* token as defined in section **1.15**. In response to a subsequent **SRQ query** message from the mainframe, the plug-in may also report new status for the input impedance if that function was changed. The plug-in will send the **SRQ no_report** message for subsequent **SRQ query** messages.

When the user presses a front panel button, the plug-in will send an SRQ. In response to the **SRQ query** message, the plug-in will send the **channel_id status** message indicating which channel's button was pressed. Subsequent **SRQ query** messages will receive an **SRQ no_report** message in response.

When the plug-in detects the addition or removal of a probe, it will send an SRQ. The plug-in will send the **probe status** message indicating the new probe status. In addition, the plug-in, in response to subsequent **SRQ query** messages may also send status or menu definition messages for any of its functions that were changed as a result of the change in probes. The plug-in will continue to send status messages until all new status has been reported. At that point, the plug-in will send the **SRQ no_report** message in response to the **SRQ query** message.

When the user presses the probe id button, the plug-in will send an SRQ. In response to the **SRQ query** message, the plug-in will send the **probe_id status** message indicating which channel's button was pressed. Subsequent **SRQ query** messages will receive an **SRQ no_report** message in response.

At power up, the plug-in will send an SRQ. See section 3.1 for details of the power up sequence.

If several events occur simultaneously, the plug-in will report status in a prioritized manner until all status has been reported. The prioritization is defined in the plug-in's EIS.

1.20 Probes

Generic plug-ins will handle most probe functions without interaction with the mainframe. All generic functions affected by probes will be handled transparently by generic plug-ins. These effects will be reported to the mainframe whenever a change in probes is detected. The plug-in will also report the change in probes for the purpose of keeping the calibration system informed of the system configuration (which includes probes).

There are three defined levels of probes that are supported by generic plug-ins. A level 0 probe does not support either the 7K resistive encoding or the new 11K digital encoding. The plug-in cannot detect the presence of these probes so plug-in operation presumes that no probe is attached. Level 1 probes support the 7K resistive encoding scheme. These probes only report their attenuation factor. This factor will be incorporated by the plug-in as defined below. Level 2 probes use the new 11K digital encoding scheme. These probes report several parameters in addition to their attenuation factor. Generic plug-ins will incorporate this information as defined below. See the *New Probe Interface Manual* for details on the new 11K probe interface.

The effects of a particular probe will be reported by the plug-in when it detects a probe change. This change will be the removal or addition of a level 1 or level 2 probe. The plug-in will send an SRQ to request mainframe attention then send the appropriate status messages in response to the **SRQ query** messages from the mainframe. Probe status is maintained individually by channel for single ended plug-ins and by input for differential plug-ins.

Generic plug-ins maintain two calibration constants for each probe. These are the nominal probe value and the actual probe value. The nominal probe value is the attenuation value reported by a level 1 or level 2 probe. This value may also be changed by the mainframe using the **cal probe_nom set** message. The other value is the probe actual value. This is the value of the probe's attenuation factor as measured by the plug-in during calibration for either a level 1 or level 2 probe. This value may also be changed by the mainframe using the **cal probe_act set** message. The plug-in will modify these values when a probe is attached or removed as defined below.

1.20.1 Effects of Adding a Probe

This section defines the effects of adding a level 1 or level 2 probe. The presumption is that the plug-in was previously detecting no probe (level 0) when the probe is added.

For the gain function, the plug-in will change the gain of the signal path by the attenuation value reported by the probe. Thus, if the deflection is 1V/div and a X10 probe is added, the new deflection will be 10V/div. This approach requires no internal changes to plug-in hardware to achieve the new gain setting. It also means that the new defined gain setting is always achievable. The plug-in will report the new gain setting using the **gain status** message.

For the offset function, the plug-in will attempt to maintain the previously requested offset value. If the actual offset value is different than the requested offset value, the plug-in will attempt to achieve the requested offset value. If the probe identifies itself as an offsetable probe (using the new 11K interface), single ended input plug-ins will disconnect the internal offset control and connect the control to the probe then attempt to achieve the requested offset value. For differential plug-ins, no change will be made to any offset value when an offsetable probe is attached to an input. The requested value (either plus offset or minus offset) for the input to which the probe is attached will be used to set the offset for that probe. The plug-in may, however, change which offsets are active as defined in the plug-in's EIS. The plug-in will report offset changes using the **offset status** and/or **diff_offset status** messages.

Some probes will require the input impedance to be set to 50Ω. When such a probe is attached, the plug-in will change the impedance as requested by the probe but will remember the impedance requested by the user. If the user subsequently requests a change in impedance, the plug-in will not change the impedance but will remember the requested value. The plug-in will report the impedance change using the **impedance status** message.

Offsetable probes require that the plug-in input coupling not be set to AC. When such a probe is attached, if the coupling is set to AC, the plug-in will change the coupling to DC as requested by the probe but will remember the setting requested by the user. If the user subsequently requests a change in coupling to AC, the plug-in will not change the coupling but will remember the requested setting. For differential plugins, if the coupling is VC, adding an offsetable probe will not change the coupling to DC. The user may request VC, DC or GND coupling when the probe is attached. The plug-in will report any coupling changes using the **coupling status** message.

If changes are made to the range of gain or offset, the plug-in will report those changes using the **disp_attr status** message.

The plug-in will send a **probe status** message to update the mainframe's table of probe status. This table is used during calibration to determine calibration needs. See section **5.1.2 Probe Configuration** for the definition of this message.

The plug-in will set the probe nominal value and the probe actual value to the attenutaion value reported by the probe. The plug-in will do this regardless of whether the mainframe had previously set the probe nominal or actual values or whether the plug-in had performed a calibration to set the actual value. No previous values will be remembered.

1.20.2 Effects of Removing a Probe

This section defines the effects of removing a level 1 or level 2 probe. The presumption is that the plug-in's detection of the presence of no probe (level 0) triggers these operations.

For the gain function, the plug-in will change the gain of the signal path by the attenuation of the previously attached probe. Thus, if the deflection factor is 10V/div and a 10X probe is removed, the deflection factor will be set to 1V/div. This approach requires no internal changes to plug-in hardware to achieve the new gain setting. It also means that the new defined gain setting is always achievable. The plug-in will report the new gain setting using the **gain status** message.

For the offset function, the plug-in will attempt to maintain the previously request offset value. If the actual offset value is different than the requested offset value, the plug-in will attempt to achieve the requested offset value. If the probe that was removed was an offsetable probe, single ended input plug-ins will disconnect the offset to the probe and connect the offset control internally then attempt to achieve the requested offset value. For differential plug-ins, no change will be made to any offset value when an offsetable probe is removed. The requested value (either plus offset or minus offset) for the input from which the probe is removed will be remembered by the plug-in. The plug-in may change which offsets are active as defined in the plug-in's EIS. The plug-in will report offset changes using the **offset status** and **diff_offset** status messages.

If a probe is removed that requested 50Ω input impedance, the plug-in will check the requested input impedance value. If that value is different than 50Ω, the plug-in will set the input impedance to that value and report the change using the **impedance status** message.

If a probe is removed that prevented AC coupling, the plug-in will check the requested coupling setting. If that setting is AC, the plug-in will set the input coupling to AC and report the change using the **coupling status** message.

If changes are made to the range of gain or offset, the plug-in will report those changes using the **disp_attr status** message.

The plug-in will send a **probe status** message to update the mainframe's table of probe status. This table is used during calibration to determine calibration needs. See section **5.1.2 Probe Configuration** for the definition of this message.

The plug-in will set the probe nominal value and the probe actual value both to 1.0. The plug-in will do this regardless of whether the mainframe had previously set the probe nominal or actual values or whether the plug-in had performed a calibration to set the actual value. No previous values will be remembered.

This page is intentionally left blank.

2.0 Smart Plug-Ins

2.1 General

A smart plug-in is defined as any plug-in that has capabilities that are additional to those defined for generic amplifier plug-ins. These plug-ins may also use the facilities provided by the mainframe for generic amplifier plug-ins. A smart plug-in can specify these built-in functions to be activated by identifying itself as a **both** type plug-in. See section **3.1 Startup Sequence** for more information. Smart plug-ins may interrogate the mainframe or issue commands to the mainframe using the status/command interface defined in this section.

If there is more than one smart plug-in in a mainframe, the plug-ins have equal priority for requesting mainframe functions. Each function will be assigned to the plug-in issuing the most recent request. A plug-in cannot lock out other plug-ins from using mainframe resources. A smart plug-in may not use iterative control except during autoranging or calibration. This is to prevent two smart plug-ins from attempting to control the same function in a way that could lead to an infinite loop.

2.2 Extended Menus

Mainframes will provide an extended menu interface for use only by **smart** or **both** type plug-ins. These menus are defined in this section. **Both** type plug-ins may request a combination of generic plug-in menus and extended menus.

2.2.1 Menu Messages

Plug-In menus can only be activated by a front panel operation. They cannot be selected via an external bus command.

The following messages control the operation of extended menus.

menu request menu_type [wvfm_flag]
menu request: message tokens
menu_type: token - specifies the type of menu requested
wvfm_flag: token - enables or disables waveform display

The plug-in uses this command to request a menu display. The mainframe will clear its display of any other plug-in or mainframe menus (and notify those plug-ins of its action) in preparation for displaying the new menu definition. There are three predefined *menu_type* tokens: **max**, **mix** and **min**. These specify the indicated extended menu modes. The *wvfm_flag* parameter must be included when the *menu_type* is **min**, **mix** or **max**. The *wvfm_flag* tokens **wvfm_on** and **wvfm_off** specify whether the waveforms are to be displayed or disabled. The **wvfm_on** token requests the mainframe to make no changes to the waveform display when the menu is displayed. The **wvfm_off** token requests the mainframe to disable the waveform display if it would interfere with the menu display (that is if waveforms would appear in the menu area).

The **menu request** message is used by the mainframe to request generic plug-in menus. The mainframe will use the **generic** token as the *menu_type* and will not specify a *wvfm_flag* parameter.

menu status status
menu status: message tokens
status: token - indicates menu status

The mainframe will send this status message to the plug-in when it receives a **menu request** command or when there is a change in the status of the plug-in's menus. There are three *status* tokens: **ready**, **formatted** and **removed**. The **ready** token indicates the mainframe's response to a **menu request** command. The **formatted** token indicates that the mainframe has completed the display of the plug-in's menus in response to a **menu def_smart** or a **menu change** message. The **removed** token is sent when the menus are removed either by plug-in command or other action.

If a smart plug-in sends the **menu request** message when it has an extended menu already defined, the mainframe will not send the **menu status removed** message to that plug-in because the **menu request** message implies removal. The response to a **menu request** message is always **menu status ready**.

A smart plug-in will send the **menu status** message with the **removed** token in response to a **menu status removed** message from the mainframe when the plug-in has not requested the menu to be removed.

menu def_smart lmpb [[**TDL** title] def_spec...]
menu def_smart: message tokens
lmpb: token - long message protocol byte
TDL: token - delimits the title string
title: string - title for the menu
def_spec: special - specifies the contents of a menu

The plug-in uses this command to specify the items to be displayed in a menu. The mainframe will begin the definition of a new menu.

The *lmpb* parameter is the long message protocol byte. See section **6.0 Protocols**, for details on the use of the long message protocol. Only the *lmpb* parameter is sent when the *lmpb* parameter is **ack** or **abort**. Each message sent using the long message protocol must have a complete cell definition in the *def_spec*. Cell definitions may not be split between messages using the long message protocol.

The **TDL** token when present defines the beginning of a title for the menu. The *title* is a string of up to 40 characters that may be displayed by the mainframe as the menu's title. The mainframe is not required to display the *title*; the plug-in is not required to supply a *title*.

The *def_spec* parameter specifies the contents of the menu. See the next section **Menu Definition** for details on the operation of the **menu def_smart** message and the format of the *def_spec* parameter. The mainframe will create a new menu according to the *def_spec* parameter in this message. At least one **cell_def** is required in this message.

menu touch area_id
menu touch: message tokens
area_id: us - specifies the touch area number

The mainframe sends this status message any time a touch is detected in a plug-in defined menu touch area. The *touch_id* parameter indicates the touch area specified in the menu uploaded by the plug-in that was selected by the user. The mainframe will translate from screen touch coordinates to the plug-in specified *area_id* value.

The plug-in will respond to this message with either the **menu status ready** message indicating it has accepted the **menu touch** message or a menu request, menu delete or menu change message to request further menu action. The mainframe will respond to those messages as defined.

menu change lmpb [[**TDL** title] def_spec...]
menu change: message tokens
lmpb: token - long message protocol byte
TDL: token - delimits the title
title: string - title for the menu
def_spec: special - specifies the contents of a menu

This message is sent by a plug-in to make a change to an existing menu. The parameters have the same meaning as for the **menu def_smart** message. The mainframe will make changes to the existing menu as defined in this message and respond with the **menu status formatted** message. The plug-in is not allowed to change menu types with this message.

menu delete
menu delete: message tokens

This message is sent by a plug-in to delete an existing menu. The mainframe will remove the menu display and send the **menu status** message with the **removed** token.

menu restore
menu restore: message tokens

This message is sent to a smart plug-in to restore a previously defined menu. It will be sent by the mainframe to restore a plug-in menu that was removed for a keypad operation. Smart plug-ins are required to maintain sufficient information to restore the last menu that was removed by the mainframe (using the **menu status removed** message). When the menu is restored, it must include any changes resulting from operations the have occurred since the menu was last displayed.

2.2.2 Operation

Extended menus are defined for use by smart plug-ins for display situations that are not covered by the built-in generic menus. For these menus, the plug-in will specify a list of cells to be displayed by the mainframe. When a touch is made, the mainframe will send the the *area_id* associated with the touch location to the plug-in. See **Appendix C** for details on how specific mainframes implement extended menus.

There are three types of extended menus: Maximum Mode, Mixed Mode and Minimum Mode. Only one extended menu may be defined at a time for all smart plug-ins.

Maximum Mode

In this mode there are 8 rows of 10 touch areas. Each touch area or cell is 5 characters wide and 2 lines tall. The full menu requires 50 characters by 16 lines. Normal mainframe display functions such as knob readout and cursors might not be supported during this mode.

Mixed Mode

In this mode there are 7 rows of 10 touch areas. Each touch area or cell is 5 characters wide and 2 lines tall. The full menu requires 50 characters by 14 lines. In this mode, mainframe knob display and control are available.

Minimum Mode

In this mode there is 1 row of 8 touch areas. All mainframe functions will be accessible in this mode.

In each mode, the plug-in may supply a title that the mainframe may use to label the menu. If the mainframe cannot title menus then the title will be ignored. If a plug-in does not specify a title for a menu, the mainframe may optionally title it for the plug-in.

For all modes, the *wvfm_id* flag is used to request the mainframe to turn off the waveform display if it will interfere with the display of the menu. If waveform displays do not interfere with the menu, the mainframe may ignore this command. If waveforms do interfere, the mainframe must turn them off when requested.

The mainframe, at its option, may make the display of a menu no larger than necessary. Any unused outside rows or columns might not be displayed by the mainframe. This does not limit the plug-in's use of any valid touch area in a menu. An unused row or column is defined as a row or column in which the plug-in has not sent any **cell_def** specifications. A cell specified with no text makes its row and column used and therefore required to be displayed as part of the menu (although filled with blanks). The plug-in must not define cells outside the boundaries defined by the **menu def_smart** message with a

menu change message. Changes in menu boundaries must be made with a **menu def_smart** message.

A plug-in menu may be removed by the mainframe at any time. For all modes, the mainframe will notify the plug-in when this action occurs.

The following sequence will be used to control the display and removal of plug-in menus.

The plug-in will send a **menu request** message to the mainframe. This request will indicate the type of menu requested and whether waveforms are to be displayed or not. The mainframe will respond with a **menu status** message. The mainframe will clear the display (sending a **menu status removed** message to any other plug-in that has a menu displayed) and take any other necessary action to prepare for a menu before it sends the **menu status** message. After receiving the **menu status** message with the **ready** token, the plug-in has full access to the mainframe's menus until it receives the **menu status removed** message. During this time, the plug-in may send **menu def_smart** or **menu change** commands to the mainframe and may expect **menu touch** messages from it. Sending the **menu delete** message terminates this mode. All plug-in menu messages are initiated by an SRQ and are sent in response to an **SRQ query** message.

When it receives the **menu status ready** message, the plug-in will send a list of menu items for display using the **menu def_smart** message. This list will specify the location, text and font of displayed items. The mainframe will display these items according to the rules specified in the next section **Menu Definition**. The mainframe will return the **menu status formatted** message when the display has been formatted according to the menu specification.

Whenever a touch is detected in the plug-in's menu area, the mainframe will send a **menu touch** message indicating the location of the touch. The plug-in will take action according to its own interpretation of the touch. It might change its setup, remove the menu and display another, make changes to the present menu or take no action at all. The mainframe does no interpretation of menu touches.

To change the menu type (ie. from max to mix), the plug-in must send a **menu request** message. The mainframe will delete the plug-in's existing menu (but not send a **menu status removed** message) and send the **menu status ready** message. The plug-in will send a **menu def_smart** message to define the menu.

Four actions will cause a plug-in menu to be removed. The plug-in may request the menu to be removed (using the **menu delete** message), the plug-in may request a new menu (using the **menu request** message), another plug-in may request the menu display or another mainframe function may be selected (by touching a mainframe menu area or a mainframe button). In all cases, the mainframe will send a status message to the plug-in indicating its menu was removed and set its display to the requested operation.

If a plug-in requires an exit selection means from a menu, the plug-in must supply that function. The mainframe will not supply exit means for extended menus. Plug-ins are not required to provide an exit means in each menu.

2.2.3 Menu Definition

The plug-in will upload a list of items for the mainframe to display as a menu. Each item will contain the following information:

1. Cell row and column coordinates.
2. One or two strings of ASCII text of up to 5 characters, one for each line.
3. Cell font selectors for each line.
4. Area definitions with an id and size for each area.

The cell row and column coordinates specify the cell location. They are numbered 1-10 left to right for max and mix modes, 1-8 left to right for min mode and 1-8 for max mode and 1-7 for mix mode top to bottom. Row 7 is the *pop-in* section for mix mode. Rows 1-6 are the *major* section for mix mode. The mainframe will display this list of cells in any appropriate location on its screen. Max mode menus must be displayed as a single contiguous block. Mix mode menus must be displayed as 2 areas of contiguous blocks of cells (the *major* section and *pop-in* section blocks) with the pop-in section always displayed below the major section. These two blocks do not need to be adjacent.

If a text string for a cell contains more than 5 characters, the remaining characters will not be displayed. Text will be left-justified within a cell. If a text string is empty, no text will be displayed on the corresponding line.

There are four predefined font tokens: **normal**, **touch**, **selected** and **atten**. The **normal** font token specifies the mainframe's normal text font. The **touch** font token specifies the font that the mainframe uses to identify selectable touch areas. The **selected** font token specifies the font that the mainframe uses to indicate a selected touch area. The **atten** font token is the font used by the mainframe to attract the user's attention for indicating an error or other urgent status condition. Font selectors for text lines will be specified using predefined tokens.

Cells are grouped together into touch areas. Each touch area has a unique *area_id* for that menu. The only restriction on the size of a touch area is that it be less than the size of the menu. Touch areas must be rectangular and they may not overlap. The mainframe will report screen touches only for cells that are defined as part of a touch area. The mainframe will report the touch *area_id* not the cell x, y coordinates. The mainframe will report only one touch per touch area (even if several cells in the area are touched) until no areas are touched. The mainframe will not report touches for cells that are not included in a touch area definition. The mainframe will produce the audible sound used by the mainframe to identify mainframe menu touches for plug-in extended menus only when a valid touch area is touched. Cells specified with a touch *area_id* of 0 are not touchable.

As information for each cell is uploaded in a **menu change** message, the mainframe will clear the cell of any previous text and fonts so the new display will show only new text and fonts for that cell. Cells that are not specified in the **menu def_smart** message are left blank or not displayed as part of the menu. Cells that are not specified in the **menu change** message remain as before (either blank or with text).

Areas may be deleted by a **menu change** message by defining the size of an area as zero and defining all cells of that area to belong to the 0 area. New touch areas may be created by the **menu change** message but only from rectangular groups of cells from the untouchable (0) area.

Smart plug-ins will use the following cell specification (*def_spec*) format for defining cells in a **menu def_smart** message:

{**area_def** area_id size {**cell_def** xloc yloc text_type [f1_font text1] [**DLT** f2_font text2]}...}...
area_def: token - specifies beginning of area definition
area_id: us - specifies the touch area
size: pb - specifies size of touch area
cell_def: token - specifies the start of a cell definition
xloc: us - specifies the cell column
yloc: us - specifies the cell row
text_type: token - specifies the type of text that follows
f1_font: token - specifies the font for text1
text1: text string - first text
DLT: token - delimiter for text strings
f2_font: token - specifies the font for text2
text2: text string - second text

The plug-in may send as many area and cell definitions in a menu definition command as are available. The **area_def** token defines the beginning of an area definition. All the following cell definitions up to the next **area_def** token or the end of the message apply to the the area specified by the *area_id* parameter. These cells must fall within the menu area specified by the size parameter. The first cell in the list defines the upper left hand corner position of the area. A **cell_def** must be included for each cell in the area even if that cell has no text.

The *area_id* parameter specifies the touch area to which the cell is assigned. Cells in an area with an *area_id* of 0 do not belong to any touch area and are not touchable.

The *size* parameter specifies the height and width of the area in cells. It is a packed binary parameter and has the following format:

```
b7  b6  b5  b4  b3  b2  b1  b0
├---height---┤  ├---width----┤
```

To delete a touch area, a plug-in must do two things. It must send the *area_id* to be deleted with a *size* of zero width and height. The plug-in must also assign all the cells previously assigned to that *area_id* to the 0 area.

Each cell definition is started by a **cell_def** token and terminates with either the next **cell_def** token or the end of the message.

The *xloc* and *yloc* parameters specify the location of the cell being defined relative to the menu with row 1 and column 1 in the upper left hand corner. They are unsigned short values.

The *text_type* token may be either **text** or **descript**. The **text** token specifies that *text1* is always put on the first line of the cell and *text2* is always put on the second line. The **descript** token specifies that *text1* is the descriptor string and *text2* is the status string. The mainframe will put these texts into the status and descriptor lines as appropriate for that mainframe. The **DLT** token is used to delimit the text for the second line because the first text string is optional.

The *f1_font* parameter specifies the font style for all characters in *text1*. It may be **normal**, **touch**, **selected** or **atten**. The *f2_font* parameter specifies the font style for all characters in *text2*. It may be **normal**, **touch**, **selected** or **atten**.

The *text1* and *text2* strings consist of all the displayable ASCII characters (20_{16} - $7E_{16}$) plus the characters defined in Appendix B. The function normally associated with control characters (CR, LF, HT, FF, BEL, BS etc.) will not be performed. Text strings may include the escape sequences defined in Appendix B of the *Command Reference Specifications* document. The text is limited to 5 displayed characters.

If both text parameters are absent, the cell is defined as a blank cell. This type of cell is useful for defining menu boundaries for mainframes that size menus without needing to specify text to appear in the cell. The mainframe will treat a blank cell as a defined cell when determining a menu's size.

2.2.4 Menu Examples

The following examples show a sample message exchange between a smart plug-in and the mainframe used to perform a menu operation. SRQ messages shown with an asterisk (*) indicate a transport level SRQ sent to the mainframe and do not indicate an application level message.

```
mainframe                          message                        plug-in

mf saves SRQ request          <---SRQ*---                    user presses button
mf acts on SRQ request      ---SRQ query--->
                         <---menu request mix---             pi sends menu req
mf readies disp          ---menu status ready--->
mf acts on SRQ request      ---SRQ query--->
                        <---menu def_smart more---           first menu def
mf accepts menu def ---menu def_smart ack--->
                        <---menu def_smart more---           next menu def
mf accepts menu def ---menu def_smart ack--->
                        <---menu def_smart last---           last menu def
mf accepts menu def ---menu def_smart ack--->
mf displays menu     ---menu status formatted--->
mf acts on SRQ request      ---SRQ query--->
mf clears SRQ request  <---SRQ no_report---                  pi has nothing
user touches menu          ---menu touch--->
                         <---menu change last---             pi changes menu
mf accepts menu ch      ---menu change ack--->
mf displays menu     ---menu status formatted--->
mf acts on SRQ request      ---SRQ query--->
mf clears SRQ request  <---SRQ no_report---                  pi has nothing
user touches menu          ---menu touch--->
                         <---menu request max---             pi req max menu
mf deletes old menu  ---menu status ready--->
                              <---SRQ*---                    pi sends SRQ
mf acts on SRQ request      ---SRQ query--->
                        <---menu def_smart more---           pi sends first def
mf accepts menu def ---menu def_smart ack--->
                        <---menu def_smart more---           next menu def
mf accepts menu def ---menu def_smart ack--->
                        <---menu def_smart last---           last menu def
mf accepts menu def ---menu def_smart ack--->
mf displays menu     ---menu status formatted--->
mf acts on SRQ request      ---SRQ query--->
mf clears SRQ request  <---SRQ no_report---                  plug-in has nothing
user touches menu          ---menu touch--->
                            <---menu delete---               plug-in is done
mf removes menu       ---menu status removed--->
```

The above example shows a typical (but shortened) menu operation between a smart plug-in and the mainframe. To start the process, the plug-in responds to a user pressing one of its front panel buttons by sending an SRQ transport packet to the mainframe. The mainframe will save the SRQ state for that plug-in then send an **SRQ query** message to the plug-in in response.

The plug-in sends the **menu request mix** message in response to the **SRQ query** message request a mixed mode menu. The mainframe sends the **menu status ready** message when it is ready for menu formatting. Since the SRQ state is still set in the mainframe, it will send another **SRQ query** message to the plug-in. The plug-in will send the menu definition in three **menu def_smart** messages using the long message protocol as shown. When the mainframe has received and acknowledged the last **menu def_smart** message and formatted the display, it will send the **menu status formatted** message to indicate its status.

Because the mainframe's SRQ state is still set for that plug-in, it will send another **SRQ query** message. The plug-in will send the **SRQ no_report** message because it has nothing to say. No more messages will be sent until the user takes some action.

When the user touches an area in the plug-in menu, the mainframe will send the plug-in the **menu touch** message as shown. The plug-in has several options for a response. In the first response shown in the example, the plug-in takes no menu action and sends the **menu status ready** message. In the second case, the plug-in changes the existing menu by sending a single **menu change** message (only a few cells were changed, hence, the short message). The mainframe responds with the **menu status formatted** message when it has updated the menu display. In the third case, the plug-in wants to display a new menu. It sends the **menu request max** message to change to a max mode menu. The mainframe will delete the existing menu display and prepare to display a max mode menu then send the **menu status ready** message to the plug-in. The plug-in sends three **menu def_smart** messages to define the contents of the max mode menu. The mainframe sends the **menu status formatted** message when it has completed formatting the menu.

In the last case shown for a response to a menu touch, the plug-in sends the **menu delete** message to cause the mainframe to delete the menu. The mainframe will send the **menu status removed** message to indicate removal of the menu and the termination of the menu session.

Note that during the time the plug-in has a menu displayed, the message traffic between the mainframe and plug-in is not restricted to menu messages. There may be knob messages, for example, interspersed with the menu messages. The following example shows such a case:

```
mainframe                          message                    plug-in

mf saves SRQ request         <---SRQ*---                   user presses button
mf acts on SRQ request    ---SRQ query--->
                        <---menu request mix---           pi sends menu req
mf readies disp         ---menu status ready--->
mf acts on SRQ request    ---SRQ query--->
                       <---menu def_smart more---         first menu def
mf accepts menu def ---menu def_smart ack--->
                       <---menu def_smart more---         next menu def
mf accepts menu def ---menu def_smart ack--->
                       <---menu def_smart last---         last menu def
mf accepts menu def ---menu def_smart ack--->
mf displays menu     ---menu status formatted--->
mf acts on SRQ request    ---SRQ query--->
mf clears SRQ request  <---SRQ no_report---              pi has nothing
user touches menu         ---menu touch--->
                        <---menu status ready---          pi accepts touch
user turns knob             ---knob change--->
                            <---knob update---            pi responds
mf updates display    ---knob status ready--->
mf deletes menu        ---menu status removed--->
                       <---menu status removed---         plug-in responds
```

In this example, the knob turn is detected by the mainframe which causes a knob transaction between the mainframe and the plug-in. Subsequently, the mainframe (for one reason or another) deletes the plug-in's menu. It notifies the plug-in of this action with the **menu status removed** message. The plug-in responds with the **same** message. This terminates the menu session.

2.2.5 Status Message Display

Mainframes supply a display facility that allows smart plug-ins to supply and update a status message that gives information about the plug-in's operation and status. This message will be displayed by the mainframe in a location that will not interfere with the waveform display. This status information may displace other mainframe status information if necessary. This function does not support touches. All text in this area will be displayed in the mainframe's normal font. This function will be activated by user via a plug-in front panel button.

The plug-in will update this status area as necessary to inform the user of changes in its operation. This status may be reassigned by the mainframe when the mainframe needs that area to display other status (usually requested by the user). The mainframe will notify the plug-in when its status display is removed. Rotating the control knobs will not cause the status area to be removed.

This area is defined as 2 lines of 50 characters of text each. The following messages are defined to support this function:

status_disp set [DLT1 text1] [DLT2 text2]
status_disp set: message tokens
DLT1: token - delimiter for line 1 text
text1: string - text for line 1
DLT2: token - delimiter for line 2 text
text2: string - text for line 2

This message is sent by a smart plug-in to define the text to be displayed in the status display area. The **DLT1** token delimits the first line of text. The *text1* parameter is the text for line 1 and may be up to 50 characters. The **DLT2** token delimits the second line of text. The *text2* parameter is the text for line 2 and may be up to 50 characters. Either **DLT1** and *text1* or **DLT2** and *text2* or both may be sent. At least one line of text must be included in this message. The plug-in must send the entire message to be displayed each time the display is to be updated. The plug-in may not edit existing displayed text.

status_disp status status
status_disp status: message tokens
status: token - specifies status display status

This message is sent by the mainframe in response to a **status_disp set** message. The **ready** *status* token indicates the mainframe has formatted and displayed the status message. The mainframe will send the **status_disp status** message with the **removed** *status* token whenever it removes the plug-in's status display. The plug-in will discontinue updating the status area and send the **status_disp status** message with the **removed** *status* token as the response.

2.3 Knobs and Keypad

Mainframes will provide two control knobs and a keypad for use by smart plug-ins.

A plug-in requests knob assignments by sending a knob request command to the mainframe. The mainframe will reply with a status message indicating the knob assignment was completed. The mainframe will save previous assignments to the knobs before it sends the knob status message. Both the knobs and the keypad are always assigned as a group to a plug-in. The plug-in may individually assign functions to each knob. If a knob is not assigned, its display is blank and it controls no function. Recommended practice is that plug-ins will not request the knob function without assigning a function to both knobs (possibly the same function to both).

When a keypad entry is made, the mainframe will indicate to the plug-in the knob to which the keypad and, hence, the entry is assigned. The mainframe will also provide selection means for changing the assignment of the keypad and a means for activating the keypad.

The plug-in will send a **knob request** message to the mainframe to initiate knob assignment.When it receives the **knob status ready** message, the plug-in will send a **knob def** message to define the parameters for both knobs. This message will include information for the knob status displays. The mainframe will label the knobs and return the **knob status formatted** message to the plug-in when it is finished. The plug-in may send new knob display messages without needing to request knob control until the knobs are reassigned.

Whenever the mainframe detects a change in a knob position it will send the **knob change** message to the plug-in indicating which knob was changed, the number of detents it was changed and the direction the knob was turned. The plug-in will respond with either a **knob update** or **knob def** message.

The mainframe must provide a means for selecting and assigning the keypad to either knob. The knobs and keypad may be reassigned at any time by the mainframe. Knob control may also be released by the plug-in. The mainframe will send a **knob status removed** message to the plug-in whenever its knob assignment is removed. When the knobs are released by the plug-in, the knob display and control will revert to the function that was previously controlled.

2.3.1 Knob Messages

knob request
knob request: message tokens

The plug-in uses this command to request assignment of the control knobs. The mainframe will respond with the **knob status** message.

knob status status
knob status: message tokens
status: token - specifies the status of the knobs

The mainframe uses this status message to return knob status to the plug-in. There are three knob status tokens: **ready**, **formatted** and **removed**. The **ready** token is used to indicate assignment of the knobs. The **formatted** token indicates completion of the knob display as requested by a **knob def** command. The **removed** token is used by the mainframe to indicate that the knobs and keypad have been reassigned. This might be in response to a plug-in command or some other mainframe action.

knob def which title min max value units type {control resol}...
[which title min max value units type {control resol}...]
knob def: message tokens
which: token - specifies the knob
title: string - the title for the knob
min: float - the minimum legal value
max: float - the maximum legal value
value: float - the present setting of the function
units: string - indicates knob units
type: token - indicates scale type
control: token - indicates type of control
resol: float - indicates knob resolution

This command is used by the plug-in to label the knobs. This message may specify the labelling for one or both knobs.

The *which* parameter specifies to which knob the definition applies using two tokens: **knob1** and **knob2**. The *title* parameter is the label that is associated with the knob. It is limited to 15 characters. The *min* and *max* parameters are used to display the limits of the control range. The *value* parameter specifies the present setting of the control. The plug-in will check for out of range conditions. If a value is out of range, the plug-in will modify the knob display to show the nearest legal value and set the function to that value. The *units* parameter specifies the units appropriate for the knob. It is limited to 10 characters.

The *type* parameter specifies the type of knob control. The **linear** token selects linear scaling. The **dbm_2** token selects log base 2 scaling. The **dbm_10** token selects log base 10 scaling. The **step_125** token selects 1-2-5 sequence scaling. The *control* and *resol* parameter pairs specify the type of control provided for the function and the resolution for that type. There are three *control* types: **coarse**, **medium** and **fine**. Each has a *resol* parameter associated with it that defines the resolution of that control. The *resol* parameter associated with the **fine** *control* type indicates the best achievable resolution of the control and thus may be used to determine how many significant digits to display for the knob function. The mainframe will use the *resol* parameters to round or truncate (according to the mainframe's human interface specification) values entered on the keypad before they are sent to the plug-in. The **fine** *control* type and *res* parameters are required to define the function resolution. The **coarse** and **medium** *control* types are optional. The mainframe is not required to support the **medium** *control* type. If a mainframe does not have medium control capability, it will ignore the **medium** *control* token.

If two definitions are sent for the same knob in the same message, the second one applies.

knob update which value
knob update: message tokens
which: token - indicates which knob
value: float - indicates new value

This message is sent by a smart plug-in to change the present value of a knob display without needing to redefine all the parameters of the knob. The *which* parameter specifies to which knob the message applies and is either **knob1** or **knob2**. The *value* parameter specifies the value to be displayed.

knob change which control value
knob change: message tokens
which: token - specifies which knob
control: token - specifies the control type
value: short - specifies the direction and number of detents

This message is used by the mainframe to indicate knob changes to the plug-in. The *which* token will be either **knob1** or **knob2** to indicate which knob was changed. The *control* parameter specifies which type of control was selected by the user. It is either **coarse**, **medium** or **fine**. The *value* parameter is a short integer indicating the direction and amount of change. The absolute value will indicate the number of detents the knob was rotated since the last **knob change** message. The sign indicates the direction - positive sign indicates clockwise rotation, negative sign indicates counter-clockwise rotation.

knob keypad which value
knob keypad: message tokens
which: token - indicates which knob
value: float - the value entered by the user

This status message is used by the mainframe to send a keypad value to the plug-in when the keypad is assigned to a plug-in. The mainframe will send this message after the user has completed an entry. The *which* parameter indicates to which knob the keypad is assigned and is either **knob1** or **knob2**. The *value* parameter is the value entered by the user. The mainframe will display entry-in-progress information while numbers are being entered. The mainframe will handle any editing that might be done using keypad keys. If an entry is canceled, the mainframe will not send a **knob keypad** message to the plug-in.

The plug-in will respond to the **knob change** and **knob keypad** messages with either the **knob status ready** message, the **knob update** message or the **knob delete** message.

knob delete
knob delete: message tokens

The plug-in uses this command request removal of the knobs from the plug-in. The mainframe will remove the knobs and send the **knob status removed** message.

2.3.2 Knob Examples

The following examples show message exchanges between a smart plug-in and a mainframe that are used to assign and control the knobs. SRQ messages shown with and asterisk (*) indicate a transport level SRQ sent to the mainframe and do not indicate an application level message.

```
mainframe                          message                  plug-in

mf saves SRQ state           <---SRQ*---                user pushes button
mf acts on SRQ state       ---SRQ query--->
                         <---knob request---            pi requests knobs
mf assigns knobs       ---knob status ready--->
mf acts on SRQ state       ---SRQ query--->
                           <---knob def---              pi defines knobs
mf formats display ---knob status formatted--->
mf acts on SRQ state       ---SRQ query--->
                          <---SRQ no_report             pi has nothing
user turns knob           ---knob change--->
                          <---knob update---            pi updates knobs
mf formats display ---knob status formatted--->
user turns knob           ---knob change--->
                          <---knob update---            pi updates knobs
mf formats display ---knob status formatted--->
mf saves SRQ state           <---SRQ*---                user pushes button
mf acts on SRQ state       ---SRQ query--->
                          <---knob delete---            pi done with knobs
mf removes knobs      ---knob status removed--->
mf acts on SRQ state       ---SRQ query--->
                          <---SRQ no_report             pi has nothing
```

In this example, the knobs are assigned as the result of a user touch of a front panel button. The plug-in requests the knob function using the knob request message then defines the knobs using the knob def message. Two knob turns by the user are reported by the mainframe to the plug-in which sends changes to the knob display back to the mainframe. The plug-in requests the knobs to be removed after another user touch of the plug-in front panel.

Another means of termination of the knob function is shown below:

```
mainframe                          message                  plug-in

mf saves SRQ state           <---SRQ*---                user pushes button
mf acts on SRQ state       ---SRQ query--->
                         <---knob request---            pi requests knobs
mf assigns knobs       ---knob status ready--->
mf acts on SRQ state       ---SRQ query--->
                           <---knob def---              pi defines knobs
mf formats display ---knob status formatted--->
mf acts on SRQ state       ---SRQ query--->
                          <---SRQ no_report             pi has nothing
user turns knob           ---knob change--->
                          <---knob update---            pi updates knobs
mf formats display ---knob status formatted--->
user turns knob           ---knob change--->
                          <---knob update---            pi updates knobs
mf formats display ---knob status formatted--->
other mf operation  ---knob status removed--->
                   <---knob status removed---           pi acks knob status
```

In this example, the user selects an alternative mainframe operation that causes the knobs to be reassigned. The mainframe will inform the plug-in of this action. The plug-in responds with the **knob status removed** message acknowledging the action.

2.4 Cursors

Mainframes will provide measurement display means for use by plug-ins called cursors. There are four cursor modes: **horizontal**, **vertical**, **paired** and **split**.

The **horizontal** and **vertical** modes provide a visual display of values using lines that extend the length or height of the display area. In these modes, the cursors are associated with the axes of the selected trace, not the trace itself. Thus, the cursors may be positioned independently from the trace and do not need to intersect the trace. Cursor positioning is controlled from plug-ins by selecting axis values. All mainframes will provide the **horizontal** and **vertical** cursor modes. The **horizontal** cursor mode measures horizontal distance using vertical lines. The **vertical** cursor mode measures vertical distance using horizontal lines.

The **paired** and **split** modes provide a visual indication of selected waveform points. A marker is provided that is always associated with a waveform point. Cursor positioning is controlled from plug-ins by selecting waveform point numbers. Only digitizing mainframes provide the **paired** and **split** modes.

Cursors are selectable in the **mix** and **min** menu modes but not the **max** mode. Menus are not required to use the cursor function. The knob and cursor functions are mutually exclusive. Since the mainframe uses the knobs to control the cursors, the plug-in may not assign the knobs to another function when using the cursors. If a plug-in has requested the knob function then subsequently requests the cursor function, the mainframe will notify the plug-in that the knobs have been reassigned. If a plug-in has requested the cursor function then subsequently requests the knob function, the mainframe will notify the plug-in that the cursors have been removed.

Operation

The following messages are used to control the cursor function.

cursor set mode [pos1 pos2 id1 id2 scale offset resol units [wvfm_id]]
cursor set: message tokens
mode: token - indicates cursor type
pos1: float - position of cursor 1
pos2: float - position of cursor 2
id1: string - id for cursor 1
id2: string - id for cursor 2
scale: float - scale factor for cursor values
offset: float - offset value for cursors
resol: float - resolution of cursor function
units: string - cursor function units
wvfm_id: integer - specifies waveform number

This message is sent by a plug-in to request the use of the cursor function. The mainframe will respond with **a cursor status** **message** indicating the actual status of the cursors.

The *mode* parameter selects the cursor mode - **paired**, **split**, **horizontal**, **vertical** or **off**. The **paired** mode selects a pair of cursors to be displayed on a single waveform. The **split** mode selects a single cursor to be displayed on each of two different waveforms. In this mode, the *wvfm_id* parameter is included and specifies the waveform identification of the waveform on which to place cursor 2. The *wvfm_id* parameter is included only when the *mode* is **split**. The **horizontal** and **vertical** modes select horizontal or vertical cursors, respectively. When the **off** *mode* is selected, the cursor display is turned off. The *pos1* and *pos2* parameters are not used when **off** is specified.

The *pos1* and *pos2* parameters specify the initial position of cursor 1 and cursor 2 respectively. The values of these parameters are waveform point numbers for the **paired** and **split** modes and divisions from center screen for the **horizontal** and **vertical** modes.

The *id1* and *id2* parameters are strings that the plug-in uses to identify the function being measured by the cursors. These strings will be displayed just to the left of each scaled cursor value. The *id1* string will be placed with cursor 1. The *id2* string will be placed with cursor 2.

The *scale* parameter specifies a scaling factor the mainframe will use to convert divisions to the plug-in units being measured by the cursors. The mainframe will multiply the cursor position in divisions by the *scale* parameter to calculate the proper display value.

The *offset* paramter specifies the offset value of the cursor in scaled units. The mainframe will add the *offset* value to the scaled cursor value before display. Cursor display can be calculated as follows:

disp_value = (*scale* * cursor_pos) + *offset*

Where disp_value is the displayed value in plug-in cursor units, cursor_pos is the cursor position in divisions minus position and *scale* and *offset* are the values from the **cursor set** message.

The *resol* parameter specifies the resolution of the units being measured. The mainframe will use this value with the scaled cursor value to determine how may digits are meaningful for display. The formula is as follows:

$$\text{num\_digits} = \text{int}(\log_{10}(\frac{\text{present value}}{\text{resolution}})) + 1$$

The int function truncates the value toward 0.

The *units* parameter is a string defining the plug-in units being measured. The mainframe will append this string to the knob display. The knob display will be formatted as follows:

idn[value*scale+offset]units

Where idn is the *id1* or *id2* string for that cursor, value is the cursor value in divisions, scale is the *scale* value, offset is the *offset* value and units is the *units* string.

If the cursors are already displayed, this message is used to change the position or type of cursors or to change the knob display.

The plug-in will use the **mf_display set** message to select the waveform on which to place the cursors.

The mainframe will send a **cursor status** message to the plug-in each time the knobs or keypad change the position of the cursors. The mainframe will control the size of cursor steps for each knob detent.

cursor limits min1 max1 min2 max2
cursor limits: message tokens
min1: float - indicates the minimum value allowed for cursor 1
max1: float - indicates the maximum value allowed for cursor 1
min2: float - indicates the minimum value allowed for cursor 2
max2: float - indicates the maximum value allowed for cursor 2

This message is used by plug-ins to specify the limits allowed by the cursors. The *min1* and *max1* parameters specify the minimum and maximum limits for cursor 1. The *min2* and *max2* parameters specify the minimum and maximum limits for cursor 2. The mainframe will not allow the cursors to be positioned outside the range specified by the plug-in. The mainframe will send the **cursor status ready** message in response.

cursor status status [pos1 pos2] [value1 value2]
cursor status: message tokens
status: token - indicates cursor status
pos1: float - position of cursor 1
pos2: float - position of cursor 2
value1: float - value of waveform point at cursor 1
values: float - value of waveform point at cursor 2

This message is sent by the mainframe to indicate cursor status. It is sent each time the position or status of the cursors is changed. The *status* parameter indicates the status of the cursors: **ready**, **na** or **removed**. The **ready** token indicates that the cursors have been set as requested by the plug-in and is sent in response either to a **cursor set** or **cursor limits** message or when there is a change in the control knobs. The **na** token is sent by a realtime only mainframe in response to a **cursor set** command when the **paired** or **split** cursor *mode* is requested. The **removed** token indicates the cursors are no longer assigned to the plug-in. The *pos1* and *pos2* parameters indicate the actual position of cursor 1 and cursor 2 and are only sent with the **ready** *status* token. These positions might not correspond with the values requested by the plug-in. These are the values that are displayed by the mainframe. These values will be in divisions from center screen for **horizontal** and **vertical** modes and vertical or horizontal units for **paired** and **split** modes. The *value1* and *value2* parameters indicate the value of the waveform point on which cursor 1 or cursor 2 resides, respectively. These values are included only if the cursor mode is **paired** or **split**. They are in waveform units.

2.5 Front Panel Settings

The mainframe will request a smart plug-in to store its existing front panel setup when requested by the user. The smart plug-in will access its own stored settings buffer to save the status necessary to restore the current front panel setting. The mainframe will also request smart plug-ins to restore a previously saved setting. The smart plug-in will access its stored settings buffer to return it to a previous front panel setting. The plug-in must be able to store up to 10 front panel settings. These settings will be stored in plug-in non-volatile memory.

The following messages are used to manage front panel settings storage and recall:

setting store number
setting store: message tokens
number: us - selects setting number

This command is sent to smart plug-ins to cause them to save their existing front panel settings. The *number* parameter is used to identify a particular setting. The smart plug-in will access its stored settings buffer to save its front panel settings and respond with the **setting status ready** message when it has completed the task.

setting recall number
setting recall: message tokens
number: us - selects setting number

This message is sent to smart plug-ins to cause them to restore their front panel settings to a previously saved value. The *number* parameter indicates which previously stored setting is desired. The smart plug-in will access its stored settings buffer to restore its settings to the previously saved values and respond with the **setting status ready** message when it has completed the task.

setting status status
setting status: message tokens
status: token - indicates plug-in setting status

The plug-in will send this message when it has completed the setting operation requested by the mainframe in response to a **setting store** or **setting recall** message. The plug-in will send the **ready** *status* token when it has completed the requested setting operation. The plug-in will send the **na** *status* token if the setting requested by the mainframe is not valid or not available.

2.6 Interactions With the Mainframe and Other Plug-Ins

Smart Plug-Ins may request system configuration information from the mainframe. This will tell the smart plug-in what other plug-ins are in the system. If they are generic plug-ins, all generic plug-in function values and parameters can be requested. If they are smart plug-ins, their plug-in type can be requested.

2.6.1 Generic Plug-Ins

Smart plug-ins will not access generic plug-in functions by communicating directly with the generic plug-in. Generic plug-in functions will be included in the set of functions provided by the mainframe that smart plug-ins may access. The five basic functions - gain, offset, input coupling, input impedance and bandwidth limit will be controlled via commands sent to the mainframe. The generic plug-in's display and trigger functions will be controlled via the smart plug-in's access to the mainframe new trace and trigger trace description functions.

The mainframe will provide smart plug-ins with information about generic plug-ins display, trigger and auxiliary channel definitions. Smart plug-ins may request this information using the **sys_config query** command.

The following command is provided to allow smart plug-ins to control generic plug-ins that are installed:

generic command compartment length command params
generic command: message tokens
compartment: char - plug-in compartment - left, center or right
length: us - specifies the number of following bytes
command: message tokens (2) - valid command for generic plug-ins
params: special - value of new setting

The *compartment* parameter indicates the plug-in compartment for which the command is intended. The *length* parameter specifies the number of bytes following the *length* parameter that compose the command to the plug-in. The *command* parameter is a message token pair that specifies any generic plug-in command defined in section **1.0 Generic Plug-ins**. The *params* parameters are the parameters that are normally associated with the specific generic plug-in command specified by the *command* parameter. See the definition of these commands in the generic plug-in section for more information.

The status messages that are returned from these commands will be used by the mainframe to update its status and will also be sent to the requesting smart plug-in. If an error is detected, the error message will also be sent to the smart plug-in. The smart plug-in is required to handle error conditions as they arise.

2.6.2 Smart Plug-Ins

Smart plug-ins may communicate with each other using the messages defined in this section. Smart plug-ins will not communicate in any way except through the interface defined in this standard.

There is a single message that is used to route information between smart plug-ins. Smart plug-ins will first determine the system configuration before attempting to send messages. Messages cannot be sent to generic plug-ins.

smart message dest src length message
smart message: message tokens
dest: char - indicates message destination
src: char - indicates message source
length: us - specifies the number of following bytes
message: special - the message data

This message is used by smart plug-ins to communicate with each other. The *dest* parameter indicates the destination compartment of the message. The mainframe will use this information to send the message to the correct plug-in compartment. The *src* parameter indicates the compartment name of the message source. It is set by the sending plug-in, not the mainframe. The receiving plug-in will use this information to return messages to the sending plug-in. The *dest* and *src* parameters can be determined using the **sys_config query** message defined in the next section. The *length* parameter specifies the number of bytes following the *length* parameter. The *message* parameter is the message data to be sent between the plug-ins. The format is specified by the plug-in designers and is not interpreted by the mainframe and is not part of this specification. Each **smart message** message requires a response from the receiving smart plug-in.

2.6.3 System

Smart plug-ins may get system configuration information by sending the **sys_config query** message to the mainframe:

sys_config query
sys_config query: message tokens

The mainframe will respond with the following status message giving the system configuration:

sys_config status compartment {report_compart name uid pi_type [disp_channels trig_channels aux_channels] [**minus_coupl**] [**lower_bandw**] [**diff_offset**] [**no_invert**] [**trig_view**] [**trig_out**] [**dig** channels] [**get**]}...
sys_config status: message tokens
compartment: char - indicates occupied plug-in compartment
report_compart: char - indicates reporting plug-in compartment
name: string - Pioneer nomenclature, 16 char limit
uid: string - unit identification, 16 char limit
pi_type: token - indicates plug-in type
disp_channels: us - number of display channels
trig_channels: us - number of trigger channels
aux_channels: us - number of auxiliary trigger channels
minus_coupl: token - indicates minus coupling capability
lower_bandw: token - indicates lower bandwidth capability
diff_offset: token - indicates differential offset capability
no_invert: token - indicates no invertsion capability
trig_view: token - indicates trigger view capability
trig_out: token - indicates trigger output capability
dig: token - indicates digitizing capability

channels: us - indicates the number of digitized channels
get: token - indicates group execute trigger capability

This message gives information about each contents of each plug-in compartment in the system. The *compartment* parameter specifies the name of the compartment in which the smart plug-in is installed. It is a single character the mainframe uses to identify that plug-in compartment: L, C, or R. The smart plug-in will use this response to form external interface command syntax for its parse tables by prepending this character and the underscore to each external command.

The *report_compart* parameter specifies the compartment for which the following plug-in characteristics are being reported. It is a single character the mainframe uses to identify that plug-in compartment: L, C, or R.

The *pi_type* parameter indicates the plug-in type: **generic**, **both**, **smart**, **old** or **empty**. Smart plug-ins may use this parameter to form external bus commands or for formatting menus for display to be consistent with mainframe usage. The *name* parameter is the Pioneer nomenclature name for the plug-in in the specified compartment. The *uid* parameter is the unit identifier reported by the plug-in. See section **3.0 Common Messages**. The *disp_channels*, *trig_channels* and *aux_channels* parameters specify the number of display, trigger and auxiliary trigger output channels, respectively. The **minus_coupl** token indicates a generic plug-in that has differential inputs. The **lower_bandw** token indicates a generic plug-in that supports the lower bandwidth function. The **diff_offset** token indicates a generic plug-in that supports the differential offset function.

The **no_invert** token indicates a generic plug-in that does not have display or trigger signal inversion capability. This affects the operation of the **display set** and **trigger set** messages defined in section **1.16 New Trace**.

The **trig_view** token indicates the plug-in supports the trigger view function. The **trig_out** token indicates the plug-in is a triggering plug-in and provides a single output trigger on both the display and trigger outputs. The **dig** token indicates the plug-in is a digitizing plug-in with *channels* number of digitized inputs. The **get** token indicates a plug-in with group execute trigger capability.

See the **plugin_config status** message in section **3.1** for more information on the meaning of these parameters. This message may contain other single tokens that define future plug-in capabilities. Smart plug-ins must be able to ignore capability tokens they do not understand.

mf_display query
mf_display query: message tokens

This command is sent by smart plug-ins to determine display status. The mainframe will send the following status message as the response:

mf_display status {status wvfm_id source_desc}...
mf_display status: message tokens
status: token - indicates display status
wvfm_id: us - trace number
source_desc: string - source description

The *status* parameter identifies the status of the indicated trace. There are five status tokens. The **selected_live** *status* token indicates a selected waveform that is a live waveform. The **selected_stored** *status* token indicates a selected waveform that is a stored waveform. The **live** *status* token indicates a non-selected live waveform. The **stored** *status* token indicates a non-selected stored waveform. The **none** *status* token indicates no waveforms are displayed. The *wvfm_id* parameter specifies the trace number from 1 to 8. The *source_desc* is a string that defines the source input description for the specified trace. It has the same format as the TRACE<ui> DESCRIPTION link argument defined in the *Command Reference Specifications 11K Series Family of Products* document. The plug-in will be able to scan this string to obtain useful information about the input channels that are included in the source description.

mf_display set wvfm_id
mf_display set: message tokens
wvfm_id: us - indicates selected trace

This message is sent by a smart plug-in to select a displayed trace from the list received in a **mf_display status** message. The *wvfm_id* parameter indicates the trace to be selected and must be a *wvfm_id* value reported by the **mf_display status** message. The mainframe will select the specified trace and return the **mf_display status** message to the plug-in. Valid *wvfm_id* values range from 1 to 8.

mf_trigger status
mf_trigger status: message tokens

This message is sent to a smart plug-in that has identified itself as having a trigger output (using the **plugin_config status** message) when that plug-in's trigger output is selected for a sweep trigger or other function. This will allow the plug-in to use the following message to set trigger parameters to appropriate values whenever it is selected as the trigger source. The plug-in will send the **mf_trigger status** message if it does not wish to change the mainframe's trigger parameters.

mf_trigger set level slope coupling
mf_trigger set: message tokens
level: float - selects trigger level
slope: token - selects trigger slope
coupling: token - selects trigger coupling

This message is sent to a mainframe by a smart plug-in with a trigger output to set the trigger parameters for all sweeps or other functions for which the plug-in's trigger output is the trigger source. The *level* parameter specifies the setting of the trigger level control in divisions from center screen. The *slope* parameter selects the slope of the trigger control and may be either **plus** or **minus**. The *coupling* parameter selects the trigger input coupling and may be either **AC** or **DC**.

mainframe message message
mainframe message: message tokens
message: see text

This message is sent by a smart plug-in to access internal mainframe commands. These commands are mainframe specific. The mainframe will send the **mainframe message** with appropriate message contents as a response. The level of functionality supported by the mainframe is reported in the **mf_id status** message by the *level* parameter. The contents of the **mainframe message** message are not defined in this interface specification.

2.7 Updates

The updates function allows a smart plug-in to monitor the changes made in generic plug-ins. The mainframe will handle update reports to smart plug-ins for generic plug-ins.

Generic Plug-Ins

Smart plug-ins may request updates on changes in generic amplifier plug-in functions using the following message:

update request compartment mode
update request: message tokens
compartment: character - specifies the plug-in compartment
mode: token - specifies the mode

The *compartment* parameter is a character identified by the **sys_config status** message (see **3.0 Common Messages**). The *mode* parameter specifies whether this function is being enabled or disabled. There are two valid tokens: **on** and **off**. When the mainframe receives this command with the **on** mode token, it will enable updates from the specified compartment to the requesting smart plug-in. Thereafter, the mainframe will send an **update status** message to the requesting plug-in each time a change is made in any of the generic functions of the specified plug-in. When this message is received with the **off** mode token, the mainframe will not send any further **update status** messages for that compartment. Note that this message applies only to compartments with a generic plug-in. The smart plug-in must get initial generic plug-in status using the **generic command** message with function status requests.

The mainframe will send the status information to the plug-in requesting updates using the following message:

update status status [compartment function param]...
update status: message tokens
status: token - indicates update status
compartment: character - specifies the plug-in compartment
function: message tokens (2) - specifies the function
param: variable type - function parameters

The *status* parameter specifies the type of response. The **ready** *status* token is sent by the mainframe in response to an **update request** message from the plug-in. No other parameters are included when the status token is ready. The **status** *status* token is sent by the mainframe when it is sending generic plug-in status to the smart plug-in. In this case, the rest of the parameters will be included. The *compartment* parameter specifies the compartment for which the status information applies. It is one of the compartment characters identified by the **sys_config status** message (L,C,R). The *function* parameter specifies which function the mainframe is reporting change information for. It is one of the status message token pairs defined in section **1.0 Generic Amplifier Plug-Ins**. The *param* parameters are the parameters for the specified function as defined in section **1.0 Generic Amplifier Plug-Ins.**

Smart Plug-Ins

Smart plug-ins may request updates from other smart plug-ins by sending messages defined by the plug-ins using the **smart message** message. Mainframes do not support updates from smart plug-ins.

2.8 Error Handling

Smart plug-ins will use the error messages defined in section **3.2 Error Handling.** For each error detected, smart plug-ins will send the appropriate error message with a status byte and an ASCII message for display. The mainframe will display the message as specified. The mainframe will handle any external ASCII interface requirements for the type of error message sent by the plug-in.

2.9 ASCII Interface

This plug-in/mainframe software interface provides a method for allowing smart plug-ins to add ASCII external interface commands to the list of commands accepted by the mainframe. The intent of this function is to provide the user with a view of the instrument (mainframe and plug-ins) as a single device rather than requiring the user to address each component individually.

To provide this capability, smart plug-ins will upload a special set of tables that will be used by the mainframe's ASCII interface parser. These tables will include all the information necessary for the mainframe to recognize valid plug-in external commands and send semantic actions to the plug-in when those commands are received.

The following section outlines the functions that are necessary to support this facility. The next section defines the interface messages that will be used to transfer the tables and the semantic actions between the plug-ins and the mainframe. Subsequent sections describe Mainframe/Plug-In Interface commands that support functions that are specific to a particular type of ASCII interface.

2.9.1 Overview

Plug-In ASCII interface commands will be defined in a BNF format by the plug-in designer. This BNF will be fed to translators during development which will produce tables to be stored in the plug-in. These tables will be uploaded to the mainframe at powerup. The mainframe provides a scanner and a parser for interpreting ASCII interface commands. The mainframe's scanner and parser will use the tables uploaded by the plug-ins to translate plug-in commands received over the external interface to semantic actions. These semantic actions will be sent to the plug-in when plug-in commands are recognized.

There are two types of mainframe parsers and scanners, type **E** and type **R**. Because these two types use different table formats, there will be two translators in the design group and two sets of tables in the plug-in. The mainframe will ask for the appropriate table using the commands defined in the next section. Both mainframes will send the same semantic actions when plug-in commands are recognized. Detailed specifications for the translators, tables, scanners and parsers will be found in Appendix A. The following paragraphs give overviews of each.

Translators

Both translators will accept a specified format for describing smart plug-in ASCII interface commands. This format will be a context free LL(1) grammar in Backus-Naur Form (BNF). Terminals and semantic actions of the grammar will be explicitly defined preceding the grammar. Semantic actions will be embedded in the grammar at places determined by the plug-in designer.

Each translator will choose terminal and non-terminal token values as appropriate. The plug-in designer will choose semantic action token values. Each translator will produce a token map defining terminal token values to be used by the mainframe's scanner. Translator type **E** will produce a production list and an LL (1) parse table in addition to its token map. Translator type **R** will produce a production list and a link list in addition to its token map.

Tables

The token map will contain a mapping of the token names (ASCII strings) to the values assigned by the translator. The type **E** token map will also contain abbreviation information. The abbreviations are chosen by the plug-in designer and specified in the plug-in BNF file.

The production lists will contain tokenized representations of the productions of the plug-in grammar. The token values are those defined in the token map for the production list type (**E** or **R**).

The parse table produced by the type **E** translator is a mapping of terminals and nonterminals to productions. Because this matrix has many zero entries, sparse matrix reduction techniques will be used to reduce the size of this table as stored in the plug-in.

The link list produced by the type **R** translator will contain a list of productions and addresses to be used at power up time.

There are size limitations for each of these tables. See Appendix A for specifications.

Upload

At powerup, the mainframe will request the type of tables appropriate for that mainframe (**E** or **R**). The plug-in will upload the tables requested. **E** type tables will be loaded into a separate space reserved for plug-ins. **R** type tables will be linked with the mainframe's tables during the powerup process.

Lexical Analyzers

The lexical analyzers in the mainframes convert the stream of ASCII input characters to a stream of tokens using both the mainframe and plug-in token maps produced by the associated translator. The stream of tokens is passed to the parser under control of the parser.

Parsers

The mainframe parsers match the stream of tokens from the scanner with plug-in (or mainframe) productions and emit semantic actions when appropriate. If the token stream does not match a production, a syntax error is declared and a recovery process is begun.

2.9.2 Messages

The following interface messages are used to transfer the information necessary to support the functions described above.

external req_table ttype ftype
external req_table: message tokens
ttype: token - specifies table type
ftype: token - specifies format type

This command is sent by the mainframe to request plug-in table information. There are two table types, **E** and **R** for the *ttype* parameter. The *ftype* parameter specifies the bus format. The only token presently defined is **C_F** (for Codes and Formats). The plug-in will respond with the **external table_data** message.

external table_data lmpb [ttype ftype data]
external table_data: message tokens
lmpb: token - long message protocol
ttype: token - specifies table type
ftype: token - specifies format type
data: special - table information

This message transfers plug-in table information to the mainframe. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter. There are two table types, **E** and **R** for the *ttype* parameter. A plug-in will send **a none** token for the *ttype* parameter if it does not have a table for the requested format. The *ftype* parameter specifies the bus format. The only token presently defined is **C_F**. The format for the table data is specified in Appendix A.

external message lmpb [length message]
external message: message tokens
lmpb: token - long message protocol
length: us - number of following bytes
message: special - the interface message

This message returns an ASCII interface message to a plug-in. The *message* is composed of semantic action tokens and parameters defined in the plug-in grammar. The format for the message data is specified in Appendix A. The *length* parameter specifies the number of following bytes. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter.

external status status
external status: message tokens
status: token - indicates plug-in execution status

This message is sent in response to an **external message, external fp_mode, external long_form, external set_data, external interf_data** or **external oper_compl** message. The *status* token is **ready**. A plug-in will always respond with either this status message or one of the defined error messages to each interface message.

2.9.3 GPIB Codes and Formats

This section defines the Mainframe/Plug-In Software Interface commands that are necessary to support specific functions defined by the Tektronix Codes and Formats document and the *Command Reference Specifications 11K Series Family of Products* document.

Init

The effect of this command is to set the instrument to a known state. When the mainframe receives this command over the bus, it will send **plugin_config init** messages to each plug-in to cause them to reset all settings to predefined states. See section **3.0 Common Messages** for the definition of the **plugin_config init** message.

Long Form ON/OFF

This function controls the type of command header reporting done by the instrument when a set or help query is received. If long form is disabled, commands are reported in abbreviated format. When long form is enabled, commands are reported in full length.

external long_form mode
external long_form: message tokens
mode: token - indicates reporting mode

This command is sent by the mainframe to control the type of reporting for set query commands. The *mode* parameter specifies the reporting mode. The **on** token selects full length format. The **off** token selects abbreviated format. The plug-in will send the **external status ready** message in response.

Set Query

The following two messages are used to implement the set query command defined in Codes & Formats.

external set_query type
external set_query: message tokens
type: token - indicates query type

This message is sent to each plug-in when the mainframe receives the SET? command. The *type* parameter indicates the type of response. The **ASCII** token requests the settings to be sent in ASCII encoded format. When returned as a command, this information will be processed by the ASCII interface parser in the same way as any ASCII interface command. The form of ASCII headers is determined by the **external long_form** message (see above). The **binary** token requests the settings to be sent in binary encoded format. When returned as a command, this format will bypass the parser and be returned directly to the plug-in.

external set_status lmpb [length status]
external set_status: message tokens
lmpb: token - long message protocol
length: us - number of following bytes
status: special - plug-in settings

This message is sent by a plug-in when it receives the **external set_query** message. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter. The *length* parameter specifies the number of bytes in the message following the *length* parameter. The *status* parameter is an ASCII or binary list of all plug-in functions and their present settings. Note that plug-ins should not report any query only headers for this message. The plug-in is responsible to prepend compartment designators where appropriate.

external set_data lmpb [length data]
external set_data: message tokens
lmpb: token - long message protocol
length: us - number of following bytes
data: special - plug-in settings data

This message is sent to smart plug-ins when the mainframe has received a binary setting over the external bus. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter. The *length* parameter specifies the number of bytes in the message following the *length* parameter. The *data* is the binary status that the plug-in sent in a previous **external set_status** message. The plug-in will send the **external status ready** message in response.

Help Query

The following two messages are used to implement the help query command defined in Codes & Formats.

external help_query
external help_query: message tokens

This message is sent to each plug-in when the mainframe receives the HELP? command.

external help_status lmpb [length status]
external help_status: message tokens
lmpb: token - long message protocol
length: us - number of following bytes
status: string - plug-in settings

This message is sent by a plug-in when it receives the **external help_query** message. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter. The *length* parameter specifies the number of bytes in the message following the *length* parameter. The *status* parameter is an ASCII list of all plug-in command headers. Note that plug-ins should not report any query only headers for this command. The plug-in is responsible to prepend compartment designators where appropriate.

Event Reporting

Event reporting is done using the error messages defined in section **3.2 Error Handling**. The text associated with an error message is returned over the GPIB when **long_form** is enabled. Smart plug-ins will send the error text when requested by the mainframe. It is up to the mainframe to determine whether to send the text over the GPIB.

General Reporting

Smart plug-ins may be required to send messages to the external bus as a result of a command or query. The message in this section is provided for that function.

external interf_data lmpb [length data]
external interf_data: message tokens
lmpb: token - long message protocol
length: us - indicates the number of following bytes
data: special - external bus message

This message is used by smart plug-ins to transfer a message to the external bus. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter. The *length* parameter specifies the number of bytes in the message following the *length* parameter. The *data* is formatted information ready for transfer over the external bus. The mainframe is not required to interpret the type or contents of *data*.

This message is also used by mainframes to pass data directly from the bus to the plug-in without running it through the parser. This may be required when binary data is sent over the bus to a plug-in. In this case, the *data* is external bus data for a plug-in. The plug-in or mainframe will send the **external status ready** message in response as appropriate.

Operation Complete

Smart plug-ins may send an operation complete status when they have completed certain operations that take a significant amount of time. These operations are determined by the plug-in designer. The plug-in will use the following message to transmit the operation complete information to the mainframe which will handle informing the GPIB of the plug-in status.

external oper_compl code index channel
external oper_compl: message tokens
code: ui - indicates operation condition
index: int - specifies text message
channel: us - indicates affected channel

This message is sent by a smart plug-in when it has completed certain operations. The *code* parameter indicates the operation condition that was completed. The *index* selects a text message that gives information about the completed operation. The *channel* parameter indicates the affected channel if any. If a channel is not appropriate, the *channel* parameter will be set to 0.

Version and Configuration Queries

These GPIB queries request information about the instrument system including the plug-ins. The mainframe will use the messages defined in section **3.0 Common Messages** to get the information necessary to respond to these queries.

Local Lockout

The purpose of this function is to prevent the operation of the instrument from the front panel controls within certain restrictions and rules. There are two modes of operation, locked and unlocked.

There are two special buttons called the display ON/OFF button and probe ID button. The plug-in will send the **channel_id status** or **probe_id status** message respectively when either of these buttons is activated. Smart plug-ins that use the generic capabilities will also supply a generic ID button for each input channel. Smart plug-ins that do not have display channels may supply an ID button whose operation is similar in function to the generic ID button. However, this button may cause the smart plug-in to perform certain actions (such as putting up status menus) while the generic amplifier plug-in ID button action is only sent as a status message to the mainframe. The mainframe decides what operation is requested when the generic plug-in ID button is pushed.

In unlocked mode, all plug-ins will respond to all front panel button operations. Pushes of the generic ID button (for both generic and smart plug-ins) will cause a message to be sent to the mainframe that the generic ID button was pushed. When other buttons on smart plug-ins are pushed, the smart plug-in will send a button push message to the mainframe to notify it that a front panel button has been pushed. The mainframe will not try to interpret the action required from the pushing the button but it may notify the ASCII interface that a button was pushed. The smart plug-in will take appropriate action in response to the button that was pressed.

In locked mode, smart plug-ins will still report the push of a generic ID button to the mainframe. However, smart plug-ins will take no action that would normally be associated with the pressing of that button. All other front panel buttons will be ignored. The smart plug-ins will not notify the mainframe or take any action when those buttons are pushed.

Generic plug-ins will not be informed of changes in the front panel mode. They will always report presses of the display ON/OFF or probe id buttons. The mainframe will determine whether or not to take any action in response to the button push based on the local lockout state.

To select local lockout mode, a mainframe will send **external fp_mode** messages to each plug-in.

external fp_mode mode
external fp_mode: message tokens
mode: token - indicates front panel mode

This message selects the front panel operational mode to be *mode*. The **locked** token causes the plug-in to enter the front panel locked mode. The **unlocked** token causes the plug-in to respond to all front panel controls. The plug-in will send the **external status ready** message in response.

external fp_button button
external fp_button: message tokens
button: us - indicates the button

This message is sent by a smart plug-in when one of its front panel buttons is pressed and if the front panel status is not **locked**. The *button* parameter is a number that indicates which button was pressed. This number only has meaning to the plug-in. The mainframe may use this message to generate an SRQ for the external bus. If a smart plug-in also has a channel id button it will use the **channel_id** message for reporting action on that button.

Group Execute Trigger

This function allows the external interface to initiate a predefined action. Smart plug-ins may specify the use of this function at powerup. See section **3.1 Powerup Sequence** for details. When the mainframe receives a Group Execute Trigger from the external bus it will notify plug-ins that have sent the **get** token using the following message:

external get
external get: message tokens

This message is sent only to smart plug-ins that have sent the **get** token in the **plugin_config status** message. The plug-in action taken when this message is received is specified in the plug-in's EIS.

2.10 New Trace and Triggering

Smart plug-ins are allowed to define new trace entries known as functions. For **E** type mainframes, information about plug-in functions will be separate from the table information defined in section **2.9 ASCII Interface**. For **R** type mainframes, additional information about function descriptions will be included in the plug-in tables for the mainframe parser. This section defines the limits of plug-in new trace capabilities that are supported by the mainframes. The details of function definition and upload format are specified in Appendix A.

When a smart plug-in function has been selected by the user, the mainframe will send a description of the selection to the plug-in in reverse Polish format. The plug-in has the option of notifying the mainframe that it cannot perform the requested operation. The information about functions uploaded for type **R** mainframes will contain sufficient information to prevent the mainframe from allowing the user to select operations that the plug-in (or the mainframe) cannot perform.

The following messages are used to control smart plug-in functions:

function select mode status seq_num [data]
function select: message tokens
mode: token - selects function mode
status: token - indicates selection status
seq_num: us - specifies sequence number
data: special - describes requested operation

This message is sent by the mainframe when a plug-in function is selected from its new trace menu. The *mode* parameter specifies whether the function is a trigger or display function output using two tokens: **trigger** and **display**. The *status* parameter indicates the status of the function and may be either **enable** or **disable**. When the **enable** token is sent, the plug-in will enable the output (if possible) of the operation specified by the *data* parameter for the sequence number specified by the *seq_num* parameter. The *seq_num* parameter specifies the position in the display output sequencer for the function and is a value from 1 to 12. The **disable** status token causes the plug-in to remove the function from the specified sequence. The *data* parameter is sent only with the **enable** status token. The format for the *data* parameter is specified in Appendix A.

function status status
function status: message tokens
status: token - indicates plug-in function status

This message is sent by a plug-in in response to a **function select** message. The *status* token indicates the status of the plug-in function operation and is either **ready** or **unable**. The **ready** token indicates the plug-in has performed the requested operation. The **unable** token indicates the plug-in cannot perform the requested operation. In this case, the plug-in will make no changes to its display sequencer output.

2.11 Trigger View

Mainframes will provide a facility to display the triggering point for smart plug-ins in the right compartment. This display may be either the actual waveform supplied by the plug-in in the right compartment or a marker generated by other means that indicates the instant of triggering. The plug-in in the right compartment must be the trigger source for at least one timebase for this function to operate.

This function requires a calibration procedure. See the section **5.4.6 Trigger View Calibration** in section **5.0 Calibration** for details. The mainframe will manage any constants needed to calibrate the trigger view function and will be responsible for using those constants to display the triggering point at the proper location on the display.

Smart plug-ins in the right compartment will notify the mainframe during the powerup sequence if they have this capability. See section **3.1 Start-up Sequence** for details. When the mainframe receives this notification, it will provide means for selecting the trigger view function in its new trace menu if the trigger view display is an actual waveform.

The trigger view function may be selected from either the mainframe's new trace menu or a menu from the smart plug-in. The signal output for display is always available at the AB-13 and AB-11 interface signal lines (see the *11000 Series Plug-In to Main Frame Interface Manual*).

trig_view request status
trig_view request: message tokens
status: token - selects operation

When the trigger view function is selected from a plug-in's menu, the plug-in will send this message to the mainframe to cause it to activate the trigger view function. The *status* token may be either **on** or **off**. The plug-in will send the **on** *status* token when the trigger view function is selected from one of its menus. It will send the **off** *status* token when the trigger view function is disabled from one of its menus.

The mainframe will send this message to a smart plug-in when the trigger view function has been selected by a mainframe menu. The plug-in will use this information to display proper status in any of its menus that might include the status of the trigger marker.

trig_view status status
trig_view status: message tokens
status: token - selects operation

The mainframe or the plug-in will send this message in response to the **trig_view request** message. The *status* token may be either **on** or **off**.

2.12 Auto-range

Mainframes will provide a means for controlling smart plug-in autoranging. The mainframe will coordinate selection of autoranging so that only one unit in the system is attempting to autorange at a time.

autorange request status
autorange request: message tokens
status: token - indicates autorange status

This message is sent to smart plug-ins to control their autoranging function. The **enable** *status* token is sent to enable a smart plug-in's autoranging function. The **disable** *status* token terminates the plug-in's autorange operation.

autorange status status
autorange status: message tokens
status: token - indicates autorange status

The plug-in will send this message in response to the **autorange request** message when it has set itself for autoranging. The *status* token may be either **enable** or **disable**.

2.14 Waveform Transfer

This section describes the method used for transferring waveform information from plug-ins that digitally acquire data to a mainframe for digital storage and display.

During the powerup sequence, a plug-in that has digitizing capability will notify the mainframe that it has such a capability and how many channels it can acquire. The mainframe will use this information to provide a means for the user to select the digitally acquired traces for display. See section **3.1 Powerup Sequence** for more information. The number of traces displayed at one time may be limited by the capabilities of the mainframe.

When the user selects a plug-in digitized input for display, the mainframe will request the plug-in to begin sending acquired data. This request will also include pertinent information about the data expected by the mainframe: number of points, data format etc. The plug-in will send a set of data each time it completes an acquisition. The data will include a header that describes the characteristics of the data followed by the actual acquired data.

The acquisition parameters including vertical size and position, input coupling, bandwidth limit and input impedance will be controlled by the user through normal mainframe supplied menus and controls. The mainframe will send generic plug-in **display set** and **trigger set** messages to the digitizing smart plug-in to control the sequencer input to the plug-in's digitizer. This interface will be identical to the interface defined for generic plug-ins with one exception. The exception is that the plug-in does not supply an analog signal output to the mainframe but instead supplies digitized data. The mainframe must adjust its acquisition and display accordingly.

The following messages are used to control and transfer data from the plug-in to the mainframe.

> **trace request** status wvfm_id [points format]
> **trace request:** message tokens
> status: token - specifies trace status
> wvfm_id: unsigned integer - trace number
> points: unsigned integer - specifies number of points
> format: token - specifies trace data format

This message is sent to a digitizing plug-in when the user has selected one of its traces for display from a mainframe menu. The *wvfm_id* parameter is the sequence number of a channel combination specified by a **display set** command. The *status* parameter indicates the status of the identified trace: **enable** or **disable** . When the **enable** token is sent, the identified trace has been selected. The plug-in will send trace data as often as it is available until the **disable** token for that trace is sent. The plug-in will terminate acquisition for the specified trace when it receives the **disable** token. The *points* parameter specifies the number of trace points expected by the mainframe. The plug-in will adjust its acquisition to meet this value. If the expected number of points is greater than the plug-in can acquire, the plug-in will send as many points as it can and notify the mainframe of how many are sent using the trace header. The format parameter specifies the data format expected by the mainframe. The formats available are **integer** and **fraction**. The *points* and *format* parameters are not sent when the trace status token is **disable**

trace status status
trace status: message tokens
status: token - indicates status

This message is sent by the plug-in in response to a **trace request** message. It indicates the plug-in has received the message and is ready to begin acquisition. This message is also sent by the mainframe in response to a **trace data** message. The **ready** *status* token indicates the mainframe has accepted the data and is ready for the next set.

trace data lmpb [**HDLT** header] [**DDLT** data]
trace data: message tokens
lmpb: token - long message protocol
HDLT: token - delimits trace header
header: special - specifies trace parameters
DDLT: token - delimits the trace data
data: special - trace data

This message is sent by a digitizing plug-in to transfer acquired data when it has received the **trace status** command with the **display** status token. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for information on the use and meaning of this parameter. The **HDLT** token signifies the beginning of the header. The *header* contains information about the trace data and is defined below. The **DDLT** token delimits the start of trace data. The *data* is the trace data as defined below. The *header* is optional. The mainframe will use previously supplied header information if a header is not included with trace data.

The trace header has the following format:

wvfm_id format points x_inc x_zero x_mult x_unit y_zero y_mult y_unit
wvfm_id: unsigned integer - trace number
format: token - trace data format
points: ui - number of points sent
x_inc: float - time between points
x_zero: float - zero location in horizontal divisions
x_mult: float - horizontal units per division
x_unit: string - horizontal unit name
y_zero: float - zero location in vertical divisions
y_mult: float - vertical units per division
y_unit: string - vertical units name

The *wvfm_id* parameter specifies the trace number for which the header and following data apply. The *format* parameter specifies the data format - **integer** or **fraction**. The *points* parameter specifies the number of points being sent. The *x_inc* parameter specifies the time between each data point. The *x_zero* parameter specifies the zero position in horizontal divisions. The *x_mult* parameter specifies the units per division. The *x_unit* parameter is the name of the horizontal unit of measurement. The *y_zero* parameter specifies the zero location in vertical divisions. The *y_mult* parameter specifies the vertical units per division. The *y_unit* parameter is the name of the vertical unit of measurement.

Plug-In digitized trace data is a list of acquired data values in the format specified by the header. The values are sent as a continuous list not separated by delimiters. The list contains exactly the number of points specified in the header.

This page is intentionally left blank.

3.0 Common Messages

3.1 Startup Sequence

The first part of the startup sequence involves initializing the SDI interface, starting up the transport protocol and completing self-tests. These subjects are covered in section **4.0 Self-tests.** The following sequence of events will occur when entering normal operational mode.

The mainframe will use the type information reported by each plug-in at power up (using the **plugin_config status** message) to set its configuration tables. (Some mainframes will lose this information during self-test and diagnostics. These mainframes will use the **plugin_config query** message to get plug-in type information after diagnostics are completed.)

After the configuration is determined, smart plug-ins will request system configuration information using the **sys_config query** message. The mainframe will send the **sys_config status** message to inform the smart plug-in of the system configuration. This information is required by the smart plug-ins in order to properly fill out their external interface parse tables.

Smart plug-ins will then be asked for their external interface tables using the **external req_table** message and will send those tables using the **external table_data** message. After smart plug-ins have sent their tables, they may request any other information about the system (such as generic plug-in status using the **generic command** message or requesting updates using the **update request** message) as defined in section **2.0 Smart Plug-Ins.**

Generic plug-ins will be asked for their generic menu definitions using the **menu request generic** message. These will be uploaded as defined in section **1.10 Generic Menus** using the **menu def_generic** message. Mainframes will set generic plug-ins to values saved by the mainframe for that type of plug-in.

The final power up step is calibration if required. This will be done according to the rules defined in section **5.0 Calibration.** When calibration is complete, the instrument will enter its normal operating state.

The following messages are used to determine plug-in capabilities during the powerup sequence.

plugin_config query
plugin_config query: message tokens

This message is sent by the mainframe to request plug-in configuration information.

plugin_config status pi_type [disp_channels trig_channels aux_channels] [**minus_coupl**] [**lower_bandw**] [**diff_offset**] [**no_invert**] [**trig_view**] [**trig_out**] [**dig** channels] [**get**]

plugin_config status: message tokens
pi_type: token - indicates plug-in type
disp_channels: us - number of display channels
trig_channels: us - number of trigger channels
aux_channels: us - number of auxiliary trigger channels
minus_coupl: token - indicates minus coupling capability
lower_bandw: token - indicates lower bandwidth capability
diff_offset: token - indicates differential offset capability
no_invert: token - indicates no inversion capability
trig_view: token - indicates trigger view capability
trig_out: token - indicates trigger output capability
dig: token - indicates digitizing capability
channels: us - indicates the number of digitized channels
get: token - indicates group execute trigger capability

This message is sent by each plug-in in response to the **plugin_config query** message from the mainframe and in response to the initial power-up **SRQ query** message. The *pi_type* token indicates the type of plug-in: **generic**, **smart** or **both**. The **generic** *pi_type* has only generic amplifier channel capabilities defined in the first section of this document. The **smart** *pi_type* has only smart capabilities and does not use any of the generic functions. The **both** *pi_type* uses both interfaces. This plug-in type has input channels that conform to the definitions of generic plug-ins but also has other capabilities that are beyond those defined for generics. For the purposes of this document, references to generic plug-ins refer to plug-ins that identify themselves as either **generic** or **both**. References to smart plug-ins refer to plug-ins that identify themselves as either **smart** or **both**.

The *disp_channels*, *trig_channels* and *aux_channels* parameters indicate how many display, trigger or auxiliary trigger channels, respectively, a **generic** or **both** type of plug-in has. These definitions start with channel 1 (that is, if a plug-in has 4 display channels and 2 trigger channels, the 2 trigger channels *must* be channel 1 and channel 2). A **smart** plug-in will not report any channels.

The **minus_coupl** token indicates a generic plug-in that has differential inputs. The **lower_bandw** token indicates a generic plug-in that supports the lower bandwidth function. The **diff_offset** token indicates a generic plug-in supports the differential offset functions and the **diff_offset** messages (**diff_offset set**, **diff_offset query** and **diff_offset status**).

The **no_invert** token indicates a generic plug-in that does not have display or trigger signal inversion capability. This affects the operation of the **display set** and **trigger set** messages defined in section **1.16 New Trace**.

The **trig_view** token indicates that the plug-in has a trigger view function to be used for identifying the trigger location. This capability will cause the mainframe to use the trigger view calibration procedure defined in section **5.0 Calibration** and also put a trigger view trace selection in its new trace menu if it has that capability. See section **2.11 Trigger View** for smart plug-ins for more details.

The **trig_out** token indicates the plug-in has a fixed output trigger signal that may be used for display and triggering. A plug-in that uses this function may not have any generic channels. The mainframe will put a selector for this plug-in in its trigger source menu.

The **dig** token indicates the plug-in is a digitizing plug-in and has digitized channels available for display. The mainframe will put selection means in its new trace menu for these traces. See section **2.14 Waveform Transfer** for more details. The *channels* parameter is used only with the **dig** token and indicates how many digitized channels the plug-in has for display. A plug-in that specifies this capability must also identify itself as a **both** type plug-in so the mainframe may control the input parameters of the digitized channels using the generic interface.

The **get** token indicates the plug-in has group execute trigger capability. The mainframe will notify these plug-ins each time it receives a group execute trigger message from the external interface. See section **2.9.3 GPIB Codes and Formats** for more details.

The mainframe handler for this message should be written to be able to ignore unknown tokens that may be sent by the plug-in. These tokens will specify the presence of future functions and capabilities. This will be the means by which future plug-ins and mainframes may add functionality to this interface.

menu request generic
menu request: message tokens
generic: token - requests generic plug-in menus

This message is sent by the mainframe to each generic plug-in to request them to upload their basic menus. The plug-ins will respond using the messages defined in the generic plug-in section **1.10 Generic Menus**.

3.2 Error Handling

Protocol system error handling is described in section 6.

There are five error conditions defined for this interface: Command Errors, Execution Errors, Internal Errors, Execution Warnings and Internal Warnings.

The following messages are used to communicate plug-in error conditions to the mainframe.

error generic status code index channel
error generic: message tokens
status: token - indicates type of error
code: ui - indicates error category
index: int - selects error message
channel: us - indicates which channel

error smart status code index channel
error smart: message tokens
status: token - indicates type of error
code: ui - indicates error category
index: int - selects error message
channel: us - indicates which channel

The plug-in will send this message to the mainframe when it detects an error condition. Smart plug-ins will send the **error smart** message and generic plug-ins will send the **error generic** message. Plug-ins that are the **both** type will use the appropriate message to report each error condition based on whether the error was with the generic or smart interface. The mainframe will handle error reporting both to the display and the external bus as defined for each error type. See the section **GPIB Error Reporting** below for details. The *status* parameter identifies the error type using the following tokens:

command_error - This token identifies a command error. This type of error is generated when the plug-in receives an incorrect command.

exec_error - This token identifies an execution error. This type of error is generated when the plug-in receives a command it cannot execute. IF SETTINGS DIDN'T CHANGE

internal_error - This token indicates an internal error. This type of error is generated when a plug-in has detected a serious problem with its operation not as a result of a command.

exec_warning - This token indicates an execution warning. This type of error is generated when the plug-in can execute a command but the results of the execution may be unexpected and the user needs to be notified.
IF SETTINGS CHANGED SOMEHOW

internal_warning - This token indicates an internal warning. This type of error is generated when a plug-in detects a problem with its operation that is not serious but requires user notification.

The *code* parameter specifies the category of error detected. These are broad categories that have predefined meaning. The value and categories are defined in the *Command Reference Guidelines 11K Series Family of Products* document. The *channel* parameter indicates the channel (if any) to which the error is related. This parameter will be inserted at a specified place in the text of the message indicated by the *index* parameter when the mainframe formats the message. If the *channel* parameter is 0, the error is not a channel related error.

The *index* parameter for smart plug-ins specifies a string of text that gives detailed information about the type of error. When the mainframe is ready to format the text for display or transmission, it will request the error text from the smart plug-in using the **error req_text** message and the *code* and *index* parameters. The smart plug-in will return the error text using the **error text** message. The *index* parameter sent by the smart plug-in must have the most significant bit set (must be a negative integer). This identifies the error message as a smart plug-in message. This will allow the mainframe to sort out mainframe and plug-in error messages that are stacked in a queue waiting for external bus transmission.

The *index* parameter for generic plug-ins specifies the type of function for which an out-of-range condition was detected. When the *status* byte is **exec_warning** and the *code* value is 550 or if the *status* byte is **exec_error** and the *code* value is 205, the *index* parameter indicates which function was out-of-range according to the following table:

index	**function**
1	vertical size
2	vertical position
3	bandwidth limit
4	input impedance

The mainframe will substitute the name of the indicted function in the text of the error message when it is displayed or sent over the external bus. The *index* parameter has no meaning for generic plug-ins when reporting errors other than the out-of-range condition. A zero value will be used for the *index* parameter in these cases.

For both generic and smart plug-in error texts, there are embedded escape sequences that may be used to insert channel and plug-in compartment information. This allows a single message format to be used for any plug-in compartment or channel rather than having separate messages for each channel and compartment. The escape sequences use the percent sign to identify the insertion type and location in the text. They are defined as follows:

%a - specifies the channel number. When the mainframe is formatting either generic or smart plug-in error messages, it will replace this sequence of characters with the numeric value of the *channel* parameter of the **error** message.

%A - specifies the generic function type for parameter out-of-range condition. This will be replaced by the mainframe with the function name indicated by the *index* parameter of the **error generic** message. Smart plugins must not use the %A construct in error texts.

%b - specifies the compartment identifier. When the mainframe is formatting either generic or smart plug-in error messages, it will replace this sequence of characters with the compartment identifier of the plug-in that sent the **error** message. This identifier is a single character (eg. L, C, R).

%B - specifies the compartment name. When the mainframe is formatting either generic or smart plug-in error messages it will replace this sequence of characters with the compartment name of the plug-in that sent the **error** message. The compartment name is a word describing the compartment (eg. left, center, right).

When the percent sign is followed by any other character, that character only will be displayed. Thus, %% causes a single % to be displayed.

error req_text code index
error req_text: message tokens
code: ui - specifies the error category
index: int - specifies the error text

This message is sent by the mainframe to a smart plug-in that has previously sent an **error smart** message to request the error text associated with the error. The *code* parameter indicates the error type and is the same as the *code* parameter sent by the plug-in in the **error smart** message. The *index* parameter indicates the error string and is the same as the *index* parameter sent by the plug-in in the **error smart** message.

error text text
error text: message tokens
text: string - the error message

This message is sent by a smart plug-in in response to a **error req_text** message from the mainframe. The *text* parameter is a character string describing the error condition. It is limited to 50 displayed characters. It may include %a, %b and %B constructs as defined above. The mainframe will format this string when these constructs are used. Note that if the plug-in wishes to identify an external plug-in command header, it is the plug-in's responsibility to place an underscore in the proper place following the compartment name as defined for plug-in command headers in the *Command Reference Specifications 11K Series Family of Products* document.

GPIB Error Reporting

Codes and Formats for GPIB defines several error categories that are identified using the status bytes. The plug-in/mainframe interface error *status* codes defined above correspond exactly to C & F status bytes of the same name. When the mainframe receives an error message from a plug-in, it will assert SRQ as appropriate and save the status byte for later retrieval for the bus. The *code* parameter defined for the **error generic** and **error smart** messages above corresponds to the event code defined for Codes and Formats. The event code descriptions are predefined and are specified in the *Command Reference Specifications 11K Series Family of Products* document. The *index* parameter defined for the error message is not related directly to Codes & Formats error reporting. The message defined by the *index* parameter will be reported to the GPIB if longform is enabled. The mainframe will decide when to request the error text from a smart plug-in.

In general, errors that result from the execution of a plug-in command from the mainframe will be reported to the source of the command. Errors that result from commands from the mainframe's human interface will be displayed on the screen. Errors that result from commands from the external bus will be sent over the bus. Errors that are not related to a specific command will be broadcast both to the external bus and the screen. An example of this type of error is an input overload condition for a generic amplifier. This error does not result from the execution of a command so the error message must be sent to all control sources.

3.3 Other Common Commands

The mainframe will use the following message to determine all plug-in nomenclature and software versions:

plugin_id query
plugin_id query: message tokens

This message will cause the plug-in to return its 11000 series nomenclature and software version numbers using the following status message:

plugin_id status name version
plugin_id status: message tokens - indicates plug-in id message
name: string - 11000 series nomenclature
version: string - specifies the software version number

The *name* parameter specifies the functional type of a plug-in using the new 11000 series nomenclature and is limited to 10 characters. The *version* parameter specifies the software version number for the plug-in. It is also limited to 10 characters.

plugin_uid query
plugin_uid query: message tokens

This message is sent by the mainframe to request the unit identification number of a plug-in. This number is set to the plug-in's serial number during manufacturing. The plug-in will respond with the following message:

plugin_uid status uid
plugin_uid status: message tokens
uid: string - plug-in identification number

The plug-in uses this message to report its identification number in response to a **config query uid** message from the mainframe. The *uid* parameter is a string that is set during manufacture or during service at a Tek service center to the serial number of the plug-in. It is limited to 10 characters. This value might, but should not, be changed by the user in the field.

plugin_uid set uid
plugin_uid set: message tokens
uid: string - plug-in identification number

The mainframe uses this message to set the value of the plug-in's identification number. This may be a string of up to 10 ASCII characters. The *uid* parameter must be unique for each instrument of the same type. Since this value might be set by the user, it may not be used for determining software or hardware versions or revisions.

plugin_config init
plugin_config init: message tokens

This message is sent by the mainframe to initialize a plug-in to its EIS defined settings. Each plug-in will set itself to those settings and return present setting status to update the stored settings buffer. Generic plug-ins will respond with the status messages defined in section **7.0 Messages** for this command. Smart plug-ins will presume that all mainframe functions (such as knobs and menus) have been removed when this message is sent. The smart plug-in does not need to send messages to release those functions.

cal const_query
cal const_query: message tokens

This message is sent by the mainframe to request calibration constants for any plug-in.

cal const_status status const_num value
cal const_status: message tokens
status: token - indicates constant set status
const_num: unsigned integer - specifies which constant
value: float - calibration constant

This message is sent by a plug-in in response to the **cal const_query** or **cal const_set** message from the mainframe. The status parameter indicates whether the plug-in is set up to modify calibration constants. The **enable** *status* token indicates the plug-in is enabled for changing calibration constants. The **disable** *status* token indicates the plug-in is prevented from changing calibration constants.

The *const_num* parameter indicates which constant is being reported. The *value* parameter is the calibration constant value in floating point format. The mainframe will convert these to ASCII format for transmission over the external ASCII interface.

When this message is sent with the **disable** status token in response to a **cal const_set message**, the plug-in will report the previous value of the constant.

cal const_set const_num value
cal const_set: message tokens
const_num: ui - specifies which constant
value: float - calibration constant

This message is sent by the mainframe to set calibration constants in the plug-in. The *const_num* parameter specifies which constant is to be set to *value*.

Plug-ins will have an internal hardware control that will enable or disable the setting of calibration constants. This is to provide security for those constants so that any change can be detected.

Both smart and generic plug-ins will use these messages to transfer calibration constants to the external interface.

cal const_query_num
cal const_query_num: message tokens

This message is sent by the mainframe to request the number of calibration constants in a plug-in. This is to support the query all syntax of GPIB.

cal const_num number
cal const_num: message tokens
number: ui - specifies the number of cal constants

This message is sent by the plug-in in response to the **cal const_query_num** message to report the number of calibration constants in the plug-in. The mainframe may query for the complete list of constants by querying for each constant individually using the query message defined above.

SRQ query
SRQ query: message tokens

This message is sent by a mainframe to request a plug-in to send status or commands when the mainframe has detected a plug-in SRQ. The SRQ is defined in section **6.0 Protocols**. The plug-in will respond with a single command or status message. If a command message is sent, the mainframe will respond with the appropriate status message. The mainframe will then repeat the **SRQ query** message. The plug-in will respond either with another status or command message or with the **SRQ no_report** message. The mainframe will continue sending **SRQ query** messages until it receives the **SRQ no_report** message.

SRQ no_report
SRQ no_report: message tokens

This message is sent by a plug-in in response to an **SRQ query** message to indicate that it has no other status or commands to send. The mainframe will clear the SRQ status for that plug-in and will not send any more **SRQ query** messages to it. See section **6.0 Protocols** for more details on the operation of plug-in SRQ's.

4.0 Self-test

Instrument self-test control falls into two categories: power-up self-tests and diagnostic mode. At power-up, the mainframe and each of the plug-ins performs a series of self-tests to check for functional operation of circuits. If there are no failures of these tests, instrument operation will continue to the next step of the power-up sequence. If there are failures or when a request is made by the user, the instrument will enter the diagnostic mode. This mode is used to locate and assist in diagnosing circuit faults. It provides a menu display using either the mainframe's normal display or an ASCII interface to a remote display. The user also may be allowed to request normal operation of the instrument whether or not tests have failed.

The tests that are run in power-up mode will be available for execution in diagnostic mode wherever possible. This is to prevent the situation of a powerup test indicating a failure that a diagnostic test cannot locate because the tests are different. In the diagnostic mode, the user will have explicit control over which tests are run and how they are run. In the power-up test mode, the tests are run once and only summaries of failures are given. In the diagnostic mode, diagnostic information is available to the user to assist in troubleshooting instrument faults.

Generic plug-ins will provide sufficient information to support the diagnostic mode. Access to individual routines or areas will be limited by the capability of the plug-in. Generic plug-ins will support all levels (block, area and routine) defined in this interface.

Mainframes will provide two facilities for plug-ins to assist in testing. These are the voltage source and measurement capabilities identified in section **5.5 Calibrated Voltage Reference** and **5.6 Voltage Measurement**.

4.0.1 Plug-In Communication Initialization

This section defines the sequence of events that cause communication to be established between the mainframe and the plug-in.

When the plug-in has completed its kernel tests after power up, it will begin sending SRQ's to the mainframe. When the mainframe is ready to incorporate plug-ins, it will look for plug-in SRQ's. The plug-in must begin sending SRQ's less than 2 seconds after the power is applied. Time 0 at power up is specified as the time the +5.1 digital supply crosses 4.75 volts. Plug-in time delay from this threshold crossing to release of the processor reset line is included in the 2 seconds.

If the plug-in gets a handshake error when the SRQ byte is sent or if it detects the SRQ byte was removed, it will immediately place another SRQ byte in the transmit buffer. This effectively sets the duty cycle of the first SRQ at 100%. The mainframe will begin looking for the plug-in SRQ after 2 seconds from power on and will continue to look until 2.1 seconds after power on. If the mainframe does not receive a plug-in SRQ within this time window, the mainframe may assume there is not an 11k plug-in in that compartment. The .1 second window is specified to cover differences in the tolerance of threshold detectors and reset timers in the mainframe and plug-ins and the rise time of the +5.1 digital supply.

If the mainframe takes longer than 2 seconds to reach the point of checking for plug-ins it need only wait the remaining time to 2.1 seconds to check for plug-ins. If the mainframe takes longer than 2.1 seconds to check for plug-ins it may assume that if the SRQ is not received at the first check there is not an 11k plug-in in that compartment.

When the mainframe recognizes a plug-in SRQ, it will send the **SRQ query** message. The plug-in will send the **plugin_config status** message in response. The plug-in is now ready to receive commands from the mainframe.

4.1 Startup Self-tests

Startup self-tests check most portions of the instrument system for functionality. The mainframe will coordinate plug-in self-tests after each plug-in has performed initial testing. The mainframe will also coordinate the testing of the display and trigger signal paths. The sequence of events is as follows:

1. The mainframe and plug-ins will do kernel and SDI testing. This SDI test will be a loopback test. When a plug-in has completed these tests, it will send a plug-in SRQ and wait for a response from the mainframe. The plug-in will continue sending SRQ's until it gets a response.

2. When the mainframe completes its initial tests, it will wait for the plug-in SRQ. When it receives an SRQ, it will identify that plug-in as an 11000 series plug-in.

3. Plug-ins will wait for a message from the mainframe. This message will either request the plug-in to complete self tests, begin the diagnostic mode or skip both and enter normal operating mode. At this time, the mainframe will also determine system configuration and report system status to smart plug-ins.

4. After the mainframe has completed all of its self-tests and if diagnostic mode is not being entered, the mainframe will request each plug-in in turn (left, center, right) to complete self-testing. When a plug-in receives this message, it may use the mainframe functions that are normally available for self-tests.

5. When a plug-in has completed all of its self-tests, it will notify the mainframe using the self test status message that it is through and will report the first error found, if any. The mainframe will request the next plug-in to complete its self tests. The order will be left, center, right.

6. The plug-in in the right compartment may request generic plug-ins to be put into calibrated voltage reference pass-through mode to complete testing of the auxiliary trigger signal lines. See section **4.1.3 Auxiliary Trigger Line Test**.

7. When these tests are complete, the mainframe will notify each plug-in that self-testing is finished.

If there are errors in any of the plug-in self-tests except the kernel and loop back SDI tests, the index of the first failed routine will be saved and self-test status will be reported with the message the plug-in uses to notify the mainframe that it has completed self-tests.

After completing its kernel and SDI loopback tests, the plug-in will send an SRQ whether the tests pass or fail. The plug-in will repeat sending the SRQ even if it detects that the SRQ byte was transmitted successfully by monitoring the handshake error status of the SDI. It will send SRQ's indefinitely while waiting for an **SRQ query** message from the mainframe. When it receives the **SRQ query** message it will stop sending SRQ's and send the **plugin_config status** message.

Because some mainframes will not support plug-in SRQ's during self-test and diagnostics (except for the initial power up SRQ), plug-ins may not send SRQ's to the mainframe during self-tests and diagnostics.

4.1.1 Messages

The following messages are used to control plug-in testing and to report test results. Except for the **test begin** message, these messages are legal only during test and diagnostic modes. The test mode is defined as the time from either power up or the **test begin** message is sent to the plug-ins until the **test end** message is received from all plug-ins. See the diagram at the end of section **4.0**. The only test message that is valid during diagnostic or normal operational modes is **test begin**. Responses to the other messages are undefined when not in test mode.

test complete
test complete: message tokens

This command is sent to a plug-in to instruct it to complete its self-tests. When a plug-in receives this command it may use the mainframe functions that are normally available during testing. These are the calibration voltage source function and the voltage measurement function. The mainframe will connect the reference ground of the CVR to the plug-in when it sends the **test complete** message. In this way, the mainframe will coordinate the use of those functions. A plug-in may perform testing that does not require mainframe functions prior to receiving this message (and after establishing communication) but it must be able to interrupt those tests if necessary to receive the message correctly.

Plug-in SRQ's are not required for the plug-in to request calibrated voltage source and measurement functions. These messages may be sent during the time the plug-in is executing its self tests. There are five valid messages that a plug-in may send after it receives the **test complete** or **test begin** message: **test status**, **cal set_cvr**, **cal make_meas** and **cal cvr_connect**. This last message is only valid for a smart plug-in. When the plug-in sends the **test status ready** message, this mode is terminated.

test status status [block_id area_id routine_id fault_id]
test status: message tokens
status: token - indicates plug-in status
block_id: us - indicates diagnostic block
area_id: us - indicates diagnostic area
routine_id: us - indicates diagnostic routine
fault_id: char - indicates routine status

This message is sent by the plug-in either in response to the **test begin** message or to indicate completion of self testing. When sent in response to the **test begin** message, the plug-in will send the **busy** *status* token only.

When reporting test results, the plug-in will send the **ready** *status* token and include the various id status parameters. The mainframe will respond with **test status ready** only. If the tests are all passed successfully, the plug-in will send the message with the *fault_id* parameter set to '0' (character). All others will be set to 0 (integer). If a test fails, the plug-in will retain information about that test for diagnostic mode and report the status of the failed routine. The mainframe will use this status to decide whether to enter the diagnostic mode or to continue with normal operations.

The *block_id*, *area_id* and *routine_id* parameters are the value of the block, area and routine number of the test that failed. Values may range from 0 to 11. See the diagnostic section for the definitions of the *block_id*, *area_id* and *routine_id* parameters. The *fault_id* is a character representing fault information or '0' for pass. See the **diag status routine** message. Plug-Ins will set the *block_id* to 1, 2, or 3 for each compartment the plug-in occupies. The leftmost compartment will be assigned the smallest block number increasing to the right.

test begin
test begin: message tokens

This message is sent to plug-ins by the mainframe to execute power-up self-tests (other than kernel tests) after the instrument has entered the normal operational mode. This is the only valid test message in that mode. When the plug-in receives this message it will begin its power-up self-tests in step 3 as defined above. The plug-in will not send SRQ's to start the sequence. It is required that all stored information about plug-in operating modes will be preserved (by the plug-in) so the plug-in may be restored to its previous state. The plug-in and mainframe will continue the power-up selftest sequence as defined above. When testing is completed, the mainframe and plug-ins will continue with the power-up sequence. See section **3.1 Startup Sequence** for more information.

4.1.2 Signal Path Test

Generic plug-ins are responsible for testing the display signal path using the calibrated voltage reference and measurement capabilities of the mainframe. The mainframe may place the plug-in into calibrated voltage reference pass-through mode to test trigger and alternate display signal paths. The message for selecting this mode is the **cal cvr_mode** message.

cal cvr_mode mode [gain]
cal cvr_mode: message tokens
mode: token selects the mode
gain: float - plug-in gain value

This message, when sent to a generic amplifier plug-in with the **enter** *mode* token requests the plug-in to connect all channel inputs to the calibrated voltage reference source. The mainframe (or another plug-in) may use this mode to complete testing or to do calibration. The plug-in will set its gain to the most appropriate value for testing and report that value using the *gain* parameter. This parameter is only sent by the plug-in when the *mode* token is **enable.** The mainframe will connect the calibrator ground reference to the plug-in that receives this message with the **enter** *mode* token.

The **ground** *mode* token requests the plug-in to connect its input to ground instead of the CVR. The **CVR** *mode* token requests the plug-in to reconnect its input to the CVR.

When the **exit** *mode* token is sent, the plug-in will reconnect its channel inputs to the input connectors and restore the state of the generic functions. The mainframe will disconnect the calibrator reference ground from the plug-in.

When the plug-in is in pass-through mode, the mainframe (or other plug-in) may request changes in the gain or offset or other input functions of the generic plug-in using the normal set of generic commands but the mainframe or plug-in is responsible to restore previous settings when the testing or calibration is completed. See section **5.3.10 CVR Pass-Through Mode** for more details.

4.1.3 Auxiliary Trigger Line Test

A plug-in in the right compartment that uses the auxiliary trigger lines may test those lines by requesting the mainframe to place generic plug-ins in the calibrated voltage reference pass-through mode. The plug-in in the right compartment can request calibrated voltages to be sent through each generic plug-in to the auxiliary trigger lines using the **cal cvr_connect message.**

> **cal cvr_connect** compartment mode [gain]
> **cal cvr_connect:** message tokens
> compartment: char - selects the plug-in compartment
> mode: token - selects the mode
> gain: float - plug-in gain value

When the mainframe receives this message with the **enter** *mode* token, it will send a **cal cvr_mode enter** message to the plug-in identified by the *compartment* parameter. It will also connect the calibrator ground reference to that compartment. When it receives the **cal cvr_mode** response from the generic plug-in, the mainframe will send the **cal cvr_connect** message back to the requesting plug-in. This **cal cvr_connect** message sent to the smart plug-in will also include the *gain* value reported by the generic plug-in in the **cal cvr_mode** message. The smart plug-in may then perform tests or make measurements using the calibrated voltage reference which is passed through the plug-in to the display and trigger outputs and also to each auxiliary trigger output.

The smart plug-in may request gain or offset or other input function changes of the generic plug-in (using the **generic command** message) but is responsible to restore the original values when it is done with the test.

The **cal cvr_connect** message with the **ground** *mode* token requests the generic plug-in to connect its inputs to ground instead of the CVR. The **CVR** *mode* token requests the generic plug-in to reconnect its inputs to the CVR.

When the **cal cvr_connect** message with the **exit** *mode* token is sent, the mainframe will send a **cal cvr_mode exit** message to the plug-in specified by the *compartment* parameter and disconnect the calibrator ground reference from that plug-in. When it receives the **cal cvr_mode** response from the plug-in, the mainframe will send the **cal cvr_connect** message with the **exit** *mode* token back to the requesting plug-in.

The **cal cvr_connect** is not restricted in use for the right plug-in compartment only. It may be used by any smart plug-in that requires the pass-through mode of any generic plug-in.

4.1.4 Exiting Test Mode

> **test end**
> **test end:** message tokens

This message is sent to each plug-in to indicate the termination of the self-test mode. The plug-in will enter diagnostic mode upon receipt of this message and will send the **diag status** message (see below) as a response. The mainframe must send the **diag exit** message to exit the diagnostic mode and enter normal operation mode.

4.2 Diagnostic Mode

Diagnostic mode is provided to assist the user in troubleshooting instrument errors. There are several fundamental underlying assumptions:

1. Each processor in the instrument system (includes plug-ins) must do its own kernel test independently. If a kernel is not working, further diagnostics will not be available for the module controlled by that kernel. A kernel is defined as the microprocessor, ROM and associated control circuits necessary to execute programs (need not include RAM).

2. For plug-ins, the SDI must also be operational for diagnostic mode to be useful. Plug-Ins will test their SDI's independently.

3. The mainframe must be fully functional before an attempt is made to troubleshoot a plug-in. Plug-In tests may be run when the mainframe is not fully functional but the results are questionable.

4. Each diagnostic test must be able to run independently from all other tests. There will be no tests that depend on a state left by a previous test.

5. Either a mainframe's normal display is available for displaying and selecting menus or an ASCII interface to an external device (such as a terminal) is available.

To simplify the diagnostic process, the instrument is broken down into blocks and areas. A block is any large section of circuitry within the instrument. Plug-Ins are designated as a single block per compartment. Within each block are circuit areas. A plug-in may have up to 11 areas per block. Each area may have several routines that test the area. There may be up to 11 routines per area. Information about the status of blocks, areas and routines will be displayed by the mainframe. The user may select execution of individual routines.

The plug-in will upload a configuration table that describes the tests that it has available for diagnostic mode. The mainframe will use this information to build menus that allow the user to select individual tests for execution. These tests may be selected to run to completion even if errors are discovered or to terminate the test and report results immediately when an error is found.

4.2.1 Messages

The following messages are used to control the operation of plug-in diagnostics. Except for the **diag enter** message, these messages are

legal only during the diagnostic mode. The diagnostic mode is defined **as** the time from the mainframe sending either the **diag enter** or the **test end** message to the receipt from the plug-ins of the **diag exit** **message**. See the diagram at the end of section **4.0**.

diag enter
diag enter: message tokens

This message is sent to request the plug-in to enter diagnostic mode. The plug-in will prepare itself to perform diagnostic testing when it receives this message. It will send the **diag status ready** | message as the response.

diag exit
diag exit: message tokens

This message is sent to indicate the termination of diagnostic mode. The plug-in will exit the diagnostic mode, send the **diag status ready** message and wait for the next message from the mainframe.

diag request
diag request: message tokens

The mainframe may use this message to request the test configuration of a plug-in for use in generating menus. This message may be sent whenever the mainframe requires the configuration of the plug-in and is not restricted to diagnostic mode.

diag config lmpb {**BDL** block_name {**ADL** menu_type area_name {**RDL** routine_type routine_name [**FDL** fault_id field1 field2 field3]}... }... }...
diag config: message tokens
lmpb: token - long message protocol byte
BDL: token - block delimiter
block_name: string - block name
ADL: token - area delimiter
menu_type: token - indicates type of routine menu
area_name: string - area name
RDL: token - routine delimiter
routine_type: token - indicates type of routine
routine_name: string - routine name
FDL: token - fault status delimiter
fault_id: character - indicates test status
field1: float, ul, ui or us - fault info
field2: float, ul, ui or us - fault info
field3: float, ul, ui or us - fault info

This message sends the plug-in's test configuration to the mainframe. It is sent in response to a **diag request** or **test end** message. The order of items listed is significant because the mainframe uses that order to select routines for execution in the **diag exec** message.

The *lmpb* parameter is the long message protocol byte. See section **6.0 Protocols** for its definition and use. This message may be split into smaller messages only just prior to one of the delimiters (**BDL**, **ADL**, **RDL** or **FDL**).

The **BDL** token identifies the beginning of a block field. It is immediately following by the *block_name* parameter which defines the name of the block. Each block field must be followed by at least one area field.

The **ADL** token identifies the beginning of an area field. There may be several area fields defined for each block. The *menu_type* parameter indicates the type of routines found in the area. There are seven types of routines: **minmax_int**, **minmax_long**, **minmax_float**, **digital_int**, **digital_long**, **digital_mix** and **digital_short**. See the explanation of the **diag status** message for more information on the meaning of these types. The *area_name* parameter is the name of the area.

The **RDL** token identifies the beginning of a routine field. The *routine_type* parameter indicates the type of routine - **auto**, **auto_inter**, **auto_nr** or **manual_inter**. The **auto**, **auto_inter** and **auto_nr** tokens indicate routines whose results are analyzed by the system. The **auto** token indicates a routine that may be run without any user intervention. The **auto_inter** (automatic, interactive) token indicates a routine that requires user setup but then can be run without further user intervention. The **auto_nr** is a routine that is automatic but was not run during the powerup selftest. The **manual_inter** (manual, interactive) token indicates a routine that requires user interpretation. No results are returned when this routine is executed. Generally, **manual_inter** routines are used for providing stimulus for probing circuits. The *routine_name* parameter is the name of the routine. There may be several routine fields for each area field.

This configuration information implies execution order. If a routine fails during powerup tests, the mainframe may use the returned index to mark all routines except those marked **auto_nr** listed prior to the indicated routine as having passed and those that follow as not having been executed. In diagnostic mode, the user may select execution of any routine; no order is enforced.

The **FDL** token identifies the beginning of fault status information. These parameters are optional. The *fault_id*, *field1*, *field2* and *field3* parameters have the same format as those defined for the **diag status** message.

diag exec block_id area_id routine_id mode
diag exec: message tokens
block_id: us - selects block
area_id: us - selects area
routine_id: us - selects routine
mode: token - selects testing mode

This message is sent by the mainframe to execute a test routine. The *block_id* and *area_id* parameters select the plug-in block and area that contain the routine. The *routine_id* parameter is the routine to be executed. The values of these parameters are based on the configuration uploaded by the plug-in. Each selects the n^{th} entry from the configuration table where n equals the *block_id*, *area_id* or *routine_id* parameter. Selection values start with 1 (not 0).

The *mode* parameter specifies the testing mode. The **single** token requests the plug-in to execute the test once and return status. The **cycle_halt** token requests the plug-in to terminate a test when an error is found. The **cycle** token selects repetitive execution mode. In both cycling modes, the plug-in may request voltages and measurements from the mainframe.

diag halt
diag halt: message tokens

This message is sent by the mainframe to terminate the looping of a test. The plug-in will halt looping the test after the next test completion and report test results using the **diag status** message.

diag status status [loops faults fault_id field1 field2 field3]
diag status: message tokens
status: token - indicates plug-in status
loops: ui - indicates number of loops
faults: ui - indicates number of faults found
fault_id: char - fault identifier
field1: float, ul, ui or us - fault data
field2: float, ul, ui or us - fault data
field3: float, ul, ui or us - fault data

This message is sent by a plug-in in response to a **test end**, a **diag enter**, **diag exit** or **diag exec** command from the mainframe. The status token may be **ready**, **error** or **routine**. The **ready** *status* token indicates that the plug-in is ready to perform the requested action. The **error** *status* token indicates for the **diag exec** command that the plug-in detected an error with the command. The rest of the parameters are included only when the *status* token is **routine**.

When sent with the **routine** *status* token, this message returns routine status after execution of a test routine. The plug-in will send this message when a test has been completed or when a test is terminated by an error (in **cycle_halt** mode) or by the **diag halt** message. Complete test results will be reported whether the test passed or failed.

The *loops* parameter indicates the number of loops performed by the test. The *faults* parameter indicates the number of faults detected by the test. When the maximum value is reported for either the *loops* or *faults* parameters it indicates that at least that many loops or faults were run or detected. The *fault_id* is an indicator of the type of fault found and is combined with the *block_id*, *area_id* and *routine_id* values to form an index for reporting purposes. A *fault_id* of '0' indicates no errors were detected and the routine passed its tests. A *fault_id* of '1' - '9' or 'A' - 'F' indicates a failure. A '?' indicates an option that is not found. A ' ' (blank) indicates a test that returns no results. The *field* parameters sent with a ' ' (blank) *fault-id* have no meaning.

For minmax type routines which are indicated by the **minmax_int**, **minmax_long** and **minmax_float** area tokens, the *field1* parameter indicates the minimum expected result, the *field2* parameter indicates the maximum expected result and the *field3* parameter indicates the actual measured result. For digital type routines which are indicated by the **digital_int**, **digital_long**, **digital_mix** and **digital_short** area tokens, the *field1* parameter indicates an address, the *field2* parameter indicates the expected result and the *field3* parameter indicates the actual result. For **digital_int** type routines, all the *field* parameters are unsigned integers. For **digital_long** routines, all the *field* parameters are unsigned long integers. For **digital_mix** routines, the address parameter (*field1*) is unsigned long and the expected and actual parameters (*field2* and *field3*) are unsigned integers. For **digital_short** routines, the address parameter (*field1*) is an unsigned integer and the expected and actual parameters (*field2* and *field3*) are unsigned short integers. Displays of all types of values may be limited by the mainframe to 5 characters.

4.2.2 Display and Trigger Signal Paths

In diagnostic mode, the display and trigger signal path tests will be selectable from a mainframe diagnostic menu. The interface to the plug-in will be identical in the start-up test or diagnostic modes. Plug-ins will do limited testing of these paths during their own diagnostics.

4.2.3 Serial Data Interface Test

The following message may be sent during powerup or diagnostic mode to test the SDI communication between the mainframe and the plug-in.

diag com_test pattern
diag com_test: message tokens
pattern: special

The *pattern* sent by the mainframe in this test is the following sequence of 16 bytes: FF, 00, 01, 02, 04, 08, 10, 20, 40, 80, 55, AA, 33, CC, DD, BB. The plug-in will echo this message with all bytes inverted. Normal protocol handshaking will be observed during this test. The plug-in will check the sequence number and checksum of the **diag com_test** message and send an ACK (as normal) if both are correct.

The only other way to determine fault (plug-in or mainframe) for the SDI communication path is to request the user to turn the power off and change the plug-ins. The user must be given instructions for this procedure when the serial data interface test has failed. This sequence has no impact on the software interface.

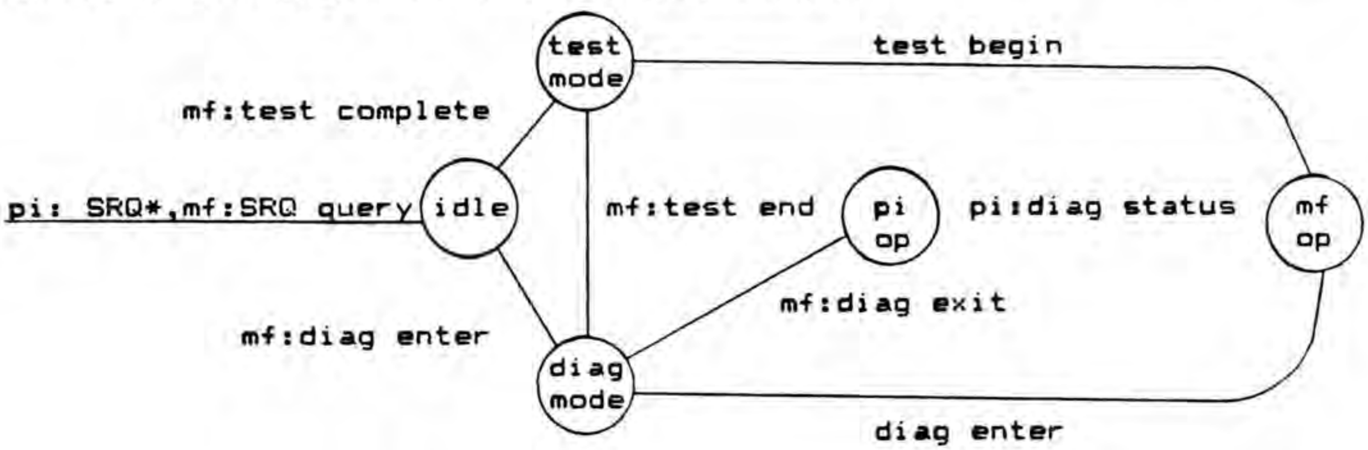
4.2.4 Auxiliary Trigger Line Diagnostics

Plug-Ins that use the auxiliary trigger lines will provide a menu selection for running the auxiliary trigger line tests. When this test is selected and the **diag exec** command is given, the right plug-in will test the auxiliary trigger lines as defined in section **4.1.3 Auxiliary Trigger Line Tests.**

To assist in locating a fault, the user may remove the right plug-in and run the display and trigger signal path tests. These tests also exercise the auxiliary trigger signal lines. The user can then determine the location of a fault by observing the activity during these tests.

4.3 Summary

The following chart shows the possible sequences of events that are possible involving the self test and diagnostic modes both at power up and during normal operations. It also shows the messages that are used to change from one mode to another.



The mainframe will determine system configuration during the idle state and report that information to all smart plug-ins.

Following are some example message sequences that illustrate a typical sequence of events for testing and diagnostics between a mainframe and a single plug-in. SRQ messages marked with an asterisk (*) are transport level SRQ packets and are not application level messages. This sequence shows the messages for a smart plug-in in the right compartment.

mainframe	message	plug-in
power on		power on
	<---SRQ*---	ready
performs internal tests		
responds to SRQ	---query SRQ--->	
	<---plugin_config status---	
ready for testing, connects cvr	---test complete--->	
	<---test status busy---	begins testing
	<---cal set_cvr---	requests cal volts
sets cal volts	---cal cvr_ready--->	
	<---cal meas_req---	request meas
makes measurement	---cal meas_status--->	
.	.	.
.	.	.
.	.	.
	<---cal cvr_connect---	right tests aux line
sends to generic	---cal cvr_mode--->	generic sets mode
	<---cal cvr_mode---	reports status
sends to right	---cal cvr_connect--->	right tests auxiliary trigger lines
	<---cal cvr_connect---	right done
sends to generic	---cal cvr_mode--->	generic sets mode
	<---cal cvr_mode---	reports status
sends to right	---cal cvr_connect--->	
.	.	repeated for other
.	.	generic plugin
.	.	
	<---test status---	completed tests
all tests are done	---test end--->	enters diag mode,
	<---diag status---	reports status
no errors reported,		
leave diag mode	---diag exit--->	enters normal mode,
enters normal mode	<---diag status---	sends status

In the following example, the sequence shown above is followed but instead of exiting the diagnostic mode immediately, the mainframe requests diagnostic configuration information and executes some diagnostic tests.

mainframe	message	plug-in
.	.	.
.	.	.
all tests are done	---test end--->	enters diag mode,
	<---diag status---	reports status
requests config info	---diag request--->	
	<---diag config more---	sends info
	---diag config ack--->	
	<---diag config more---	
	---diag config ack--->	
	<---diag config last---	
	---diag config ack--->	
execute test	---diag exec--->	runs test
	<---diag status routine---	reports results
execute test	---diag exec--->	runs test
halt test	---diag halt--->	halts test
	<---diag status routine---	reports results
end diag mode	---diag exit--->	enters normal mode,
enter normal mode	<---diag status---	sends status

5.0 Calibration

The purpose of the calibration function is to provide the user with an instrument that meets tight accuracy specifications and does not require expensive time-invariant electrical circuits to maintain that accuracy.

The calibration process uses a voltage standard provided by the mainframes. The Calibrated Voltage Reference circuit (CVR) to supplies accurate voltages to the circuits to be calibrated. The outputs of these circuits are measured by the mainframe's D to A converter (which may be the standard digitizer for a digitizing mainframe or a special converter for an analog mainframe). Each device in the instrument system requests voltages and measurements in order to characterize circuits for which that device has a calibration adjustment control. Changes to the control are made from calculations based on a series of voltage requests and measurements.

The techniques used to achieve the desired accuracy for each signal path are specified in the *11000 Series Accuracy System* specification. System accuracy specifications and requirements are also specified in that document.

This section of the Software Interface specification defines calibration messages and specifies how they are used to perform and maintain calibration.

5.1 Configuration Determination

The 11000 Series Accuracy System uses configuration information to determine calibration needs. System configuration in terms of mainframes and plug-ins is determined at power up. Configuration information concerning the probes is also determined at power up and is updated any time thereafter a probe change is detected by a plug-in. This information is used to determine calibration requirements and to define entry and exit criteria for the enhanced and normal accuracy modes.

Details on the process of determining the calibration mode, sequences of events and the requirements for entry and exit of each mode are defined in the *11000 Series Accuracy System* specification.

5.1.1 System Configuration

In order to maintain system accuracy while not re-calibrating components that do not need it, the 11000 Series Accuracy System uses system configuration information to determine whether any device in the system must be calibrated after the warm-up time. If the mainframe and plug-in system has not changed since the power was turned off, no calibration is required. If any of the plug-ins are new to the mainframe, those plug-ins must be calibrated. Mainframe and plug-in type and unit id number are used to determine configuration changes.

See section **3.1 Startup Sequence** for details on the the sequence of events during power up and at what point the configuration messages are sent.

mf_id report type uid version level
mf_id report: message tokens
type: string - indicates the mainframe type
uid: string - indicates the mainframe's unit id
version: string - indicates the mainframe's software version
level: token - indicates mainframe functional level

This message is sent by the mainframe during the power up sequence to notify the plug-in of its capabilities and identification. The *type* parameter indicates the mainframe type and is a string of up to 10 characters specifying the mainframe type using the 11000 series nomenclature. The *uid* parameter is the unit identification number of the mainframe. This should always match the serial number of the mainframe. The *version* parameter specifies the software version for that mainframe. The *level* parameter specifies the level of functionality supported by that mainframe.

The plug-in will use the *type* and *uid* parameters to determine whether it is installed in the same or a different mainframe than during the last power-up. The plug-in will compare these values to those saved in non-volatile memory. The plug-in will report the results of this comparison in the **mf_id config** message then save the new values in non-volatile memory. The *version* and *level* parameters are provided to notify smart plug-ins of specific capabilities that a particular mainframe may have. They are used in conjunction with the **mainframe message** message. See section **2.6.3 System**.

mf_id config status
mf_id config: message tokens
status: token indicates plug-in status

This message is sent by the plug-in in response to the **mf_id report** message. It is used to report whether the plug-in is installed in the same mainframe as during the previous power on or whether the mainframe is new to the plug-in. The **old** *status* token indicates the plug-in is installed in the same mainframe as previously. The **new** *status* token indicates the plug-in is installed in a new mainframe and may require calibration.

5.1.2 Probe Configuration

Generic plug-ins will inform the mainframe each time a change is detected in the probes attached to that plug-in. The mainframe will use this information in determining which probes need to be calibrated in order to maintain or enter the enhanced calibration mode. The user will be notified using appropriate means which probes require calibration. Separate messages defined in the following sections are used to cause the plug-in to perform probe calibration and report the results of that calibration.

probe status channel input level [type uid]
probe status: message tokens
channel: us - specifies the channel
input: token - specifies which input
level: us - specifies the probe level
type: string - specifies the probe type
uid: string - specifies the probe unit id

This message is sent by the plug-in whenever it detects a change (either addition or removal) of a probe. The *channel* parameter specifies the channel for which the plug-in is reporting a change. The *input* parameter specifies which input is being reported for each channel. The **plus** *input* token indicates that status is being reported for the input of a single ended channel or the plus input of a differential channel. The **minus** *input* token indicates the status is being reported for the minus input of a differential channel only. The *level* parameter specifies the probe level detected by the plug-in. Level 0 specifies no probe is detected. Level 1 probes use resistive encoding and provide no information other than the probe attenuation factor. Level 2 probes provide additional information about probe parameters including probe type and probe uid. The *type* parameter specifies the probe type reported by a level 2 probe using the Tekprobe nomenclature. The *uid* parameter specifies the unit id reported by a level 2 probe. The *type* and *uid* parameters are only included in this message when the probe level is 2.

Since the plug-in cannot differentiate between no probe connected and a level 1 1x probe, the plug-in will report level 0 for both cases. The mainframe may allow the user to identify those channels that have 1x probes. The mainframe will not report this information to the plug-in.

probe query channel
probe query: message tokens
channel: us - specifies the channel

This message is sent by the mainframe to request the present probe status of the specified *channel*. The plug-in will send the **probe status** message as the response. For differential channels, two **probe status** messages will be sent, one for the probe on the minus input and one for the probe on the plus input. The plus **probe status** message will always be sent last.

5.2 Calibration Messages

This section defines the messages that are used to control the operation of the calibration functions.

busy query
busy query: message tokens

This message is sent by the mainframe only to smart plug-ins to determine if the calibration process may be started. It may be sent any time during normal operation. The plug-in will reply with the following message:

busy status status
busy status: message tokens
status: token – indicates plug-in status

This message indicates smart plug-in busy status. The **idle** *status* token indicates the plug-in is not busy and the calibration process may be started. The **busy** *status* token indicates the plug-in is in a busy state and calibration must wait. If a plug-in reports its status as **busy**, it is required that the plug-in subsequently enter a state that will allow calibration to be performed and will immediately notify the mainframe. The plug-in will send a plug-in SRQ then send the **busy status** message with the **idle** *status* token in response to the **SRQ query** message.

When the smart plug-in has sent the **busy status idle** message, it must not start any other operations. Therefore, all front panel buttons will be ignored. The plug-in will remove any menus that are displayed and release any mainframe functions (such as cursors or the knobs) prior to sending the **busy status idle** message. The only valid message that the mainframe may send to a smart plug-in after that plug-in has sent the **busy status idle** message is **cal mode** (see below).

Generic plug-ins are not sent the **busy query** message. Therefore, by definition, generic plug-ins may not have a "busy" mode. Generic plug-ins are sent the **cal mode** message directly.

To begin calibration, the mainframe will notify each plug-in using the **cal mode** message that the calibration process is beginning. This is to insure that all other activity is halted and only calibration messages will be transmitted between the mainframe and plug-ins (see **busy status** above).

cal mode mode [meas_opt] [compartment [compartment]]
cal mode: message tokens
mode: token – specifies the calibration mode
meas_opt: float – specifies the optimal measurement band
compartment: char – specifies the new configuration

This message selects the mode of operation for calibration. The selection of calibration modes is determined by the mainframe based on the needs of the system.

The **enter** *mode* token specifies the beginning of the calibration process. The mainframe will send the **cal mode enter** message to each plug-in at the beginning of full calibration. This token defines the beginning of *Initial Calibration Mode*. See section 5.3.2.

The **hold** *mode* token also defines the beginning of calibration mode but is sent during new plug-in configuration calibration or during probe calibration to plug-ins that do not need to be calibrated. This token defines the beginning of *Hold Calibration Mode* defined in section 5.3.3.

The **exit** *mode* token specifies the end of the calibration process. See section 5.3.9.

When the calibration process is begun, the the first **cal mode** message must contain an **enter** or **hold** *mode* token. When the calibration process is completed, the last **cal mode** message must contain the **exit** *mode* token.

The **dp_cal_woc** *mode* token specifies the beginning of *Display Path Calibration Mode - Without CVR*. See section 5.3.4 for details of this mode. The **dp_cal_wc** mode token specifies the beginning of *Display Path Calibration Mode - With CVR*. See section 5.3.5 for details of this mode. The **dtp_imb_woc** *mode* token specifies the beginning of *Display/Trigger Imbalance Determination Mode*. See section 5.3.6 for details of this mode. These tokens are only sent to plug-ins that support the generic interface.

The **new_config** *mode* token is only sent to smart plug-ins during new configuration calibration after instrument warmup. The *compartment* parameters are included in the **cal mode** message only when the *mode* token is **new_config**. The *compartment* parameters specify the compartments that contain plug-ins that are new since the last calibration. See section 5.3.7 for details of this calibration mode.

The **other** *mode* token specifies the beginning of time allowed for a smart plug-in to perform calibration not related to the display or trigger paths. See section 5.3.8 for details of events during the *Other Calibration Mode*.

The **special** mode token causes a plug-in to begin the calibration of special functions. These are functions that require calibration based on criteria other than the normal system criteria. See section **5.4.7 Special Plug-in Calibration** for the details of this procedure.

The *meas_opt* parameter is included when the *mode* token is **dp_cal_woc**, **dp_cal_wc**, **dtp_imb_woc**, **new_config** or **other**. The *meas_opt* parameter specifies the optimal measurement band for the mainframe in divisions from center screen. Plug-ins will make adjustments as required to cause measurements to be made within 5% of the *meas_opt* value specified by the mainframe. The value sent by the mainframe will always be positive. The measurement bands must be symmetrical around center screen. Different bands may be specified for each of the modes (**dp_cal_woc**, **dp_cal_wc**, **dtp_imb_woc**, **new_config** and **other**).

cal status status [channels]
cal status: message tokens
status: token - indicates status
channels: special - indicates failed channel

This message is sent by a plug-in in response to the **cal mode** message and other calibration messages. The plug-in will send the

ready *status* token when the mainframe sends the **cal mode enter** or **cal mode hold** messages. In these cases, the **cal status** message indicates the plug-in has entered the requested calibration mode and is ready to perform calibration. The plug-in will send the **busy** *status* token when the mainframe sends the **cal mode** message with the **dp_cal_woc**, **dp_cal_wc**, **dtp_imb_woc**, **new_config** or **other** *mode* tokens. In these cases, the **cal status** message indicates the plug-in has begun the procedure requested by the mainframe. The mainframe will not send any further unsolicited messages to the plug-in until it receives the **cal status complete** message.

When the plug-in has completed any of these calibration modes, it will send the **cal status** message. The **complete** *status* token indicates the plug-in has completed the specified procedure. If the plug-in has detected an error in the calibration process, it will send the **fail** or **fail_chan** *status* token. The **fail** *status* token indicates the failure was not related to a channel. The **fail_chan** *status* token indicates the failure was related to a channel. In this case, the plug-in will also send the *channels* parameter to indicate which channel or channels failed calibration. The *channels* parameter has the following format:

b7	b6	b5	b4	b3	b2	b1	b0
X	X	X	X	CH4	CH3	CH2	CH1

If a bit is set, the indicated channel failed calibration.

The mainframe will send the **cal status** message with the **ready** *status* token in response to the **cal mf_imbalance** and **cal pi_imbalance** messages from the plug-ins.

The plug-in will report mainframe and plug-in imbalances that it measures during some modes of calibration. The following messages are used to report those values.

cal mf_imbalance mf_value
cal mf_imbalance: message tokens
mf_value: float - mainframe imbalance value

This message is sent by a generic plug-in to report the mainframe input imbalance value, *mf_value*. This value will be the display or trigger path input imbalance as determined by the calibration mode. The plug-in will measure and report the display or trigger path input imbalance during the calibration modes defined below.

cal pi_imbalance {channel pre_invert post_invert}...
cal mf_imbalance: message tokens
channel: us - specifies the channel
pre_invert: float - pre-inversion imbalance value
post_invert: float - post-inversion imbalance value

This message is sent by the plug-in to report plug-in channel output imbalances. These values will apply to either the display or trigger outputs as determined by the calibration mode in which the message is sent. The *channel* parameter specifies the channel for which the following imbalance values apply. The *pre_invert* parameter gives the imbalance value that occurs prior to the inversion circuit. The *post_invert* parameter gives the imbalance value that occurs after the inversion circuit. The plug-in may report all channels in one message or it may report each channel in a separate message. If a channel is determined to be non-functional, the plug-in is not required to report imbalance values for that channel.

All values reported by the plug-in (*mf_value*, *pre_invert*, *post_invert*) are in divisions from center screen.

5.3 Calibration Modes

The calibration process is coordinated between plug-ins by the mainframe. Since there is only one voltage reference circuit, only one plug-in may request voltages at a time. The CVR is assigned to one compartment at a time. See section **5.5 Calibrated Voltage Reference** for details on controlling the assignment of the CVR. The mainframe must also perform some internal calibration prior to allowing plug-ins to perform calibration.

The mainframe's measurement capability may be time-shared between plug-ins on demand from the plug-ins. This allows all the plug-ins to perform certain parts of the calibration process in parallel with other parts.

The calibration process is performed as a series of modes. These modes define the state of the interface between the plug-in and the mainframe and most may be thought of as plug-in modes. The *Initial Calibration Mode* and the *Final Calibration Mode* are system modes. Since the mainframe operates up to three plug-ins during calibration, the mainframe may place each of those plug-ins in different modes as required. The procedures discussed in section **5.4 Calibration Procedures** show how the different modes are combined to perform the appropriate calibration procedures.

In each mode, certain requirements are made of both the plug-ins and the mainframe. Each mode has limitations on what is expected and available from plug-ins and the mainframe. Each of the modes is defined below. Entry and exit criteria and limitations and expectations of each mode are defined.

During these modes, the plug-ins will not respond to front panel operations. Probe id and channel id button pushes will not be reported to the mainframe except as required by calibration processes. Smart plug-ins may not request any functions of the mainframe except as defined for each calibration mode. The mainframe or plug-ins will not send messages other than those defined for calibration.

5.3.1 Entering Calibration Mode

The 11000 Series Accuracy System determines when calibration is required and which devices in the system must be calibrated. This determination is based on changes in system configuration, time and temperature changes or input from the user and is specified in the *11000 Series Accuracy System* document.

Before calibration mode can be entered, the mainframe must determine whether a smart plug-in is busy. Busy is defined as a state which, if interrupted by the calibration process, may cause the system to lose information. It is up to the plug-in designer to determine which plug-in states will result in the plug-in reporting busy status and delay the operation of calibration.

The mainframe will use the **busy query** message to determine the state of smart plug-ins. When all smart plug-ins have sent the **busy status idle** message, the mainframe will send the **cal mode** message with the appropriate *mode* tokens to each plug-in to begin the calibration process.

5.3.2 Initial Calibration Mode

The mainframe must calibrate its measurement circuits prior to allowing signal path calibration by the plug-in. Since this calibration process is entirely internal to the mainframe, there are no messages defined for the interface to support it. This mode is entered when the mainframe sends a **cal mode enter** message to each plug-in. This mode is terminated when the mainframe places the plug-ins into any of the calibration modes defined below.

When a plug-in receives the **cal mode enter** message, it may begin any internal calibration that does not required the use of the CVR or the voltage measurement circuits in the mainframe. The plug-in must be able to respond to incoming messages from the mainframe. Generic plug-ins must turn their display and trigger outputs off.

5.3.3 Hold Calibration Mode

This mode is entered when the mainframe sends the **cal mode hold** message to a plug-in. That plug-in will enter calibration mode with the limitations as defined for *Initial Calibration Mode* but will not perform any calibration procedures. This mode is provided for new configuration or probe calibration to cause a plug-in not involved with the required calibration to be in a defined quiet state. This mode is terminated when the mainframe sends the **cal mode** message with the **exit** *mode* token.

In the case of new configuration calibration, the mainframe will send the **cal mode enter** message to the plug-in or plug-ins requiring calibration and the **cal mode hold** message to all other plug-ins. The plug-ins receiving the **cal mode enter** message will begin calibration. Those receiving the **cal mode hold** message will enter calibration mode (and be ready to accept calibration commands) but will not perform any calibration procedures.

In the case of probe calibration (when other calibration is not required), the mainframe will send the **cal probe begin** message to the plug-in required to calibrate a probe and the **cal mode hold** message to all other plug-ins. See section **5.4.5 Probe Calibration** for more information on probe calibration.

5.3.4 Display Path Calibration - Without CVR

This mode is provided for generic plug-ins to begin display path calibration. This mode is not valid for smart plug-ins that do not support the generic plug-in interface. In this mode, the plug-in may request measurements from the mainframe but may not request voltages from the CVR. This mode allows the plug-ins to perform calibration calculations in parallel. The mainframe will time share the measurement facility as requested by the plug-ins. This mode may be run concurrently with any other calibration mode.

This mode is entered when the mainframe sends the **cal mode dp_cal_woc** message. During this mode, the plug-in is allowed to request measurements using the **cal measure** message without requiring the use of the plug-in SRQ. The mainframe is not allowed to send any unsolicited messages to the plug-in during this time. Four messages from the plug-in are valid during this time: **cal measure, cal meas_set, cal mf_imbalance** and **cal status**. This mode is terminated when the plug-in sends the **cal status** message with the **complete, fail** or **fail_chan** *status* token.

In this mode, the calibration process determines the mainframe input imbalance measured by the plug-in in the display signal path. This value is reported to the mainframe using the **cal mf_imbalance** message.

5.3.5 Display Path Calibration - With CVR

In this mode, the plug-in will complete the calibration process of the display path. It determines calibration constants that will be used by the plug-in for correcting gain errors for the entire display path (to the mainframe measurement point) and constants for the plug-in to use in correcting internal imbalances. All functions related to the signal path (gain, offset, bandwidth limits etc.) are calibrated and adjusted during this mode. The plug-in is assigned the CVR and may request voltages from it to perform the calibration. All generic plug-ins to be fully calibrated must be placed in the *Display Path Calibration Mode - With CVR* during calibration. This mode is not valid for smart plug-ins that do not support the generic plug-in interface. No calibration constants are reported to the mainframe during this mode. This mode may not be requested by the mainframe of one plug-in concurrently with any other mode in another plug-in that uses the CVR.

This mode is entered when the mainframe sends the **cal mode dp_cal_wc** message. During this mode, the plug-in is allowed to request measurements and voltages using the **cal measure, cal meas_set** and **cal cvr_set** messages without requiring the use of the plug-in SRQ. The mainframe is not allowed to send any unsolicited messages to the plug-in during this time. Only four messages from the plug-in are valid during this time: **cal measure, cal meas_set, cal cvr_set** and **cal**

status. This mode is terminated when the plug-in sends the **cal status** message with the **complete**, **fail** or **fail_chan** *status* token.

5.3.6 Display/Trigger Imbalance Determination Mode

In this mode, the plug-in will report the imbalances it measures in the display or trigger signal paths but will make no internal adjustments to compensate for those errors. The plug-in will report the mainframe input imbalance and imbalance values for each channel using the **cal mf_imbalance** and **cal pi_imbalance** messages defined above. This mode is not valid for smart plug-ins that do not support the generic plug-in interface. The plug-in may not request the use of the CVR during this mode. This mode may be run concurrently with any other calibration mode.

In this mode, each generic plug-in will request measurements and perform calculations on the display or trigger path imbalances. This mode may be run concurrently with any other calibration mode. Since measurements are required to perform the calculations, the mainframe will time share the measurement facility as requested by the plug-ins. The mainframe will determine to which path the results reported by the plug-in apply by selecting the path on which the measurements requested by the plug-in are made. The plug-in is not required to know which signal path is being measured. This mode may be repeated by the mainframe for each signal path in the mainframe.

This mode is entered when the mainframe sends the **cal mode dtp_imb_woc** message. During this mode, the plug-in is allowed to request measurements using the **cal measure** message without requiring the use of the plug-in SRQ. The mainframe is not allowed to send any unsolicited messages to the plug-in during this time. Five messages from the plug-in are valid during this time: **cal measure**, **cal meas_set**, **cal mf_imbalance**, **cal pi_imbalance** and **cal status**. This mode is terminated when the plug-in sends the **cal status** message with the **complete**, **fail** or **fail_chan** *status* token. All generic plug-ins to be calibrated must be placed in the *Display/Trigger Imbalance Determination Mode* during calibration for at least one display and one trigger path.

5.3.7 New Configuration Calibration Mode

This mode is provided for smart plug-ins that must perform calibration when another plug-in is new to the system. This is the case for a triggering plug-in that uses the auxiliary trigger lines when a new generic plug-in has been added to the system.

This mode is entered when the mainframe sends the **cal mode** message with the **new_config** *mode* token. The mainframe will assign the CVR to the plug-in when it sends this message. This mode is not valid for generic plug-ins. This mode may not be run concurrently with any other calibration mode that requires the use of the CVR.

During this mode, the smart plug-in is allowed to request voltages and measurements from the mainframe and is also allowed to control generic plug-ins in the system using the **generic command** message. The plug-in may also use the **cal cvr_connect** message to assign the CVR to one of the generic plug-ins in the system. All of these messages may be sent without requiring the use of the plug-in SRQ. The mainframe is not allowed to send any unsolicited messages during this mode. Valid messages from the plug-in are: **cal cvr_set**, **cal measure**, **cal meas_set**, **cal cvr_connect**, **generic command** and **cal status complete**. This mode is terminated when the plug-in sends the **cal status** message with the **complete** or **fail** *status* token.

5.3.8 Other Calibration Mode

This mode is provided to allow smart plug-ins to complete calibration of non-signal path related functions. The plug-in is assigned the CVR and may request measurements and voltages. A triggering plug-in that uses the auxiliary trigger lines and is new to the system will calibrate those lines in this mode. See section **5.4.4 Auxiliary Trigger Calibration**. This mode is not valid for generic plug-ins. This mode may not be run concurrently with any other calibration mode that uses the CVR.

This mode is entered when the mainframe sends the **cal mode other** message. During this mode, the plug-in is allowed to request measurements and voltages using the **cal measure**, **cal meas_set** and **cal cvr_set** messages. Smart plug-ins may also send the **cal cvr_connect** and **generic command** messages to control generic plug-ins. All of these messages may be sent without requiring the use of the plug-in SRQ. The mainframe is not allowed to send any unsolicited messages to the plug-in during this time. Valid messages from the plug-in during this time are: **cal measure**, **cal meas_set**, **cal cvr_set**, **cal cvr_connect**, **generic command** and **cal status complete**. This mode is terminated when the plug-in sends the **cal status** message with the **complete** or **fail** *status* token. This mode can be entered only after the display path calibration with CVR has been completed.

5.3.9 Final Calibration Mode

This mode is entered by default when all other appropriate calibration modes have been completed. In this mode, the mainframe may use the calibrated display and trigger signal paths to complete calibration of its internal circuits. The mainframe may also request calibration of probes and may request the trigger view function to be calibrated. See sections **5.4.5 Probe Calibration** and **5.4.6 Trigger View Calibration**. Smart plug-ins that use the auxiliary trigger lines will calibrate those lines during this mode.

The mainframe may use the **cal cvr_mode** message to cause a generic plug-in to enter CVR pass-through mode (see section **5.3.10 CVR Pass-Through Mode**). The mainframe may use the standard set of generic plug-in commands to control generic plug-in functions. Plug-ins may not send any unsolicited messages during this mode. This mode is terminated (and calibration in general) when the mainframe sends the **cal mode** message with the **exit** *mode* token to all plug-ins.

When the **exit** *mode* token is received, plug-ins will exit from the calibration mode. They resume scanning of their front panels and will respond to all normal operating commands.

5.3.10 CVR Pass-Though Mode

This mode is provided to allow the mainframe and smart plug-ins to complete calibration of their internal circuits by using a generic plug-in to pass the reference voltage through to its outputs.

cal cvr_mode mode [gain]
cal cvr_mode: message tokens
mode: token - specifies the cvr mode
gain: float - generic plug-in gain setting

This message is sent by the mainframe to a generic plug-in to cause it to connect all of its inputs to the CVR circuit rather than the external input. This allows the mainframe (or other plug-in) to make calibration measurements on one of the signal paths provided by the generic plug-in.

The **enter** *mode* token, when sent by the mainframe, requests the plug-in to enter CVR pass-through mode. The plug-in will connect its input to the CVR signal and will also adjust all of its functions to provide the optimal settings for performing calibration. The plug-in will save its present settings for the gain, offset, coupling, bandwidth, impedance, display and trigger functions for restoration when the CVR pass-through mode is exited.

Offset will be set to 0. The input will be DC coupled to the CVR. Bandwidth will be set to the optimal value for noise reduction in the plug-in. This bandwidth setting will not be so low as to affect the settling time for measurements. The input impedance will be set to a value appropriate for connection to the CVR. The display and trigger output combinations will be set to channel 1 only, not inverted. None of the settings of these functions will be reported.

The gain will be set to the optimal gain setting for that plug-in for calibration. The plug-in will report this gain value when it sends the **cal cvr_mode** message with the **enter** *mode* token. The mainframe or smart plug-in that requested the CVR pass-through mode will use this gain value and the default offset setting to perform calibration of their circuits.

The mainframe or a smart plug-in may request changes in the gain or offset functions or the trigger and display outputs of a generic plug-in in CVR pass-through mode only under the following conditions: The mainframe or smart plug-in must query the generic plug-in for its settings prior to placing the generic plug-in in CVR pass-through mode. After requesting the generic plug-in to enter CVR pass-through mode, the mainframe or smart plug-in may request changes in the gain or offset functions or the display and trigger outputs as long as the final request sets the plug-in to the settings that it had prior to entry of the CVR pass-through mode. This is required because the plug-in will restore the settings of these functions to the values set by the last function set command. The input coupling, bandwidth limit or input impedance functions should not be changed during CVR pass-through mode. They have no effect on the operation of this mode but will affect the settings when CVR pass-through mode is exited.

The **CVR** *mode* token requests the plug-in to connect its input to the CVR circuit. This will normally be sent after a **cal cvr_mode** message with the **ground** *mode* token to re-connect the input to the CVR. The plug-in will send the **cal cvr_mode** message with the **CVR** *mode* token when its input is connected to the CVR.

The **ground** *mode* token requests the plug-in to connect its input to reference ground. The plug-in will send the **cal cvr_mode** message with the **ground** *mode* token when its input is connected to ground.

The **exit** *mode* token requests the plug-in to exit the CVR pass-through mode. The plug-in will restore all function settings to the values that were active when CVR pass-through mode was entered (unless a change was requested during CVR pass-through mode). When the plug-in has completed the restoration and connected its input to the input signal, it will send the **cal cvr_mode** message with the **exit** *mode* token.

cal cvr_connect compartment mode [gain]
cal cvr_connect: message tokens
compartment: char - specifies the plug-in compartment
mode: token - specifies the connect mode

This message is sent by a smart plug-in to request the mainframe to connect the CVR to the specified *compartment* and to send a **cal cvr_mode** message to the plug-in in that compartment. The mainframe will send the **cal cvr_mode** message to the generic plug-in using the *mode* token sent in the **cal cvr_connect** message by the smart plug-in. The mainframe will wait for the generic plug-in to respond then report the results to the smart plug-in. When the generic plug-in reports its gain setting in response to a **cal cvr_mode enter** message, the mainframe will report that *gain* value to the smart plug-in in the **cal cvr_connect** message.

The **enter** *mode* token requests the generic plug-in to enter CVR pass-through mode as defined above. The plug-in will modify its settings as defined and connect its input to the CVR. The mainframe will wait for the response from the generic plug-in then report the mode and gain setting to the requesting smart plug-in.

The **CVR** *mode* token requests the input to be connected to the CVR circuit. This will normally be sent after a **cal cvr_connect** with the **ground** *mode* token to re-connect the input to the CVR.

The **ground** *mode* token requests the input to be connected to reference ground.

The **exit** *mode* token requests the mainframe to disconnect the ground reference from the specified compartment and send a **cal cvr_mode exit** message to the plug-in. The plug-in will exit the CVR pass-through mode and restore all function settings to the values that were active when CVR pass-through mode was entered (unless a change was requested during CVR pass-through mode). When the plug-in has completed the restoration and connected its input to the input signal, it will send the **cal cvr_mode** message with the **exit** *mode* token. The mainframe will then send the **cal cvr_connect** message with the **exit** *mode* token to the requesting smart plug-in.

The **cal cvr_connect** message may be sent after the smart plug-in has received the **cal mode** message with the **other** or **new_config** *mode* tokens and until the smart plug-in sends the **cal status** message indicating the calibration mode has been competed. The **cal cvr_connect** message may also be sent during diagnostic mode. See section **4.1.3 Auxiliary Trigger Line Test.**

5.4 Calibration Procedures

This section defines the calibration procedures for different calibration operations. This section also defines some of the procedures and messages associated with those procedures that are not covered explicitly by the calibration modes defined above.

5.4.1 Power-Up Calibration

Immediately after power-on diagnostics are completed, the mainframe will check the system configuration. If there are new generic plug-ins installed in the system, the mainframe will begin the power-up calibration procedure. The intent of this procedure is to cause newly installed generic plug-ins to correct for the imbalances of the new mainframe in which they are installed. This procedure also allows a smart plug-in that uses the auxiliary trigger lines to make adjustments for new generic auxiliary trigger outputs.

The mainframe will place all plug-ins that are not new to the system in the *Hold Calibration Mode* (**cal mode hold** message). The new generic plug-ins will be sent the **enter** *mode* token then placed in the *Display Path Calibration Mode - Without CVR* (**cal mode dp_cal_woc** message). In this mode, the plug-in will measure and report the mainframe input imbalance.

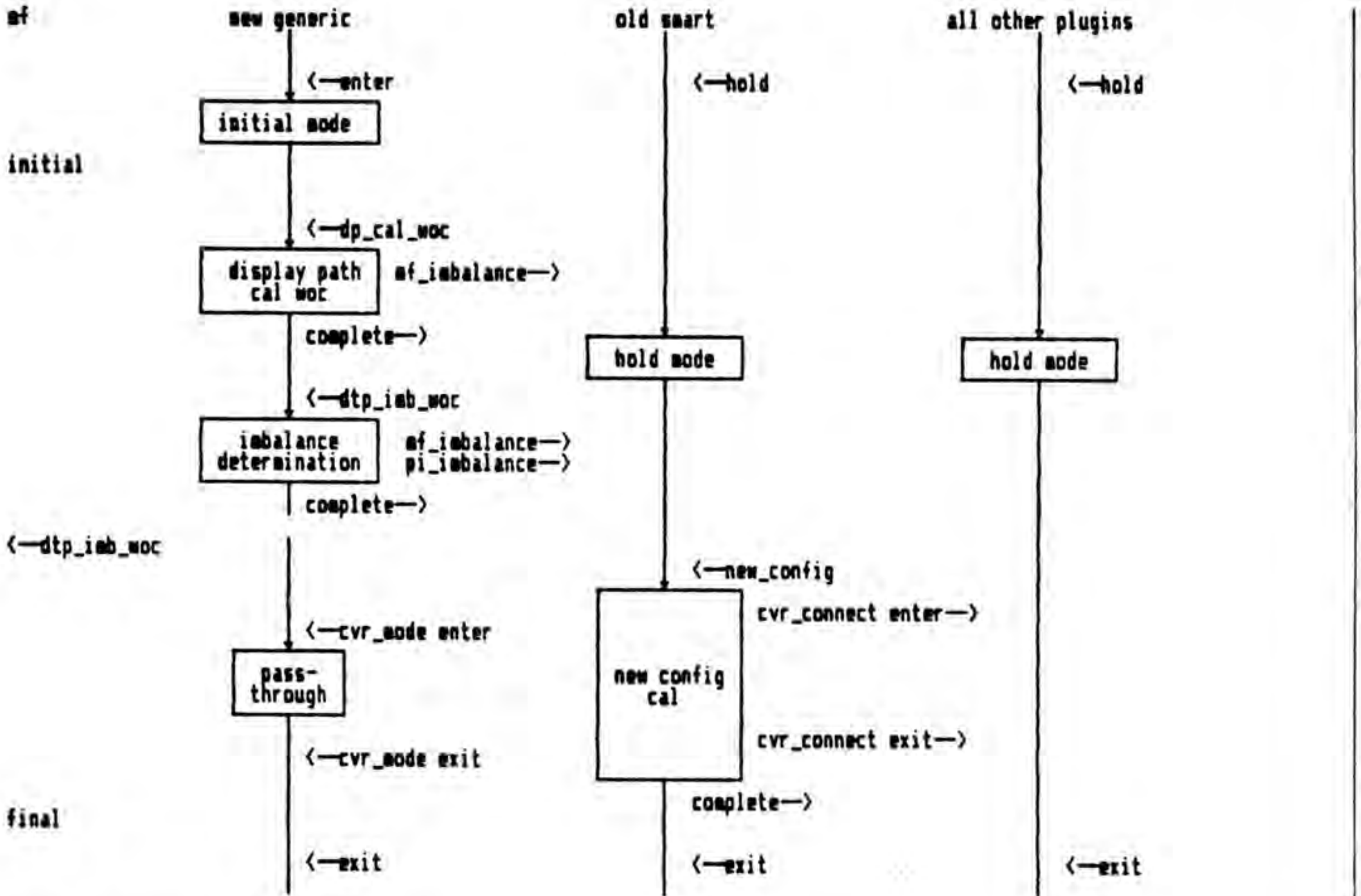
When each new generic plug-in has completed the *Display Path Calibration Mode - Without CVR*, the mainframe will place that plug-in into the *Display/Trigger Imbalance Determination Mode* (**cal mode dtp_imb_woc** message). The plug-in will measure and report plug-in channel output and mainframe input imbalances. The mainframe will repeat this mode for each of its display and trigger paths.

When new plug-ins have completed their calibration, smart plug-ins that are not new to the system will be placed in the *New Configuration Calibration Mode*. In this mode, the smart plug-ins may perform calibration related to the new plug-ins installed in the system (such as auxiliary trigger line calibration). Smart plug-ins will update only those constants that relate to the new plug-ins.

When all new generic plug-ins have completed the *Display Path Calibration Mode*, the mainframe will end the calibration procedure by sending the **cal mode** message with the **exit** *mode* token to all plug-ins. Generic plug-ins will make no changes to their internal calibration constants during this procedure except to correct for the mainframe's input imbalance.

The following chart shows the sequence of events for this calibration procedure. The tokens shown are either **cal mode** *mode* tokens (such as **enter** or **dp_cal_woc**) or are secondary message tokens (such a **mf_imbalance**). These labels show where the messages represented by these labels would occur in the sequence of events. Labels with a left arrow (<--) are sent by the mainframe. Labels with a right arrow (-->)are sent by the plug-in. The boxes in the chart represent modes defined above. Each box has a label that identifies the mode represented by the box. Note that after the imbalance determination mode, the mainframe may either move to the next mode or repeat the imbalance determination mode for another signal path (by sending the **cal mode dtp_imb_woc** message).

The column labelled "new generic" shows the sequence of modes in which the mainframe will place a generic plug-in that is new to the system during power-on calibration. The column labelled "old smart" shows the sequence of modes in which the mainframe will place a smart plug-in. The column labelled "all other plug-ins" shows the modes in which all other plug-ins will be operated during this procedure. The column labelled "mf" shows the mainframe mode.



5.4.2 Warm Calibration

This procedure is provided to cause a plug-in that is new to the system to be fully calibrated in the new system. This procedure is performed automatically at the end of the warm-up period unless auto-enhanced mode is enabled. See the *11000 Series Accuracy System* specification for details on when this procedure is run.

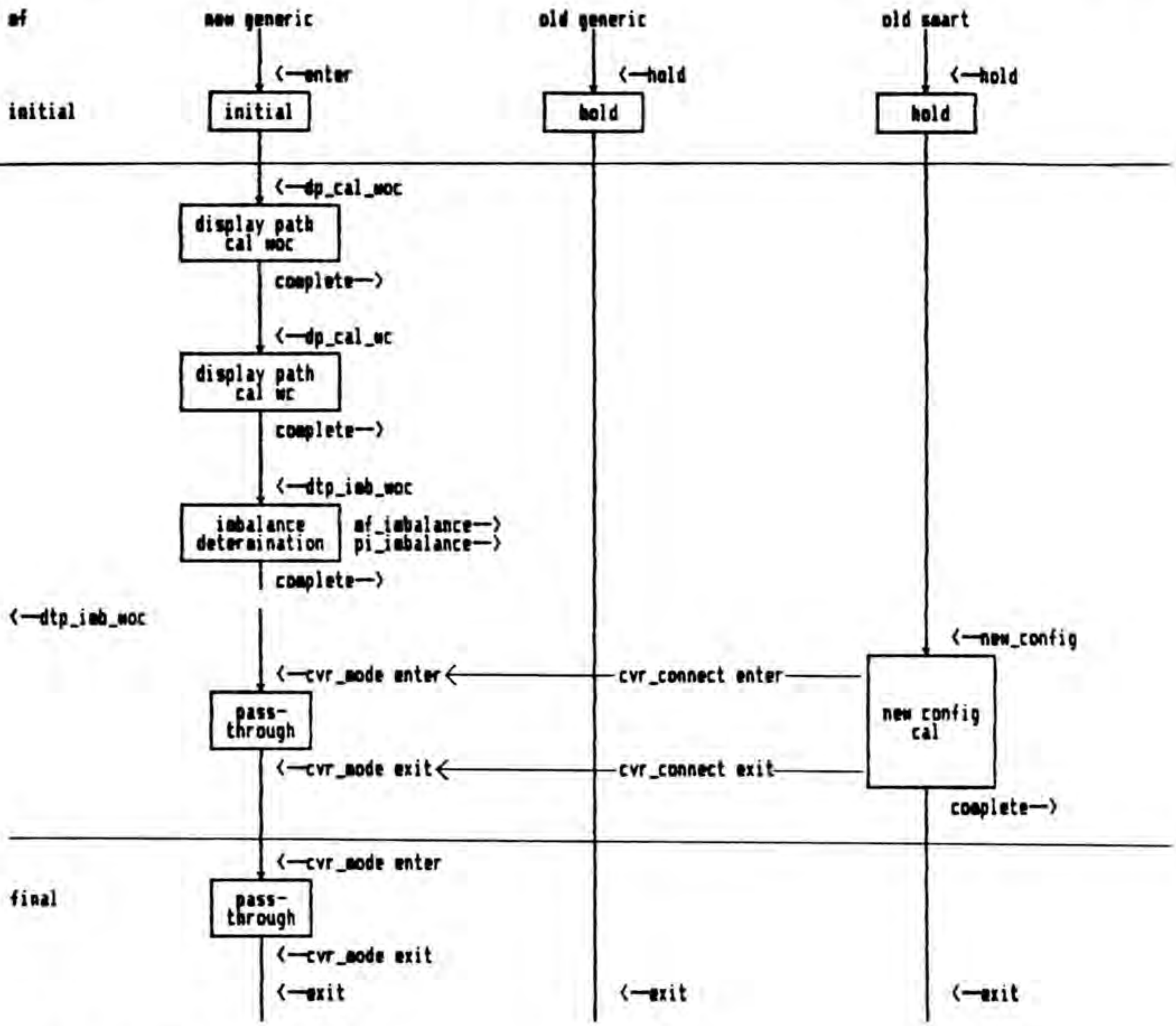
The mainframe will place all plug-ins that are not new to the system in the *Hold Calibration Mode*. Each new plug-in will be stepped through a full calibration procedure as defined in the next section.

When new plug-ins have completed their calibration, smart plug-ins that are not new to the system will be placed in the *New Configuration Calibration Mode*. In this mode, the smart plug-ins may perform calibration related to the new plug-ins installed in the system (such as auxiliary trigger line calibration). Smart plug-ins will update only those constants that relate to the new plug-ins.

When all smart plug-ins have completed their calibration, the mainframe will terminate this calibration process by sending the **cal mode** message with the **exit** *mode* token.

The following chart shows the sequence of events for this calibration procedure. The tokens shown are either **cal mode** *mode* tokens (such as **enter** or **dp_cal_woc**) or are secondary message tokens (such as **mf_imbalance**). These labels show where the messages represented by these labels would occur in the sequence of events. Labels with a left arrow (<—) are sent by the mainframe. Labels with a right arrow (—>) are sent by the plug-in. The boxes in the chart represent modes defined above. Each box has a label that identifies the mode represented by the box. Note that after the imbalance determination mode, the mainframe may either move to the next mode or repeat the imbalance determination mode for another signal path (by sending the **cal mode dtp_imb_woc** message).

The column labelled "new generic" shows the sequence of modes in which the mainframe will place a generic plug-in that is new to the system during the warm calibration procedure. The column labelled "old generic" shows the modes in which generic plug-ins that are not new to the system will be operated during this procedure. The "old smart" column shows the modes for smart plug-ins that are not new to the system. The column labelled "mf" shows the mainframe mode.



5.4.3 Full Calibration

The full calibration procedure allows the system to enter the enhanced accuracy mode by calibrating all functions in the system. The procedure is different for generic and smart plug-ins.

1. All plug-ins will be placed in the *Initial Calibration Mode* (**cal mode enter** message). The mainframe will perform calibration required to support measurement and CVR request and any other calibration needed to be done prior to signal path calibration.

2. When the mainframe has completed its initial calibration, all generic plug-ins will be placed in the *Display Path Calibration Mode - Without CVR* (**cal mode dp_cal_woc** message).

3. When the left-most generic plug-in completes the *Display Path Calibration Mode - Without CVR*, it will be placed in the *Display Path Calibration Mode - With CVR* (**cal mode dp_cal_wc** message). Other generic plug-ins will not be placed into this mode until all plug-ins in compartments to the left have completed this mode.

4. When all generic plug-ins have completed the *Display Path Calibration Mode*, each generic plug-in will be placed in the *Display/Trigger Imbalance Determination Mode* (**cal dtp_imb_woc** message). All generic plug-ins may operate in this mode concurrently.

5. When all generic plug-ins have completed the *Display/Trigger Imbalance Determination Mode*, each smart plug-in will be placed, one at a time, in the *Other Calibration Mode* (**cal mode other** message).

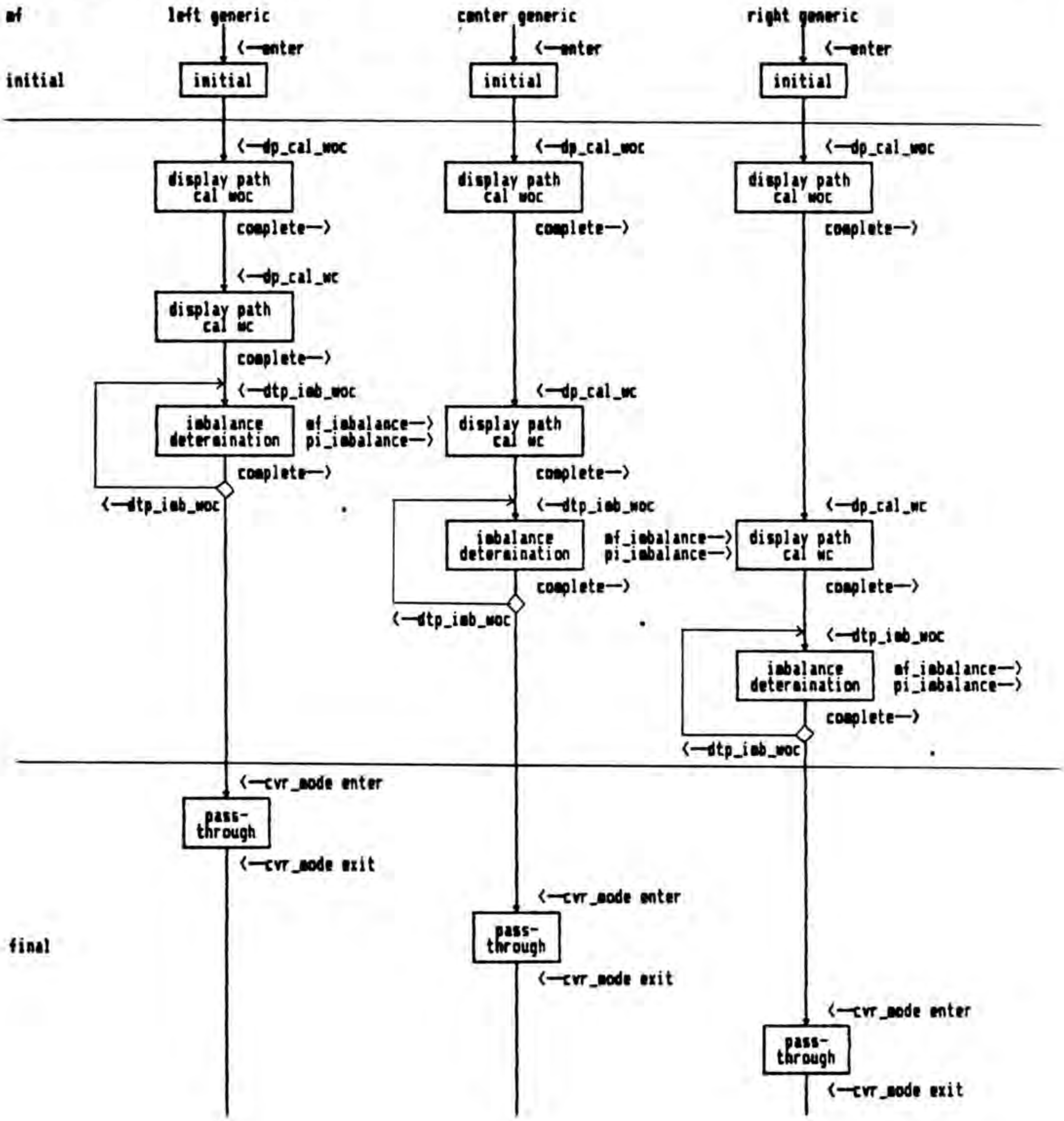
6. When all smart plug-ins have completed the *Other Calibration Mode*, the mainframe will enter the *Final Calibration Mode*.

7. When the mainframe has completed its final calibration, the calibration process is completed and the mainframe will terminate the calibration process for each plug-in.

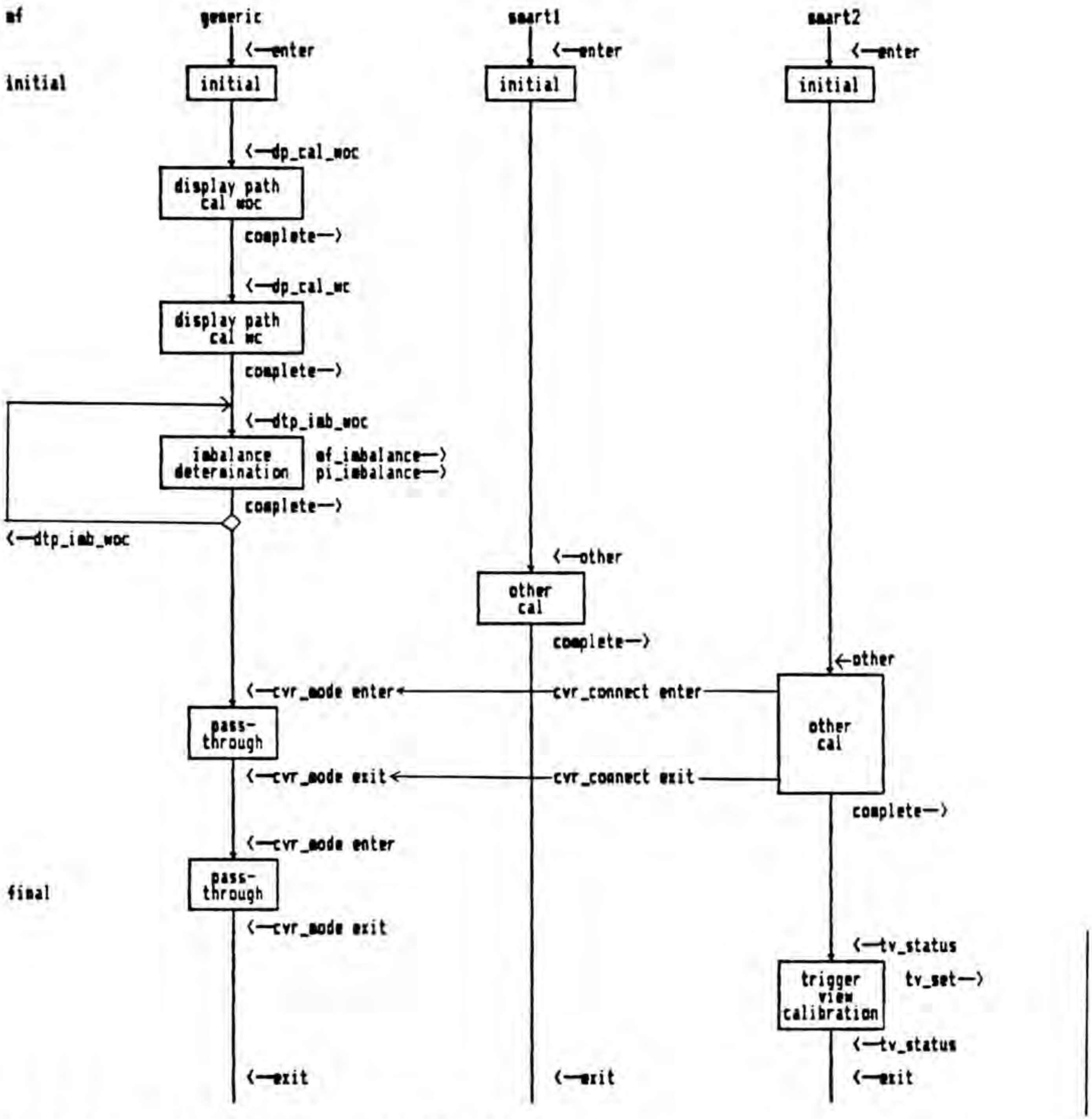
The following charts show examples of the sequences of events for different configurations of plug-ins for a full calibration procedure. The tokens shown are either **cal mode** *mode* tokens (such as **enter** or **dp_cal_woc**) or secondary message tokens (such as **mf_imbalance**). These labels show where the messages represented by these labels would occur in the sequence of events. Labels with a left arrow (<—) are sent by the mainframe. Labels with a right arrow (—>) are sent by the plug-in. The boxes in the chart represent modes defined above. Each box has a label that identifies the mode represented by the box. Note that after the imbalance determination mode, the mainframe may either move to the next mode or repeat the imbalance determination mode for another signal path (by sending the **cal mode dtp_imb_woc** message).

The columns labelled "left generic", "center generic", "right generic" show the sequence of modes in which the mainframe will place a generic plug-in in the left, center and right compartments, respectively. The column labelled "mf" shows the mainframe mode.

This first chart shows the case for a configuration containing three generic plug-ins:



This next chart shows the sequence for one generic plug-in and two smart plug-ins, one of which calibrates the auxiliary trigger lines:



5.4.4 Auxiliary Trigger Calibration

This procedure allows a triggering plug-in in the right compartment to determine the calibration constants required to achieve the desired accuracy in the auxiliary trigger signal paths.

Auxiliary trigger path calibration is performed by the right plug-in when that plug-in receives the **cal mode** message with the **other** or **new_config** *mode* tokens. As part of the auxiliary trigger calibration process, the right plug-in may request the CVR to be connected to the left or center plug-in (using the **cal cvr_connect** message) and request voltages to be sent through those plug-ins to the auxiliary trigger lines (using the **cal cvr_set** message).

The right plug-in may also control the display and offset and other input functions of the left and center plug-ins using the **generic command** message. Function status will be reported using the standard generic plug-in status messages. The right plug-in must restore settings to the original values when the procedure is completed. The initial values may be determined using the generic plug-in query messages.

The right plug-in must explicitly enable the auxiliary trigger outputs of generic plug-ins using the **aux_trig set** generic command message. See section **1.17 Auxiliary Triggers** for more information. This will normally be done during diagnostic testing after power-up. Generic plug-ins default to auxiliary trigger outputs off when they are powered-up.

5.4.5 Probe Calibration

This procedure is used to calibrate specified probes. The probe calibration procedure may be started by the mainframe any time during normal operation as requested by the user or it may begin probe calibration at the end of the full calibration process during the *Final Calibration Mode* as defined above.

To begin the probe calibration process from normal operating mode, the mainframe must place all plug-ins that will not be calibrating a probe into the *Hold Calibration Mode* using the **cal mode** message. See section **5.3.3 Hold Calibration Mode**. The mainframe will cause a plug-in to begin probe calibration using the following message:

cal probe status input channel
cal probe: message tokens
status: token - specifies requested mode or status
input: token - specifies the input
channel: us - specifies the channel

The **begin** *status* token requests the plug-in to begin calibration of the probe specified by the *channel* parameter. The plug-in will send the **cal probe** message with the **busy** *status* token to indicate it has started the probe calibration process. The **completed** *status* token is sent by the plug-in when it has successfully completed calibration of the requested probe. The **error** *status* token indicates the plug-in detected an error with the calibration process and was unable to complete calibration. The **connect** *status* token indicates the plug-in has determined the probe is possibly not connected to the calibrator output. In all cases, the plug-in reports the *channel* on which calibration was done or attempted. The **plus** *input* token specifies the plus input of a differential channel or the input of a single ended channel. The **minus** *input* token specifies the minus input of a differential channel only. The *channel* parameter specifies which channel is to be calibrated.

The mainframe will connect the calibrated voltage reference to the external calibrator connector before it sends the **cal probe** message. The plug-in will request voltages from the CVR using the **cal cvr_set** message. These voltages will be sent to the external connector through the probe to the plug-in. The plug-in will request measurements to be made by the mainframe and will calculate calibration constants as necessary for the probe.

Manual Probe Calibration

Mainframes are provided direct access to the calibration constants used by plug-ins to define probe characteristics. There are two constants that the plug-in uses: probe nominal and probe actual. The probe nominal calibration constant is the ideal attenuation value of the probe. This value will normally be reported by level 1 and level 2 probes. Normally, the plug-in will not modify this value. The probe actual calibration constant is the actual attenuation of the probe as measured by the plug-in. This measurement will normally be made during probe calibration.

These constants are updated whenever a probe is removed or installed. When a probe is installed, the plug-in will set both the nominal and actual calibration constants to the value reported by the probe using either coding technique (7K or 11K). The actual calibration constant will be updated as a result of performing the probe calibration procedure. When a probe is removed, both constants are set to 1.0.

To provide for unique situations for probes that do not use either coding method or for transducers with non-standard conversion factors, the mainframe may modify both probe calibration constants individually for each channel. The plug-in will modify these calibration constants based on the above events whether or not the mainframe has made modifications. The following messages are defined to give mainframes access to these constants.

cal probe_nom set input channel value
cal probe_nom: message tokens
set: token - specifies the set command
input: token - selects the input
channel: us - selects the channel
value: float - the cal constant value

This message sets the nominal probe calibration constant. The **plus** *input* token selects the plus input of a differential channel or the input of a single ended channel. The **minus** *input* token specifies the minus input of a differential channel only. The *channel* parameter specifies which channel's constant is to be updated. The *value* parameter specifies the new absolute value of the probe nominal calibraton constant. There are no incremental commands provided to change this constant. The range of this constant is limited only by the range of a single precision floating point number.

cal probe_nom query input channel
cal probe_nom: message tokens
query: token - specifies the query message
input: token - selects the input
channel: us - selects the channel

This message is sent to query the present value of the probe nominal calibration constant. The **plus** *input* token selects the plus inpout of a differential channel or the input of a single ended channel. The **minus** *input* token selects the minus input of a differential channel only. The *channel* parameter selects the channel for which the probe nominal value is requested.

cal probe_nom status input channel value
cal probe_nom: message tokens
status: token - specifies the status response
input: token - selects the input
channel: us - selects the channel
value: float - the value of the probe nominal cal constant

This message is sent by the plug-in to report the present status of the probe nominal calibration constant. It is sent in response to either the **cal probe_nom set** or **cal probe_nom query** messages. The **plus** *input* token specifies the plus input of a differential channel or the input of a single ended channel. The **minus** *input* token specifies the minus input of a differential channel only. The *channel* parameter specifies which channel's constant is being reported. The *value* parameter is the present value of the probe nominal calibration constant.

cal probe_act set input channel value
cal probe_act: message tokens
set: token - selects the absolute setting mode
input: token - specifies the input
channel: us - specifies the channel
value: float - specifies the actual value

This message sets the probe actual calibration constant to an absolute value. The **set** token selects the absolute setting mode. The **plus** *input* token selects the plus input of a diffenetial channel of the input of a single ended channel. The **minus** *input* token selects the minus input of a differential channel only. The *channel* parameter specifies which channel's constant is to be changed. The *value* parameter specifies the new absolute value of the probe actual calibration constant. There are no incremental commands provided to change this constant.

For this message, the plug-in will determine the range of the control. Whent the limit of the control is exceeded, the plug-in will set the value to the maximum or minimum and report that value using the **cal probe_act status** message. No error message will be generated.

cal probe_act query input channel
cal probe_act: message tokens
query: token - selects the query mode
input: token - selects the input
channel: us - selects the channel

This message is sent to query the present value of the probe actual calibration constant. The **plus** *input* token specifies the plus input of a differential channel or the input of a single ended channel. The **minus** *input* token specifies the minus input of a differential channel only. The *channel* parameter specifies the channel for whcich the probe actual value is requested.

cal probe_act status input channel value
cal probe_set: message tokens
status: token - specifies the status response
input: token - selects the input
channel: us - selects the channel
value: float - the value of the probe actual cal constant

This message is sent by the plug-in to report the present status of the probe actual calibration constant. It is sent in response to the **cal probe_act set** and **cal probe_act query** messages. The **plus** *input* token specifies the plus input of a differentail channel of the input of a single ended channel. The **minus** *input* token specifies the minus input of a differentail channel only. The *channel* parameter specifies whcich channel's constant is being reported. The *value* parameter is the present value of the probe actual calibration constant.

This page is intentionally left blank.

5.4.6 Trigger View Calibration

The trigger view function provides a means for identifying the triggering point on the screen. There are two methods of performing this function. One method uses a nominal value provided by the triggering plug-in to place a software defined marker on the screen at the approximate point of triggering. The second method displays the actual trigger output of the triggering plug-in and adjusts the display based on the trigger view calibration procedure to provide a more precise indication of the triggering point. This section defines the messages that are used to provide a calibrated trigger view marker for smart plug-ins that support that function.

cal tv_status status [channel slope level]
cal tv_status: message tokens
status:token selects trigger view calibration mode
channel: us - specifies the reference input channel
slope: token - selects trigger slope selection
level: float - selects trigger level

This message is used by plug-ins and the mainframe to control the trigger view calibration procedure. The mainframe will initiate the procedure during the calibration process in the *Final Calibration Mode* (see section **5.3.9 Final Calibration Mode**). This will be done after all plug-ins have completed all other calibration modes. The mainframe will begin the procedure by sending either the **begin** *status* token or the **send_val** *status* token.

If the mainframe sends the **send_val** *status* token, it indicates that the mainframe does not have trigger view calibration capability and is requesting a nominal value from the plug-in that it can use to place a trigger view marker on the screen. In this case, the plug-in will send the **cal tv_nom** message to indicates the nominal delay value. The receipt of this message terminates the trigger view calibration process.

If the mainframe sends the **begin** *status* token, it indicates the mainframe has trigger view calibration capability and is requesting the plug-in to begin the calibration process. The mainframe will include the *channel*, *slope* and *level* parameters when it sends the **begin** *status* token. The *channel* parameter specifies the reference channel to be used by the plug-in. The values 1 - 4 refer to the left plug-in channels 1 - 4. The values 5 - 8 refer to the center plug-in calibration was done or attempted. The *channel* parameter specifies which channel is to be calibrated.

The mainframe will connect the calibrated voltage reference to the external calibrator connector before it sends the **cal probe** message. The plug-in will request voltages from the CVR using the **cal cvr_set** message. These voltages will be sent to the external connector through the probe to the plug-in. The plug-in will request measurements to be made by the mainframe and will calculate calibration constants as necessary for the probe.

5.4.6 Trigger View Calibration

The trigger view function provides a means for identifying the triggering point on the screen. There are two methods of performing this function. One method uses a nominal value provided by the triggering plug-in to place a software defined marker on the screen at the approximate point of triggering. The second method displays the actual trigger output of the triggering plug-in and adjusts the display based on the trigger view calibration procedure to provide a more precise indication of the triggering point. This section defines the messages that are used to provide a calibrated trigger view marker for smart plug-ins that support that function.

cal tv_status status [channel slope level]
cal tv_status: message tokens
status:token selects trigger view calibration mode
channel: us – specifies the reference input channel
slope: token – selects trigger slope selection
level: float – selects trigger level

This message is used by plug-ins and the mainframe to control the trigger view calibration procedure. The mainframe will initiate the procedure during the calibration process in the *Final Calibration Mode* (see section **5.3.9 Final Calibration Mode**). This will be done after all plug-ins have completed all other calibration modes. The mainframe will begin the procedure by sending either the **begin** *status* token or the **send_val** *status* token.

If the mainframe sends the **send_val** *status* token, it indicates that the mainframe does not have trigger view calibration capability and is requesting a nominal value from the plug-in that it can use to place a trigger view marker on the screen. In this case, the plug-in will send the **cal tv_nom** message to indicates the nominal delay value. The receipt of this message terminates the trigger view calibration process.

If the mainframe sends the **begin** *status* token, it indicates the mainframe has trigger view calibration capability and is requesting the plug-in to begin the calibration process. The mainframe will include the *channel*, *slope* and *level* parameters when it sends the **begin** *status* token. The *channel* parameter specifies the reference channel to be used by the plug-in. The values 1 – 4 refer to the left plug-in channels 1 – 4. The values 5 – 8 refer to the center plug-in channels 1 - 4. The *slope* parameter specifies the slope of the input trigger signal's reference edge. It will be either **plus** or **minus**. The level parameter specifies the level of the input trigger signal. The plug-in will set its internal trigger circuits to conform to the values of the *channel*, *slope* and *level* parameters. The plug-in will send the **cal tv_set** message when it is ready as a response to the **cal tv_status begin** message.

The mainframe will make measurements (with assistance from the user) to determine timing parameters related to the trigger signal to allow the mainframe to line up the trigger signal with the reference channel display. The mainframe will save any constants necessary to adjust the timing. These constants will not be reported to the smart plug-in.

The **complete** *status* token is sent by the mainframe to indicate that it has completed the delay measurement. The **ready** *status* token is sent in response by the plug-in to terminate the trigger view calibration procedure.

cal tv_nom value
cal tv_nom: message tokens
value: float - nominal delay value

This message is sent by a smart plug-in in response to the **cal tv_status** message with the **send_val** *status* token. The *value* parameter indicates the nominal delay value of the plug-in in seconds. This value is preprogrammed into the plug-in and is not measured by the plug-in during calibration.

cal tv_set level slope coupling
cal tv_set: message tokens
level: float - specifies mainframe triggering level
slope: token - specifies mainframe triggering slope
coupling: token - specifies mainframe triggering coupling

This message is sent by a smart plug-in when it has received a **cal tv_status** message with the **begin** *status* token. The plug-in will set its internal triggering parameters to match the values specified by the mainframe in the **cal tv_status** message. When it has done this, it will send the **cal tv_set** message. The *level* parameter indicates the output level of the plug-in's trigger output. The mainframe will set its triggering level to this value. The *slope* parameter specifies the slope of the reference edge of the plug-in's output trigger signal. The *slope* token values may be **plus** or **minus**. The *coupling* parameter specifies the input coupling for the mainframe's trigger input. The *coupling* tokens may be either **AC** or **DC**.

5.4.7 Special Plug-in Calibration

This procedure is provided for plug-ins (smart or generic) that have calibration requirements that are not covered by the default system values for time and temperature. These plug-ins may request calibration of special functions based on internal criteria.

channels 1 - 4. The *slope* parameter specifies the slope of the input trigger signal's reference edge. It will be either **plus** or **minus**. The level parameter specifies the level of the input trigger signal. The plug-in will set its internal trigger circuits to conform to the values of the *channel*, *slope* and *level* parameters. The plug-in will send the **cal tv_set** message when it is ready as a response to the **cal tv_status begin** message.

The mainframe will make measurements (with assistance from the user) to determine timing parameters related to the trigger signal to allow the mainframe to line up the trigger signal with the reference channel display. The mainframe will save any constants necessary to adjust the timing. These constants will not be reported to the smart plug-in.

The **complete** *status* token is sent by the mainframe to indicate that it has completed the delay measurement. The **ready** *status* token is sent in response by the plug-in to terminate the trigger view calibration procedure.

cal tv_nom value
cal tv_nom: message tokens
value: float - nominal delay value

This message is sent by a smart plug-in in response to the **cal tv_status** message with the **send_val** *status* token. The *value* parameter indicates the nominal delay value of the plug-in in seconds. This value is preprogrammed into the plug-in and is not measured by the plug-in during calibration.

cal tv_set level slope coupling
cal tv_set: message tokens
level: float - specifies mainframe triggering level
slope: token - specifies mainframe triggering slope
coupling: token - specifies mainframe triggering coupling

This message is sent by a smart plug-in when it has received a **cal tv_status** message with the **begin** *status* token. The plug-in will set its internal triggering parameters to match the values specified by the mainframe in the **cal tv_status** message. When it has done this, it will send the **cal tv_set** message. The *level* parameter indicates the output level of the plug-in's trigger output. The mainframe will set its triggering level to this value. The *slope* parameter specifies the slope of the reference edge of the plug-in's output trigger signal. The *slope* token values may be **plus** or **minus**. The *coupling* parameter specifies the input coupling for the mainframe's trigger input. The *coupling* tokens may be either **AC** or **DC**.

5.4.7 Special Plug-in Calibration

This procedure is provided for plug-ins (smart or generic) that have calibration requirements that are not covered by the default system values for time and temperature. These plug-ins may request calibration of special functions based on internal criteria.

The mainframe is not required to know whether a plug-in requires special calibration. The plug-in will ask for that calibration at appropriate times defined below. The plug-in is required to ask the mainframe for the present accuracy mode using the messages defined below. The plug-in will use that information to determine whether to request special function calibration. The plug-in may also notify the mainframe that it has exited the enhanced accuracy mode. This will cause the system to exit the enhanced accuracy mode.

The special function will always be calibrated during the normal, full or warm calibration procedures. The plug-in will not need to request special calibration during those procedures. The special function calibration will not be needed to maintain normal accuracy for the plug-in.

When the mainframe has terminated a full or warm-up calibration procedure (with the **cal mode exit** message), the plug-in with the special calibration function will ask the mainframe for the present calibration mode using the following message:

cal acc_query
cal acc_query: message tokens

This message may be sent any time during normal operating mode. The mainframe will respond with the following message:

cal acc_status mode
cal acc_status: message tokens
mode: token - indicates accuracy mode

The *mode* token specifies the accuracy mode: **warmup**, **normal**, **enhanced** or **auto_enhanced**. The **warmup** *mode* token indicates the system is in the warmup mode. The **normal** *mode* token indicates the system is in the normal accuracy mode. Plug-ins with special calibration requirements may not request special function calibration in either of these modes. The enhanced calibration mode token indicates the system has entered the enhanced calibration mode. In this mode, a plug-in with special calibration requirements will not request a calibration but will notify the mainframe that it is no longer in enhanced calibration mode using the **cal acc_status** message with the **normal** *mode* token. The **auto_enhanced** *mode* token indicates the system is in the auto enhanced mode. In this mode, a plug-in with special calibration requirements will request calibration of that function using the following message:

cal req_cal
cal req_cal: message tokens

The mainframe will respond with either the **cal mode** message causing the plug-in to begin special calibration or with the **cal acc_status** message indicating the system is no longer in the enhanced accuracy state and special function calibration is not required. The system may leave the auto enhanced mode without notifying a plug-in that has special calibration requirements. Thus, the plug-in may request special function calibration even though the system is not in auto enhanced mode.

When the mainframe determines that special function calibration is appropriate, it will place the requesting plug-in in the *Special Calibration Mode* and all other plug-ins in the *Hold Calibration Mode*.

The *Special Calibration Mode* is entered when the plug-in receives the **cal mode** message with the **special** *mode* token. During this mode, the plug-in is assigned the CVR and may request voltages and measurements without requiring the use of the plug-in SRQ. The mainframe is not allowed to send any unsolicited messages to the plug-in during this time. Only three messages from the plug-in are valid during this time: **cal measure**, **cal cvr_set** and **cal status** complete. This mode is terminated when the plug-in sends the **cal status** message with the **complete** or **fail** *status* token. If the fail status token is reported, it indicates the plug-in has failed to achieve enhanced accuracy. The system accuracy mode will be changed to normal mode.

5.5 Calibrated Voltage Reference

The Calibrated Voltage Reference, or CVR, is the voltage source standard that is used by the 11000 Series Accuracy System to provide accurate voltages for measurements. It is a two terminal device with an output and a ground sense line. Because the ground sense line may be assigned to only one plug-in compartment at a time (or to the mainframe), the mainframe must control the allocation of the CVR.

Plug-ins may request voltage outputs from the CVR only when they have received the **cal mode** message with the **dp_cal_wc**, **new_config** or **other** *mode* token and until the plug-in sends the **cal status** message indicating completion of the selected calibration mode. A plug-in may also request voltages during the probe calibration mode as defined in section **5.4.5 Probe Calibration**. During these times, smart plug-ins may request the CVR to be assigned to another plug-in compartment. This is the method used by the right plug-in for calibrating the auxiliary trigger lines.

cal cvr_set value
cal cvr_set: message tokens
value: float - requested voltage value

This message is used by plug-ins to request a voltage setting from the CVR. The *value* parameter specifies the voltage value (in volts) requested by the plug-in. The CVR circuit will set the voltage output to the nearest achievable value and return the **cal cvr_status** message:

cal cvr_status value
cal cvr_status: message tokens
value: float - specifies CVR output value

This message is sent by the mainframe in response to the **cal cvr_set** message when the CVR circuit output has settled. The value reported by the mainframe is the expected output value from the CVR and may not exactly match the value requested by the plug-in. The mainframe will never report an error for a **cal cvr_set** message but will always set the CVR to some value and report it.

The mainframe or a smart plug-in may request a generic plug-in to enter CVR Pass-Through Mode. In this mode, the CVR is connected to the specified generic plug-in and the plug-in connects its input to the CVR. The mainframe or smart plug-in may then request voltages which are sent through the generic plug-in and out to the plug-in's display, trigger and auxiliary trigger outputs. See section **5.3.10 CVR Pass-Through Mode** for the details of this mode.

5.6 Voltage Measurement

The mainframe provides a voltage measurement circuit that measures the voltage displacement on the display or trigger output of a plug-in compartment. Which output is measured is determined by the **cal mode** message (see section **5.3 Calibration Modes**). The mainframe will always measure the appropriate output of the plug-in requesting the measurement. Smart plug-ins may not request measurements of the outputs of other plug-ins.

cal measure
cal measure: message tokens

This message is sent by a plug-in to request the mainframe to make a measurement on its output. The mainframe will take appropriate steps (such as adjusting the number of averages and the length of time over which the averaging is done) in order to insure proper accuracy of the measurement based on the noise reported by the plug-in in the **cal meas_set** message. The mainframe will also take into account standard noise generated by the CVR and mainframe signal paths. The mainframe will measure the plug-in's output and report the result using the following message:

cal meas_value value
cal meas_value: message tokens
value: float – measured deflection value

This message reports the value measured by the mainframe. The *value* reported is in divisions of deflection.

cal meas_set time
cal meas_set: message tokens
time: token – specifies measurement time

This message is used by a plug-in to set the measurement time parameter for mainframe measurements on that plug-in's output. The *time* tokens indicate relative time required to average out noise that is above the nominal defined for a plug-in. The default time is x1. Available alternate times are **x2**, **x4** and **x8** which specify the indicated amount of additional noise in the signal path. The mainframe will save the measurement time status for each plug-in individually.

The plug-in may request a change in measurement time anytime during calibration procedures.

6.0 Interface Protocols

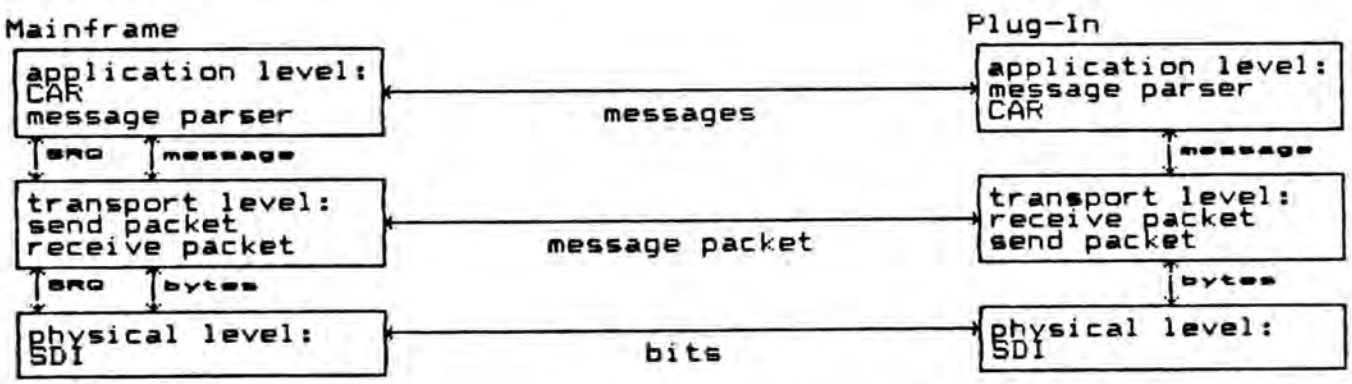
This section defines the means and methods for transferring messages between mainframes and plug-ins.

The first sub-section deals with application level protocols that are required to handle specific situations. These protocols do not apply to all messages. Application level software consists of the routines that create and interpret interface messages. Examples of these might be a message parser or a command action routine.

The second sub-section defines the transport level protocol. The transport level protocol applies to all messages and is the means for reliable transfer. Transport level software consists of the routines that send and receive packets and implement the protocol defined in the Transport Protocol section. These routines do not interpret the contents of the message; they just transfer data reliably between mainframes and plug-ins.

The third sub-section is a glossary of terms used throughout this section.

The protocol levels identified above may be represented graphically as follows:



The application and transport levels show examples of software that resides at that level. There is no software at the physical level. In the example, the command action routine on the left (CAR) communicates with the message parser on the right via messages. The send packet routine on the left communicates with the receive packet routine on the right by sending packets. Each level communicates with adjacent levels above and below as shown. Note that only the mainframe on the left implements the SRQ handling.

6.1 Application Protocols

These protocols apply to software that resides at the application level. Each routine that creates or interprets messages that are transferred across the mainframe/plug-in interface must implement the protocols as defined. Some protocols apply to all messages, some only to a few.

6.1.1 Message Formats

Each message is identified by two bytes called message tokens. These bytes uniquely determine the meaning of the message. Parameters may also be included as part of a message and convey additional information. The parameters will always follow the message token bytes in a prescribed order. Although some parameters may be optional, the order of parameters is always fixed as specified for each message.

When a message is transferred from the transport level to the application level there are two alternative methods of indicating the length of the message: byte count or termination byte.

The byte count method supplies a single byte unsigned integer prefix to each message to indicate the length of the message. This byte count is prepended to a message by the transport level when it passes the message to the application level. The application program will use this value to keep track of the number of remaining bytes as it scans the message.

In the termination byte method, each message is terminated by the **terminator** token (00_{16}). All messages have been constructed such a way that the termination byte will not be confused with any other token; message and parameter tokens do not use the value 00_{16}. The **terminator** token only has significance for variable length messages. In variable length messages, all optional or additional parameters are identified by a preceding token that is distinct from the **terminator** token. A token with the value 00_{16} (the **terminator** token) always indicates the end of the message. The **terminator** token is not sent as part of the transport message packet data but is added by the transport level when it sends the message to the application level.

6.1.2 Command/Status Protocol

Each message that is sent by the mainframe or a plug-in that requests an action is termed a command message. Each command message has a corresponding status message. A command transaction is defined as starting with the transmission of the command message and terminating with the reception of the status message. There may be intermediate messages sent by the receiver prior to the sending of the final status message but these must be defined as part of the command transaction. No other command messages may be sent by either plug-in or mainframe until the current command transaction has been completed.

The purpose of this protocol is to force synchronization of events. The mainframe or a plug-in will always know when a requested action is completed. This guarantees that operations will be performed in a sequential manner.

Following are some examples of this protocol:

```
mainframe              message                 plug-in
knob change            ---gain set fine--->    pi sets gain
mf updates display     <---gain status---
knob change            ---gain set fine--->    pi detects error
mf handles error       <---error generic---
mf updates display     <---gain status---      pi reports status

mainframe              message                 plug-in
knob change            ---gain set coarse--->  pi changes offset
mf updates display     <---offset status---
knob change
mf updates display     <---gain status---      pi reports gain
                       ---gain set coarse--->  pi sets gain
mf updates display     <---gain status---
```

Notice in the second example that even though the knob change was detected by the mainframe prior to the plug-in sending the **gain status** message, a new **gain set coarse** message was not sent until the mainframe had received the **gain status** message from the plug-in. The brackets on the right show the command transaction groupings.

6.1.3 Plug-In SRQ's

Plug-Ins will not issue asynchronous messages. Whenever a plug-in requires mainframe attention, it will send the transport level SRQ. The plug-in will then wait for the **SRQ query** message from the mainframe to send the message. The **SRQ query** message is defined in section **3.0 Common Messages**.

When the mainframe detects a plug-in SRQ (by reading its SRQ status for that plug-in, see section **6.2**), it will send the SRQ query message. The plug-in will respond with a status message indicating either the cause of the SRQ (eg. an error condition) or resultant changes in status (eg. change in gain). Smart plug-ins may also issue command messages in response to the **SRQ query** message to request an action by the mainframe. The mainframe will take action appropriate to the status or command message (and return the appropriate status message) then send another **SRQ query** message. The plug-in may reply with either the **SRQ no_report** status message or an additional status or command message. This sequence will continue until the plug-in sends the **SRQ no_report** status message. When the mainframe receives this message it will clear its SRQ status for that plug-in and send no further **SRQ query** messages.

If the plug-in does not receive an **SRQ query** message within 100ms, the plug-in will issue another SRQ. If a plug-in receives an **SRQ query** message when it has nothing to report, it will send the **SRQ no_report** status message. The mainframe will not stack SRQ requests. There will be a single SRQ status for each plug-in which is either set or reset as defined below.

The SRQ will also be used during the start-up sequence. When a plug-in powers up, it will run through its kernel self-tests then begin issuing SRQ's to the mainframe repeatedly until an SRQ is successfully transferred (no handshake errors). When the mainframe is ready, it will recognize the plug-in's SRQ and send an **SRQ query** message. The plug-in will reply with the **SRQ no_report** message.

Some examples (in these examples db stands for data base):

mainframe	message	plug-in
	——gain set abs——>	sets gain
mf updates db	<——gain status——	
knob change	——gain set coarse——>	plug-in changes gain
mf updates db	<——gain status——	
mf sets SRQ status	<——SRQ——	user removes probe
mf acts on SRQ status	——SRQ query——>	
mf updates status	<——cal probe_data——	plug-in responds
mf acts on SRQ status	——SRQ query——>	
mf updates db	<——gain status——	plug-in responds
mf acts on SRQ status	——SRQ query——>	
mf clears SRQ status	<——SRQ no_report——	plug-in responds
knob change	——gain set coarse——>	plug-in changes gain
mf updates db	<——gain status——	

mainframe	message	plug-in
	——gain set abs——>	sets gain
mf updates db	<——gain status——	
knob change	——gain set coarse——>	plug-in changes gain
mf updates db	<——gain status——	
mf sets SRQ status	<——SRQ——	user removes probe
mf acts on SRQ status	——SRQ query——>	
mf updates status	<——cal probe_data——	plug-in responds
mf acts on SRQ status	——SRQ query——>	
mf updates db	<——gain status——	plug-in responds
	<——SRQ——	user adds probe
mf acts on SRQ status	——SRQ query——>	
mf updates status	<——cal probe_data——	plug-in responds
mf acts on SRQ status	——SRQ query——>	
mf updates db	<——gain status——	plug-in responds
mf acts on SRQ status	——SRQ query——>	
mf clears SRQ status	<——SRQ no_report——	plug-in responds
knob change	——gain set coarse——>	plug-in changes gain
mf updates db	<——gain status——	

mainframe	message	plug-in
	——gain set abs——>	sets gain
mf updates db	<——gain status——	
knob change	——gain set coarse——>	plug-in changes gain
mf updates db	<——gain status——	
mf sets SRQ status	<——SRQ——	user removes probe
mf acts on SRQ status	——SRQ query——>	
mf updates status	<——cal probe_data——	plug-in responds
knob change		
mf acts on SRQ status	——SRQ query——>	
mf updates db	<——gain status——	plug-in responds
mf acts on SRQ status	——SRQ query——>	
knob change		
mf clears SRQ status	<——SRQ no_report——	plug-in responds
	——gain set coarse——>	plug-in changes gain
mf updates db	<——gain status——	

See the next section, **Long Messages**, for more examples of the use of plug-in SRQ's.

6.1.4 Long Messages

This protocol is provided to minimize the buffer size required to save a transport message packet. If there is an operation that requires more than 127 bytes of data to be transferred, the data will be sent in a series of messages. The data is reconstructed at the application level. The burden is on the receiving routine to sort out the messages as they come.

There is a single long message protocol byte at the application level for specific messages. This byte gives sufficient information to allow the processing of multiple messages to transfer a single set of data. The byte indicates four message types: **more**, **last**, **abort** and **ack**. These four types are mutually exclusive; they are defined as follows:

name	value
more	01_{16}
last	02_{16}
ack	03_{16}
abort	04_{16}

Long messages will use the following application protocol:

The initial message will contain the **more** token. When the receive routine is started, it will know (by necessity) that this is the first message and thus what parameters are to be expected in the message. These parameters are defined at the application level and have no meaning to the long message protocol. The **more** token notifies the receiving task that there is more information to be sent. The task will save its present status, send the same application level message token it just received with the **ack** long message token as the only parameter and wait for the next message. The sending task will receive the application level **ack*** and will transmit the second message. When the receiving task receives the second message, it will have saved enough of its operating state to know that the message now being received is not the first but a subsequent message and will be able to interpret its contents. It will send an application level **ack** for each message. This sequence will continue until the last message is sent by the sending task. This message will have the **last** long message token instead of **more**. This will indicate to the receiving task that this is the last message of the sequence. The receiving task will send an application level **ack** as before. This **ack** will terminate the process.

If the data to be sent normally uses multiple messages but is less than 127 bytes, the first message of the sequence may have the **last** long message token. This informs the receiver that there is only one message in the sequence.

At an appropriate time during this process, the sending or the receiving task may send a message with the **abort** long message token instead of a **more**, **last** or **ack**. This message must be sent at the expected point in the sequence (that is, the sending task may not send a **more** message followed by an **abort** message without waiting for the **ack** message from the receiver). The **abort** token will cause the transfer process to be terminated. If either the sender or the receiver sends the **abort** message, the sender will terminate the transfer and take action appropriate to non-completion of the operation and the operation will be terminated. The receiver will discard any data received, or, if not possible to discard, take action appropriate to the termination of the operation. Either the receiver or the sender will send an **ack** message in response to an **abort** message from the other device. This **ack** message marks the termination of the transfer process. The **abort** message will also cause the mainframe's SRQ status for that plug-in to be cleared.

During the transfer process, other messages are not allowed to be interspersed. This is required to fulfill the definition of command/status protocol. The plug-in will send only one SRQ to request the transfer of several messages using the long message protocol for each operation.

Here is an example of the long message protocol.

---

*Note that this acknowledge is different than the transport level ACK. The acknowledge being discussed here is interpreted at the application level and is called the application level acknowledge to distinguish it from the transport level ACK.

```
mf sets SRQ status              <----SRQ----              plug-in wants a menu
mf acts on SRQ status       ----SRQ query---->
mf begins menu op    <----menu def_smart more----         first packet
mf ready              ----menu def_smart ack---->
mf continues         <----menu def_smart more----         second packet
mf ready              ----menu def_smart ack---->
mf continues         <----menu def_smart last----         last packet
mf finishes           ----menu def_smart ack---->
mf done              ----menu status formatted---->
mf acts on SRQ status       ----SRQ query---->
mf clears SRQ status   <----SRQ no_report----             nothing more

mf sets SRQ status              <----SRQ----              plug-in wants a menu
mf acts on SRQ status       ----SRQ query---->
mf begins menu op    <----menu def_smart more----         first packet
mf ready              ----menu def_smart ack---->
mf continues         <----menu def_smart more----         second packet
mf interrupted       ----menu def_smart abort---->
mf clears SRQ status
mf takes other action<----menu def_smart ack----
mf sets SRQ status              <----SRQ----              plug-in wants a menu
mf acts on SRQ status       ----SRQ query---->
mf begins menu op    <----menu def_smart more----         first packet
mf ready              ----menu def_smart ack---->
mf continues         <----menu def_smart more----         second packet
mf ready              ----menu def_smart ack---->
mf continues         <----menu def_smart last----         last packet
mf finishes           ----menu def_smart ack---->
mf done              ----menu status formatted---->
mf acts on SRQ status       ----SRQ query---->
mf clears SRQ status   <----SRQ no_report----             nothing more
```

6.2 Transport Protocol

The transport level protocol is implemented by the software that causes bytes to be transferred across the SDI interface. Each device that uses this interface must implement this protocol fully.

6.2.1 Scope

This section specifies the means for transferring data using the serial data interface lines. Specifically, it defines the transport protocol used to reliably transfer data between mainframes and plug-ins. This section does not cover the hardware operation of the serial data interface. That is described in the *11000 Series Plug-In to Main Frame Interface Manual.*

Instruments that use this interface will be classified as either plug-ins or mainframes. Mainframes will provide a serial data interface for each plug-in slot. Each plug-in must provide a serial data interface for communication with the mainframe. Plug-Ins will not be able to communicate directly with other plug-ins.

6.2.2 Features

The absolute minimum transport system would assume that all the hardware and software in the system is error free and not provide any error detection at all. If there are errors, this system would provide no way for recovery. Command parameter errors would be passed through with resulting consequences or the system could just lock up and need a power-up reset to restart it. The following Transport System was devised to minimize these problems.

There are six major features that are used to provide reliable data transfer between mainframes and plug-ins. These are the byte count, checksum, positive acknowledge, sequence number, timers and SRQ.

Byte Count

The Byte Count specifies the number of bytes in the body of a message packet and allows the transmission of very short messages without the overhead that a fixed length message protocol would require. The Byte Count also provides a level of data reliability by requiring the message length and byte count to match. If they don't, an error is detected. The byte count does not include the header or the trailer byte.

Checksum

The Checksum provides an additional level of data reliability by checking for incorrect data bits. The method used is a spiral addition of all bytes sent before the checksum to generate the checksum. The receiver calculates a checksum as it receives a message. It then compares the checksum it generated with the one sent by the sender. If they don't match then an error is detected. See section 6.2.4 for the checksum algorithm.

Acknowledge

Positive Acknowledges are used to provide error correction. If an error is detected by the receiver for a message, it will not return a positive acknowledge for that message. This lack of an acknowledge (after a time interval has elapsed, see the timers described below) will cause the sender to repeat the message. Without this feature, the receiver has no recourse on an error except to ignore the erroneous message.

The acknowledge is sent as a single byte with the value 55_{16} and with the EOI bit set. This distinguishes the ACK (or positive acknowledge) byte from all other bytes.

Sequence Numbers

Sequence Numbers provide an additional level of reliability above the Positive Acknowledges. They prevent confusion as to whether a message is a repeat message (perhaps because the acknowledge got lost) or a new message. The range of the sequence number is determined by the number of unacknowledged messages allowed. In this protocol, the limit is one unacknowledged message, hence the sequence number range from 0 to 1 is sufficient. Sequence numbers are set to 0 at power up.

Timers

Timers are used to prevent system lockup. If both sides of the interface are waiting for the other to complete a transfer (due to transmission errors) unless there is some sort of timeout, the system is locked up and recovery is not possible. Also, if the messages get out of sync (due to a problem with the byte count) either timers or a sync byte are required to resynchronize the system. In this protocol, timers will be used to provide resynchronization after errors that are not otherwise recoverable.

There are three virtual timers for each device in this protocol: transmit timer, receive timer and ack timer. The transmit timer is used by the transmitter to detect the failure of the receiver to transfer bytes out of its receive buffer. The receive timer is used to determine when a receiver may assume no more bytes are to be received. The ack timer determines the amount of time the sender will wait for an ACK before retransmitting the message. Note that because none of the timers ever run simultaneously in a device, this protocol may be implemented with only one actual timer per device. Although the receive and transmit timers for a particular message run at the same time, they are in different devices.

The receive timer time is 100ms. The transmit timer time is also 100ms. The ACK timer time is 200ms.

SRQ's

SRQ's are used to prevent collisions of outgoing messages. If both the plug-in and the mainframe were to transmit a message at the same time, the protocol would not otherwise be able to recover gracefully. The plug-in SRQ is used as a means for plug-ins to notify the mainframe that they have an unsolicited message to send. The mainframe will query for that message in a synchronous fashion.

A plug-in SRQ is defined as a single byte sent to the mainframe with the EOI bit set. It is distinguished from an ACK by its byte value which is AA_{16} (an ACK is 55_{16}). The SRQ is defined as a transport level packet. It does not require an ACK from the receiver. When the mainframe receives a plug-in SRQ, it will save internal SRQ status individually for each plug-in. The mainframe will examine this status when it does not have another message to transmit. If the status for a particular plug-in is set, the mainframe will send the **SRQ query** message defined in section 3 to the indicated plug-in.

Plug-Ins will not send an SRQ while they are transmitting a message. Plug-Ins may send an SRQ during message reception. Whenever the mainframe sees an SRQ it will set its SRQ status and dispose of the SRQ byte. If the mainframe is waiting for an ACK but receives an SRQ, the mainframe will set its SRQ status, dispose of the SRQ byte and continue to wait for the ACK (the ACK timer is still running).

To insure the integrity of the SRQ system, the plug-in will re-issue an SRQ if the mainframe has not sent a **SRQ query** message. The amount of time a plug-in waits between issuing SRQ's is 1 second.

6.2.3 Format

The Transport System consists of message, acknowledge and SRQ packets.

The message packet contains one header byte followed by the body of the message and a single trailer byte. The header byte is the byte count indicating the length of the rest of the message (excluding the checksum) and contains the sequence number. The sequence number is the most significant bit. This gives 7 bits for the byte count which provides for message packets as long as 127 bytes. The body may contain 0 - 127 bytes and has no meaning to the Transport System. The trailer byte is the checksum. Packets sent to generic plug-ins are limited to 32 bytes.

sn	bc	body	chksum

A message will not be split over multiple packets. Each message must be sent complete in one packet. Each packet will contain only one message. This protocol does not provide for message reconstruction from packets; there is a one-to-one correspondence between message packets and messages. Operations requiring data longer than 127 bytes will use the long message protocol. Each message in that protocol is a single packet.

There is no routing information provided in this protocol. Each device that uses the interface communicates with only one other device. There is no way that plug-ins may talk to other plug-ins at the transport level.

The acknowledge packet is a single byte with value 55_{16} that is sent with the EOI bit set. The EOI bit in combination with the byte value identifies the byte as the acknowledge packet.

The SRQ packet is a single byte with value AA_{16} that is sent with the EOI bit set. The EOI bit in combination with the byte value identifies the byte as the SRQ packet.

6.2.4 Operation

This section contains specific algorithms to transmit and receive messages using the defined interface protocol.

Packets are transferred across the interface one at a time. When a device places its first byte in the transmit hardware, it becomes the sender; the other device becomes the receiver. Both devices follow the appropriate algorithms defined below for sending and receiving packets.

The checksum is used by the receiver to check for errors in messages. It is generated by the sender by summing and shifting in an accumulator each byte that is sent. The checksum byte is sent last. The receiver generates its own checksum using the same algorithm and compares it with the one received at the end of the message. If they agree, the receiver can assume that the message was received correctly; if they don't, an error has been detected.

The checksum for message packets is calculated as follows (taken from the *Digital Product Recommended Design Practices* book):

1. Initialize the 8 bit checksum accumulator to zero.

2. Read and save the byte count from the incoming message.

3. Shift the checksum accumulator left 1 bit with bit 8 going into carry and zero into bit 0.

4. Add with carry, the accumulator and the byte read from the SDI. Ignore any carry from this operation.

5. Decrement the byte count saved in step 2.

6. Repeat steps 3 - 5 until the byte count reaches 0.

7. Compare the calculated checksum with the next byte read from the SDI. If they are identical, the message has passed the checksum test. If they are not, the message has failed the checksum test.

The header byte (which includes the sequence number and the byte count) is included in the checksum calculation.

The sender will start its transmit timer when it sends the first byte. This timer checks that the receiver removes bytes from its input buffer. If it fails to do so, the timer will expire and the sender will detect an error.

After the sender has sent a complete message it terminates the transmit timer and starts the ACK timer to wait for an acknowledge from the receiver. This is done immediately after the last byte of the message (the checksum) has been sent (ie. the transmit buffer is empty).

When a message is received correctly, the receiver sends a positive acknowledge packet. When the sender receives the positive acknowledge, the message transfer is completed and the sender's ACK timer is stopped. If the receiver detects an error, no positive acknowledge is sent. The sender's ACK timer expires and indicates that the receiver did not receive the message correctly (or the acknowledge was lost). The sender may then repeat the message. This means that if the receiver receives a message correctly it must send the positive acknowledge within the time limit of the sender's ack timer.

Note that the positive acknowledges are not acknowledged. If they were this would lead to an infinite loop of acknowledges and lock up the interface.

The sequence number is used to prevent duplicate messages from being sent to the application level. Each message contains a sequence number. If the receiver receives a message correctly but the acknowledge is lost, the sender will repeat the message and the receiver will have duplicate correct messages. Using the sequence number, the receiver knows whether each message received is a repeat message or a new message. When the receiver receives a message with the same sequence number as the one it just received it discards the message but still sends the positive acknowledge to notify the sender. After each complete transfer (defined by the receipt of an acknowledge) the sender increments its sequence number. The receiver increments its receive sequence number after it has verified a correct checksum and sent the acknowledge if the sequence number did not indicate a duplicate message. See the diagrams below.

There are actually four sequence numbers that are kept by each pair of communicating devices. Each device has separate send and receive sequence numbers. Each sequence number is changed according to the rules defined below. The send and receive sequence numbers in each device are independent.

When a device transmits a message, it will use the value of the send sequence number at the start of transmission. It will change its send sequence number when it receives an acknowledge.

When a device receives the checksum byte of a message packet it will compare that checksum with the checksum it calculated. If they are the same, it will send an ACK. Next, it will check the sequence numbers. If the sequence number received in the packet is the same as the device's receive sequence number, the device will send the message on to the next level and change the sequence number. If the sequence numbers are different, the device assumes the packet is a repeat packet and will discard the message (the ACK has already been sent).

There are two problems that can occur if the byte count is received incorrectly - the receiver may wait for nonexistent bytes to be sent because the byte count was too high or the receiver may truncate the message because the byte count was too low.

The receive timer is designed to take care of these problems. After the first byte of a message is received, the receiver starts the receive timer. The complete message must be sent before this timer expires. If it is not, the receiver discards all the bytes it has received and does not send an acknowledge. The sender will then know that a problem has occurred and will repeat the message. This situation could happen if the byte count is received incorrectly and the receiver is waiting for bytes that will never be sent.

The other problem is if the byte count is too small, the receiver will accept only part of a message. In this case, the checksum will most likely be incorrect because it will just be a byte in the body of the packet. The receiver will then discard all the bytes that it has received and all additional bytes until its receive timer expires. It will then expect the beginning of a new message (because it has not sent the positive acknowledge within the time required by the sender). This means that all messages must be sent completely within the time of the receive timer.

6.2.5 Errors

When a protocol error is detected by a mainframe, the mainframe may notify the user that there is a communication problem with a plug-in. If a plug-in detects an error it may indicate the error condition using its front panel indicators (if there are any). After an error condition is reported, the mainframe or plug-in will return to the idle transmit and receive states as shown in the flow charts.

When a plug-in transmitter detects a transmit timer expiration it will attempt 4 additional retries. The mainframe will attempt 9 additional retries. If these all fail, the plug-in or the mainframe will terminate the transmit process and report an error condition.

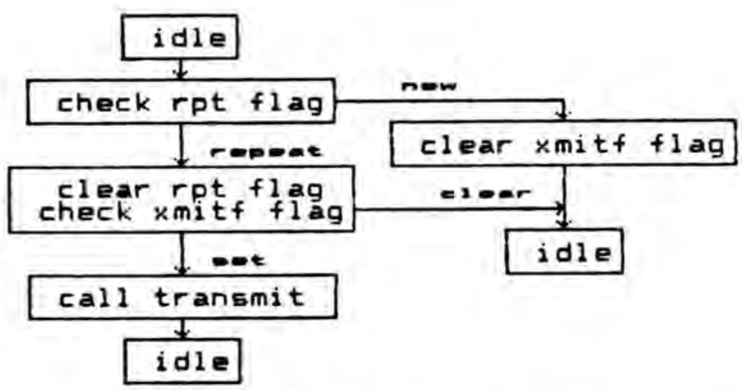
When a transmitter detects an ACK timer expiration, the plug-in will re-send the message up to four additional times. The mainframe will make 9 additional retries. If all retries fail, the transmitter will report an error condition.

When a receiver detects a checksum error it will discard the message and send no ACK. It will also discard all incoming bytes until its receive timer has expired. Five consecutive errors will cause the receiver to report an error.

If the receiver detects a receive timer expiration it will discard any bytes received and expect the beginning of a new message. Five consecutive errors will cause the receiver to report an error.

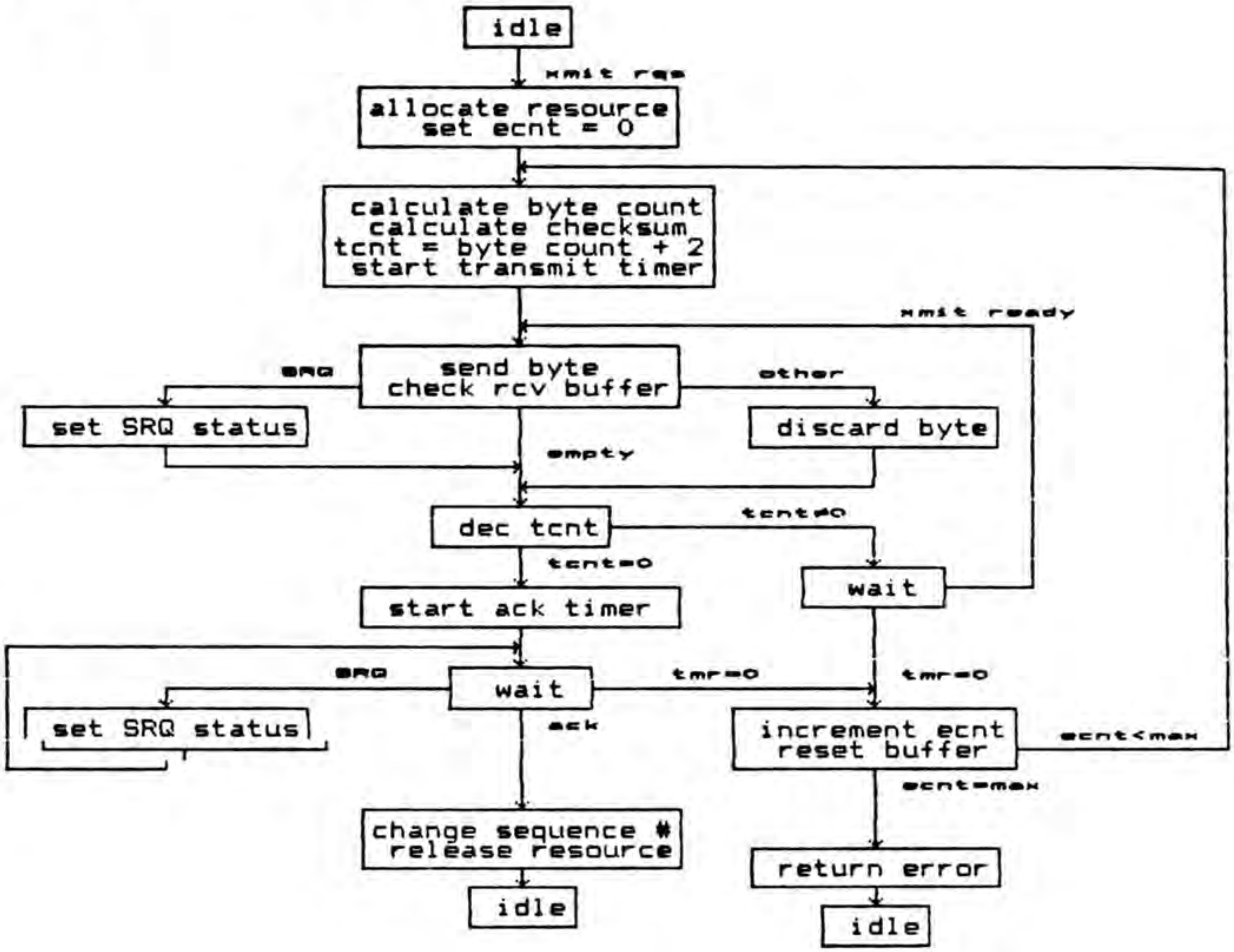
There may be separate retry counters for the receive and transmit functions or there may be one counter for both functions. When two counters are used, the receive counter is reset when a message is received correctly and the transmit counter is reset when a message is transmitted correctly. In the case of a single counter, the counter is reset when a message is sent or received correctly.

When a plug-in fails to successfully transmit a message, it will not discard the message. It will wait for the next message to be sent from the mainframe. If that message is a repeat message, the plug-in will send the message it previously attempted to transmit. If the message received from the mainframe is a new message, the plug-in will discard the message it was attempting to transmit. The plug-in will use the following algorithm to determine the proper action for disposal of the message in the transmit buffer after failure to transmit:

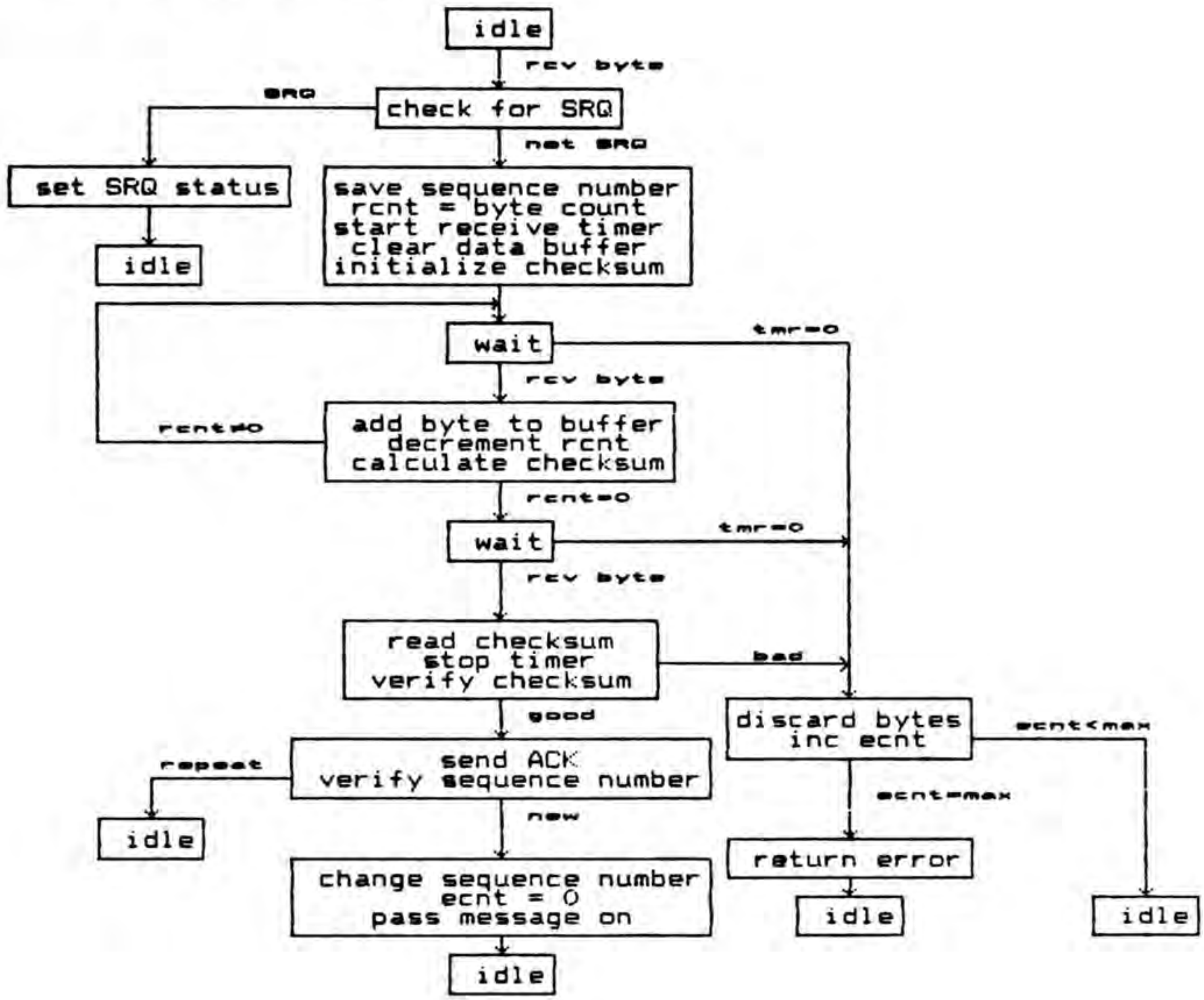


This procedure is executed immediately after each message is received.

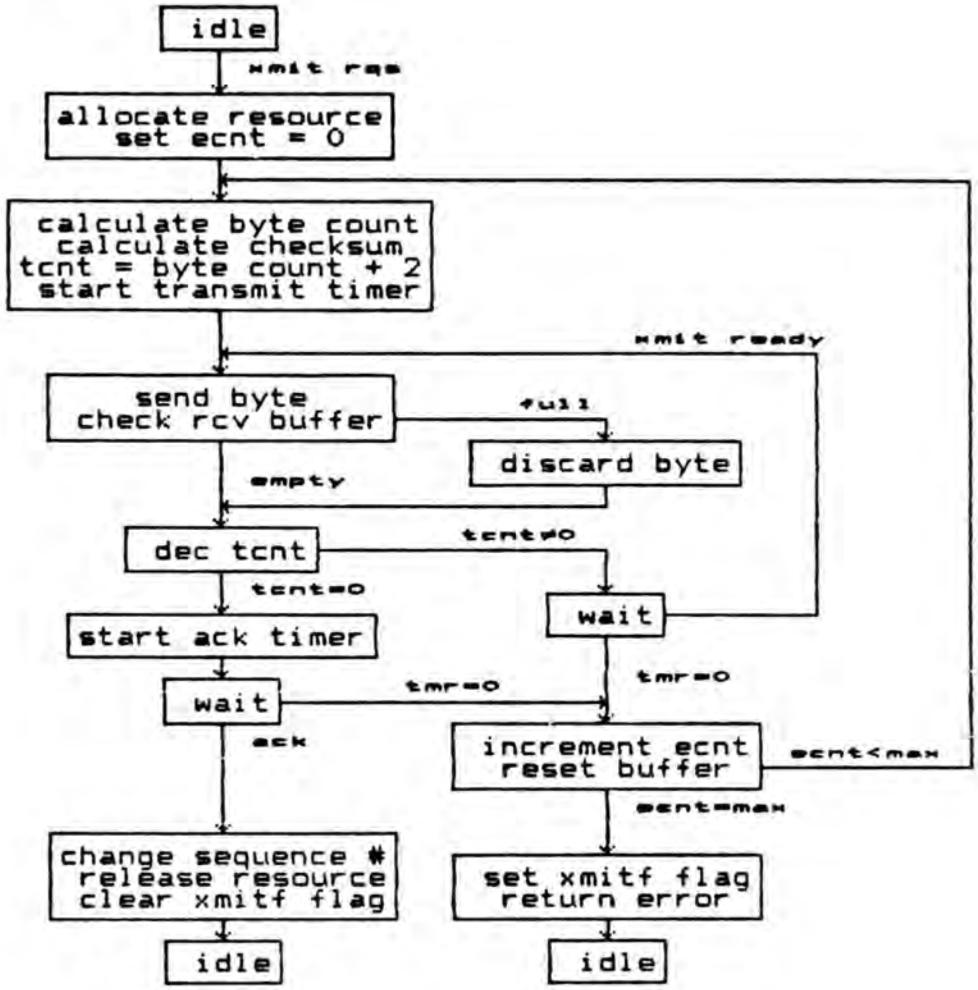
6.2.6.1 Mainframe Transmit



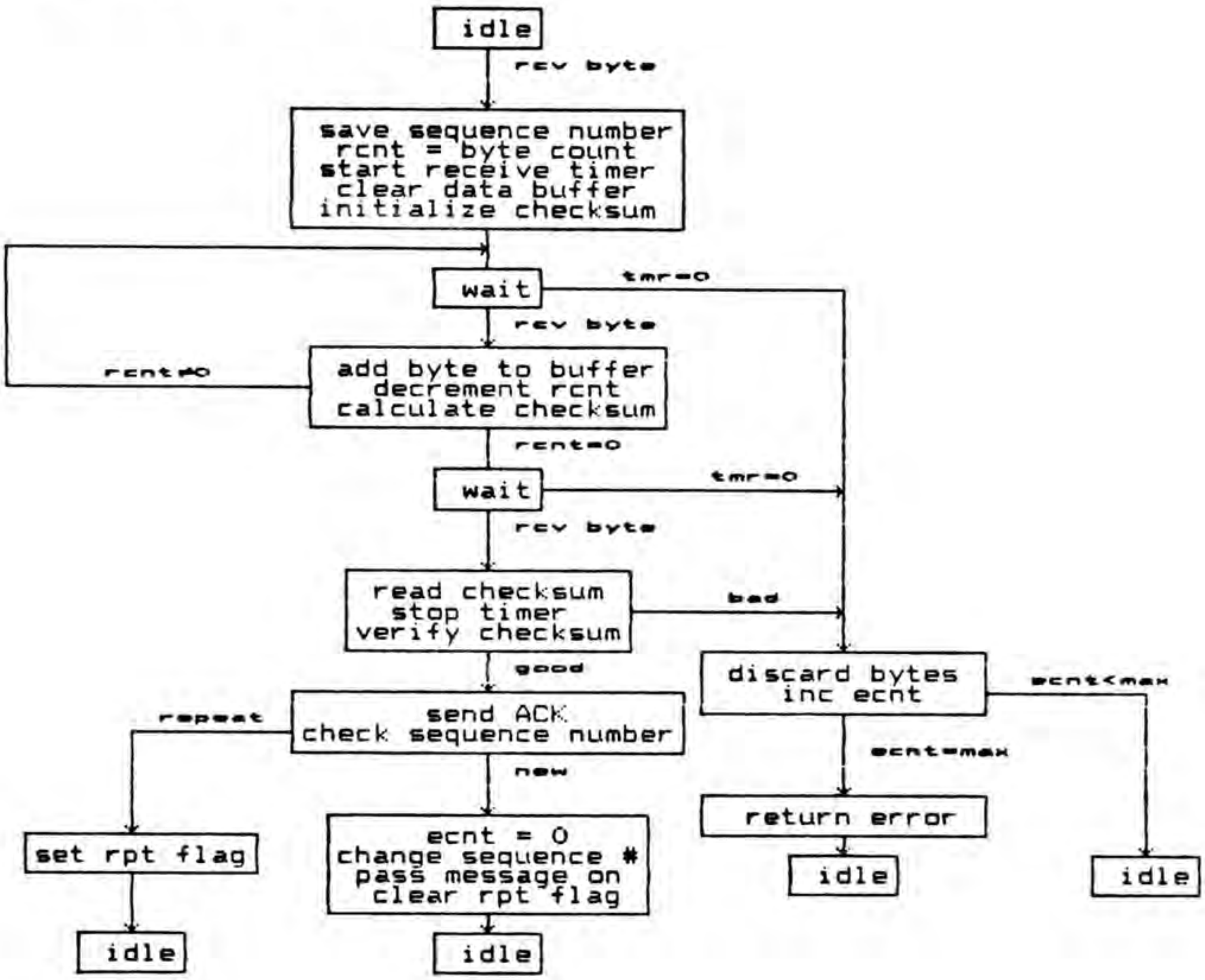
6.2.6.2 Mainframe Receive



6.2.6.3 Plug-In Transmit



6.2.6.4 Plug-In Receive



6.3 Definitions

This section contains definitions of terms used in section 6. Terms shown in *italics* are defined in this section.

acknowledge: a *transport packet* that indicates the reception of a correct *message packet*. It is used as part of the *transport protocol*.

ack timer: *timer* used by the *sender* to wait for an *acknowledge* from the *receiver*.

application level: The level at which the *messages* are interpreted and responses are formed.

body: the *message* to be transferred by the *transport system* in a *message packet*. The *data* has no meaning to the *transport system*.

byte count: specifies the number of bytes in the *body* of a *message packet*. The *byte count* is placed in the 7 least significant bits of the *header* byte for transmission. It is part of the *transport protocol*.

checksum: a data integrity check used as part of the *transport protocol*. It is computed using a spiral add algorithm.

data: the information in the form of one or more *messages* to be transferred between a mainframe and a plug-in.

device: a generic term for an instrument connected to the interface that uses the *transport protocol*. Each device contains a *sender* and *receiver*.

EOI: end or identify is an identity bit created by the SDI hardware that can be used to identify an individual byte being transferred across the interface.

error correction: the ability to correct an error that was detected. Taking action to overcome an error rather than just reporting its detection.

header: the first byte of a *message packet* that contains the *sequence number* and *byte count* in a format specified by the *transport protocol*.

levels: the transport protocol may be divided logically into modules that communicate vertically to different *levels* or horizontally across the interface at the same *level*. Each *level* has a defined interface to the *levels* above and below it and to the associated *level* across the interface.

lockup: the situation where a plug-in and a mainframe are both waiting for the other to send the next byte when neither is expecting to send a byte.

message: the series of bytes in the *body* of the *message packet* that convey information at the *application level*.

message packet: a set of bytes composed of a *header*, a *body* and a *trailer*. Its format is defined by the *transport protocol* and it is used to transfer a *message* between mainframes and plug-ins.

message transfer: the process for completely transferring a *message* without errors between a mainframe and a plug-in. A *message transfer* is begun when the first byte of a *message packet* is sent and terminates when the *acknowledge* is received.

receiver: the part of the mainframe or plug-in software that executes the receive algorithm defined by the *transport protocol*.

receive timer: *timer* used by the *receiver* to determine how long to wait for incoming bytes.

reliability: the ability to transfer a set of *data* bytes between mainframes and plug-ins with a high degree of confidence in the accuracy of the transfer.

sender: the part of the mainframe or plug-in software that executes the send algorithm define by the *transport protocol*.

sequence number: a number sent as part of a *message packet* that distinguishes that packet from the previous and next packets. The *sequence number* is placed in the most significant bit of the *header* byte for transmission. It is used as part of the *transport protocol*.

SRQ: a *transport packet* that a plug-in uses to notify the mainframe that is has a message to send.

timer: causes an event to occur after a predetermined period of time. Timers are used as part of the *transport protocol*.

trailer: the last byte of a *message packet* that contains the *checksum*.

transmit timer: a *timer* used by the *sender* to check for the *receiver* accepting bytes.

transport level: The level at which data is formed into *packets* in preparation for transmission across the interface.

transport packet: a set of bytes that compose a complete set of information for transferring a *message* between mainframes and plug-ins. There are three types of packets: *message packets*, *acknowledges* and *SRQ's*.

transport protocol: a system of messages, events and definitions used by the *transport system* to reliably transfer data between mainframes and plug-ins. It is composed of *byte counts*, *checksums*, *acknowledges*, *sequence numbers*, *timers* and *SRQ's*. In addition, the transport protocol defines how these items interact to transfer a *message*.

transport system: a mechanism to reliably transfer *messages* between mainframes and plug-ins. This mechanism uses the *transport protocol* to perform error checking and correction.

7.0 Message Definitions

This section contains a complete list of the messages defined for the software interface.

The general form of a message page gives pertinent information about each message in a specific format. It is split into sections each of which begins with an identifying name. Each section is defined in the following paragraphs.

The *name:* section gives the name by which the message is identified throughout the document. This name is symbolic. Each name is actually represented by two or three token values. The first two token values are the primary and secondary message tokens defined for the interface. The third token, when used, is a parameter to the message defined by the first two and is used to distinguish messages that perform different parts of the same operation (such as **gain set abs** vs **gain set coarse**).

The *syntax:* section shows the syntax of the message. This includes the name identifier and all parameters in the order in which they are to appear in the message. Several special characters are used to identify specific information. Curly braces ({}) are used for grouping parameters and are usually associated with repetitions. Brackets ([]) are used to identify parameters that are optional. The text of the command page will specify the usage of these parameters. A string of dots (...) is used to identify a parameter or group of parameters that may be repeated. The text will identify limits and characteristics of the repetition of parameters. Vertical bars (|) indicate alternation, that is, only one of the indicated parameters may be used at a time. **Boldface** is used to identify tokens, that is, parameters that have fixed values that are specified in this document. All message names are shown in **boldface**. Other parameters are shown in *italics* when used in the text of the message page.

The *type:* section identifies the type of message. A message is either a command or a status message. Command messages always require a status message reply. Messages identified as plug-in command messages are messages that are sent to a plug-in to cause it to take some action. Plug-In status messages are sent from a plug-in to indicate plug-in status. Mainframe command messages are messages sent by a plug-in to request action of the mainframe. Mainframe status messages report mainframe status to the plug-in. Messages identified as generic plug-in messages are used only with generic plug-ins. Messages identified as smart plug-in messages are used only with smart plug-ins. Messages identified as plug-in messages may be used with both types of plug-ins. Messages that use the long message protocol may send status both directions. The information in the type field indicates the direction of information transfer for the message contents and does not include long message protocol responses.

The *message tokens:* section defines the values for the message tokens that identify the message. The values shown are single byte (8 bit) values represented by the symbolic names.

The *parameters:* section defines the parameters of the message. The name and type of the parameter is shown along with appropriate limits or values for that parameter.

Parameter Types

Parameter types and notation are defined as follows:

notation	meaning
short	short integer - this is a single byte value. 7 bit magnitude with a sign bit. Range is +127 to -128
us	unsigned short - this is a single byte value with no sign bit. 8 bit magnitude. Range is 0 to 255.
int	integer - this is a double byte value. 15 bit magnitude with a sign bit. Range is +32767 to -32768.
ui	unsigned integer - this is a double byte value with no sign. 16 bit magnitude. Range is 0 to 65535.
long	long integer - this is a four byte value. 31 bit magnitude with a sign bit. Range is $+2^{31}-1$ to -2^{31}.
ul	unsigned long - this is a four byte value with no sign bit. 32 bit magnitude. Range is 0 to $2^{32}-1$.
float	floating point - this is a four byte signed floating point value in IEEE format. 23 bit magnitude, 8 bit exponent, 1 sign bit.
double	floating point - this is an eight byte signed floating point value in IEEE format. 52 bit magnitude, 11 bit exponent, 1 sign bit.
string	string - this is a string of ASCII characters terminated by the NUL character (00_{16}) and includes the set of escape sequences defined in Appendix B of the Command Reference Specifications document. Note that where strings are shown limited to a specific number of characters, the limit applies to displayed characters only. Characters defined as a two character escape sequence occupy only one display location. The limit also does not include the NUL terminator character.

char	character - this is a single byte unsigned value that is typically (but not always) a printable ASCII value.
pb	packed binary - this is a single byte split into two nibbles of four bits each. Each nibble represents a value from 0 to 16.
token	when shown as a parameter in a message, this type is a specified list of byte values that are defined in this specification. The token name is a symbolic substitute for the absolute token value.
byte	this type is used to define absolute hex values for tokens.
special	this type, when used in a message, is not a general type but indicates that the parameter is defined in the text and does not have one of the meanings defined above.

All references to channel numbers as unsigned integers in all sections will use the value 1 to indicate channel 1, 2 for channel 2 etc. Thus, a four channel plug-in will accept values 1 - 4 for its channel selection parameter for all of its commands.

All parameter types of more than one byte will be transmitted least significant byte first.

The NaN floating point value used for terminating floating point lists is the value 0xffc00000. Plus infinity is defined as 0x7f800000. Minus infinity is defined as 0xff800000.

Parameters of special types will be defined as necessary for messages that need them.

The *usage:* section identifies the usage of the message. It specifies the details of the parameters, what the message means and how is to be used. It identifies relationships to other messages and to general system operation. It specifies how parameters are checked for validity and range and how errors are identified.

The *response:* section identifies the response for the message. No responses are shown for most status messages.

The *execution time:* section specifies performance requirements generally in terms of the time allowed for a response to a command.

The *error handling:* section identifies error conditions that might be encountered with the use of the message and how they are reported.

The *also see:* section is a cross reference for other related messages.

The *command transactions:* section defines all the command transactions that involve the message being defined. A command transaction is a defined series of messages (usually two) used perform a specific function. Once the first message of a command transaction is sent, the order specified in the *command transactions* section must be followed to completion.

Each command transaction is shown as a series of messages separated by a colon (:). The colon is used as a separater for clarity and has no meaning to the interface. In some cases, intermediate messages are optional. They are enclosed in square brackets ([]). In some cases, there are alternative transactions that may be performed in response to the starting message. These alternatives are shown on subsequent lines. If there is more than one command transaction for a message, all possibilities are shown. The meaning of each command transaction is discussed in the text of the *usage* section.

name: **autorange request**

syntax: **autorange request** status

type: smart plug-in command

message tokens:

autorange ::= $1F_{16}$

request ::= 04_{16}

parameters:

name	**type**	**values**
status	token	**enable** ¦ **disable**
enable	byte	01_{16}
disable	byte	02_{16}

usage: This message is sent to a smart plug-in when its autoranging function is selected by a mainframe menu.

response: The plug-in will send the **autorange status** message.

error handling: none

also see: **autorange status**

command transactions:

autorange request : **autorange status**

name: **autorange status**

syntax: **autorange status** status

type: smart plug-in status

message tokens:

autorange ::= $1F_{16}$

status ::= 03_{16}

parameters:

name	**type**	**values**
status	token	**enable** ¦ **disable**
enable	byte	01_{16}
disable	byte	02_{16}

usage: This message is sent by a smart plug-in in response to the **autorange request** message. This message indicates the smart plug-in has entered the requested autoranging mode.

response: none

error handling: none

also see: **autorange request**

command transactions:

autorange request : autorange status

name: **aux_trig query**

syntax: **aux_trig query**

type: generic plug-in command

message tokens:

aux_trig ::= 29_{16}

query ::= 02_{16}

parameters: none

usage: This message requests a generic plug-in to report the status of its auxiliary trigger outputs.

response: The plug-in will send the **aux_trig status** message to report the auxiliary trigger status.

error handling: none

also see: **aux_trig status**, **aux_trig set**

command transactions:

aux_trig query : aux_trig status

name: **aux_trig set**

syntax: **aux_trig set** mode

type: generic plug-in command

message tokens:

aux_trig ::= 29_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
mode	token	**on** | **off**
on	byte	01_{16}
off	byte	02_{16}

usage: This message is sent by the mainframe to request a generic plug-in to enable or disable its auxiliary trigger outputs. The **on** *mode* token request the outputs to be enabled. The **off** *mode* token requests the outputs to be disabled. A smart plug-in in the right compartment will send this message in a **generic command** message to turn the auxiliary triggers on or off.

response: Generic plug-ins will send the **aux_trig status** message as the response.

error handling: none

also see: **aux_trig status**, **aux_trig query**, **generic command**

command transactions:

aux_trig set : aux_trig status

name: **aux_trig status**

syntax: **aux_trig status** mode

type: generic plug-in status

message tokens:

aux_trig ::= 29_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
mode	token	**on** ! **off**
on	byte	01_{16}
off	byte	02_{16}

usage: This message is sent by a generic plug-in in response to an **aux_trig set** or **aux_trig query** message. The **on** *mode* token indicates the auxiliary trigger outputs are enabled. The **off** *mode* token indicates the auxiliary trigger outputs are disabled.

response: none

error handling: none

also see: **aux_trig query**, **aux_trig set**

command transactions:

aux_trig query : aux_trig status
aux_trig set : aux_trig status

name: **bandwidth query lower**

syntax: **bandwidth query lower** channel

type: generic plug-in command

message tokens:

bandwidth ::= 05_{16}

query ::= 02_{16}

parameters:

name	**type**	**values**
lower	token	02_{16}
channel	us	**1 - 4**

usage: This message requests the plug-in to report the lower bandwidth limit for the specified *channel*.

response: The plug-in will report the lower bandwidth value using the **bandwidth status lower** message.

execution time: The plug-in will send the first byte of the **bandwidth status lower** message within xxms of sending the acknowledge transport packet for the **bandwidth query lower** message.

error handling: If the plug-in does not have lower bandwidth selection capability, the plug-in will report the error using the **error generic** message with the **command_error** *status* token and the *code* parameter set to 157 to indicate the problem. If a channel number is received as part of a **bandwidth query** command that a plug-in does not have, the plug-in will take unspecified action and return a **bandwidth status** message indicating the action it took.

also see: **bandwidth status lower**, **bandwidth set lower**

command transactions:

bandwidth query lower : bandwidth status lower
bandwidth query lower : error generic

name: bandwidth query upper

syntax: bandwidth query upper channel

type: generic plug-in command

message tokens:

bandwidth ::= 05_{16}

query ::= 02_{16}

parameters:

name	type	values
upper	token	01_{16}
channel	us	1 - 4

usage: This message requests the plug-in to report the upper bandwidth limit for the specified *channel*.

response: The plug-in will report the upper bandwidth value using the **bandwidth status upper** message.

execution time: The plug-in will send the first byte of the **bandwidth status upper** message within xxms of sending the acknowledge transport packet for the **bandwidth query upper** message.

error handling: If a channel number is received as part of a **bandwidth query** message that a plug-in does not have, the plug-in will take unspecified action and return a **bandwidth status** message indicating the action it took.

also see: **bandwidth status upper, bandwidth set upper**

command transactions:

bandwidth query upper : bandwidth status upper

name: **bandwidth set lower**

syntax: **bandwidth set lower** channel value

type: generic plug-in command

message tokens:

bandwidth ::= 05_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
lower	token	02_{16}
channel	us	1 - 4
value	float	

usage: This message requests a new lower bandwidth limit of *value* for the specified *channel*. The plug-in will set the lower bandwidth limit to the value nearest the requested *value*. The meaning of nearest is defined by the plug-in. This message controls a bandwidth limiting function in the plug-in and does not specify system bandwidth.

response: The plug-in will report the new lower bandwidth value using the **bandwidth status lower** message. If the plug-in determines that the bandwidth value is out of range, it will make no change to the bandwidth setting.

execution time: The plug-in will send the first byte of either the **bandwidth status lower** message or the **error generic** message within xxms of sending the acknowledge transport packet for the **bandwidth set lower** message.

error handling: If the plug-in does not have lower bandwidth selection capability, the plug-in will report the error using the **error generic** message with the **command_error** *status* token and the *code* parameter set to 157 to indicate the problem.

If a channel number is received as part of a **bandwidth set** message that a plug-in does not have, the plug-in will take unspecified action and return a **bandwidth status** message indicating the action it took.

If a bandwidth value is determined to be out of range, the plug-in will send an **error generic** message with the **exec_error** *status* token and the *code* parameter set to 205 to indicate the problem.

also see: **bandwidth status lower, error generic**

command transactions:

bandwidth set lower : [error generic :] bandwidth status lower
bandwidth set lower : error generic

name: **bandwidth set upper**

syntax: **bandwidth set upper** channel value

type: generic plug-in command

message tokens:

bandwidth ::= 05_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
upper	token	01_{16}
channel	us	1 - 4
value	float	

usage: This message requests a new upper bandwidth limit of *value* for the specified *channel*. The plug-in will set the upper bandwidth limit to the value nearest the requested *value*. The meaning of nearest is defined by the plug-in. This message controls a bandwidth limiting function in the plug-in and does not specify system bandwidth.

response: The plug-in will report the new upper bandwidth value using the **bandwidth status upper** message. If the bandwidth value is determined to be out of range, the plug-in will make no change to the bandwidth limit.

execution time: The plug-in will send the first byte of either the **bandwidth status upper** message or the **error generic** message within xxms of sending the acknowledge transport packet for the **bandwidth set upper** message.

error handling: If a channel number is received as part of a **bandwidth set** message that a plug-in does not have, the plug-in will take unspecified action and return a **bandwidth status** message indicating the action it took.

If a bandwidth value is determined to be out of range, the plug-in will send an **error generic** message with the **exec_error** *status* token and the *code* parameter set to 205 to indicate the problem.

also see: **bandwidth status upper, bandwidth set lower, error generic**

command transactions:

bandwidth set upper : [error generic :] bandwidth status upper

name: **bandwidth status lower**

syntax: **bandwidth status lower** channel value

type: generic plug-in status

message tokens:

bandwidth ::= 05_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
lower	token	02_{16}
channel	us	1 - 4
value	float	

usage: This message is used by a generic plug-in to report its lower bandwidth limit status. It is sent in response to a **bandwidth set lower** message or a **bandwidth query lower** message. The *channel* parameter identifies which channel's limit is being reported. The *value* parameter specifies the present setting of the lower bandwidth limit.

response: none

error handling: none

also see: **bandwidth set lower, bandwidth query lower**

command transactions:

bandwidth query lower : bandwidth status lower
bandwidth set lower : [error generic :] bandwidth status lower

name: **bandwidth status upper**

syntax: **bandwidth status upper** channel value

type: generic plug-in status

message tokens:

bandwidth ::= 05_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
upper	token	01_{16}
channel	us	1 - 4
value	float	

usage: This message is used by a generic plug-in to report its upper bandwidth limit status. It is sent in response to a **bandwidth set upper** message or a **bandwidth query upper** message. The *channel* parameter identifies which channel's limit is being reported. The *value* parameter specifies the present setting of the upper bandwidth limit.

response: none

error handling: none

also see: **bandwidth set upper**, **bandwidth query upper**

command transactions:

bandwidth query upper : bandwidth status upper
bandwidth set upper : [error generic :] bandwidth status upper

name: **busy query**

syntax: **busy query**

type: smart plug-in command

message tokens:

busy ::= 27_{16}

query ::= 02_{16}

parameters: none

usage: This message is sent by the mainframe to request the busy status of a smart plug-in. The mainframe will use the returned status to determine whether to begin a calibration cycle or whether to wait.

response: The smart plug-in will send the **busy status** message in response.

error handling: none

also see: **busy status**

command transactions:

busy query : busy status

name: **busy status**

syntax: **busy status** status

type: smart plug-in status

message tokens:

busy ::= 27_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
status	token	**busy** | **idle**
busy	byte	27_{16}
idle	byte	01_{16}

usage: This message is sent by a smart plug-in in response to a **busy query** message. If the plug-in is in a state that will allow calibration to begin, the plug-in will send the **idle** *status* token. If the plug-in is not in a state that will allow calibration to begin (such as armed for a trigger etc.), the plug-in will send the **busy** *status* token. The next time the plug-in reaches a state that would allow calibration to be performed, the plug-in will notify the mainframe that it is ready by sending an SRQ then sending the **busy status** message with the **idle** *status* token in response to the **SRQ query** message.

response: none

error handling: none

also see: **busy query**, **SRQ query**

command transactions:

busy query : busy status
SRQ query : busy status

name: cal acc_query

syntax: cal acc_query

type: mainframe command

message tokens:

cal ::= 17_{16}

acc_query ::= 02_{16}

parameters: none

usage: This message is sent by a smart plug-in with a special calibration function to request the present accuracy state of the accuracy system. The plug-in will send this message after the end of a new configuration or full calibration procedure. The end of these procedures is identified by the receipt of the **cal mode exit** message. The plug-in will send an SRQ then send the **cal acc_query** message in response to the subsequent **SRQ query** message.

response: The mainframe will send the **cal acc_status** message to indicate the present accuracy system state.

error handling: none

also see: cal acc_status

command transactions:

cal acc_query : cal acc_status

name: **cal acc_status**

syntax: **cal acc_status** mode

type: mainframe/smart plug-in status

message tokens:

cal ::= 17_{16}

acc_status ::= 13_{16}

parameters:

name	**type**	**values**
mode	token	**warmup** | **normal** | **enhanced** | **auto_enhanced**
warmup	byte	01_{16}
normal	byte	09_{16}
enhanced	byte	02_{16}
auto_enhanced	byte	03_{16}

usage: This message is sent by the mainframe in response to a **cal acc_query** message to notify a plug-in of the accuracy system state. The **warmup** *mode* token indicates the accuracy system is in the warmup mode and enhanced accuracy is not available. The **normal** *mode* token indicates the accuracy system is in the normal accuracy state. The **enhanced** *mode* token indicates the accuracy system is in the enhanced accuracy state. This state will not be automatically maintained. The **auto_enhanced** *mode* token indicates the accuracy system is in the auto enhanced accuracy state. In this state, a plug-in with special calibration requirements must request calibration of the mainframe to maintain the enhanced accuracy state. This request will be sent when the plug-in's internal criteria for needing a special calibration have been met.

This message is also sent by a plug-in with special calibration requirements when it has exited the enhanced accuracy **state** when the system accuracy **state is enhanced.** See the *11000 Series Accuracy System* document. The plug-in will send the **normal** *mode* token in this case.

response: When this message is sent by a plug-in, the mainframe will send a **cal acc_status** message in reply.

error handling: none

also see: cal acc_query

command transactions:

cal acc_query : cal acc_status
cal acc_status : cal acc_status
cal req_cal : cal acc_status

name: **cal const_num**

syntax: **cal const_num** number

type: **plug-in status**

message tokens:

cal ::= 17_{16}

const_num ::= 17_{16}

parameters:

name	**type**	**values**
number	int	

usage: This message is sent by a plug-in in response to the **cal const_query_num** message to report the number of calibration constants in the plug-in. The *number* parameter specifies the number of calibration constants. The mainframe may query for any constant value from 0 to *number* using the **cal const_query** message.

response: none

error handling: none

also see: **cal const_query_num**, **cal const_query**, **cal_const_status**, **cal const_set**

command transactions:

cal const_query_num : cal const_num

name: **cal const_query**

syntax: **cal const_query** const_num

type: plug-in command

message tokens:

cal ::= 17_{16}

const_query ::= 10_{16}

parameters:

name	**type**	**values**
const_num	ui	

usage: This message is sent by the mainframe to request a calibration constant from a plug-in. The *const_num* parameter is the calibration constant identification number used by the plug-in.

response: The plug-in will send the **cal const_status** message as the response and will report the requested calibration constant value.

error handling: none

also see: **cal const_set**, **cal const_num**, **cal query_const_num**, **cal const_status**

command transactions:

cal const_query : cal const_status

name: **cal const_query_num**

syntax: **cal const_query_num**

type: plug-in command

message tokens:

cal ::= 17_{16}

const_query_num ::= 16_{16}

parameters: none

usage: This message is sent by the mainframe to determine the number of calibration constants contained in a plug-in.

response: The plug-in will send the **cal const_num** message as the response.

error handling: none

also see: **cal const_num**, **cal const_query**, **cal_const_status**

command transactions:

cal const_query_num : cal const_num

name: cal const_set

syntax: cal const_set const_num value

type: plug-in command

message tokens:

cal ::= 17_{16}

const_set ::= 11_{16}

parameters:

name	**type**	**values**
const_num	ui	
value	float	

usage: This message is sent by the mainframe to set calibration constant values in a plug-in. The *const_num* parameter specifies the constant number for which the *value* parameter applies. The plug-in will set the calibration constant associated with *const_num* to the value specified by the *value* parameter.

response: The plug-in will send the **cal const_status** message to report constant status.

error handling: none

also see: **cal const_status**, **cal const_query**, **cal const_num**, **cal const_query_num**

command transactions:

cal const_set : cal const_status

name: **cal const_status**

syntax: **cal const_status** const_num value

type: plug-in status

message tokens:

cal ::= 17_{16}

const_status ::= 12_{16}

parameters:

name	**type**	**values**
const_num	ui	
value	float	

usage: This message is sent by the plug-in in response to a **cal const_set** or **cal const_query** message. The plug-in will send a *const_num value* pair as specified by the **cal const_set** or **cal const_query** messages. The *const_num* parameter specifies the constant number whose value is reported by the *value* parameter.

response: none

error handling: none

also see: **cal const_set**, **cal const_query**, **cal const_num**, **cal const_query_num**

command transactions:

cal const_set : cal const_status
cal const_query : cal const_status

name: cal cvr_connect

syntax: cal cvr_connect compartment mode [gain]

type: mainframe command

message tokens:

cal ::= 17_{16}

cvr_connect ::= 08_{16}

parameters:

name	**type**	**values**
compartment	char	'L' | 'C' | 'R'
mode	token	**enter** | **exit** | **ground** | **CVR**
enter	byte	01_{16}
exit	byte	02_{16}
ground	byte	03_{16}
CVR	byte	04_{16}
gain	float	

usage: This message is sent by a smart plug-in to request the mainframe to connect the CVR to the specified *compartment* and to send a **cal cvr_mode** message to the plug-in in that compartment. The mainframe will send the **cal cvr_mode** message to the generic plug-in using the *mode* token sent in the **cal cvr_connect** message by the smart plug-in. The mainframe will wait for the generic plug-in to respond then report the results to the smart plug-in. When the generic plug-in reports its gain setting in response to a **cal cvr_mode enter** message, the mainframe will report that *gain* value to the smart plug-in in the **cal cvr_connect** message.

The **enter** *mode* token requests the generic plug-in to enter CVR pass-through mode as defined above. The plug-in will modify its settings as defined in section **5.3.10 CVR Pass-Through Mode** and connect its input to the CVR. The mainframe will wait for the response from the generic plug-in then report the *mode* and *gain* setting to the requesting smart plug-in.

The **CVR** *mode* token requests the input to be connected to the CVR circuit. This will normally be sent after a **cal cvr_connect** with the **ground** *mode* token to re-connect the input to the CVR.

The **ground** *mode* token requests the input to be connected to reference ground.

The **exit** *mode* token requests the mainframe to disconnect the ground reference from the specified compartment and send a **cal cvr_mode exit** message to the plug-in in that compartment. The plug-in will exit the CVR pass-through mode and restore all function settings to the values that were active when CVR pass-through mode was entered (unless a change was requested during CVR pass-through mode). When the plug-in has completed the restoration and connected its input to the input signal, it will send the **cal cvr_mode** message with the **exit** *mode* token. The mainframe will then send the **cal cvr_connect** message with the **exit** *mode* token to the requesting smart plug-in.

The *gain* parameter specifies the gain setting of the plug-in. This setting is chosen by the plug-in to be the optimal choice for calibration. It is included in the response form the mainframe only when the *mode* token is **enter**.

response: The mainframe will send the **cal cvr_connect** message to the requesting plug-in when it has completed the connection change and the plug-in in the specified *compartment* has returned a **cal cvr_mode** message.

error handling: none

also see: **cal cvr_mode**, **cal cvr_set**, **cal cvr_status**

command transactions:

cal cvr_connect : cal cvr_connect

name: cal cvr_mode

syntax: cal cvr_mode mode [gain]

type: plug-in command

message tokens:

cal	::= 17_{16}
cvr_mode	::= 09_{16}

parameters:

name	**type**	**values**
mode	token	**enter** | **exit** | **ground** | **CVR**
enter	byte	01_{16}
exit	byte	02_{16}
ground	byte	03_{16}
CVR	byte	04_{16}
gain	float	

usage: This message is sent by the mainframe to a generic plug-in to cause it to connect all of its inputs to the CVR circuit rather than the external inputs.

The **enter** *mode* token requests the plug-in to enter CVR pass-through mode. The plug-in will connect its input to the CVR signal and will also adjust all of its functions to provide the optimal settings for performing calibration. The plug-in will save its present settings for the gain, offset, coupling, bandwidth, impedance, display and trigger functions for restoration when the CVR pass-through mode is exited.

Offset will be set to 0. The input will be DC coupled to the CVR. Bandwidth will be set to the optimal value for noise reduction in the plug-in. This bandwidth setting will not be so low as to affect the settling time for measurements. The input impedance will be set to a value appropriate for connection to the CVR. The display and trigger output combinations will be set to channel 1 only, not inverted. None of the settings of these function will be reported.

The gain will be set to the optimal gain setting for that plug-in for calibration. The plug-in will report this gain value when it sends the **cal cvr_mode** message with the **enter** *mode* token. The mainframe or smart plug-in that requested the CVR pass-through mode will use this gain value and the default offset setting to perform calibration of their circuits.

The **CVR** *mode* token requests the plug-in to connect its input to the CVR circuit. This will normally be sent after a **cal cvr_mode** message with the **ground** *mode* token to re-connect the input to the CVR. The plug-in will send the **cal cvr_mode** message with the **CVR** *mode* token when its input is connected to the CVR.

The **ground** *mode* token requests the plug-in to connect its input to reference ground. The plug-in will send the **cal cvr_mode** message with the **ground** *mode* token when its input is connected to ground.

The **exit** *mode* token requests the plug-in to exit the CVR pass-through mode. The plug-in will restore all function settings to the values that were active when CVR pass-through mode was entered (unless a change was requested during CVR pass-through mode). When the plug-in has completed the restoration and connected its input to the input signal, it will send the **cal cvr_mode** message with the **exit** *mode* token.

The *gain* parameter specifies the gain setting of the plug-in. This setting is chosen by the plug-in to be the optimal choice for calibration. It is included in the response ~~form~~ from the plug-in only when the *mode* token is **enter**.

response: The plug-in will send the **cal cvr_mode** message with the appropriate *mode* token and possibly a gain value as a response.

error handling: none

also see: **cal cvr_connect, cal cvr_set, cal cvr_status**

command transactions:

cal cvr_mode : cal cvr_mode

name: **cal cvr_set**

syntax: **cal cvr_set** value

type: mainframe command

message tokens:

cal ::= 17_{16}

cvr_set ::= 04_{16}

parameters:

name	**type**	**values**
value	float	0 – ±10

usage: This message requests the mainframe to set its calibrated voltage reference to the specified *value* in volts. The mainframe will check if *value* is within its achievable limits. If it is, the mainframe will set the calibrator to the requested *value*. If it is not, the mainframe will set the calibrator to its maximum or minimum value. If *value* specifies more resolution than the mainframe calibrator can achieve, the mainframe will round *value* to the nearest achievable setting and set the calibrator to that value. This message does not change the connection of the calibrator reference ground.

response: The mainframe will report the new calibrator setting to the plug-in that requested the change using the **cal cvr_status** message.

error handling: none

also see: **cal cvr_status**

command transactions:

cal cvr_set : cal cvr_status

name: **cal cvr_status**

syntax: **cal cvr_status** value

type: mainframe status

message tokens:

cal ::= 17_{16}

cvr_status ::= 05_{16}

parameters:

name	**type**	**values**
value	float	0 - ±10

usage: The mainframe will send this message to a plug-in that has requested a calibrator setting using the **cal cvr_set** message. The *value* reported will be the actual calibrator output in volts.

response: none

error handling: none

also see: **cal cvr_set**

command transactions:

cal cvr_set : cal cvr_status

name: **cal measure**

syntax: **cal measure**

type: mainframe command

message tokens:

cal ::= 17_{16}

measure ::= 06_{16}

parameters: none

usage: A plug-in uses this message to request a voltage measurement to be made by the mainframe. The mainframe will make a measurement on the signal path (display or trigger) appropriate to the calibration mode in effect at the time of the request.

Based on the time value specified by the **cal meas_set** message, the mainframe will take appropriate steps (such as adjusting the number of averages and the length of time over which the averaging is done) in order to insure proper accuracy of the measurement. The mainframe will also take into account standard noise generated by the CVR and mainframe signal paths.

response: The mainframe will send the value measured to the requesting plug-in using the **cal meas_value** message.

error handling: none

also see: **cal meas_value**

command transactions:

cal measure : cal meas_value

name: **cal meas_set**

syntax: **cal meas_set** time

type: mainframe command/status

message tokens:

cal	::= 17_{16}
meas_set	::= 01_{16}

parameters:

name	**type**	**values**
time	token	**x1** ¦ **x2** ¦ **x4** ¦ **x8**
x1	byte	01_{16}
x2	byte	02_{16}
x4	byte	03_{16}
x8	byte	04_{16}

usage: This message is sent by a plug-in to set the requirements for measurements made by the mainframe in response to a **cal measure** message. The *time* tokens indicate the relative amount of time determined by the plug-in needed to perform an accurate measurement based on the noise of the plug-in signal path being measured. The token values are relative to the standard value of **x1**. **x2** indicates twice as much noise as **x1**. **x4** and **x8** indicated 4 and 8 times as much noise, respectively, as **x1**. The mainframe will maintain the state of this request for each plug-in individually. The mainframe will make measurements for each plug-in based on the *time* specified by that plug-in.

response: The mainframe will send the **cal meas_set** message with the requested *time* parameter as the response.

error handling: none

also see: **cal measure**, **cal meas_value**

command transactions:

cal meas_set : **cal meas_set**

name: **cal meas_value**

syntax: **cal meas_value** value

type: mainframe status

message tokens:

cal ::= 17_{16}

meas_value ::= 07_{16}

parameters:

name	**type**	**values**
value	float	0 - ±10

usage: The mainframe uses this message to report the value measured in response to a **cal measure** message. The *value* indicates divisions of deflection. Divisions are defined as 50mV per division at the plug-in/mainframe interface connector.

response: none

error handling: none

also see: **cal meas_value**

command transactions:

cal measure : cal meas_value

name: **cal mf_imbalance**

syntax: **cal mf_imbalance** mf_value

type: generic plug-in status

message tokens:

cal ::= 17_{16}

mf_imbalance ::= $0B_{16}$

parameters:

name	**type**	**values**
mf_value	float	

usage: This message is sent by a generic plug-in to report plug-in mainframe input imbalance. This message is sent to report either display path or trigger path imbalances. The calibration mode determines the path for which the values are being reported.

The *mf_value* parameter is the mainframe imbalance value in divisions as measured by the plug-in. The plug-in will measure and report this value at times defined by the calibration procedures.

response: The mainframe will send the **cal status** message with the **ready** token as the response.

error handling: none

also see: **cal status**

command transactions:

cal mf_imbalance : cal status

name: **cal mode**

syntax: **cal mode** mode [meas_opt] [compartment [compartment]]

type: plug-in command

message tokens:

cal	::=	17_{16}
mode	::=	$0A_{16}$

parameters:

name	type	values
mode	token	**enter** ¦ **exit** ¦ **hold** ¦ **dp_cal_woc** ¦ **dp_cal_wc** ¦ **dtp_imb_woc** ¦ **new_config** ¦ **other** ¦ **special**
enter	byte	01_{16}
exit	byte	02_{16}
hold	byte	07_{16}
dp_cal_woc	byte	03_{16}
dp_cal_wc	byte	04_{16}
dtp_imb_woc	byte	05_{16}
new_config	byte	08_{16}
other	byte	06_{16}
special	byte	09_{16}
meas_opt	float	0 - 5
compartment	char	'L', 'C', 'R'

usage: This message is sent to plug-ins to select the different modes of calibration.

The **enter** *mode* token notifies the plug-in that the calibration process is being started. The plug-in will enter the *Initial Calibration Mode*.

The **hold** *mode* token notifies the plug-in that the calibration process is being started and requests the plug-in to enter the *Hold Calibration Mode*.

The **exit** *mode* token notifies the plug-in that the calibration process is completed. The plug-in will return to normal operating mode.

The **dp_cal_woc** *mode* token causes a generic plug-in to begin calibration of the display path without the use of the CVR. The plug-in will report the mainframe input imbalance value. The plug-in may not request CVR voltages in this mode.

The **dp_cal_wc** *mode* token causes a generic plug-in to complete signal path calibration. In this mode, the plug-in may request CVR voltages and will complete calibration of the display signal paths.

The **dtp_imb_woc** *mode* token causes a generic plug-in to measure and report imbalance values for the display and trigger signal paths for each channel. The plug-in will report these values to the mainframe. The plug-in may not request CVR voltages in this mode.

The **new_config** *mode* token is sent to a smart plug-in to cause it to calibrate circuits affected by a change in the system configuration. This will be used by triggering plug-ins for calibrating the auxiliary trigger lines when a new generic plug-in is installed in the system.

The **other** *mode* token causes a smart plug-in to perform any calibration that is not related to the display or trigger signal paths but requires the use of the CVR. This mode will be entered after the display and trigger paths have been calibrated.

The **special** *mode* token causes a plug-in to begin the calibration of special functions. These are functions that require calibration based on criteria other than the normal system criteria.

The *meas_opt* parameter is included when the *mode* token is **dp_cal_woc, dp_cal_wc, dtp_imb_woc, new_config, special** or **other**. This value specifies the mainframe's bands of optimal measurement performance in divisions from center screen. The measurement bands have a width of ±5%. This value is always positive. It can be different for each mode.

The *compartment* parameters are included only when the *mode* token is **new_config**. These parameters indicate the presence of a new plug-in in the system configuration. This information will be used by a smart plug-in to calibrate circuits (ie. the auxiliary trigger lines) related to the new plug-in. At least one and no more than two *compartment* parameters must be sent with the **new_config** *mode* token.

response: The plug-in will send the **cal status** message with the **busy** *status* token when the *mode* token is **dp_cal_woc, dp_cal_wc,**

dtp_imb_woc, **new_config**, **special** or **other**. The plug-in will send the **ready** *status* token when the *mode* token is **enter**, **hold** or **exit**.

error handling: none

also see: **cal status**

command transactions:

cal mode : cal status

name: **cal pi_imbalance**

syntax: **cal pi_imbalance** {channel pre_invert post_invert}...

type: generic plug-in status

message tokens:

cal ::= 17_{16}

pi_imbalance ::= $0C_{16}$

parameters:

name	**type**	**values**
channel	us	1 - 4
pre_invert	float	
post_invert	float	

usage: This message is sent by a generic plug-in to report plug-in channel imbalances. This message is sent to report either display path or trigger path imbalances. The calibration mode determines which values are being reported.

The *channel* parameter indicates for which channel the following imbalances apply.

The *pre_invert* parameter is the imbalance value for the indicated channel that occurs prior to the inversion circuit. The *post_invert* parameter is the imbalance value that occurs after the inversion circuit for the indicated channel. There is a *pre_invert* and *post_invert* imbalance pair for each specified channel and they are in order of lowest to highest channel number. These values are in divisions.

The plug-in may report all channels in a single message or may send individual messages for each channel. If a channel is determined to be non-functional, the plug-in is not required to report imbalance values for that channel.

response: The mainframe will send the **cal status** message with the **ready** token.

error handling: none

also see: **cal status**

command transactions:

cal pi_imbalance : cal status

name: **cal probe**

syntax: **cal probe** status input channel

type: mainframe/generic plug-in command/status

message tokens:

cal	::= 17_{16}
probe	::= 15_{16}

parameters:

name	**type**	**values**
status	token	**begin** ¦ **busy** ¦ **completed** ¦ **error** ¦ **connect**
begin	byte	02_{16}
busy	byte	27_{16}
completed	byte	01_{16}
error	byte	12_{16}
connect	byte	03_{16}
input	token	**plus** ¦ **minus**
plus	byte	01_{16}
minus	byte	02_{16}
channel	us	1 - 4

usage: This message is used to control the calibration of probes. It is sent either during the *Final Calibration Mode* as part of calibration or during normal operation when requested by the user.

When sent by the mainframe with the **begin** *status* token, this message requests a generic plug-in to calibrate the probe connected to the specified *input* and *channel*. The **plus** *input* token specifies the plus input of a differential channel or the input of a single ended channel. The **minus** *input* token specifies the minus input of a differential channel only. The plug-in will send the **cal probe** message with the **busy** *status* token as a response. When it has completed the calibration of the probe, the plug-in will send the **cal probe** message with the **completed** *status* token. The *channel* parameter will indicate the channel that was successfully calibrated.

If the plug-in cannot complete the calibration of the probe, it will send the **cal probe** message with the **error** *status* token. If the plug-in determines that the probe is not connected to the mainframe's calibrator, it will send the **cal probe** message with the **connect** *status* token.

response: The plug-in will respond with the **cal probe busy** message. The mainframe will respond with the **cal probe completed** message in response to the **cal probe** message with the **complete**, **error** or **connect** status tokens from the plug-in.

error handling: included

also see:

command transactions:

cal probe : cal probe

name: **cal probe_act query**

syntax: **cal probe_act query** *input channel*

type: generic plug-in command

message tokens:

cal ::= 17_{16}

probe_act ::= 18_{16}

parameters:

name	**type**	**values**
query	token	02_{16}
input	token	**plus** ¦ **minus**
plus	byte	01_{16}
minus	byte	02_{16}
channel	us	1 - 4

usage: This message is sent by the mainframe to request the value of the probe actual calibration constant. This is the value that represents the probe's actual attenuation either as measured by the plug-in or as set by the mainframe. The **plus** *input* token specifies the plus input of a differential channel or the input of a single ended channel. The **minus** *input* token specifies the minus input of a differential channel only. The *channel* parameter specifies the channel for which the probe actual value is requested.

response: The plug-in will send the **cal probe_act status** message to report the value of the probe actual calibration constant.

error handling: If the *channel* parameter specifies an illegal channel number, the plug-in will report the probe actual value for a valid channel. No error message will be generated. If the **minus** *input* token is received by a plug-in that does not have differential channels, the plug-in will send the **error generic** message with the **command_error** *status* token and the *code* value set to 157.

also see: cal probe_act status, cal probe_act set, cal probe_nom query, cal probe_nom status, cal probe_nom set

command transactions:

cal probe_act query : cal probe_act status

name: **cal probe_act set**

syntax: **cal probe_act set** input channel value

type: generic plug-in command

message tokens:

cal ::= 17_{16}

probe_act ::= 18_{16}

parameters:

name	**type**	**values**
set	token	01_{16}
input	token	**plus** ¦ **minus**
plus	byte	01_{16}
minus	byte	02_{16}
channel	us	1 - 4
value	float	

usage: This message is sent by the mainframe to change the probe actual value. This value represents the probe's actual attenuation either as measured by the plug-in or as set by the mainframe using this message. The **plus** *input* token specifies the plus input of a differential channel or the input of a single ended channel. The **minus** *input* token specifies the minus input of a differential channel only. The *channel* parameter specifies which channel's constant is to be changed. The *value* parameter specifies the new value of the probe actual calibration constant.

For this message, the plug-in determines the range of the control. The range will be at least q2% of the probe nominal value. If the mainframe requests a value outside these limits, the plus-in will set the value to the minimum or maximum value and report that value using the **cal probe_act status** message.

response: The plug-in will send the **cal probe_act status** message to report the new value of the probe actual calibration constant.

error handling: If the *channel* parameter specifies an illegal channel number, the plug-in will report the probe actual value for a valid channel. No error message will be generated. If the **minus** *input* token is received by a plug-in that does not have differential channels, the plug-in will send the **error generic** message with the **command_error** *status* token and the *code* value set to 157.

also see: **cal probe_act query, cal probe_act status, cal probe_nom query, cal probe_nom set, cal probe_nom status**

command transactions:

cal probe_nom set : cal probe_nom status

name: **cal probe_act status**

syntax: **cal probe_act status** input channel value

type: generic plug-in status

message tokens:

cal ::= 17_{16}

probe_act ::= 18_{16}

parameters:

name	type	values
set	token	01_{16}
input	token	**plus** ¦ **minus**
plus	byte	01_{16}
minus	byte	02_{16}
channel	us	1 - 4
value	float	

usage: This message is sent by a generic plug-in in response to either the **cal probe_act set** or **cal probe_act query** messages. The **plus** *input* token specifies the plus input of a differential channel or the input of a single ended channel. The **minus** *input* token specifies the minus input of a differential channel only. The *channel* parameter specifies which channel's constant is being reported. The *value* parameter is the present value of the probe actual calibration constant.

response: none

error handling: none

also see: **cal probe_act query**, **cal probe_act set**, **cal probe_nom query**, **cal probe_nom set**, **cal probe_nom status**

command transactions:

cal probe_act query : cal probe_act status
cal probe_act set : cal probe_act status

name: **cal probe_nom query**

syntax: **cal probe_nom query** input channel

type: generic plug-in command

message tokens:

cal ::= 17_{16}

probe_nom ::= 19_{16}

parameters:

name	type	values
query	token	02_{16}
input	token	**plus** ¦ **minus**
plus	byte	01_{16}
minus	byte	02_{16}
channel	us	1 - 4

usage: This message is sent by the mainframe to request the value of the probe nominal calibration constant. This is the value that represents the probe's nominal attenuation either as reported by the probe or as set by the mainframe. The **plus** *input* token specifies the plus input of a differential channel or the input of a single ended channel. The **minus** *input* token specifies the minus input of a differential channel only. The *channel* parameter specifies the channel for which the probe nominal value is requested.

response: The plug-in will send the **cal probe_nom status** message to report the value of the probe nominal calibration constant.

error handling: If the *channel* parameter specifies an illegal channel number, the plug-in will report the probe nominal value for a valid channel. No error message will be generated. If the **minus** *input* token is received by a plug-in that does not have differential channels, the plug-in will send the **error generic** message with the **command_error** *status* token and the *code* value set to 157.

also see: **cal probe_nom status, cal probe_nom set, cal probe_act query, cal probe_act status, cal probe_act set**

command transactions:

cal probe_nom query : cal probe_nom status

name: **cal probe_nom set**

syntax: **cal probe_nom set** input channel value

type: generic plug-in command

message tokens:

cal ::= 17_{16}

probe_nom ::= 19_{16}

parameters:

name	type	values
set	token	01_{16}
input	token	**plus** | **minus**
plus	byte	01_{16}
minus	byte	02_{16}
channel	us	1 - 4
value	float	

usage: This message is sent by the mainframe to change the probe nominal value. This value represents the probe's nominal attenuation either as reported by the probe or as set by the mainframe using this message. The **plus** *input* token specifies the plus input of a differential channel or the input of a single ended channel. The **minus** *input* token specifies the minus input of a differential channel only. The *channel* parameter specifies which channel's constant is to be changed. The *value* parameter specifies the new value of the probe nominal calibration constant.

response: The plug-in will send the **cal probe_nom status** message to report the new value of the probe nominal calibration constant.

error handling: If the *channel* parameter specifies an illegal channel number, the plug-in will report the probe nominal value for a valid channel. No error message will be generated. If the **minus** *input* token is received by a plug-in that does not have differential channels, the plug-in will send the **error generic** message with the **command_error** *status* token and the *code* value set to 157.

also see: **cal probe_nom query**, **cal probe_nom status**, **cal probe_act query**, **cal probe_act set**, **cal probe_act status**

command transactions:

cal probe_nom set : cal probe_nom status

name: **cal probe_nom status**

syntax: **cal probe_nom status** input channel value

type: generic plug-in status

message tokens:

cal ::= 17_{16}

probe_nom ::= 19_{16}

parameters:

name	type	values
set	token	01_{16}
input	token	**plus** ! **minus**
plus	byte	01_{16}
minus	byte	02_{16}
channel	us	1 - 4
value	float	

usage: This message is sent by a generic plug-in in response to either the **cal probe_nom set** or **cal probe_nom query** messages. The **plus** *input* token specifies the plus input of a differential channel or the input of a single ended channel. The **minus** *input* token specifies the minus input of a differential channel only. The *channel* parameter specifies which channel's constant is being reported. The *value* parameter is the present value of the probe nominal calibration constant.

response: none

error handling: none

also see: **cal probe_nom query**, **cal probe_nom set**, **cal probe_act query**, **cal probe_act set**, **cal probe_act status**

command transactions:

cal probe_nom query : cal probe_nom status
cal probe_nom set : cal probe_nom status

name: **cal req_cal**

syntax: **cal req_cal**

type: mainframe command

message tokens:

cal 17_{16}

req_cal 14_{16}

parameters: none

usage: This message is sent by a plug-in with a special calibration function. This message will be sent when the accuracy system is in the auto enhanced mode and the plug-in has determined that its special function requires calibration. The plug-in will base its decision on the latest information it received in response to a **cal acc_query** message.

response: The mainframe will send the **cal mode** message with the appropriate *mode* token to start calibration if the accuracy system is in the auto enhanced accuracy mode. Since the accuracy system may have exited the auto enhanced mode without notifying the plug-in, the mainframe will send the **cal acc_status** message with the appropriate *mode* token if the accuracy system is not in auto enhanced mode.

error handling: none

also see: **cal acc_query**, **cal acc_status**, **cal mode**

command transactions:

cal req_cal : cal mode
cal req_cal : cal acc_status

name: cal status

syntax: cal status status [channels]

type: mainframe/plug-in status

message tokens:

cal	::= 17_{16}
status	::= 03_{16}

parameters:

name	**type**	**values**
status	token	**ready** | **busy** | **complete** | **fail** | **fail_chan**
ready	token	01_{16}
busy	byte	27_{16}
complete	byte	04_{16}
fail	byte	02_{16}
fail_chan	byte	03_{16}
channels	special	b7 b6 b5 b4 b3 b2 b1 b0 X X X X CH4 CH3 CH2 CH1

usage: This message is sent by a plug-in to report calibration status and values. The *status* parameter indicates the state of the plug-in.

The **ready** *status* token is used when this message is sent as a response to another calibration message and indicates the plug-in is has entered the requested mode and is ready for the next message.

The **busy** *status* token indicates the plug-in is performing the requested calibration operation. It is sent in response to a **cal mode** message with the **dp_cal_woc**, **dp_cal_wc**, **dtp_imb_woc**, **new_config**, **special** or **other** mode tokens.

The **complete** *status* token indicates the plug-in has completed the active calibration mode and has not detected any errors with the calibration process.

The **fail** *status* token indicates the plug-in has failed to achieve a calibrated state. Diagnostics may be invoked to determine the cause of the failure.

The **fail_chan** *status* token indicates the failure of a generic plug-in to calibrate one of its channels. In this case, the *channels* parameter indicates the channels that failed.

The *channels* parameter indicates which channels have failed and is only included when the *status* tokens **fail_chan**. If a bit is set, the indicated channel failed calibration. If a bit is reset, the indicated channel passed calibration.

response: The mainframe will send the **cal status** message with the **ready** *status* token in response to the plug-in sending the **cal status** message with the **complete**, **fail** or **fail_chan** *status* token.

error handling: none

also see: **cal mode**, **cal pi_imbalance**, **cal mf_imbalance**

command transactions:

cal mode : cal status
cal status : cal status

name: **cal tv_nom**

syntax: **cal tv_nom** value

type: smart plug-in status

message tokens:

cal	::= 17_{16}
tv_nom	::= $0E_{16}$

parameters:

name	**type**	**values**
value	float	

usage: This message is sent by the plug-in in response to the **cal tv_status** message with the **send_val** *status* token. The *value* parameter represents the nominal delay through the plug-in in seconds.

response: none

error handling: none

also see: **cal tv_status**

command transactions:

cal tv_status : cal tv_nom

name: **cal tv_set**

syntax: **cal tv_set** level slope coupling

type: smart plug-in status

message tokens:

cal	::=	17_{16}
tv_set	::=	$0F_{16}$

parameters:

name	type	values
level	float	
slope	token	**plus** ! **minus**
plus	byte	01_{16}
minus	byte	02_{16}
coupling	token	**AC** ! **DC**
AC	byte	03_{16}
DC	byte	01_{16}

usage: This message is sent by a triggering plug-in in the right compartment in response to the **cal tv_status** message with the **begin** *status* token.

The *level* parameter specifies the level the mainframe should set its trigger for responding to the plug-in's trigger output.

The *slope* parameter specifies the slope the mainframe should set its trigger to.

The *coupling* parameter specifies the trigger coupling the mainframe should select.

response: none

error handling: none

also see: **cal tv_status**

command transactions:

cal tv_status : cal tv_set

name: **cal tv_status**

syntax: **cal tv_status** status [channel slope level]

type: mainframe/smart plug-in command/status

message tokens:

cal ::= 17_{16}

tv_status ::= $0D_{16}$

parameters:

name	**type**	**values**
status	token	**begin** ¦ **complete** ¦ **send_val** ¦ **ready**
begin	byte	02_{16}
complete	byte	04_{16}
send_val	byte	03_{16}
ready	byte	01_{16}
channel	us	1 - 8
slope	token	**plus** ¦ **minus**
plus	byte	01_{16}
minus	byte	02_{16}
level	float	-5.0 to +5.0

usage: This message is sent by the mainframe to perform trigger view calibration.

The **begin** *status* token requests the plug-in in the right compartment to connect its trigger output to the specified auxiliary trigger line. The mainframe will make measurements of the delay through the plug-in. The *channel*, *slope* and *level* parameters are included when the *status* token is **begin**.

The **completed** *status* token indicates the mainframe has completed the delay measurement.

The **send_val** *status* token notifies the plug-in that the mainframe does not have the capability to display a trigger signal from the right compartment. The mainframe will instead place a marker on the screen. The plug-in is requested by this *status* token to send the nominal value of delay through the plug-in using the **cal tv_nom** message. The mainframe will use this value to properly place the trigger view marker.

The **ready** *status* token is used by the plug-in to report its status when the mainframe sends the **cal tv_status completed** message.

The *channel* parameter indicates the reference channel that will be used by the mainframe for making the delay measurement. The plug-in will set itself to trigger on that channel. Values 1 - 4 refer to left plug-in channels 1 - 4. Values 5 - 8 refer to center plug-in channels 1 - 4. This parameter is sent only when the *status* token is **begin**.

The *slope* parameter indicates the slope to which the plug-in should set its internal trigger. The **plus** *slope* token selects plus slope. The **minus** *slope* token selects minus slope. This parameter is sent only when the *status* token is **begin**.

The *level* parameter specifies the midpoint of the trigger signal on the specified auxiliary trigger line. The plug-in will set its trigger level (or threshold comparison level) to this value to insure maximum accuracy of the delay measurement. This parameter is sent only when the *status* token is **begin**.

response: The plug-in will send the **cal tv_set** message in response to the **cal tv_status begin** message. The plug-in will send the **cal tv_nom** message in response to the **cal tv_status send_val** message. The plug-in will send the **cal tv_status ready** message in response to the **cal tv_status completed** message.

error handling: none

also see: **cal tv_set**, **cal tv_nom**

command transactions:

cal tv_status : cal tv_set
cal tv_status : cal tv_nom
cal tv_status : cal tv_status

name: **channel_id status**

syntax: **channel_id status** channel

type: generic plug-in status

message tokens:

channel_id ::= $0C_{16}$

status ::= 03_{16}

parameters:

name	**type**	**values**
channel	us	1 - 4

usage: This message is used by generic plug-ins to report the status of the front panel display on/off button. This message will be sent whenever a channel id button is pressed.

response: none

execution time:

error handling: none

also see: **probe_id**

command transactions:

SRQ query : channel_id status

name: **coupling query minus**

syntax: **coupling query minus** channel

type: generic plug-in command

message tokens:

coupling ::= 03_{16}

query ::= 02_{16}

parameters:

name	**type**	**values**
minus	token	02_{16}
channel	us	1 – 4

usage: This message requests the plug-in to report the input coupling setting for the inverting input of the specified differential *channel*. This message is only valid for differential input plug-ins.

response: The plug-in will report the coupling status using the **coupling status minus** message.

execution time: The plug-in will send the first byte of either the **coupling status minus** message or the **error generic** message within xxms of sending the acknowledge transport packet for the **coupling query minus** message.

error handling: If the plug-in does not have a differential input for the specified channel, the plug-in will report the error using the **error generic** message with the **command_error** *status* token and the *code* parameter set to 157 to indicate the problem.

If a channel number is received as part of a **coupling query** message that a plug-in does not have, the plug-in will take unspecified action and return a **coupling status** message indicating the action it took.

also see: **coupling status minus**, **coupling set plus**, **coupling status plus**

command transactions:

coupling query minus : coupling status minus
coupling query minus : error generic

name: **coupling query plus**

syntax: **coupling query plus** channel

type: generic plug-in command

message tokens:

coupling ::= 03_{16}

query ::= 02_{16}

parameters:

name	**type**	**values**
plus	token	01_{16}
channel	us	1 - 4

usage: This message requests the plug-in to report the input coupling setting for the specified *channel* for single ended channels. The plug-in will report the coupling setting for the non-inverting input for differential channels.

response: The plug-in will report the coupling status using the **coupling status plus** message.

execution time: The plug-in will send the first byte of the **coupling status plus** message within xxms of sending the acknowledge transport packet for the **coupling query plus** message.

error handling: If a channel number is received as part of a **coupling query** message that a plug-in does not have, the plug-in will take unspecified action and return a **coupling status** message indicating the action it took.

also see: **coupling status plus**, **coupling set minus**, **coupling status minus**

command transactions:

coupling query plus : coupling status plus

name: **coupling set minus**

syntax: **coupling set minus** channel coupl

type: generic plug-in command

message tokens:

coupling ::= 03_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
minus	token	02_{16}
channel	us	1 - 4
coupl	token	**DC** | **OFF** | **AC** | **VC**
DC	byte	01_{16}
OFF	byte	02_{16}
AC	byte	03_{16}
VC	byte	04_{16}

usage: This message requests a new input coupling setting for the inverting input of the specified differential *channel*. This message is only valid for differential input plug-ins. The plug-in will check the *coupl* parameter for a legal coupling selection. If it is valid, the plug-in will change the input coupling to the requested type. Otherwise, no change will be made.

response: The plug-in will report the new coupling status using the **coupling status minus** message. If an error is detected, the plug-in will report the previous coupling setting. For differential plug-ins, if the offset is changed, the plug-in will send **offset status** and **diff_offset status** messages to report the new offset values.

execution time: The plug-in will send the first byte of either the **coupling status minus** message or the **error generic** message within xxms of sending the acknowledge transport packet for the **coupling set minus** message.

error handling: If the plug-in does not have a differential input, the plug-in will make no change and will report the error using the **error generic** message with the **command_error** *status* token and the *code* parameter set to 157 to indicate the problem.

If a token is received that is a valid coupling token but is a coupling which the plug-in does not support, the plug-in will take no action and return an **error generic** message with a **command_error** *status* token and the *code* parameter set to 284. If the coupling token is not a valid coupling token, the plug-in will take no action and return an **error generic** message with a **command_error** *status* token and the *code* parameter set to 157. It will report the present coupling setting using the **coupling status** message.

If a channel number is received as part of a **coupling set** command that a plug-in does not have, the plug-in will take unspecified action and return a **coupling status** message indicating the action it took.

also see: **coupling status minus, coupling set plus, coupling status plus, offset status, diff_offset status**

command transactions:

coupling set minus : [offset status :] [diff_offset status :] [error generic :] coupling status minus

name: **coupling set plus**

syntax: **coupling set plus** channel coupl

type: generic plug-in command

message tokens:

coupling ::= 03_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
plus	token	01_{16}
channel	us	1 - 4
coupl	token	**DC** ¦ **OFF** ¦ **AC** ¦ **VC**
DC	byte	01_{16}
OFF	byte	02_{16}
AC	byte	03_{16}
VC	byte	04_{16}

usage: This message requests a new input coupling setting for the specified *channel* for single ended channels and requests a new input coupling for the non-inverting input of differential channels. The plug-in will check the *coupl* parameter for a legal coupling selection. If it is valid, the plug-in will change the input coupling to the requested type. Otherwise, no change will be made.

response: The plug-in will report the new coupling status using the **coupling status plus** message. If an error is detected, the plug-in will report the previous coupling setting. For differential plug-ins, if the offset is changed, the plug-in will send **offset status** and **diff_offset status** messages to report the new offset values

execution time: The plug-in will send the first byte of either the **coupling status plus** message or the **error generic** message within xxms of sending the acknowledge transport packet for the **coupling set plus** message.

error handling: If a token is received that is a valid coupling token but is a coupling which the plug-in does not support, the plug-in will take no action and return an **error generic** message with a **command_error** *status* token and the *code* parameter set to 284. If the coupling token is not a valid coupling token, the plug-in will take no action and return an **error generic message** with a **command_error** *status* token and the *code* parameter set 157. It will report the present coupling setting using the **coupling status message**.

If a channel number is received as part of a **coupling set** message that a plug-in does not have, the plug-in will take unspecified action and return a **coupling status** message indicating the action it took.

also see: **coupling status plus, coupling set minus, coupling status minus, offset status, diff_offset status**

command transactions:

coupling set plus : [offset status :] [diff_offset status :] [error generic :] coupling status plus

name: **coupling status minus**

syntax: **coupling status minus** channel coupl

type: generic plug-in status

message tokens:

coupling ::= 03_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
minus	byte	02_{16}
channel	us	1 - 4
coupl	token	**DC** ¦ **OFF** ¦ **AC** ¦ **VC**
DC	byte	01_{16}
OFF	byte	02_{16}
AC	byte	03_{16}
VC	byte	04_{16}

usage: This message is used by a generic plug-in to report the inverted input coupling status of the specified differential channel. It is sent in response to the **coupling set minus** message or the **coupling query minus** message. It will also be sent if a change of probes requires the coupling to be changed. The *channel* parameter identifies which channel's coupling is being reported. The *coupl* parameter indicates the input coupling of the *channel*.

response: none

error handling: none

also see: **coupling set minus**, **coupling query minus**

command transactions:

coupling query minus : coupling status minus
coupling set minus : [offset status :] [diff_offset status :] [error generic :] coupling status minus

name: **coupling status plus**

syntax: **coupling status plus** channel coupl

type: generic plug-in status

message tokens:

coupling ::= 03_{16}

status ::= 03_{16}

parameters:

name	type	values
plus	token	01_{16}
channel	us	1 - 4
coupl	token	**DC** ¦ **OFF** ¦ **AC** ¦ **VC**
DC	byte	01_{16}
OFF	byte	02_{16}
AC	byte	03_{16}
VC	byte	04_{16}

usage: This message is used by a generic plug-in to report the coupling status of a singled ended channel or the non-inverting input coupling status of a differential channel. It is sent in response to the **coupling set plus** message or a **coupling query plus** message. It will also be sent if a change of probes requires the coupling to be changed. The *channel* parameter identifies which channel's coupling is being reported. The *coupl* parameter indicates the input coupling of the *channel*.

response: none

error handling: none

also see: **coupling set plus, coupling query plus**

command transactions:

coupling query plus : coupling status plus
coupling set plus : [offset status :] [diff_offset status :] [error generic :] coupling status plus

name: **cursor limits**

syntax: **cursor limits** min1 max1 min2 max2

type: mainframe command

message tokens:

cursor ::= $1D_{16}$

limits ::= 02_{16}

parameters:

name	**type**	**values**
min1	float	
max1	float	
min2	float	
max2	float	

usage: This message is sent by a smart plug-in to define the limits for cursors. Since the mainframe controls the cursors directly from the function control knobs, the plug-in may use this message as a means to place limits on the range of cursors. The *min1* and *max1* parameters specify minimum and maximum values for cursor 1. The *min2* and *max2* parameters specify the minimum and maximum values for cursor 2. The cursors may be moved to the specified *min* or *max* value but not beyond. If the present cursor position lies outside the specified minimum or maximum value, the cursor will be set to the limit value.

In **horizontal** and **vertical** cursor modes (see the **cursor set** message), the *min* and *max* parameters specify cursor limits in divisions horizontally or vertically from center screen. Resolution is at least 100 points per division. If additional resolution is specified by the plug-in, the mainframe may truncate or round the *min* or *max* limit to the nearest achievable value. If limits of greater than ±5 divisions are specified, the limits become either +5 or -5 divisions.

In the **paired** or **split** modes, the *min* and *max* values specify waveform points and must be integers greater than 0 and less than the number of points on the waveform. If a *max* value is specified greater than the number of points on a waveform, the upper cursor limit becomes the last point of the waveform. If the *min* value is specified less than 0, the lower cursor limit becomes the first point of the waveform.

Cursors are not restricted from "crossing over", that is, cursor 1 may be placed to the right of cursor 2 (in **paired** or **split** mode) as long as both cursors are within the respective *min* and *max* specifications. Cursors may also cross over in the **horizontal** and **vertical** modes.

There is no implied connection between cursor 1 and cursor 2. Each is controlled independently and does not affect the other.

response: The mainframe will respond with a **cursor status ready** message indicating the position of cursors.

error handling: none

also see: **cursor set, cursor status**

command transactions:

cursor limits : cursor status ready

name: **cursor set**

syntax: **cursor set** mode [pos1 pos2 id1 id2 scale offset resol units [wvfm_id]]

type: mainframe command

message tokens:

cursor ::= $1D_{16}$

set ::= 01_{16}

parameters:

name	**type**	**values**
mode	token	**paired** ¦ **split** ¦ **horizontal** ¦ **vertical** ¦ **off**
paired	byte	01_{16}
split	byte	03_{16}
horizontal		byte 04_{16}
vertical	byte	05_{16}
off	byte	02_{16}
pos1	float	
pos2	float	
id1	string	
id2	string	
scale	float	
offset	float	
resol	float	
units	string	
wvfm_id	us	

usage: This message is sent by a smart plug-in to specify cursor mode and position. The *mode* parameter specifies the cursor mode. The **paired** *mode* token specifies a pair of waveform type cursors that are placed on a single waveform. The cursors are placed on the selected waveform which may be specified by the plugin using the **mf_display set** message. (Waveforms may be selected using the **mf_display set** message.) The **split** *mode* token specifies cursors split between two waveforms. Cursor 1 is placed on the selected waveform and cursor 2 is placed on the waveform specified by the *wvfm_id* parameter. If *wvfm_id* specifies the selected waveform, both cursors will be placed on that waveform. The *wvfm_id* parameter is included only when the *mode* token is **split**.

For the **paired** and **split** modes, the *pos1* and *pos2* parameters specify cursor positions for cursor 1 and cursor 2 in waveform points. *pos1* and *pos2* must be integers between 0 and the maximum number of points on the selected waveform. If a value of less than zero is specified, the mainframe will place the cursor on the first point of the waveform. If a value of greater than the length of the waveform is specified, the mainframe will place the cursor on the last point of the waveform.

The **horizontal** mode token specifies a pair of cursors indicated by vertical lines useful for measuring horizontal displacement. The **vertical** mode token specifies a pair of cursors indicated by horizontal lines useful for measuring vertical displacement.

For the **horizontal** and **vertical** modes, the *pos1* and *pos2* parameters specify cursor position in divisions horizontally or vertically from center screen respectively. Resolution is at least 100 points per division. If finer resolution is specified, the mainframe will truncate or round the position value to the nearest achievable value. The *pos1* and *pos2* values are limited to q5 division. Positions that are specified outside those limits will cause the cursor to be placed at the limit.

The **off** *mode* token requests the mainframe to disable the cursor function. The *wvfm_id*, *pos1* and *pos2* parameters are not sent when *mode* is **off**.

All mainframes provide **horizontal** and **vertical** cursor modes. Only digitizing mainframes provide **paired** and **split** cursor modes.

The *id1* and *id2* parameters are strings that the plug-in uses to identify the function being measured by the cursors. These strings will be displayed just to the left of each scaled cursor value. The *id1* string will be placed with cursor 1. The *id2* string will be placed with cursor 2.

The *scale* parameter specifies a scaling factor the mainframe will use to convert divisions to the plug-in units being measured by the cursors. The mainframe will multiply the cursor position in divisions by the *scale* parameter to calculate the proper display value.

The *offset* parameter specifies a value to be added by the mainframe after scaling and before display. The displayed value will be cursor_pos * *scale* + *offset*.

The *resol* parameter specifies the resolution of the units being measured. The mainframe will use this value with the scaled cursor value to determine how may digits are meaningful for display.

The *units* parameter is a string defining the plug-in units being measured. The mainframe will append this string to the knob display.

The total knob display (including the *id* string, the number of digits for the value display and the *units*) is limited to 15 characters.

response: The mainframe will send a **cursor status ready** message in response to indicate cursor operation. If a plug-in requests **paired** or **split** modes in a non-digitizing mainframe, the mainframe will send the **cursor status na** message. It is the plug-in's responsibility to take appropriate action when **paired** or **split** modes are not available.

error handling: see response above

also see: **cursor status, cursor limits**

command transactions:

cursor set : cursor status

name: **cursor status**

syntax: **cursor status** status [pos1 pos2 [value1 value2]]

type: mainframe status

message tokens:

cursor	::= $1D_{16}$
status	::= 03_{16}

parameters:

name	**type**	**values**
status	token	**ready** | **na** | **removed**
ready	byte	01_{16}
na	byte	02_{16}
removed	byte	03_{16}
pos1	float	
pos2	float	
value1	float	
value2	float	

usage: This message is sent by the mainframe in response to a **cursor set** or **cursor limits** message from a smart plug-in. The *status* token indicates the cursor status being reported. The **ready** *status* token indicates that the mainframe has displayed the cursors and the cursors are connected to the function control knobs.

The *pos1* and *pos2* parameters indicate the positions of cursor1 and cursor 2 respectively. They are only sent with the **ready** *status* token. When **paired** or **split** cursor mode is active, the *pos1* and *pos2* parameters indicate cursor horizontal position in waveform points. The first waveform point is indicated by the value 0. When **horizontal** or **vertical** cursor mode is active, the *pos1* and *pos2* parameters specify cursor position in divisions horizontally or vertically from center screen, respectively.

The *value1* and *value2* parameters are sent when **paired** and **split** cursor *mode* is active. They indicate the waveform point vertical values at cursor 1 and cursor 2, respectively.

The **na** *mode* token is sent by non-digitizing mainframes when the **paired** or **split** modes are requested. Non-digitizing mainframes do not support **paired** or **split** modes. The plug-in is responsible to take appropriate action when this message is received.

The **removed** *mode* token is sent when the plug-in has requested the mainframe to remove the cursors.

This message is also sent with the **removed** *status* token when the cursors are no longer assigned to the plug-in.

response: When this message is sent as unsolicited status when the mainframe has reassigned the cursors, the plug-in will send the **cursor status removed** message as as response to the mainframe. Otherwise, there is no response.

error handling: none

also see: **cursor set**, **cursor limits**

command transactions:

cursor set : cursor status
cursor limits : cursor status
cursor status removed : cursor status removed

name: **diag com_test**

syntax: **diag com_test** pattern

type: plug-in command

message tokens:

diag ::= 16_{16}

com_test ::= 07_{16}

parameters:

name	**type**	**values**
pattern	special	sequence of FF, 00, 01, 02, 04, 08, 10, 20, 40, 80, 55, AA, 33, CC, DD, BB hex values.

usage: This message is sent to test the SDI interface. Normal protocol handling is observed during this test. The plug-in will accept the transport header, **diag** and **com_test** message token bytes to determine which message is being sent. It will invert each byte of the *pattern* and form a reply message of the inverted bytes.

response: The plugin will send the **diag com_test** message with all data bytes inverted.

error handling: none

also see:

command transactions:

diag com_test : diag com_test

name: diag config

syntax: diag config lmpb {**BDL** block_name {**ADL** menu_type area_name {**RDL** routine_type routine_name [**FDL** fault_id field1 field2 field3]}... }... }...

type: plug-in status

message tokens:

diag	::=	16_{16}
config	::=	08_{16}

parameters:

name	type	values
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
BDL	token	42_{16}
block_name	string	up to 10 characters
ADL	token	41_{16}
menu_type	token	**minmax_int** ¦ **minmax_long** ¦ **minmax_float** ¦ **digital_int** ¦ **digital_long** ¦ **digital_mix** ¦ **digital_short**
minmax_int	byte	01_{16}
minmax_long	byte	02_{16}
minmax_float	byte	03_{16}
digital_int	byte	04_{16}
digital_long	byte	05_{16}
digital_mix	byte	06_{16}
digital_short	byte	07_{16}
area_name	string	up to 10 characters

RDL	token	52_{16}
routine_type	token	**auto** ¦ **auto_inter** ¦ **auto_nr** ¦ **manual_inter**
auto	byte	01_{16}
auto_inter	byte	02_{16}
auto_nr	byte	03_{16}
manual_inter	byte	04_{16}
routine_name	string	up to 10 characters
FDL	byte	46_{16}
fault_id	char	'0' - '9' ¦ 'A' - 'F' ¦ '?' ¦ ' '
field1	variable, see text	
field2	variable, see text	
field3	variable, see text	

usage: This message is sent by a plug-in to report its test configuration. It is sent in response to a **diag request** or **test end** message. The order that routines are listed in this message indicates the order that they are executed during the power up sequence.

The *lmpb* parameter is the long message protocol byte. See section **6.0 Interface Protocols** for its definition and use. The **diag config** message may be split into smaller messages only just prior to one of the delimiters (**BDL**, **ADL**, **RDL** or **FDL**).

The **BDL** token identifies the beginning of a block field. It is immediately following by the *block_name* parameter which defines the name of the block. It may be up to 10 characters long. Each block field must be followed by at least one area field.

The **ADL** token identifies the beginning of an area field. There may be up to 11 area fields defined for each block. The *menu_type* parameter indicates the type of routines found in the area. This parameter specifies the meaning and type of the field parameters in the **FDL** field. The *menu_type* tokens and how they affect the **FDL** field parameters are described below. The *area_name* parameter is the name of the area and is limited to 10 characters.

The **RDL** token identifies the beginning of a routine field. The *routine_type* parameter indicates the type of routine.

The **auto**, **auto_inter** and **auto_nr** *routine_type* tokens indicate routines whose results are analyzed by the system. The **auto** token indicates a routine that may be run without any user intervention. The **auto_inter** (automatic, interactive) token indicates a routine that requires user setup but then can be run without further user intervention. The **auto_nr** token indicates a routine that is automatic but was not run during the powerup selftest.

The **manual_inter** (manual, interactive) *routine_type* token indicates a routine that requires user interpretation. No results are returned when this routine is executed. Generally, **manual_inter** routines are used for providing stimulus for probing circuits.

The *routine_name* parameter is the name of the routine. There may be up to 11 routine fields for each area field.

This configuration information implies execution order. If a routine fails during powerup tests, the mainframe may use the returned index to mark all routines except those marked **auto_nr** listed prior to the indicated routine as having passed and those that follow as not having been executed. In diagnostic mode, the user may select execution of any routine; no order is enforced.

The **FDL** token identifies the beginning of fault status information. These parameters are optional. The *fault_id* indicates the type of fault discovered: '0' indicates no fault, '1' - '9' and 'A' - 'F' indicate a fault of a specific type was discovered. These fault values may be used to guide a trouble shooter in diagnosing a problem. A *fault_id* of '?' indicates an option that is not installed. A *fault_id* of ' ' indicates a test that does not return results. In this case, the following *field* parameters will be included in the message but will have no meaning.

The *field1*, *field2* and *field3* parameters give information about a fault that was discovered. The type and meaning of these parameters are defined by the *menu_type* parameter in the **ADL** field as follows:

For minmax type routines indicated by the **minmax_int**, **minmax_long** and **minmax_float** *menu_type* tokens, the *field1* parameter indicates the minimum expected result, the *field2* parameter indicates the maximum expected result and the *field3* parameter indicates the actual measured result. For **minmax_int** routines, the *field* parameters are integers. For **minmax_long** routines, the *field* parameters are long integers. For **minmax_float** routines, the *field* parameters are single precision floating point values.

For digital type routines which are indicated by the **digital_int**, **digital_long**, **digital_mix** and **digital_short** *menu_type* tokens, the *field1* parameter indicates an address, the *field2* parameter indicates the expected result and the *field3* parameter indicates the actual result. For **digital_int** type routines, the *field* parameters are all unsigned integers. For **digital_long** routines, the *field* parameters are all unsigned long integers. For **digital_mix** routines, the address parameter (*field1*) is unsigned long and the expected and actual parameters (*field2* and *field3*) are unsigned integers. For **digital_short** routines, the address parameter (*field1*) is an unsigned integer and the expected and actual parameters (*field2* and *field3*) are unsigned short integers. Displays of all types of *field* values may be limited to 5 characters.

response: none

error handling: none

also see: **diag request**

command transactions:

diag request : diag config

name: **diag enter**

syntax: **diag enter**

type: plug-in command

message tokens:

diag ::= 16_{16}

enter ::= 01_{16}

parameters: none

usage: This message is sent by the mainframe to each plug-in to cause the plug-in to enter diagnostic mode.

response: The plug-in will send the **diag status ready** message when it has entered diagnostic mode.

error handling: none

also see: **diag status**

command transactions:

diag enter : diag status ready

name: **diag exec**

syntax: **diag exec** block_id area_id routine_id mode

type: plug-in command

message tokens:

diag	::= 16_{16}
exec	::= 05_{16}

parameters:

name	type	values
block_id	us	1 - 11
area_id	us	1 - 11
routine_id	us	1 - 11
mode	token	**single** | **cycle** | **cycle_halt**
single	byte	01_{16}
cycle	byte	02_{16}
cycle_halt	byte	03_{16}

usage: The mainframe sends this message to a plug-in to cause it to execute a specific diagnostic routine. The *block_id* parameter specifies the block from which the routine is selected. The *area_id* parameter specifies the area from which the routine is selected. The *routine_id* parameter indicates the selected routine. The *block_id*, *area_id* and *routine_id* values indicate the ordered number of the block, area or routine as listed by the plug-in in the **diag config** message.

The *mode* parameter specifies the testing mode. The **single** token requests the plug-in to execute the test once and return status using the **diag status routine** message. The **cycle_halt** token requests the plug-in to execute the test accumulating errors and incrementing the loop count until an error is found. The **cycle** token selects repetitive execution mode. The plug-in will continue to execute the test accumulating errors and incrementing the loop count until interrupted by the **diag halt** message from the mainframe.

During testing, the plug-in may request voltages and measurements using the **cal measure** and **cal cvr_set** messages.

response: The plug-in will send a **diag status routine** message as a response after it has completed the test or detected a failure. The plug-in will not return a response if in the **cycle_halt** mode no error is detected or if the mainframe specifies the **cycle** mode.

error handling: none

also see: **diag status routine, diag halt**

command transactions:

diag exec : diag status routine
diag exec ... diag halt : diag status routine

name: **diag exit**

syntax: **diag exit**

type: plug-in command

message tokens:

diag ::= 16_{16}

exit ::= 02_{16}

parameters: none

usage: This message is sent by the mainframe to request the plug-ins to leave the diagnostic mode and enter normal operational mode. The plug-in must enter operational mode and be ready to accept normal operational messages before it sends the response.

response: The plug-in will send the **diag status ready** message after it has entered normal operational mode.

error handling: none

also see: **diag enter**, **diag status**

command transactions:

diag exit : diag status ready

name: **diag halt**

syntax: **diag halt**

type: plug-in command

message tokens:

diag ::= 16_{16}

halt ::= 06_{16}

parameters: none

usage: This message is sent by the mainframe to request a plug-in to terminate a test in progress. The test may be any routine started by the **diag exec** message. The plug-in will terminate execution of the test at the end of the next loop and report its status.

response: The plug-in will send the **diag status routine** message in response indicating the status of the previously executed routine.

error handling: none

also see: **diag exec, diag status routine**

command transactions:

diag exec ... diag halt : diag status routine

name: **diag request**

syntax: **diag request**

type: plug-in command

message tokens:

diag ::= 16_{16}

request ::= 04_{16}

parameters: none

usage: The mainframe will send this message to a plug-in to request its diagnostic configuration.

response: The plug-in will send the **diag config** message in response.

error handling: none

also see: **diag config**

command transactions:

diag request : diag config

name: **diag status** 0 2 4 5

syntax: **diag status** status [loops faults fault_id field1 field2 field3]

type: plug-in status

message tokens:

diag	::= 16_{16}
status	::= 03_{16}

parameters:

name	type	values
status	token	**ready** ¦ **routine** ¦ **error**
ready	byte	01_{16}
routine	byte	02_{16}
error	byte	12_{16}
loops	ui	0 - 65535
faults	ui	0 - 65535
fault_id	char	'0' - '9' ¦ 'A' - 'F' ¦ '?' ¦ ' '
field1	variable, see text	
field2	variable, see text	
field3	variable, see text	

usage: This message is sent by a plug-in in response to a **test end**, **diag enter**, **diag exit**, **diag exec** or **diag halt** message. When sent in response to the **test end**, **diag enter** or **diag exit** messages, the status parameter will be **ready**. In this case, the rest of the parameters will not be included.

The **routine** *status* token indicates the message is reporting the results of a test execution. In this case, the rest of the parameters are included in the message.

The *loops* parameter indicates the number of test repetitions that were performed by the test. The *faults* parameter indicates how many errors were detected by the test. The *fault_id* indicates the type of fault discovered: '0' indicates no fault, '1' - '9' and 'A' - 'F' indicate a fault of a specific type was discovered. These fault values may be used to guide a trouble shooter in diagnosing a problem. A *fault_id* of '?' indicates and option that is not installed. A *fault_id* of ' ' indicates a test that does not return results. In this case, the following *field* parameters will be included in the message but will have no meaning.

The *field1*, *field2* and *field3* parameters give information about a fault that was discovered. For minmax type routines indicated by the **minmax_int**, **minmax_long** and **minmax_float** *menu_type* tokens, the *field1* parameter indicates the minimum expected result, the *field2* parameter indicates the maximum expected result and the *field3* parameter indicates the actual measured result. For **minmax_int** routines, the *field* parameters are integers. For **minmax_long** routines, the *field* parameters are long integers. For **minmax_float** routines, the *field* parameters are single precision floating point values.

For digital type routines which are indicated by the **digital_int**, **digital_long**, **digital_mix** and **digital_short** *menu_type* tokens, the *field1* parameter indicates an address, the *field2* parameter indicates the expected result and the *field3* parameter indicates the actual result. For **digital_int** type routines, the *field* parameters are all unsigned integers. For **digital_long** routines, the *field* parameters are all unsigned long integers. For **digital_mix** routines, the address parameter (*field1*) is unsigned long and the expected and actual parameters (*field2* and *field3*) are unsigned integers. For **digital_short** routines, the address parameter (*field1*) is an unsigned integer and the expected and actual parameters (*field2* and *field3*) are unsigned short integers. Displays of all types of *field* values may be limited to 5 characters.

If a plug-in detects an error in the **diag exec** message, it will send the **error** *status* token in the **diag status** message to indicate it cannot execute the requested test.

response: none

error handling: none

also see: **test end, diag enter, diag exec, diag halt**

command transactions:

test end : diag status ready
diag enter : diag status ready
diag exit : diag status ready
diag exec : diag status routine
diag exec ... diag halt : diag status routine

name: **diff_offset query**

syntax: **diff_offset query** control channel

type: generic plug-in command

message tokens:

diff_offset ::= $2A_{16}$

query ::= 02_{16}

parameters:

name	**type**	**values**
control	token	**amp** ¦ **vc** ¦ **plus** ¦ **minus**
amp	byte	04_{16}
vc	byte	03_{16}
plus	byte	01_{16}
minus	byte	02_{16}
channel	us	1 - 4

usage: This message is sent to a generic plug-in that supports the differential offset function to request the status of one of the internally maintained offset values. The *channel* parameter specifies channel for which the offset value is requested. The *control* parameter specifies which control value is requested. The **amp** *control* token requests the amplifier offset value. The **vc** *control* token requests the comparison voltage value. The **plus** *control* token requests the offset value sent to the probe connected to the plus input. The **minus** *control* token requests the offset value sent to the probe connected to the minus input.

response: The plug-in will send the **diff_offset status** message to report the requested value.

error handling: If the plug-in does not support the differential offset message, it will send an **error generic** message with a **command_error** *status* token and the *code* value set to 157.

also see: **diff_offset set, diff_offset status**

command transactions:

diff_offset query : diff_offset status

name: **diff_offset set**

syntax: **diff_offset set** control channel value

type: generic plug-in command

message tokens:

diff_offset ::= $2A_{16}$

set ::= 01_{16}

parameters:

name	**type**	**values**
control	token	**amp** | **vc** | **plus** | **minus**
amp	byte	04_{16}
vc	byte	03_{16}
plus	byte	01_{16}
minus	byte	02_{16}
channel	us	1 - 4
value	float	

usage: This message is sent to a generic plug-in that supports the differential offset function to control one of its internally maintained offset values. The *value* parameter specifies the new value of the offset function. The *channel* parameter specifies the channel for which the change is requested. The *control* parameter specifies which control value is to be changed. The **amp** *control* token specifies the amplifier offset function. The **vc** *control* token specifies the comparison voltage function. The **plus** *control* token specifies the offset sent to the probe connected to the plus input. The **minus** *control* token specifies the offset sent to the probe connected to the minus input.

response: none

error handling: If a plug-in does not support the differential offset function, it will send an **error generic** message with a **command_error** *status* token and the *code* value set to 157.

If the *value* parameter is outside the achievable limits of the plug-in at the present gain setting, the plug-in will report an error using the **error generic** message with the **exec_warning** *status* token, the *code* parameter set to 550 and the *index* set to 2 to indicate the problem.

also see: **diff_offset query, diff_offset status**

command transactions:

diff_offset set : diff_offset status

name: **diff_offset status**

syntax: **diff_offset status** control channel value

type: generic plug-in status

message tokens:

diff_offset ::= $2A_{16}$

status ::= 03_{16}

parameters:

name	**type**	**values**
control	token	**amp** ¦ **vc** ¦ **plus** ¦ **minus**
amp	byte	04_{16}
vc	byte	03_{16}
plus	byte	01_{16}
minus	byte	02_{16}
channel	us	1 – 4
value	float	

usage: This message is sent in response to either the **diff_offset query** message or the **diff_offset status** message. The *value* parameter indicates the present setting of the specified control. The *channel* parameter specifies the channel for which the offset value is being reported. The *control* parameter specifies which control value is being reported. The **amp** *control* token specifies the amplifier offset value. The **vc** *control* token specifies the comparison voltage value. The **plus** *control* token specifies the offset value of the probe connected to the plus input. The **minus** *control* token specifies the offset value of the probe connected to the minus input.

response: none

error handling: none

also see: **diff_offset query**, **diff_offset set**

command transactions:

diff_offset query : diff_offset status
diff_offset set : diff_offset status

message **name: display query**

syntax: display query

type: generic plug-in command

message tokens:

display ::= 06_{16}

query ::= 02_{16}

parameters: none

usage: This message requests the plug-in to report the display output selection.

response: The plug-in will report the combination list using the **display status** message.

execution time: The plug-in will send the first byte of the **display status** message within xxms of sending the acknowledge transport packet for the **display query** message.

error handling: none

also see: **display status**

command transactions:

display query : display status

name: **display set**

syntax: **display set** num comb...

type: generic plug-in command

message tokens:

display ::= 06_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
num	us	1 - 12
comb	special	bits 7 6 5 4 3 2 1 0 meaning P4 E4 P3 E3 P2 E2 P1 E1 En = 0 => channel n is off En = 1 => channel n is on Pn = 0 => channel n is +up Pn = 1 => channel n is inverted

usage: This message requests a new display output selection. The plug-in will set its display sequencer to the specified *comb*inations. The *num* parameter indicates the number of combinations being specified. There may be up to 12 *comb*inations. A complete list of combinations must be sent each time a change is made to any combination. If there are fewer than 12 *comb*inations specified, the plug-in will set the remaining combinations to all 0's. The number of combinations in the list specifies the length of the channel switching sequence. The mainframe must append null combinations (all 0's) as necessary to match the number of combinations to the length of the sequence.

response: The plug-in will report the new combination list using the **display status** message.

execution time: The plug-in will send the first byte of either the **display status** message within xxms of sending the acknowledge transport packet for the **display set** message.

error handling: If there are no combinations, the plug-in will make no change to any combination. If there are more than 12 combinations, the additional combinations will be ignored.

also see: **display status**

command transactions:

display set : display status

name: **display status**

syntax: **display status** num comb...

type: generic plug-in status

message tokens:

display ::= 06_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
num	us	1 - 12
comb	special	bits 7 6 5 4 3 2 1 0 meaning P4 E4 P3 E3 P2 E2 P1 E1 En = 0 => channel n is off En = 1 => channel n is on Pn = 0 => channel n is +up Pn = 1 => channel n is inverted

usage: A generic plug-in uses this message to report the present status of its display outputs. It is sent in response to a **display set** or **display query** message. The plug-in will report the status of all defined *comb*inations. The *num* parameter specifies the number of combinations being reported. The number of combinations reported will match the number specified by the mainframe. There may be up to 12 *comb*inations.

response: none

error handling: none

also see: **display set**

command transactions:

display query : display status
display set : display status

name: **disp_attr query**

syntax: **disp_attr query**

type: generic plug-in command

message tokens:

disp_attr ::= 18_{16}

query ::= 02_{16}

parameters: none

usage: This message is sent to generic plug-ins to request the present status of the display parameters for the gain, offset, bandwidth and input impedance functions and a delay parameter.

response: The plug-in will send the **disp_attr status** message with all non-default parameters in response.

error handling: none

also see: **disp_attr status**

command transactions:

disp_attr query : disp_attr status

name: **disp_attr status**

syntax: **disp_attr status** lmpb {channel [**gain** gain_min gain_max gain_res] [**offset** offset_min offset_max offset_res] [**diff_offset** amp_res vc_res plus_res minus_res] [**bandwidth** bwl_res] [**impedance** imp_res] [**delay** rel_delay] **EOD**}...

type: generic plug-in status

message tokens:

disp_attr ::= 18_{16}

status ::= 03_{16}

parameters:

name	type	values
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	03_{16}
channel	us	1 - 4
gain	token	01_{16}
gain_min	float	
gain_max	float	
gain_res	us	
offset	token	02_{16}
offset_min	float	
offset_max	float	
offset_res	float	
diff_offset	token	$2A_{16}$
amp_res	float	
vc_res	float	
plus_res	float	

minus_res float

bandwidth token 05_{16}

bwl_res us

impedance token 04_{16}

imp_res us

delay token 03_{16}

rel_delay float

EOD token 06_{16}

usage: This message is sent by a generic plug-in to report the display status of its gain, offset, bandwidth and impedance functions. It also reports a plug-in delay value. When sent in response to the **disp_attr query** message, all non-default values will be reported. When sent in response to an **SRQ query** message, only changed values will be reported.

The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter.

The *channel* parameter specifies the channel for which the following list of parameters applies.

The **gain** token defines the beginning of a gain parameter list. The *gain_min* parameter specifies the minimum available gain setting. The *gain_max* parameter specifies the maximum available gain setting. The *gain_res* parameter specifies the number of display digits of resolution that are meaningful for the gain function.

The **offset** token defines the beginning of an offset parameter list. The *offset_min* parameter specifies the minimum available offset setting. The *offset_max* parameter specifies the maximum available offset setting. The *offset_res* parameter specifies the resolution of the offset control in volts. The number of digits to be displayed is a function of the present offset setting and the offset resolution according to the following equation:

$$\text{num\_digits} = \text{int}(\log_{10}(\frac{\text{present value}}{\text{resolution}})) + 1$$

The int function truncates the value toward 0. This value represents the step size of the fine offset control.

The **diff_offset** token defines the beginning of a differential offset parameter list. The *amp_res* parameter specifies the resolution of the amplifier offset function. The *vc_res* parameter specifies the resolution of the vc function. The *plus_res* parameters specifies the resolution of the offset control to the probe connected to the plus input. The *minus_res* parameter specifies the resolution of the offset control to the probe connected to the minus input.

The **bandwidth** token defines the following value to apply to the bandwidth limit function. The *bwl_res* parameter specifies the number of display digits of resolution that are meaningful for the bandwidth limit function.

The **impedance** token defines the following value to apply to the input impedance function. The *imp_res* parameter specifies the number of display digits of resolution that are meaningful for the input impedance function.

The **delay** token defines the following value to be the relative delay through the plug-in. The *rel_day* is the delay difference from a the reference delay setting to the present setting.

The **EOD** token defines the end of a channel display parameter definition. Following the **EOD** token is either another channel definition or the end of the message.

response: none

error handling: none

also see: **disp_attr query**

command transactions:

disp_attr query : disp_attr status

name: **error generic**

syntax: **error generic** status code index channel

type: plug-in status

message tokens:

error ::= 12_{16}

generic ::= 20_{16}

parameters:

name	type	values
status	token	**command_error** ¦ **exec_error** ¦ **internal_error** ¦ **exec_warning** ¦ **internal_warning**
command_error	byte	21_{16}
exec_error	byte	22_{16}
internal_error	byte	23_{16}
exec_warning	byte	25_{16}
internal_warning	byte	26_{16}
code	ui	
index	int	
channel	us	

usage: This message is sent by a **generic** or a **both** plug-in when it has detected a generic error. The *status* parameter indicates the category of error that was detected. There are five categories:

The **command_error** token identifies a command error. This type of error is generated when the plug-in receives an incorrect command.

The **exec_error** token identifies an execution error. This type of error is generated when the plug-in receives a command it cannot execute.

The **internal_error** token indicates an internal error. This type of error is generated when a plug-in has detected a serious problem with its operation not as a result of a command.

The **exec_warning** token indicates an execution warning. This type of error is generated when the plug-in can execute a command but the results of the execution may be unexpected and the user needs to be notified.

The **internal_warning** token indicates an internal warning. This type of error is generated when a plug-in detects a problem with its operation that is not serious but requires user notification.

The *code* parameter specifies the message code for the category. These codes are defined in the *Command Reference Specifications* document.

The *index* parameter specifies the type of function for which an out-of-range condition was detected. When the *status* byte is **exec_warning** and the *code* value is 550 or if the *status* byte is **exec_error** and the *code* value is 205, the *index* parameter indicates which function was out-of-range according to the following table:

index	**function**
1	vertical size
2	vertical position
3	bandwidth limit
4	input impedance

The mainframe will substitute the name of the indicted function in the text of the error message when it is displayed or sent over the external bus. The *index* parameter has no meaning for generic plugins when reporting errors other than the out-of-range condition. A zero value will be used for the *index* parameter in these cases.

The *channel* parameter indicates the channel associated with the error. If the error is not associated with a channel, the *channel* parameter will be set to 0.

When this error message is generated in response to an asynchronous event (ie. not a command from the mainframe), the plug-in will send an SRQ to the mainframe. The **error generic** message will be sent as the response to the subsequent **SRQ query** message.

response: none

error handling: this is it

also see: **error smart**

command transactions:

SRQ query : error generic

name: **error req_text**

syntax: **error req_text** code index

type: smart plug-in command

message tokens:

error ::= 12_{16}

req_text ::= 01_{16}

parameters:

name	**type**	**values**
code	ui	
index	int	

usage: The mainframe will send this message to a smart plug-in to request error text for a specific error. The mainframe will use the *code* and *index* parameters in this message are those reported by the smart plug-in in a previous **error smart** message.

response: The plug-in will respond with the **error text** message to supply the text to the mainframe.

error handling: This is it

also see: **error smart**, **error text**, **external oper_compl**

command transactions:

error req_text : error text

name: **error smart**

syntax: **error smart status** code index channel

type: plug-in status

message tokens:

error	::= 12_{16}
smart	::= 21_{16}

parameters:

name	**type**	**values**
status	token	**command_error** ¦ **exec_error** ¦ **internal_error** ¦ **exec_warning** ¦ **internal_warning**
command_error	byte	21_{16}
exec_error	byte	22_{16}
internal_error	byte	23_{16}
exec_warning	byte	25_{16}
internal_warning	byte	26_{16}
code	ui	
index	int	bit 15 must be set
channel	us	

usage: This message is sent by a **smart** or a **both** plug-in when it has detected an error. The *status* parameter indicates the category of error that was detected. There are five categories:

The **command_error** token identifies a command error. This type of error is generated when the plug-in receives an incorrect command.

The **exec_error** token identifies an execution error. This type of error is generated when the plug-in receives a command it cannot execute.

The **internal_error** token indicates an internal error. This type of error is generated when a plug-in has detected a serious problem with its operation not **as a** result of a command.

The **exec_warning** token indicates an execution warning. This type of error is generated when the plug-in can execute a command but the results of the execution may be unexpected and the user needs to be notified.

The **internal_warning** token indicates an internal warning. This type of error is generated when a plug-in detects a problem with its operation that is not serious but requires user notification.

The *code* parameter specifies the message code for the category. These codes are defined in the *Command Reference Specifications* document.

The *channel* parameter indicates the channel associated with the error. If the error is not associated with a channel, the *channel* parameter will be set to 0.

The *index* parameter is used to indicate a specific message associated with the error. The mainframe will use this parameter in the **error req_text** message to select the proper text for the error. Bit 15 of the *index* parameter must be set (negative integer) by the smart plug-in when it sends the **error smart** message. The mainframe uses this bit to separate smart plug-in index values from mainframe index values.

When this error message is generated in response to an asynchronous event (ie. not a command from the mainframe), the plug-in will send an SRQ to the mainframe. The **error smart** message will be sent as the response to the subsequent **SRQ query** message.

response: none

error handling: This is it

also see: **error generic, error req_text, error text**

command transactions:

SRQ query : error smart

name: **error text**

syntax: **error text** text

type: smart plug-in status

message tokens:

error ::= 12_{16}

text ::= 03_{16}

parameters:

name	type	values
text	string	includes %a, %b and %B constructs, excludes %A construct

usage: This message is sent by a smart plug-in in response to an **error req_text** message from the mainframe. This message is used to supply error message text for display or transmission over the external bus by the mainframe.

The *text* parameter is the actual text to be sent or displayed and is limited to 40 displayed characters. This text may contain escape sequences in addition to those normally defined for strings. These are the %a, %b and %B sequences (the %A sequence is not included for smart plug-ins). They have the following meaning:

%a - specifies the channel number. When the mainframe is formatting either generic or smart plug-in error messages, it will replace this sequence of characters with the numeric value of the *channel* parameter of the **error smart** message.

%A - is reserved for generic function types for parameter out-of-range conditions. This will be replaced by the mainframe with the function name indicated by the *index* parameter of the **error generic** message. Smart plugins must not use the %A construct in error texts.

%b - specifies the compartment identifier. When the mainframe is formatting either generic or smart plug-in error messages, it will replace this sequence of characters with the compartment identifier of the plug-in that sent the **error smart** message. This identifier is a single character (eg. L, C, R).

%B - specifies the compartment name. When the mainframe is formatting either generic or plug-in error messages it will replace this sequence of characters with the compartment name of the plug-in that sent the **error smart** message. The compartment name is a word describing the compartment (eg. left, center, right).

When the percent sign is followed by any other character, that character only will be displayed. Thus %% causes a single % to be displayed.

response: none

error handling: This is it

also see: **error req_text**, **error smart**, **external oper_compl**

command transactions:

error req_text : error text

name: **external fp_button**

syntax: **external fp_button** button

type: smart plug-in command

message tokens:

external ::= $1B_{16}$

fp_button ::= $0E_{16}$

parameters:

name	**type**	**values**
button	us	

usage: This message is sent by a smart plug-in when one of its front panel buttons is pressed and the front panel mode is not **locked**. (See **external fp_mode.**) The *button* parameter is a code number that indicates which front panel button was pressed. This value only has meaning to the smart plug-in that issues this message. The mainframe may use this message to trigger the sending of a service request over the external bus indicating a button was pressed. The definition of which buttons are indicated by which numbers is specified in the smart plug-in's EIS.

The plug-in must send an SRQ when the button is pressed then send this message in response to the subsequent **SRQ query** message.

response: none

error handling: none

also see: **external fp_mode**

command transactions:

SRQ query : external fp_button

name: **external fp_mode**

syntax: **external fp_mode** mode

type: smart plug-in command

message tokens:

external ::= $1B_{16}$

fp_mode ::= 10_{16}

parameters:

name	type	values
mode	token	**locked** | **unlocked**
locked	byte	01_{16}
unlocked	byte	02_{16}

usage: This message is sent to smart plug-ins to control front panel modes. When the **locked** *status* token is sent, the plug-ins will not report front panel button pushes. They will also not take any action when a front panel button is pressed. When the **unlocked** *status* token is sent, the plug-in will resume normal operation in response to front panel button presses.

response: The plug-in will send the **external status** message with the **ready** *status* token indicating it has entered the requested mode.

error handling: none

also see: **external fp_button, external status**

command transactions:

external fp_mode : external status ready

name: **external get**

syntax: **external get**

type: smart plug-in command/status

message tokens:

external ::= $1B_{16}$

get ::= $0F_{16}$

parameters: none

usage: This message is sent by the mainframe to smart plug-ins to cause them to execute the group execute trigger function defined for the external interface. It is only sent to smart plug-ins that have specified in the **plugin_config status** message that they support the group execute trigger function using the **get** token.

response: The plug-in will perform the specified action and return the **external get** message.

error handling: none

also see: **config_plugin status**

command transactions:

external get : external get

name: **external help_query**

syntax: **external help_query**

type: smart plug-in command

message tokens:

external ::= $1B_{16}$

help_query ::= 09_{16}

parameters:

usage: This message is sent to each smart plug-in when the mainframe receives the help query command from the external interface.

response: The plug-in will send the **external help_status** message in response.

error handling: none

also see: **external help_status**

command transactions:

external help_query : external help_status

name: **external help_status**

syntax: **external help_status** lmpb [length status]

type: smart plug-in status

message tokens:

external ::= $1B_{16}$

help_status ::= $0A_{16}$

parameters:

name	**type**	**values**
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
length	us	
status	string	

usage: This message is sent by smart plug-ins when they have received the **external help_query** message. Smart plug-ins use this message to report information about commands supported by that plug-in over the external bus. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter. The *length* parameter specifies the number of bytes in the message following the *length* parameter. The *status* parameter is a character string of ASCII text in bus compatible format indicating the commands supported by the plug-in. The plug-in is responsible to prepend compartment identifiers to plug-in command names.

response: none

error handling: none

also see: external help_query

command transactions:

external help_query : external help_status

name: **external interf_data**

syntax: **external interf_data** lmpb [length data]

type: mainframe/smart plug-in command

message tokens:

external ::= $1B_{16}$

interf_data ::= $0B_{16}$

parameters:

name	**type**	**values**
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
length	us	
data	bytes	

usage: This message is sent by a smart plug-in when it has a requirement to send information out the external interface (usually in response to a command from the interface). The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter. The *length* parameter specifies the number of bytes in the message following the *length* parameter. The *data* is interface ready information to be sent by the mainframe over the interface.

response: The mainframe will send the **external status ready** message when it has completed the operation.

error handling: none

also see: **external status**

command transactions:

external interf_data : external status ready

name: **external long_form**

syntax: **external long_form** mode

type: smart plug-in command

message tokens:

external ::= $1B_{16}$

long_form ::= 05_{16}

parameters:

name	**type**	**values**
mode	token	**on** ¦ **off**
on	byte	01_{16}
off	byte	02_{16}

usage: This message is sent by the mainframe to smart plug-ins when it has received a command from the external bus to change the state of reporting mode. The **on** *mode* token enables the long form of responses to **external set_query** and **external help_query** commands. The **off** *mode* token enables the short form of reporting.

response: The plug-in will send the **external status** message with the **ready** *mode* token when it has entered the specified mode.

error handling: none

also see: **external help_query**, **external help_status**, **external set_query**, **external set_status**, **external status**

command transactions:

external long_form : external status ready

name: **external message**

syntax: **external message** lmpb [length message]

type: smart plug-in command

message tokens:

external ::= $1B_{16}$

message ::= 01_{16}

parameters:

name	**type**	**values**
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
length	us	
message	bytes	

usage: This message is sent to smart plug-ins when the mainframe has received an external interface message for that plug-in. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter. The *length* parameter specifies the number of bytes in the message following the *length* parameter. The *message* parameter is a list of bytes produced by the mainframe's external interface parser based on the table uploaded by the plug-in.

response: The plug-in will send the **external status ready** message when it has completed the requested action.

error handling: The plug-in will report errors using the standard error handling messages **error smart**, **error req_text** and **error text**.

also see: **external status ready**

command transactions:

external message : external status ready

name: **external oper_compl**

syntax: **external oper_compl** code index channel

type: smart plug-in command

message tokens:

external ::= $1B_{16}$

oper_compl ::= $0C_{16}$

parameters:

name	**type**	**values**
code	ui	
index	int	
channel	us	

usage: This message is sent by a plug-in when it has completed an action from a specific list defined for this message. The *code* parameter specifies the type of operation that was completed. The *index* parameter indicates a specific message associated with the completed operation. The text of the message may be requested by the mainframe using the **error req_text** message and will be reported by the plug-in using the **error text** message. The *channel* parameter indicates the channel related to the operation. If the operation is not channel related, the channel parameter will be set to 0.

response: The mainframe will send the **external status ready** message when it has completed handling the operation complete function.

error handling: none

also see: **error req_text**, **error text**, **external status**

command transactions:

external oper_compl : external status ready

name: **external req_table**

syntax: **external req_table** ttype ftype

type: smart plug-in command

message tokens:

external ::= $1B_{16}$

req_table ::= 02_{16}

parameters:

name	type	values
ttype	token	**E** ¦ **R**
E	byte	45_{16}
R	byte	52_{16}
ftype	token	**C_F**
C_F	byte	01_{16}

usage: This message is sent by the mainframe during the power up sequence to smart plug-ins to request their parse tables for the external interface. The *ttype* parameter specifies the type of table requested. The two types defined are type **E** and type **R**. The *ftype* parameter specifies the bus format being requested. Presently, the only supported format is **C_F**.

response: The plug-in will send the **external table_data** message supplying the requested data.

error handling: none

also see: **external table_data**

command transactions:

external req_table : external table_data

name: **external set_data**

syntax: **external set_data** lmpb [length data]

type: smart plug-in command

message tokens:

external ::= $1B_{16}$

set_data ::= $0B_{16}$

parameters:

name	**type**	**values**
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
length	us	
data	bytes	

usage: This message is sent to smart plug-ins when the mainframe has received a binary setting from the external bus. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter. The *length* parameter specifies the number of bytes in the message following the *length* parameter. The *data* bytes are binary values that the plug-in reported in **a** previous **external set_data** message.

response: The plug-in will send the **external status ready** message when it has set itself according to the requirements of the *data* in the **external set_data** message.

error handling: none

also see: **external set_status**, **external status**

command transactions:

external set_data : external status

name: **external set_query**

syntax: **external set_query**

type: smart plug-in command

message tokens:

external ::= $1B_{16}$

set_query ::= 06_{16}

parameters: none

usage: This message is sent to smart plug-ins to request setting data for the external bus.

response: The plug-in will send the **external set_data** message to report the requested data.

error handling: none

also see: **external set_data**

command transactions:

external **set_query** : **external set_status** name: **external set_status**

syntax: **external set_status** lmpb [length status]

type: plug-in/mainframe status

message tokens:

external ::= $1B_{16}$

set_status ::= 07_{16}

parameters:

name	**type**	**values**
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
length	us	
status	bytes	

usage: This message is sent by smart plug-ins in response to an **external set_query** message. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter. The *length* parameter specifies the number of bytes in the message following the *length* parameter. The *data* being returned is of two formats. If the **long_form** *mode* is **on**, the data will be in ASCII, bus ready format. If the **long_form** *mode* is **off**, the data will be in binary format. This *data* may be returned to the plug-in at a future time to restore plug-in operation to the present settings.

response: none

error handling: none

also see: **external set_query**

command transactions:

external set_query : **external set_data**

name: **external status**

syntax: **external status** status

type: mainframe/smart plug-in status

message tokens:

external ::= $1B_{16}$

status ::= 03_{16}

parameters:

name	**type**	**values**
status	token	**ready**
ready	byte	01_{16}

usage: The mainframe or plug-in will send this message in response to the **external fp_mode**, **external long_form**, **external set_data**, **external oper_compl**, **external interf_data** and **external message** messages defined for this interface. This message indicates that the plug-in or mainframe has performed the requested task and is ready for the next operation.

response: none

error handling: none

also see: **external fp_mode**, **external long_form**, **external set_data**, **external message**, **external oper_compl**, **external interf_data**

command transactions:

external fp_mode : external status ready
external long_form : external status ready
external set_data : external status ready
external message : external status ready
external oper_compl : external status ready
external interf_data : external status ready

name: **external table_data**

syntax: **external table_data** lmpb [ttype ftype data]

type: smart plug-in status

message tokens:

external ::= $1B_{16}$

table_data ::= 04_{16}

parameters:

name	type	values
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
ttype	token	**E** ¦ **R**
E	byte	45_{16}
R	byte	52_{16}
ftype	token	**C_F**
C_F	byte	01_{16}
data	bytes	

usage: This message is sent by smart plug-ins in response to an **external req_table** message. The *lmpb* parameter is the long message protocol byte. See section **6.1.4 Long Messages** for the meaning and use of this parameter. The *ttype* parameter specifies the type of table requested. The two types defined are type **E** and type **R**. The *ftype* parameter specifies the bus format being requested. Presently, the only supported format is **C_F**.

response: none

error handling: none

also see: **external req_table**

command transactions:

external req_table : external table_data

name: **function select**

syntax: **function select** mode status seq_num data

type: smart plug-in command

message tokens:

function ::= $1C_{16}$

select ::= 01_{16}

parameters:

name	**type**	**values**
mode	token	**display** ¦ **trigger**
display	byte	06_{16}
trigger	byte	07_{16}
status	token	**enable** ¦ **disable**
enable	byte	01_{16}
disable	byte	02_{16}
seq_num	us	
data	bytes	

usage: This message is sent by the mainframe when a smart plug-in function is selected. Smart plug-in functions are specified in the data uploaded in the **external table_data** message.

The *mode* parameter specifies the output mode for the function. The **display** token requests the function output to be enabled or disabled on the display path. The **trigger** token requests the function output to be enabled or disabled on the trigger path.

The *status* parameter enables or disables the output. The **enable** token requests the function output to be enabled to the specified path. The **disable** token requests the function output to be disabled from the specified path.

The *seq_num* parameter specifies the combination slot in the display sequencer for which the function's output is to be enabled or disabled. Function outputs may be individually enabled or disabled for each sequence combination in either the display or trigger paths.

The *data* parameter specifies which function is selected and how the plug-in is to combine inputs to that function. This information is a list of bytes that is interpreted by the plug-in.

response: The plug-in will send the **function status** message when it has completed the requested action.

error handling: See the **function status** message.

also see: **function status**

command transactions:

function select : function status

name: **function status**

syntax: **function status** status

type: smart plug-in status

message tokens:

function ::= $1C_{16}$

status ::= 03_{16}

parameters:

name	**type**	**values**
status	token	**ready** ¦ **unable**
ready	byte	01_{16}
unable	byte	02_{16}

usage: This message is sent by a smart plug-in in response to a **function select** message. The *status* parameter indicates the plug-in status. The **ready** *status* token indicates the plug-in has changed its operation to provide the specified function. The **unable** *status* token indicates the plug-in is unable to perform the requested function.

response: none

error handling: See text

also see: **function select**

command transactions:

function select : function status

name: **gain query**

syntax: **gain query** channel

type: generic plug-in command

message tokens:

gain	::= 01_{16}
query	::= 02_{16}

parameters:

name	**type**	**values**
channel	us	**1 - 4**

usage: This message requests the plug-in to report the gain setting for the specified *channel*.

response: The plug-in will report the new gain setting using the **gain status** message.

execution time: The plug-in will send the first byte of the **gain status** message within xxms of sending the acknowledge transport packet for the **gain query** message.

error handling: If a channel number is received as part of a **gain query** message that a plug-in does not have, the plug-in will return a **gain status** message for an unspecified channel.

also see: **gain set coarse, gain set fine, gain status, error generic, offset set abs, offset status**

command transactions:

gain query : gain status

name: **gain set abs**

syntax: **gain set abs** channel value

type: generic plug-in command

message tokens:

gain	::= 01_{16}
set	::= 01_{16}

parameters:

name	**type**	**values**
abs	token	01_{16}
channel	us	1 - 4
value	float	

usage: This message requests a new gain setting of *value* units/div for the specified *channel*. It is an absolute setting command. The plug-in will check that the gain *value* lies within the achievable limits of the plug-in and the presently connected probe. If it does, the plug-in will change the gain of the specified *channel* to the requested *value*. If *value* specifies more resolution than the plug-in can achieve, the plug-in will round the requested *value* to the nearest achievable setting and set the gain to that value. A gain setting that does not correspond to a coarse gain setting is achieved by selecting the next lower coarse gain setting and adjusting the fine setting.

The plug-in will calculate a new offset *actual value* based on the new gain setting and the present offset *requested value*. If the new *actual value* is different from the old *actual value*, the plug-in will change the offset to the new *actual value*. See the **offset set abs** message. This will cause a differential offset value to be changed in a differential plug-in. See the **diff_offset set** message.

response: The plug-in will report the new gain setting using the **gain status** message. The value of gain reported will be the value to which the plug-in gain is actually set. This may be different (in the case of rounding) from the requested value. If the offset has changed, the plug-in will report the new offset value first using the **offset status** message. In differential plug-ins, a new differential offset will also be reported using the **diff_offset status** message. If the limits of the gain or offset functions are changed, the plug-in will also send the **disp_attr status** message.

name: **gain set abs**

syntax: **gain set abs** channel value

type: generic plug-in command

message tokens:

gain ::= 01_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
abs	token	01_{16}
channel	us	1 - 4
value	float	

usage: This message requests a new gain setting of *value* units/div for the specified *channel*. It is an absolute setting command. The plug-in will check that the gain *value* lies within the achievable limits of the plug-in and the presently connected probe. If it does, the plug-in will change the gain of the specified *channel* to the requested *value*. If *value* specifies more resolution than the plug-in can achieve, the plug-in will round the requested *value* to the nearest achievable setting and set the gain to that value. A gain setting that does not correspond to a coarse gain setting is achieved by selecting the next lower coarse gain setting and adjusting the fine setting.

The plug-in will calculate a new offset *actual value* based on the new gain setting and the present offset *requested value*. If the new *actual value* is different from the old *actual value*, the plug-in will change the offset to the new *actual value*. See the **offset set abs** message. This will cause a differential offset value to be changed in a differential plug-in. See the **diff_offset set** message.

response: The plug-in will report the new gain setting using the **gain status** message. The value of gain reported will be the value to which the plug-in gain is actually set. This may be different (in the case of rounding) from the requested value. If the offset has changed, the plug-in will report the new offset value first using the **offset status** message. In differential plug-ins, a new differential offset will also be reported using the **diff_offset status** message. If the limits of the gain or offset functions are changed, the plug-in will also send the **disp_attr status** message.

execution time: The plug-in will send the first byte of either the **gain status** message, the **offset status** message or the **error generic** message within xxms of sending the acknowledge transport packet for the **gain set abs** message.

error handling: If a channel number is received as part of a **gain set** message that a plug-in does not have, the plug-in will take unspecified action and return a **gain status** message indicating the action it took.

If the *value* specified is out of range, the plug-in will set to the maximum or minimum value and return an **error generic** message with an **exec_warning** *status* token and the *code* parameter set to 550 to indicate the problem. It will report the new gain setting using the **gain status** message.

also see: **gain set coarse**, **gain set fine**, **gain status**, **error generic**, **offset set abs**, **offset status**, **diff_offset status**, **disp_attr status**

command transactions:

gain set abs : [error generic :] [offset status :] [diff_offset status :] [disp_attr status :] gain status

execution time: The plug-in will send the first byte of either the **gain status** message, the **offset status** message or the **error generic** message within xxms of sending the acknowledge transport packet for the **gain set abs** message.

error handling: If a channel number is received as part of a **gain set** message that a plug-in does not have, the plug-in will take unspecified action and return a **gain status** message indicating the action it took.

If the *value* specified is out of range, the plug-in will not change its gain and return an **error generic** message with an **exec_error** *status* token and the *code* parameter set to 205 to indicate the problem. It will report the gain setting using the **gain status** message.

also see: **gain set coarse, gain set fine, gain status, error generic, offset set abs, offset status, diff_offset status, disp_attr status**

command transactions:

gain set abs : [**error generic** :] [**offset status** :] [**diff_offset status** :] [**disp_attr status** :] **gain status**

name: **gain set coarse**

syntax: **gain set coarse** channel value

type: generic plug-in command

message tokens:

gain	::= 01_{16}
set	::= 01_{16}

parameters:

name	type	values
coarse	token	02_{16}
channel	us	1 - 4
value	short	

usage: This message requests a new gain setting for the specified *channel*. The plug-in will calculate the new setting by adding *value* coarse steps to the present coarse gain setting if *value* is > 0 and subtract *value* coarse steps if *value* is < 0. If *value* is 1 or -1, the gain will be set to the next higher or present coarse setting, respectively. If *value* is 0, the gain is not changed. When the coarse gain is changed, the fine gain will be set to 0. If the calculated setting is outside the achievable limits of the plug-in and the presently connected probe, the plug-in will set the gain to the maximum or minimum coarse gain setting depending on the sign of *value*.

Coarse gain settings are defined as the sequence of 1-2-5 steps between the minimum and maximum coarse volts per division values for a plug-in.

The plug-in will calculate a new offset *actual value* based on the new gain setting and the present offset *requested value*. If the new *actual value* is different from the old *actual value*, the plug-in will change the offset to the new *actual value*. See the **offset set abs** message. This will cause a differential offset value to be changed in a differential plug-in. See the **diff_offset set** message.

response: The plug-in will report the new gain setting using the **gain status** message. The value of gain reported will be the actual gain achieved not the number of coarse steps changed. If the offset has changed, the plug-in will report the new offset value using the **offset status** message. In differential plug-ins, a new differential offset will also be reported using the **diff_offset status** message.

execution time: The plug-in will send the first byte of either the **gain status** message or the **offset status** message within xxms after it sends the acknowledge transport packet for the **gain set coarse** message.

error handling: If a channel number is received as part of a **gain set coarse** message that a plug-in does not have, the plug-in will take unspecified action and return a **gain status** message indicating the action it took.

also see: **gain set abs**, **gain set fine**, **gain status**, **offset set abs**, **offset status**, **diff_offset status**, **disp_attr status**

command transactions:

gain set coarse : [**offset status** :] [**diff_offset status** :][**disp_attr status** :] **gain status**

name: **gain set fine**

syntax: **gain set fine** channel value

type: generic plug-in command

message tokens:

gain ::= 01_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
fine	token	03_{16}
channel	us	1 – 4
value	short	

usage: This message requests a new gain setting for the specified *channel*. The plug-in will calculate the new gain setting by adding *value* fine steps to the present gain setting. Fine steps are defined as a specified percentage of the next more sensitive coarse gain setting. If the calculated gain setting is outside the achievable limits of the plug-in, the plug-in will set the gain to the maximum or minimum achievable gain setting. Otherwise, the plug-in will set the gain to the calculated setting. The plug-in will change the attenuator, the step gain and the variable control as necessary to achieve the requested gain setting.

The plug-in will calculate a new offset *actual value* based on the new gain setting and the present offset *requested value*. If the new *actual value* is different from the old *actual value*, the plug-in will change the offset to the new *actual value*. See the **offset set abs** message. This will cause a differential offset value to be changed in a differential plug-in. See the **diff_offset set** message.

response: The plug-in will report the new gain setting using the **gain status** message. The value of gain reported will be the actual gain achieved not the number of fine steps changed. If the offset has changed, the plug-in will report the new offset value using the **offset status** message. In differential plug-ins, a new differential offset will also be reported using the **diff_offset status** message.

execution time: The plug-in will send the first byte of either the **gain status** message or the **offset status** message within xxms of sending the acknowledge transport packet for the **gain set fine** message.

error handling: If a channel number is received as part of a **gain set fine** message that a plug-in does not have, the plug-in will take unspecified action and return a **gain status** message indicating the action it took.

also see: **gain set abs, gain set coarse, gain status, offset set abs**

command transactions:

gain set fine : [offset status :] [diff_offset status :] [disp_attr status :] gain status

name: **gain status**

syntax: **gain status** channel value

type: generic plug-in status

message tokens:

gain ::= 01_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
channel	us	1 – 4
value	float	

usage: A generic plug-in uses this message to report the status of the gain function. It is sent in response to any of the gain messages: **gain set abs**, **gain set coarse** and **gain set fine** or in response to the **gain query** message. It will also be sent if the probe is changed and the previous gain cannot be achieved with the new probe. The *channel* parameter identifies which channel's gain is being reported. The *value* parameter specifies the present setting of the gain function in units per division. The units are identified by the **units status** message. Minimum and maximum values for the gain control are specified with the **disp_attr** message.

response: none

error handling: none

also see: **gain set abs**, **gain set coarse**, **gain set fine**, **gain query**, **units status**, **disp_attr status**

command transactions:

gain query : gain status
gain set abs : [error generic :] [offset status :] [diff_offset status :] [disp_attr status :] gain status
gain set coarse : [offset status :[diff_offset status :]] [disp_attr status :] gain status
gain set fine : [offset status :] [diff_offset status :] [disp_attr status :] gain status

name: **generic command**

syntax: **generic command** compartment length command params

type: mainframe command

message tokens:

generic ::= 20_{16}

command ::= 02_{16}

parameters:

name	**type**	**values**
compartment	char	'L', 'C', 'R'
length	us	
command	2 tokens	see text
params	see text	

usage: This message is sent by smart plug-ins to request action or status of generic plug-ins. The *compartment* parameter indicates the compartment for which the command is destined. It is determined from the **sys_config status** message sent by the mainframe in response to a **sys_config query** message.

The *length* parameter specifies the length of the remainder of the message in bytes. This is useful for mainframe processing and is not passed on to the generic plug-in.

The *command* parameter specifies the requested command or status. This parameter may be any of the command tokens defined for generic plug-ins: **gain**, **offset**, **diff_offset**, **coupling**, **impedance**, **bandwidth**, **units**, **disp_attr**, **led**, **display**, **trigger** and **aux_trig** followed by the **set** or **query** message token as required.

The *params* parameter is the list of parameters associated with the command defined by the message tokens of the *command* parameter.

response: The mainframe will return the status response of the generic plug-in to the smart plug-in that requested the action or status.

error handling: The mainframe will return any errors reported by the generic plug-in to the smart plug-in that requested the action or status.

also see: **gain set, offset set, diff_offset set, coupling set, impedance set, bandwidth set, led set, display set, trigger set, aux_trig set, gain query, offset query, diff_offset query, coupling query, impedance query, bandwidth query, led query, display query, trigger query, aux_trig query**

command transactions:

generic command : gain status
generic command : offset status
generic command : diff_offset status
generic command : coupling status
generic command : impedance status
generic command : bandwidth status
generic command : led status
generic command : display status
generic command : trigger status
generic command : aux_trig status

name: **impedance query**

syntax: **impedance query** channel

type: generic plug-in command

message tokens:

impedance ::= 04_{16}

query ::= 02_{16}

parameters:

name	**type**	**values**
channel	us	**1 - 4**

usage: This message requests the plug-in to report the input impedance setting for the specified *channel*.

response: The plug-in will report the input impedance setting using the **impedance status** message.

execution time: The plug-in will send the first byte of the **impedance status** message within xxms of sending the acknowledge transport packet for the **impedance query** message.

error handling: If a channel number is received as part of a **impedance query** message that a plug-in does not have, the plug-in will return an **impedance status** message for an unspecified channel.

also see: **impedance status**

command transactions:

impedance query : impedance status

name: **impedance set**

syntax: **impedance set** channel value

type: generic plug-in command

message tokens:

impedance ::= 04_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
channel	us	1 - 4
value	float	

usage: This message requests a new input impedance setting. The plug-in will set the input impedance for the specified *channel* to the value that is nearest the requested *value*. The meaning of nearest is defined by the plug-in.

response: The plug-in will report the new input impedance setting using the **impedance status** message.

execution time: The plug-in will send the first byte of either the **impedance status** message or the **error generic** message within xxms of sending the acknowledge transport packet for the **impedance set** message.

error handling: If the impedance value is determined to be out of range, the plug-in will make no change to the impedance setting and report the error using the **error generic** message with the **exec_error** *status* token and the *code* parameter set to 205 to indicate the problem. If a channel number is received as part of an **impedance set** message that a plug-in does not have, the plug-in will take unspecified action and return a **impedance status** message indicating the action it took. If a low impedance input resistor is overheated, the plug-in will not change the input impedance to that setting if requested but will report an error using the **error generic** message with the **exec_error** *status* token and the *code* set to 280.

also see: **impedance status**

command transactions:

impedance set : [error generic :] impedance status

name: **impedance status**

syntax: **impedance status** channel value

type: generic plug-in status

message tokens:

impedance ::= 04_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
channel	us	1 - 4
value	float	

usage: This message is used by a generic plug-in to report its input impedance status. It is sent in response to the **impedance set** message, the **impedance query** message or when a probe change requires the input impedance to be changed. The *channel* parameter identifies which channel's impedance is being reported. The *value* parameter specifies the input impedance of that channel.

response: none

error handling: none

also see: **impedance set**, **impedance query**

command transactions:

impedance query : impedance status
impedance set : [error generic :] impedance status

name: **knob change**

syntax: **knob change** which control value

type: mainframe status

message tokens:

knob ::= 19_{16}

change ::= 06_{16}

parameters:

name	type	values
which	token	**knob1**, **knob2**
knob1	byte	08_{16}
knob2	byte	09_{16}
control	token	**coarse**, **medium**, **fine**
coarse	byte	02_{16}
medium	byte	01_{16}
fine	byte	03_{16}
value	short	

usage: This message is sent from the mainframe to the smart plug-in for which the knobs are assigned whenever the either of the knobs is rotated. The *which* parameter indicates which knob was changed: **knob1** or **knob2**. The control parameter indicates the control type that was selected when the user turned the knob. There are three control types: **coarse**, **medium** and **fine**. The plug-in will take action appropriate to the *control* type when this message is received. The *value* parameter indicates the number of knob clicks the knob was rotated. A positive sign indicates clockwise rotation; a negative sign indicates counter-clockwise rotation.

The plug-in is responsible for checking the value returned by the mainframe against the limits of the function being controlled. The plug-in will update the knob display using the **knob update** message.

response: The plug-in will send the **knob status ready** message to the mainframe.

error handling: Error reporting will be appropriate for the function to which the knob is assigned.

also see: **knob request, knob def, knob update, knob keypad**

command transactions:

knob change : knob status ready
knob change : knob update
knob change : knob delete

name: **knob def**

syntax: **knob def** which title min max value units type {control resol}... [which title min max value units type {control resol}...]

type: mainframe command

message tokens:

knob	::= 19_{16}
def	::= 01_{16}

parameters:

name	type	values
which	token	**knob1**, **knob2**
knob1	byte	08_{16}
knob2	byte	09_{16}
title	string	10 chars max
min	float	
max	float	
value	float	
units	string	10 char max
type	token	**linear**, **dbm_2**, **dbm_10**, **step_125** (when used with **coarse**)
linear	byte	04_{16}
dbm_2	byte	05_{16}
dbm_10	byte	06_{16}
step_125	byte	07_{16}
control	token	**coarse**, **medium**, **fine**
coarse	byte	02_{16}
medium	byte	01_{16}
fine	byte	03_{16}
resol	float	

usage: This message is used by smart plug-ins to define knob display parameters.

The *which* parameter specifies to which knob the following parameters apply. The *title* parameter specifies a title for the knob. It may be up to 10 characters. The *min* and *max* parameters specify the minimum and maximum limits of the function to be controlled.

The *value* parameter gives the present value of the function. The *units* parameter specifies the units of the function being controlled.

The *type* parameter defines the type of control. The **linear** token selects linear scaling. The **dbm_2** token selects log base 2 scaling. The **dbm_10** token selects log base 10 scaling. The **step_125** token selects 1-2-5 sequence scaling and is allowed for the coarse control only.

The *control* parameter defines the control type: **coarse**, **medium** or **fine**. The definition of what is a coarse, medium or fine control is up to the plug-in. The *resol* parameter is associated with the *control* type and specifies the resolution of the **coarse**, **medium** or **fine** control. It is a floating point value. The value associated with the **fine** *control* may be used to indicate the resolution for display of all knob parameters. When the **step_125** *type* token is specified, the **fine** *resol* parameter specifies resolution in percent of the present coarse step. The *fine* value will also be used to determine the number of digits to display based on the *fine* resolution and the present value. See **disp_attr status** message for the formula for calculating the number of digits to display. There may be up to three *control resol* pairs for each knob definition. The **fine** *control resol* pair must always be included to specify function resolution. The **coarse** and **medium** *control resol* pairs are optional.

The plug-in may optionally send definitions for both knobs in the same message or a single definition for either knob.

response: The mainframe will respond with a knob status message with the **formatted** token when it has completed knob formatting.

error handling: none

also see: knob def, knob status, knob update, knob change

command transactions:

knob def : knob status ready

name: **knob delete**

syntax: **knob delete**

type: mainframe command

message tokens:

knob ::= 19_{16}

delete ::= 07_{16}

parameters: none

usage: This message is sent to a mainframe by a smart plug-in to remove the knob assignment from the plug-in. The mainframe will assign the knobs as appropriate to another (or no) function.

response: The mainframe will respond with the **knob status** message with the **removed** token indicating the knobs have been reassigned.

error handling: none

also see:

command transactions:

[knob change :] knob delete : knob status removed
[knob keypad :] knob delete : knob status removed

name: **knob keypad**

syntax: **knob keypad** which value

type: mainframe status

message tokens:

knob ::= 19_{16}

keypad ::= 02_{16}

parameters:

name	**type**	**values**
which	token	**knob1, knob2**
knob1	byte	08_{16}
knob2	byte	09_{16}
value	float	

usage: This message is sent from the mainframe to a smart plug-in when the user has entered a value from the keypad. The *which* parameter specifies to which knob the value applies: **knob1** or **knob2**. The *value* parameter is the value entered by the user.

response: The smart plug-in will send the **knob status ready** message as the response.

error handling: Error reporting will be appropriate for the function to which the knob is assigned.

also see: **knob request, knob def, knob update, knob change**

command transactions:

knob keypad : knob status ready
knob keypad : knob update
knob keypad : knob delete

name: **knob request**

syntax: **knob request**

type: mainframe command

message tokens:

knob ::= 19_{16}

request ::= 04_{16}

parameters: none

usage: A smart plug-in uses this message to request the assignment of the mainframe's control knobs to a plug-in function. The mainframe will remove any previous assignment of the control knobs in preparation for assignment to the plug-in.

response: The mainframe will respond with the **knob status** message with the **ready** token when the knobs have been assigned to the plug-in.

error handling: none

also see: **knob status**, **knob def**, **knob update**, **knob change**, **knob keypad**

command transactions:

knob request : knob status ready

name: **knob status**

syntax: **knob status** status

type: mainframe/smart plug-in status

message tokens:

knob ::= 19_{16}

status ::= 03_{16}

parameters:

name	type	values
status	token	**ready**, **removed**, **formatted**
ready	byte	01_{16}
removed	byte	03_{16}
formatted	byte	02_{16}

usage: This message is sent by the mainframe to indicate knob status. The **ready** *status* token is sent in response to a **knob request** message from a plug-in. It indicates the mainframe has assigned the knobs to the plug-in and is ready to display knob parameters. The plug-in also sends this message with the **ready** *status* token in response to a **knob change** or **knob keypad** message from the mainframe.

The **removed** *status* token indicates the knobs are no longer assigned to the plug-in either by request of the plug-in using the **knob delete** message or by another mainframe action that caused the knobs to be reassigned. The **formatted** *status* token is sent in response to a **knob def** or **knob update** message and indicates the knob display has been formatted.

response: If the **knob status removed** message is sent by the mainframe in response to an input other than a **knob delete** message from the plug-in assigned to the knobs, the plug-in will return a **knob status removed** message to the mainframe.

error handling: none

also see: **knob request**, **knob def**, **knob change**, **knob update**, **keypad status**

command transactions:

knob change : knob status
knob def : knob status
knob delete : knob status

knob keypad : knob status
knob request : knob status
knob update : knob status
knob status removed : knob status removed

knob keypad : knob status
knob request : knob status
knob update : knob status
knob status removed : knob status removed

name: **knob update**

syntax: **knob update** which value

type: mainframe command

message tokens:

knob	::= 19_{16}
update	::= 25_{16}

parameters:

name	type	values
which	token	**knob1**, **knob2**
knob1	byte	08_{16}
knob2	byte	09_{16}
value	float	

usage: This message is sent to a mainframe by a smart plug-in that has already defined knob parameters to change the value of the present setting. The *which* parameter specifies which knob's value is being updated: **knob1** or **knob2**. The *value* parameter indicates the new value of the function controlled by the specified knob.

response: The mainframe will respond with the **knob status** message with the **formatted** *status* token when it has finished formatting the knob.

error handling: none

also see: **knob request**, **knob status**, **knob def**, **knob change**

command transactions:

[knob change :] knob update : knob status
[knob keypad :] knob update : knob status

name: **knob update**

syntax: **knob update** which value

type: mainframe command

message tokens:

knob	::= 19_{16}
update	::= 25_{16}

parameters:

name	type	values
which	token	**knob1, knob2**
knob1	byte	08_{16}
knob2	byte	09_{16}
value	float	

usage: This message is sent to a mainframe by a smart plug-in that has already defined knob parameters to change the value of the present setting. The *which* parameter specifies which knob's value is being updated: **knob1** or **knob2**. The *value* parameter indicates the new value of the function controlled by the specified knob.

response: The mainframe will respond with the **knob status** message with the **formatted** *status* token when it has finished formatting the knob.

error handling: none

also see: **knob request, knob status, knob def, knob change**

command transactions:

knob update : knob status

name: **led query**

syntax: **led query** channel

type: generic plug-in command

message tokens:

led ::= 08_{16}

query ::= 02_{16}

parameters:

name	**type**	**values**
channel	us	**1 - 4**

usage: This message requests the plug-in to report the front panel LED status of the specified *channel*.

response: The plug-in will report the LED status using the **led status** message.

execution time: The plug-in will send the first byte of the **led status** message within xxms of sending the acknowledge transport packet for the **led query** message.

error handling: If a channel number is received as part of an **led query** message that a plug-in does not have, the plug-in will take unspecified action and return an **led status** message indicating the action it took.

also see: **led status, led set**

command transactions:

led query : led status

name: **led set**

syntax: **led set** channel state

type: generic plug-in command

message tokens:

led	::= 08_{16}
set	::= 01_{16}

parameters:

name	type	values
channel	us	1 - 4
state	token	**on** | **off**
on	byte	01_{16}
off	byte	02_{16}

usage: This message requests a change in the plug-in front panel LED status of the specified *channel* to *state*. The plug-in will turn the LED on when the **on** token is received and turn it off when the **off** token is received.

response: The plug-in will report the LED status using the **led status** message.

execution time: The plug-in will send the first byte of the **led status** message within xxms of sending the acknowledge transport packet for the **led set** message.

error handling: If the *state* parameter is invalid, the plug-in will take unspecified action and return an **led status** message indicating the action it took. If a channel number is received as part of an **led set** message that a plug-in does not have, the plug-in will take unspecified action and return an **led status** message indicating the action it took.

also see: **led status**, **led query**

command transactions:

led set : led status

name: **led status**

syntax: **led status** channel state

type: generic plug-in status

message tokens:

led	::= 08_{16}
status	::= 03_{16}

parameters:

name	**type**	**values**
channel	us	1 - 4
state	token	**on** ¦ **off**
on	byte	01_{16}
off	byte	02_{16}

usage: A generic plug-in uses this message to report the status of the LED of the specified *channel*. It is sent in response to an **led set** message or to an **led query** message. The *state* parameter will be **off** if the LED is off and **on** if the LED is on.

response: none

error handling: none

also see: **led set**

command transactions:

led query : led status
led set : led status

name: **mainframe message**

syntax: **mainframe message** message

type: mainframe command

message tokens:

mainframe ::= 28_{16}

message ::= 01_{16}

parameters:

name	type	values
message	see text	

usage: This message is sent to a mainframe to allow plug-ins to use mainframe functions that are not defined elsewhere in this interface specification. The contents of the message are mainframe specific. Plug-ins will use the mainframe *level* parameter reported in the **mf_id report** message to determine what messages (if any) will be understood by the mainframe. The mainframe will use the **smart message** message to return status and commands to the plug-in.

response: The mainframe will send the **mainframe message** message with appropriate *message* contents as the reply to the **mainframe message** message from the plug-in.

error handling: none

also see: **smart message**

command transactions:

mainframe message : mainframe message

name: **menu change**

syntax: **menu change** lmpb [**TDL** title] [**area_def** area_id size {**cell_def** | xloc yloc text_type [f1_font text1] [**DLT** f2_font text2]}...]...

type: mainframe command

message tokens:

menu	::=	$0A_{16}$
change	::=	06_{16}

parameters:

name	type	values
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
TDL	byte	02_{16}
title	string	10 characters
area_def	byte	$0A_{16}$
area_id	us	0 - 255
size	pb	0 - 16, 0 - 16
cell_def	byte	01_{16}
xloc	us	1 - 10
yloc	us	1 - 8
text_type	token	**text** ¦ **descript**
text	byte	03_{16}
descript	byte	04_{16}
f1_font	token	**normal** ¦ **touch** ¦ **selected** ¦ **atten**
text1	string	limit 5 char
DLT	byte	08_{16}

f2_font	token	**normal** ¦ **touch** ¦ **selected** ¦ **atten**
text2	string	limit 5 char
normal	byte	09_{16}
touch	byte	05_{16}
selected	byte	06_{16}
atten	byte	07_{16}

usage: Smart plug-ins use this message to cause the mainframe to change an extended menu. This message defines the contents of each cell to be changed for the menu.

The *lmpb* parameter is the long message protocol byte and is used to transfer menu definitions that are longer than 127 bytes. See section **6.1.4 Long Messages** for the definition of the long message protocol.

The **TDL** token, when present, delimits the beginning of the title. The optional *title* parameter specifies the title for the menu. The mainframe will use this title if it can title plug-in menus. If the mainframe does not title menus, this parameter may be ignored. The **TDL** token and *title* string must both be present or absent.

The **area_def** token defines the beginning of an area definition. All of the following cell definitions up to the next **area_def** (or the end of the last message) specify cells that belong to a touch area. The touch area must be rectangular in shape. The cells must be listed in order from left to right and top to bottom. All cells encompassed by the area must be represented by a **cell_def** parameter group (even if a cell has no text).

The *area_id* parameter specifies the touch area to which the cells belong. When a 0 value is specified for the *area_id* parameter, all the following cells are non-touchable cells. The mainframe will not generate an audible click, indicate selection or generate a touch message when the user touches one of these cells. Only non-touchable cells with text need to be specified for this area. The shape of the non-touch area does not need to be rectangular nor do the cells need to be listed in any specific order.

The *size* parameter specifies the size of the associated touch area in cells. This parameter is a packed binary format and specifies the height and width of the touch area as follows:

```
b7  b6  b5  b4  b3  b2  b1  b0
├---height---┤  ├---width----┤
```

A *size* of 0 height and width indicates the area associated with the *area_id* is being deleted. All cells previously defined as part of that area become untouchable and are assigned to the 0 touch area. No changes are made to the content of these cells as a result of this change. Changes to the contents of these cells may be subsequently made in the same manner as for other cells that are not touchable.

The **cell_def** token defines the beginning of a cell definition. The following parameters apply to the cell specified by the *xloc* and *yloc* parameters. There may be as many cell definitions in a **menu change** message as there are cells (up to 80 in **max** mode).

The *xloc* parameter specifies the x location of the cell being defined. It is an unsigned short integer in the range of 1 - 10 for **max** and **mix** modes and 1 - 8 for **min** mode. The values 1 - 10 specify cells from left to right.

The *yloc* parameter specifies the y location of the cell being defined. Its range depends on the type of menu being defined. For **max** mode menus, the range is 1 - 8. For **mix** mode menus, the range is 1 - 7. For **min** mode menus, the only allowable value is 1. Row 7 of the **mix** mode menu is defined as the pop-in sub-menu and is not required to be displayed adjacent to row 6.

The *text_type* parameter specifies how the text is to be placed in the cell. The **text** token specifies that *text1* will appear in the top line and *text2* will appear in the bottom line. The **descript** token specifies that *text1* is the descriptor and *text2* is the status. The mainframe will put the text into the top and bottom lines to correspond to the format being used by the mainframe.

The *f1_font* parameter specifies the font of *text1*. The mainframe will display all of *text1* on the appropriate line in the specified font. *f1_font* may be any of the defined font tokens, **normal**, **touch**, **selected** or **atten**. The **normal** token selects the font used by the mainframe for normal text displays. The **touch** token selects the font the mainframe uses to identify areas that are touchable. The **selected** token selects the font the mainframe uses to identify an area that has been selected. The **atten** token selects the font the mainframe uses as a standout font to attract the user's attention.

The *text1* parameter defines the text to be displayed either in the top line or as the descriptor. If it is absent, the line is left blank. If *text1* contains more than 5 characters, the additional characters are truncated and not displayed.

The **DLT** token is used only when *text2* is present to define the beginning of the text2 definition.

The *f2_font* parameter specifies the font of *text2*. The mainframe will display all of *text2* on the appropriate line in the specified font. *f2_font* may be any of the defined font tokens, **normal**, **touch**, **selected** or **atten**. The **normal** token selects the font used by the mainframe for normal text displays. The **touch** token selects the font the mainframe uses to identify areas that are selectable. The **selected** token selects the font the mainframe uses to identify an area that has been selected. The **atten** token selects the font the mainframe uses as a standout font to attract the user's attention.

The *text2* parameter defines the text to be displayed either in the bottom line or as the status. If it is absent, the line is left blank. If *text2* contains more than 5 characters, the additional characters are truncated and not displayed.

response: The mainframe will send the **menu status** message with the **formatted** status token when it has completed formatting and displaying the menu.

error handling: none

also see: **menu def_smart**, **menu status**

command transactions:

[menu touch :] menu change : menu status

name: **menu def_generic**

syntax: **menu def_generic** lmpb [{menu_type channel item_list}... **EOM**]

item_list ::= coupl... **EOC** ¦ value... NaN

type: generic plug-in status

message tokens:

menu ::= $0A_{16}$

def_generic ::= 01_{16}

parameters:

name	**type**	**values**
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
menu_type	token	**plus_coupl**, **minus_coupl**, **impedance**, **upper_bandw**, **lower_bandw**
plus_coupl		byte 01_{16}
minus_coupl		byte 02_{16}
impedance	byte	04_{16}
upper_bandw		byte 03_{16}
lower_bandw		byte 05_{16}
channel	us	1 – 4
coupl	token	**DC** ¦ **OFF** ¦ **AC** ¦ **VC**
DC	byte	01_{16}
OFF	byte	02_{16}
AC	byte	03_{16}
VC	byte	04_{16}
EOC	byte	05_{16}

value	float	
NaN	float	0xffc00000
EOM	token	06_{16}

usage: This is the general form of the generic plug-in menu definition message. It is used by plug-ins to define the characteristics of generic plug-in basic menus. The five forms are listed separately and define the entries for the coupling (plus and minus), input impedance, and bandwidth limit (upper and lower) menus. This message is sent in response to a **menu request generic** message from the mainframe or whenever the entries of a generic menu need to be changed (eg. caused by changing the probe).

When sent by the plug-in (using the **more** or **last** long message tokens), this message must contain at least one menu definition. Menu parameters are not sent only when the long message token is **ack** or **abort**. See section **6.1.4 Long Messages**.

response: none

error handling: none

also see: **menu request**

command transactions:

menu request generic : menu def_generic
SRQ query : menu def_generic

name: **menu def_generic impedance**

syntax: **menu def_generic** lmpb [{**impedance** channel value... **NaN**}...**EOM**]

type: generic plug-in status

message tokens:

menu ::= $0A_{16}$

def_generic ::= 01_{16}

parameters:

name	type	values
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
impedance	token	04_{16}
channel	us	1 - 4
value	float	
NaN	float	0xffc00000
EOM	token	06_{16}

usage: This message used by generic plug-ins to define the entries of the impedance menu for the specified *channel*. It is sent in response to a **menu request generic** message from the mainframe or whenever the entries of the impedance menu need to be changed (eg. caused by changing the probe). There may be up to 16 impedance *values* sent for each of up to 4 channel menus.

This menu controls the input impedance. The mainframe will send the **impedance set** message with the selected impedance value and appropriate channel parameter when a selection is made from this menu.

When sent by the plug-in (using the **more** or **last** long message tokens), this message must contain at least one menu definition. Menu parameters are not sent only when the long message token is **ack** or **abort**.

response: none

error handling: none

also see: **menu request**, **set impedance**

command transactions:

menu request generic : menu def_generic impedance
SRQ query : menu def_generic impedance

name: **menu def_generic lower_bandw**

syntax: **menu def_generic** lmpb [{**lower_bandw** channel value... **NaN**}... **EOM**]

type: generic plug-in status

message tokens:

menu ::= $0A_{16}$

def_generic ::= 01_{16}

parameters:

name	type	
lmpb	token	**more** | **last** | **ack** | **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
lower_bandw	token	05_{16}
channel	us	1 - 4
value	float	
NaN	float	0xffc00000
EOM	token	06_{16}

usage: This message is used by generic plug-ins to define the entries of the lower bandwidth limit menu for the specified *channel*. It is sent in response to a **menu request generic** message from the mainframe or whenever the entries of a generic menu need to be changed (eg. caused by changing the probe). There may be up to 16 bandwidth *values* for each of up to 4 channel menus.

This menu controls the lower bandwidth limit function. The mainframe will send the **bandwidth set lower** message with the selected bandwidth value and appropriate channel parameter when a selection is made from this menu.

When sent by the plug-in (using the **more** or **last** long message tokens), this message must contain at least one menu definition. Menu parameters are not sent only when the long message token is **ack** or **abort.** See section **6.1.4 Long Messages.**

response: none

error handling: none

also see: **menu request, bandwidth set lower**

command transactions:

menu request generic : menu def_generic lower_bandw
SRQ query : menu def_generic lower_bandw

name: **menu def_generic minus_coupl**

syntax: **menu def_generic** lmpb [{**minus_coupl** channel coupl... **EOC**}... **EOM**]

type: generic plug-in status

message tokens:

menu ::= $0A_{16}$

def_generic ::= 01_{16}

parameters:

name	type	values
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
minus_coupl	token	02_{16}
channel	us	1 - 4
coupl	token	**DC** ¦ **OFF** ¦ **AC** ¦ **VC**
DC	byte	01_{16}
OFF	byte	02_{16}
AC	byte	03_{16}
VC	byte	04_{16}
EOC	byte	05_{16}
EOM	byte	06_{16}

usage: This message is used by generic plug-ins to define the entries of the minus input coupling menu for the specified *channel*. It is sent in response to a **menu request generic** message from the mainframe or whenever the entries of the minus coupling menu need to be changed (eg. caused by changing the probe). There may be up to 4 *coupl* parameters for each of up to 4 channel menus.

This menu controls the minus input of a differential input channel. The mainframe will send the **coupling set minus** message with the selected token and appropriate parameters when a selection is made from this menu.

When sent by the plug-in (using the **more** or **last** long message tokens), this message must contain at least one menu definition. Menu parameters are not sent only when the long message token is **ack** or **abort**. See section **6.1.4 Long Messages**.

response: none

error handling: none

also see: **menu request**, **coupling set minus**

command transactions:

menu request generic : menu def_generic minus_coupl
SRQ query : menu def_generic minus_coupl

name: **menu def_generic plus_coupl**

syntax: **menu def_generic** lmpb [{**plus_coupl** channel coupl... **EOC**}... **EOM**]

type: generic plug-in status

message tokens:

menu ::= $0A_{16}$

def_generic ::= 01_{16}

parameters:

name	type	values
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
plus_coupl	token	01_{16}
channel	us	1 - 4
coupl	token	**DC** ¦ **OFF** ¦ **AC** ¦ **VC**
DC	byte	01_{16}
OFF	byte	02_{16}
AC	byte	03_{16}
VC	byte	04_{16}
EOC	byte	05_{16}
EOM	byte	06_{16}

usage: This message is used by generic plug-ins to define the entries of the plus input coupling menu for the specified *channel*. It is sent in response to a **menu request generic** message from the mainframe or whenever the entries of the plus coupling menu need to be changed (eg. caused by changing the probe). There may be up to 4 *coupl* parameters for each of up to 4 channel menus.

This menu controls the coupling of a single ended channel or the plus input coupling of a differential input channel. The mainframe will send the **coupling set plus** message with the selected token and appropriate parameters when a selection is made from this menu.

When sent by the plug-in (using the **more** or **last** long message tokens), this message must contain at least one menu definition. Menu parameters are not sent only when the long message token is **ack** or **abort**. See section **6.1.4 Long Messages.**

response: none

error handling: none

also see: **menu request, coupling set plus**

command transactions:

menu request generic : menu def_generic plus_coupl
SRQ query : menu def_generic plus_coupl

name: **menu def_generic upper_bandw**

syntax: **menu def_generic** lmpb [{**upper_bandw** channel value... **NaN**}... **EOM**]

type: generic plug-in status

message tokens:

menu ::= $0A_{16}$

def_generic ::= 01_{16}

parameters:

name	**type**	**values**
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
upper_bandw	token	03_{16}
channel	us	1 - 4
value	float	
NaN	float	0xffc00000
EOM	byte	06_{16}

usage: This message is used by generic plug-ins to define the entries of the upper bandwidth limit menu for the specified *channel*. It is sent in response to a **menu request generic** message from the mainframe or whenever the entries of a generic menu need to be changed (eg. caused by changing the probe). There may be up to 16 bandwidth *values* for each of up to 4 channel menus.

This menu controls the upper bandwidth limit function. The mainframe will send the **bandwidth set upper** message with the selected bandwidth limit and appropriate channel parameter when a selection is made from this menu.

When sent by the plug-in (using the **more** or **last** long message tokens), this message must contain at least one menu definition. Menu parameters are not sent only when the long message token is **ack** or **abort.** See section **6.1.4 Long Messages.**

response: none

error handling: none

also see: **menu request generic, bandwidth set upper**

command transactions:

menu request generic : menu def_generic upper_bandw
SRQ query : menu def_generic upper_bandw

name: **menu def_smart**

syntax: **menu def_smart** lmpb [**TDL** title] [**area_def** area_id size {**cell_def** xloc yloc text_type [f1_font text1] [**DLT** f2_font text2]}...]...

type: mainframe command

message tokens:

menu ::= $0A_{16}$

def_smart ::= 02_{16}

parameters:

name	type	values
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
TDL	byte	02_{16}
title	string	10 characters
area_def	byte	$0A_{16}$
area_id	us	0 - 255
size	pb	0 - 16, 0 - 16
cell_def	byte	01_{16}
xloc	us	1 - 10
yloc	us	1 - 8
text_type	token	**text** ¦ **descript**
text	byte	03_{16}
descript	byte	04_{16}
f1_font	token	**normal** ¦ **touch** ¦ **selected** ¦ **atten**
text1	string	5 char

DLT	byte	08_{16}
f2_font	token	**normal** ¦ **touch** ¦ **selected** ¦ **atten**
text2	string	5 char
normal	byte	09_{16}
touch	byte	05_{16}
selected	byte	06_{16}
atten	byte	07_{16}

usage: Smart plug-ins use this message to cause the mainframe to display an extended menu. This message defines the contents of each cell for the menu.

The *lmpb* parameter is the long message protocol byte and is used to transfer menu definitions that are longer than 127 bytes. See section **6.1.4 Long Messages** for the definition of the long message protocol.

The **TDL** token, when present, delimits the beginning of the title. The optional *title* parameter specifies the title for the menu. The mainframe will use this title if it can title plug-in menus. If the mainframe does not title menus, this parameter may be ignored. The **TDL** token and *title* string must both be present or absent.

The **area_def** token defines the beginning of an area definition. All of the following cell definitions up to the next **area_def** (or the end of the last message) specify cells that belong to a touch area. The touch area must be rectangular in shape. The cells must be listed in order from left to right and top to bottom. All cells encompassed by the area must be represented by a **cell_def** parameter group (even if a cell has no text).

The *area_id* parameter specifies the touch area to which the cells belong. When a 0 value for the *area_id* parameter is specified, all the following cells are non-touchable cells. The mainframe will not generate an audible click, indicate selection or generate a touch message when the user touches one of these cells. Only non-touchable cells with text need to be specified for this area. The shape of the non-touch area does not need to be rectangular nor do the cells need to be listed in any specific order.

The *size* parameter specifies the size of the associated touch area in cells. This parameter is a packed binary format and specifies the height and width of the touch area as follows:

```
b7  b6  b5  b4  b3  b2  b1  b0
├---height---┤  ├---width----┤
```

The **cell_def** token defines the beginning of a cell definition. The following parameters apply to the cell specified by the *xloc* and *yloc* parameters. There may be as many cell definitions in a **menu def_smart** message as there are cells (up to 80 for **max** mode).

The *xloc* parameter specifies the x location of the cell being defined. It is an unsigned short integer in the range of 1 - 10 for **max** and **mix** modes and 1 - 8 for **min** mode. The values 1 - 10 specify cells from left to right.

The *yloc* parameter specifies the y location of the cell being defined. Its range depends on the type of menu being defined. For **max** mode menus, the range is 1 - 8. For **mix** mode menus, the range is 1 - 7. For **min** mode menus, the only allowable value is 1. Row 7 of the **mix** mode menu is defined as the pop-in sub-menu and is not required to be displayed adjacent to row 6.

The *text_type* parameter specifies how the text is to be placed in the cell. The **text** token specifies that *text1* will appear in the top line and *text2* will appear in the bottom line. The **descript** token specifies that *text1* is the descriptor and *text2* is the status. The mainframe will put the text into the top and bottom lines to correspond to the format being used by the mainframe.

The *f1_font* parameter specifies the font of *text1*. The mainframe will display all of *text1* on the appropriate line in the specified font. *f1_font* may be any of the defined font tokens, **normal**, **touch**, **selected** or **atten**. The **normal** token selects the font used by the mainframe for normal text displays. The **touch** token selects the font the mainframe uses to identify areas that are touchable. The **selected** token selects the font the mainframe uses to identify an area that has been selected. The **atten** token selects the font the mainframe uses as a standout font to attract the user's attention.

The *text1* parameter defines the text to be displayed either in the top line or as the descriptor. If it is absent, the line is left blank. If *text1* contains more than 5 characters, the additional characters are truncated and not displayed.

The **DLT** token is used only when *text2* is present to define the beginning of the *text2* definition.

The *f2_font* parameter specifies the font of *text2*. The mainframe will display all of *text2* on the appropriate line in the specified font. *f2_font* may be any of the defined font tokens, **normal**, **touch**, **selected** or **atten**. The **normal** token selects the font used by the mainframe for normal text displays. The **touch** token selects the font the mainframe uses to identify areas that are selectable. The **selected** token selects the font the mainframe uses to identify an area that has been selected. The **atten** token selects the font the mainframe uses as a standout font to attract the user's attention.

The *text2* parameter defines the text to be displayed either in the bottom line or as the status. If it is absent, the line is left blank. If *text2* contains more than 5 characters, the additional characters are truncated and not displayed.

response: The mainframe will send the **menu status** message with the **formatted** status token when it has completed formatting and displaying the menu.

error handling: none

also see: **menu def_smart**, **menu status**

command transactions:

menu def_smart : menu status

name: **menu delete**

syntax: **menu delete**

type: mainframe command

message tokens:

menu ::= $0A_{16}$

delete ::= 07_{16}

parameters: none

usage: A smart plug-in will send this message to request the mainframe to remove that plug-in's extended menu.

response: The mainframe will send the **menu status removed** message when it has removed the plug-in's menu.

error handling: none

also see: **menu status**

command transactions:

[menu touch :] menu delete : menu status removed

name: **menu request**

syntax: **menu request** menu_type [wvfm_flag]

type: mainframe command

message tokens:

menu	::= $0A_{16}$
request	::= 04_{16}

parameters:

name	type	values
menu_type	token	**max** ¦ **mix** ¦ **min** ¦ **generic**
max	byte	03_{16}
mix	byte	02_{16}
min	byte	01_{16}
generic	byte	20_{16}
wvfm_flag	token	**wvfm_on** ¦ **wvfm_off**
wvfm_on	byte	01_{16}
wvfm_off	byte	02_{16}

usage: This message has two uses. It is used by smart plug-ins to request the mainframe to prepare for a menu. The *menu_type* parameter specifies the type of menu the plug-in is requesting and is **min**, **mix** or **max**. The *wvfm_flag* specifies waveform status. When the **wvfm_off** token is used, the mainframe will turn off waveform displays if they will interfere with the menu display. When the **wvfm_on** token is used, the mainframe will not disable waveform displays even if they interfere with the menu display. The *wvfm_flag* parameter must be present when this message is sent by a smart plug-in.

This message is also used by the mainframe to request generic plug-in menus at power up. The mainframe will use the **generic** *menu_type* token to request generic menus and will not use the *wvfm_flag* parameter.

response: The mainframe will send a **menu status** message with the **ready** token to the requesting plug-in indicating it is prepared to display a plug-in menu of the specified type.

A generic plug-in will send a **menu def_generic** message in response.

error handling: none

also see: **menu status**, **menu def_smart**, **menu def_generic**

command transactions:

menu request : menu status
menu request : menu def_generic

name: **menu restore**

syntax: **menu restore**

type: smart plugin command

message tokens:

menu ::= $0A_{16}$

restore ::= $0B_{16}$

parameters: none

usage: This message is sent to a smart plugin to restore a menu that was previously removed by the mainframe. When restored, the menu will include any updated information that is a result of operations that occurred since the last time the menu was displayed.

response: The plugin will send the **menu def_smart** message to restore the menu.

error handling: none

also see: **menu def_smart**, **menu status**

command transactions:

menu restore : menu def_smart : menu status formatted

name: **menu status**

syntax: **menu status** status

type: mainframe status

message tokens:

menu ::= $0A_{16}$

status ::= 03_{16}

parameters:

name	type	values
status	token	**ready, formatted, removed**
ready	byte	01_{16}
formatted	byte	02_{16}
removed	byte	03_{16}

usage: This message is sent by the mainframe to indicate menu status to a smart plug-in that has requested a menu display. The **ready** token is sent in response to a **menu request** message to indicate the mainframe is ready to display a plug-in menu. The **formatted** token is sent in response to **a menu def_smart** or **menu change** message and indicates the menu display has been formatted and the mainframe is ready to send menu touches to the plug-in. The **removed** token is sent either in response to **a menu delete** message from the plug-in or whenever the menu is removed for other reasons (such as a mainframe selection that removes the plug-in's menu).

This message is also sent by a smart plug-in in response to **a menu touch** or to a **menu status removed** message when the menu was not removed by the plug-in.

When sent by the mainframe with the **removed** *status* token, this message will cause the smart plug-in to save sufficient information about the menu that was removed to be able to restore it when sent the **menu restore** message.

response: A smart plug-in will send the **menu status removed** message when it receives an unsolicited **menu status removed** message from the mainframe.

error handling: none

also see: **menu request, menu delete, menu def_smart, menu restore**

command transactions:

menu change : menu status
menu delete : menu status
menu request : menu status
menu status : menu status
menu touch : menu status

name: **menu touch**

syntax: **menu touch** area_id

type: mainframe status

message tokens:

menu ::= $0A_{16}$

touch ::= 05_{16}

parameters:

name	**type**	**values**
area_id	us	1 - 255

usage: This message is sent by the mainframe when the user has touched an area of the screen that is part of a plug-in defined menu. The *area_id* parameter indicates the touch area that was selected by the user's touch. This is the *area_id* supplied by the plug-in in the **menu def_smart** or **menu change** message. The mainframe will provide translation from screen coordinates to plug-in menu coordinates.

response: The plug-in will send the **menu status ready** message if it has no further menu transactions to perform. If the plug-in needs to update the menu in response to the menu touch, it may send a **menu change** message as the response to the **menu touch** message. The plug-in may also request a different menu using the **menu requst** message or delete the present menu using the menu delete message, also in response to the **menu touch** message.

error handling: none

also see: **menu def_smart**

command transactions:

menu touch : menu status
menu touch : menu change : menu status formatted
menu touch : menu request : menu status ready
menu touch : menu delete : menu status removed

name: **mf_display query**

syntax: **mf_display query**

type: mainframe command

message tokens:

mf_display ::= 22_{16}

query ::= 02_{16}

parameters: none

usage: This message is sent by a smart plug-in to request waveform display information from the mainframe.

response: The mainframe will send the **mf_display status** message to supply the requested information.

error handling: none

also see: **mf_display status, mf_display set**

command transactions:

mf_display query : mf_display status

name: **mf_display set**

syntax: **mf_display set** wvfm_id

type: mainframe command

message tokens:

mf_display ::= 22_{16}

set ::= 01_{16}

parameters:

name	type	values
wvfm_id	us	1 - 8

usage: This message is sent by a smart plug-in to the mainframe to select the waveform specified by the *wvfm_id* parameter. The *wvfm_id* parameter is selected from the list reported by the mainframe in the **mf_display status** message. The mainframe will make appropriate changes to the display to cause the specified waveform to become the selected waveform. The plug-in is not allowed to create or remove waveforms with this message, only to change the state of existing waveform display.

response: The mainframe will send the **mf_display status** message when it has completed the changes to the display.

error handling: none

also see: **mf_display status**, **mf_display query**

command transactions:

mf_display set : mf_display status

name: **mf_display status**

syntax: **mf_display status** status [wvfm_id source_desc {status wvfm_id source_desc}...]

type: mainframe status

message tokens:

mf_display ::= 22_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
status	token	**live** ¦ **stored** ¦ **selected_live** ¦ **selected_stored** ¦ **none**
live	byte	02_{16}
stored	byte	03_{16}
selected_live	byte	04_{16}
selected_stored	byte	05_{16}
none	byte	01_{16}
wvfm_id	us	1 - 8
source_desc	string	See text

usage: This message is sent by the mainframe in response to a **mf_display query** or **mf_display set** message from a smart plug-in. It contains information about the present status of the mainframe's waveform display. There may be status for up to 8 waveforms reported.

The *status* parameter indicates either the status of the display or the status of the associated waveform. The **none** *status* token indicates that there are no waveforms defined for display. In this case, no other parameters will be sent with the message (just the *status* parameter). The **live** *status* token indicates a live waveform that is not the selected waveform. The **stored** *status* token indicates a stored waveform that is not the selected waveform. The **selected_live** *status* token indicates the selected waveform and specifies that it is a live waveform. The **selected_stored** *status* token indicates the selected waveform and specifies that it is a stored waveform.

A live waveform is defined as a waveform that is associated to an input (either through a digitizer or an analog path). A stored waveform is defined as a digital waveform created from values stored in memory and is not associated to an input.

The *wvfm_id* parameter specifies the waveform id associated with the waveform being identified. The plug-in may use this value in the **mf_display set** message to define a specific waveform as the selected waveform.

The *source_desc* parameter is a character string describing the source of the waveform. This parameter has the same format as the TRACE<ui> DESCRIPTION link argument defined in the *Command Reference Specifications 11000 Series Family of Products* document.

response: none

error handling: none

also see: **mf_display query**, **mf_display set**

command transactions:

mf_display query : mf_display status
mf_display set : mf_display status

name: **mf_id config**

syntax: **mf_id config** status

type: plug-in status

message tokens:

mf_id	::= 14_{16}
config	::= 08_{16}

parameters:

name	type	values
status	token	**new** | **old**
new	byte	01_{16}
old	byte	06_{16}

usage: This message is sent by a plug-in in response to the **mf_id report** message. The plug-in will use the *type* and *uid* parameters from the **mf_id report** message to see if it is installed in the same mainframe as during the previous power on. If it is, the plug-in will send the **old** *status* token in the **mf_id config** message. If the mainframe is different than the previous, the plug-in will send the **new** *status* token.

response: none

error handling: none

also see: **mf_id report**

command transactions:

mf_id report : mf_id config

name: **mf_id report**

syntax: **mf_id report** type uid version level

type: mainframe status

message tokens:

mf_id ::= 14_{16}

report ::= 01_{16}

parameters:

name	**type**	**values**
type	string	10 characters
uid	string	10 characters
version	string	10 characters
level	token	**EO** ¦ **RO**
EO	byte	01_{16}
RO	byte	02_{16}

usage: This message is sent by the mainframe to report its type and capabilities. The *type* parameter is an ASCII string representing the 11000 Series nomenclature for that mainframe. The *uid* parameter is an ASCII string representing the serial number of the mainframe. The *version* parameter is an ASCII string representing the software version number of the mainframe. The *level* parameter is a token that indicates the level of support provided by that mainframe. The levels presently defined are **EO** and **RO**. These are used in conjunction with the **mainframe message** message. The plug-in will save the *type* and *uid* parameters for comparison at the next power on.

response: The plug-in will send the **mf_id config** message as a response.

error handling: none

also see: **mf_id config, mainframe message**

command transactions:

mf_id report : mf_id config

name: **mf_trigger set**

syntax: **mf_trigger set** level slope coupling

type: mainframe command

message tokens:

mf_trigger ::= 23_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
level	float	
slope	token	**plus** ¦ **minus**
plus	byte	01_{16}
minus	byte	02_{16}
coupling	token	**AC** ¦ **DC**
AC	byte	03_{16}
DC	byte	01_{16}

usage: This message is sent by a smart plug-in in response to a **mf_trigger status** message. The plug-in uses this message to set up the mainframe's trigger parameters to trigger on the plug-in's trigger output. The mainframe will change the trigger parameters of all sweeps or other functions for which the smart plug-in's trigger output is defined as part of the source description.

A smart plug-in may also send an SRQ then send this message as the response to the subsequent **SRQ query** message when the user has selected an entry from the plug-in's menu or pressed a plug-in front panel button. This allows the plug-in to update the mainframe's trigger parameters whenever the user makes input to the plug-in.

The *level* parameter specifies the triggering level in divisions from center screen.

The *slope* parameter specifies the slope of the trigger circuit. The **plus** *slope* token selects triggering on the rising slope of the plug-in's trigger output. The **minus** *slope* token selects triggering on the falling slope of the plug-in's trigger output.

name: **mf_trigger status**

syntax: **mf_trigger status**

type: mainframe status

message tokens:

mf_trigger ::= 23_{16}

status ::= 03_{16}

parameters: none

usage: This message is sent to smart plug-ins that have the trigger output function (identified using the **plugin_config status** message) when the plug-in's trigger output is selected as the source for a sweep or other function.

response: The plug-in will send the **mf_trigger set** message to select trigger parameters suitable for triggering on the plug-in's trigger output. If a smart plug-in does not wish to change the sweep parameters, it will send the **mf_trigger status** message instead of the **mf_trigger set** message.

error handling: none

also see: **mf_trigger set**, **plugin_config status**

command transactions:

mf_trigger status : mf_trigger set
mf_trigger status : mf_trigger status

name: **offset query**

syntax: **offset query** channel

type: generic plug-in command

message tokens:

offset ::= 02_{16}

query ::= 02_{16}

parameters:

name	**type**	**values**
channel	us	1 - 4

usage: This message requests the plug-in to report the offset setting for the specified *channel*.

response: The plug-in will report the new offset value using the **offset status** message. The offset value reported will always be the *actual value*. See the **offset set abs** message.

execution time: The plug-in will send the first byte of the **offset status** message within xxms of sending the acknowledge transport packet for the **offset query** message.

error handling: If a channel number is received as part of an **offset query** message that a plug-in does not have, the plug-in will take unspecified action and return a **offset status** message indicating the action it took.

also see: **offset set coarse**, **offset set fine**, **offset status**, **diff_offset query**, **diff_offset set**, **diff_offset status**

command transactions:

offset query : offset status

name: **offset set abs**

syntax: **offset set abs** channel value

type: generic plug-in command

message tokens:

offset ::= 02_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
abs	token	01_{16}
channel	us	1 - 4
value	float	

usage: This message requests a new offset setting of *value* units for the specified *channel*. It is an absolute setting command. The plug-in will check that the offset *value* lies within the achievable limits of the plug-in. If it does not, the plug-in will not change the offset. Otherwise, *value* will be rounded to the nearest achievable setting. This becomes the *requested value*.

The plug-in next checks that the *requested value* is within the achievable limits for the present gain setting. If it is, the plug-in will change the offset to the *requested value* and set the *actual value* to the *requested value*. If the *requested value* is not within the achievable limits for the present gain setting, the plug-in will set the *actual value* to the maximum available gain setting. The plug-in will set the offset to the *actual value*.

When the gain is changed, the plug-in will calculate a new *actual value* based on the new gain value and the *requested value*. If the new *actual value* is different from the old *actual value*, the plug-in will change the offset to the new *actual value*.

If the plug-in supports the differential offset function, this message will set one of the differential offset values. Which value is set is determined by the input coupling and whether an offset probe is connected to either input. The rules for this determination are specified in the plug-in's EIS. The appropriate differential offset function will be set to the *actual value*. No *requested values* are maintained individually for the differential offset functions.

response: The plug-in will report the new offset value using the **offset status** message. The offset value reported will always be the *actual value*.

execution time: The plug-in will send the first byte of either the **offset status** message or the **error generic** message within xxms of sending the acknowledge transport packet for the **offset set abs** message.

error handling: If the *value* parameter is outside the achievable limits of the plug-in at the present gain setting, the plug-in will report an error using the **error generic** message with the **exec_warning** *status* token and the *code* parameter set to 550 to indicate the problem.

If a channel number is received as part of an **offset set abs** message that a plug-in does not have, the plug-in will take unspecified action and return a **offset status** message (and possibly a **diff_offset status** message) indicating the action it took.

also see: **offset set coarse**, **offset set fine**, **offset status**, **error generic**, **gain set abs**, **diff_offset query**, **diff_offset set**, **diff_offset status**

command transactions:

offset set abs : [**diff_offset status** :] [**error generic** :] **offset status**

name: **offset set coarse**

syntax: **offset set coarse** channel value

type: generic plug-in command

message tokens:

offset ::= 02_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
coarse	token	02_{16}
channel	us	1 - 4
value	short	

usage: This message requests a new offset setting for the specified *channel*. The plug-in will calculate the new offset setting by adding *value* times the coarse step size to the present *actual value* then truncating to the nearest coarse setting. The truncation is toward the previous *actual value*. The coarse step size is defined as .25 divisions at the present gain setting.

If *value* is 0, the offset will not be changed. If the calculated setting is outside the achievable limits of the plug-in at the present gain setting, the plug-in will set the offset *requested value* to the maximum or minimum available offset value depending on the sign of *value*. Otherwise, the plug-in will set the offset *requested value* to the calculated coarse offset setting. The plug-in will then set the offset to the *requested value* and update the *actual value* with the *requested value*.

If the plug-in supports the differential offset function, this message will set one of the differential offset values. Which value is set is determined by the input coupling and whether an offset probe is connected to either input. The rules for this determination are specified in the plug-in's EIS. The appropriate differential offset function will be set to the *actual value*. No *requested values* are maintained individually for the differential offset functions.

response: The plug-in will report the new offset setting using the **offset status** message. The offset value reported will be the offset *actual value* not the number of coarse steps changed.

execution time: The plug-in will send the first byte of the **offset status** message within xxms of sending the acknowledge transport packet for the **offset set coarse** message.

error handling: If a channel number is received as part of **a offset set** message that a plug-in does not have, the plug-in will take unspecified action and return a **offset status** message (and possibly a **diff_offset status** message) indicating the action it took. Calculated offset values that are out of the achievable limits of the plug-in do not cause errors to be reported.

also see: **offset set abs**, **offset set fine**, **offset status**, **error generic**, **diff_offset query**, **diff_offset set**, **diff_offset status**

command transactions:

offset set coarse : [diff_offset status :] [error generic :] offset status

name: **offset set fine**

syntax: **offset set fine** channel value

type: generic plug-in command

message tokens:

offset ::= 02_{16}

set ::= 01_{16}

parameters:

name	type	values
fine	token	03_{16}
channel	us	1 - 4
value	short	

usage: This message requests a new offset setting for the specified *channel*. The plug-in will calculate the new offset setting by adding *value* times the fine step size to the present *actual value* then truncating to the nearest fine setting. The truncation is toward the previous *actual value*. Fine steps are defined as .025 division at the present gain setting.

The plug-in will check that this new offset value lies within the achievable limits of the plug-in at the present gain setting. If it doesn't, the offset *requested value* will be set to the maximum or minimum available offset value depending on the sign of *value*. Otherwise, the offset *requested value* will be set to the calculated value. The plug-in will set the offset to the *requested value* and update the *actual value* with the *requested value*.

If the plug-in supports the differential offset function, this message will set one of the differential offset values. Which value is set is determined by the input coupling and whether an offset probe is connected to either input. The rules for this determination are specified in the plug-in's EIS. The appropriate differential offset function will be set to the *actual value*. No *requested values* are maintained individually for the differential offset functions.

response: The plug-in will report the new offset setting using the **offset status** message. The offset value reported will be the offset *actual value* not the number of fine steps changed.

error handling: If a channel number is received as part of a **offset set** message that a plug-in does not have, the plug-in will take unspecified action and return a **offset status** message (and possibly a **diff_offset status** message) indicating the action it took. Calculated offset values that are out of the achievable limits of the plug-in do not cause errors to be reported.

also see: **offset set abs**, **offset set coarse**, **offset status**, **error generic**, **diff_offset query**, **diff_offset set**, **diff_offset status**

command transactions:

offset set fine : [**diff_offset status** :] [**error generic** :] **offset status**

name: **offset status**

syntax: **offset status** channel value

type: generic plug-in status

message tokens:

offset ::= 02_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
channel	us	1 - 4
value	float	

usage: A generic plug-in uses this message to report the status of the offset function. It is sent in response to any of the offset messages: **offset set abs**, **offset set coarse** and **offset set fine** or to the **offset query** message. It will be sent in response to any of the gain messages if the offset is changed as a result of one of those messages: **gain set abs**, **gain set coarse** and **gain set fine**. It will also be sent if the probe is changed and the previous offset cannot be achieved with the new probe. The *channel* parameter identifies which channel's offset is being reported. The *value* parameter specifies the present setting of the offset *actual value* in units. The units are identified by the **units status** message. Minimum and maximum values for the gain control are specified with the **disp_attr status** message.

response: none

error handling: none

also see: **offset set abs**, **offset set coarse**, **offset set fine**, **gain set abs**, **gain set coarse**, **gain set fine**, **units status**, **disp_attr status**, **diff_offset query**, **diff_offset set**, **diff_offset status**

command transactions:

offset set : [diff_offset status :] [error generic :] offset status
offset query : offset status

name: **plugin_config init**

syntax: **plugin_config init**

type: plug-in command

message tokens:

plugin_config ::= $0E_{16}$

init ::= 01_{16}

parameters: none

usage: This message is sent by the mainframe to any plug-in to request that plug-in to set itself to its EIS defined initial settings.

response: The plug-in will send the **plugin_config status** message as the response. For generic plug-ins, this message will be preceded by status reporting for generic plug-in functions using the following messages: **gain status**, **offset status**, **diff_offset status**, **coupling status**, **bandwidth status**, **impedance status**, **display status**, **trigger status**, **led status**, **aux_trig status** and **disp_attr status**. All display, trigger and auxiliary trigger outputs will be disabled. The **plugin_config status** message will always be the last message sent by a generic plug-in.

error handling: none

also see:

command transactions:

plugin_config init : [gain status : offset status : diff_offset status : coupling status : bandwidth status : impedance status : display status : trigger status : led status : aux_trig status : disp_attr status :] plugin_config status

name: **plugin_config init**

syntax: **plugin_config init**

type: plug-in command

message tokens:

plugin_config ::= $0E_{16}$

init ::= 04_{16}

parameters: none

usage: This message is sent by the mainframe to any plug-in to request that plug-in to set itself to its EIS defined initial settings.

response: The plug-in will send the **plugin_config status** message as the response. For generic plug-ins, this message will be preceded by status reporting for generic plug-in functions using the following messages: **gain status**, **offset status**, **diff_offset status**, **coupling status**, **bandwidth status**, **impedance status**, **display status**, **trigger status**, **led status**, **aux_trig status** and **disp_attr status**. All display, trigger and auxiliary trigger outputs will be disabled. The **plugin_config status** message will always be the last message sent by a generic plug-in.

error handling: none

also see:

command transactions:

plugin_config init : [gain status : offset status : diff_offset status : coupling status : bandwidth status : impedance status : display status : trigger status : led status : aux_trig status : disp_attr status :] plugin_config status

name: **plugin_config query**

syntax: **plugin_config query**

type: plug-in command

message tokens:

plugin_config ::= $0E_{16}$

query ::= 02_{16}

parameters: none

usage: The mainframe sends this message to each plug-in during the power up sequence to request information about the plug-in.

response: The plug-in will send the **plugin_config status** message to report information about itself.

error handling: none

also see: **plugin_config status**

command transactions:

plugin_config query : plugin_config status

name: **plugin_config status**

syntax: **plugin_config status** pi_type [disp_channels trig_channels aux_channels] [**minus_coupl**] [**lower_bandw**] [**diff_offset**] [**no_invert**] [**trig_view**] [**trig_out**] [**dig** channels] [**get**]

type: plug-in status

message tokens:

plugin_config ::= $0E_{16}$

status ::= 03_{16}

parameters:

name	**type**	**values**
pi_type	token	**generic** ¦ **smart** ¦ **both**
generic	byte	20_{16}
smart	byte	21_{16}
both	byte	01_{16}
disp_channels	us	
trig_channels	us	
aux_channels	us	
minus_coupl	token	02_{16}
lower_bandw	token	05_{16}
diff_offset	token	$2A_{16}$
no_invert	token	08_{16}
trig_view	token	26_{16}
trig_out	token	03_{16}
dig	token	04_{16}
channels	ui	
get	token	$0f_{16}$

usage: A plug-in will send this message in response to the **plugin_config query** message to report its configuration information. The mainframe will use this information in building the facilities that are available to the user.

The *pi_type* parameter specifies the plug-in type. The **generic** *pi_type* token indicates the plug-in is a generic only plug-in that supports the generic messages defined in section 1.0. The **smart** *pi_type* token indicates the plug-in is a smart only plug-in that uses the smart plug-in interface defined in section **2.0.** The **both** *pi_type* token indicates a plug-in that supports both the generic and smart interfaces.

The *disp_channels*, *trig_channels* and *aux_channels* parameters specify the number of display, trigger and auxiliary channels, respectively, that are provided by a **generic** or **both** type plug-in. Channels are identified as numbers 1 through the indicated number of channels. A **smart** plug-in will not report any channels.

The **minus_coupl** token indicates the generic plug-in has differential input channels and supports the minus coupling generic plug-in function. The **coupling set minus** message is legal only for a generic plug-in that reports the **minus_coupl** token in the **plugin_config status** message.

The **lower_bandw** token indicates the generic plug-in supports the lower bandwidth control function. The **bandwidth set lower** message is legal only for a generic plug-in that reports the **lower_bandw** token in the **plugin_config status** message.

The **diff_offset** token indicates that the plug-in supports the differential offset function. The **diff_offset set** and **diff_offset query** messages are legal only for a generic plug-in that reports the **diff_offset** token in the **plugin_config status** message.

The **no_invert** token indicates a generic plug-in that does not have display or trigger signal inversion capability. This affects the operation of the **display set** and **trigger set** messages defined in section **1.16 New Trace**.

The **trig_view** token indicates that the plug-in has a trigger view function to be used for identifying the trigger location. This capability will cause the mainframe to use the trigger view calibration procedure defined in section **5.0 Calibration** and also put a trigger view trace selection in its new trace menu if it has that capability. See section **2.11 Trigger View** for smart plug-ins for more details.

The **trig_out** token indicates the plug-in has a fixed output trigger signal that may be used for display and triggering. A plug-in that uses this function may not have any generic channels. The mainframe will put a selector for this plug-in in its trigger source menu.

The **dig** token indicates the plug-in is a digitizing plug-in and has digitized channels available for display. The mainframe will put selection means in its new trace menu for these traces. See section **2.14 Waveform Transfer** for more details. The *channels* parameter is used only with the **dig** token and indicates how many digitized channels the plug-in has for display.

The **get** token indicates the plug-in has group execute trigger capability. The mainframe will notify these plug-ins each time it receives a group execute trigger message from the external interface. See section **2.9.3 GPIB Codes and Formats** for more details.

response: none

error handling: none

also see: **plugin_config query**

command transactions:

plugin_config query : plugin_config status

name: **plugin_id query**

syntax: **plugin_id query**

type: smart plug-in command

message tokens:

plugin_id ::= $0F_{16}$

query ::= 02_{16}

parameters: none

usage: The mainframe sends this message to get the nomenclature and software version of a plug-in.

response: The plug-in will send the **plugin_id status** message to report its nomenclature and software version.

error handling: none

also see: **plugin_id status**

command transactions:

plugin_id query : plugin_id status

name: **plugin_id status**

syntax: **plugin_id status** name version

type: smart plug-in status

message tokens:

plugin_id ::= $0F_{16}$

status ::= 03_{16}

parameters:

name	**type**	**values**
name	string	10 characters
version	string	10 characters

usage: This message is sent by a plug-in in response to a **plugin_id query** message from the mainframe. The *name* parameter indicates the plug-in type using 11000 Series nomenclature. The *version* parameter indicates the version of the software resident in the plug-in.

response: none

error handling: none

also see: **plugin_id query**

command transactions:

plugin_id query : plugin_id status

name: **plugin_uid query**

syntax: **plugin_uid query**

type: plug-in command

message tokens:

plugin_uid ::= 10_{16}

query ::= 02_{16}

parameters: none

usage: This message is sent by the mainframe to obtain the plug-in's unit identification number.

response: The plug-in will send the **plugin_uid status** message in response to report the unit identification number.

error handling: none

also see: **plugin_uid status**, **plugin_uid set**

command transactions:

plugin_uid query : plugin_uid status

name: **plugin_uid set**

syntax: **plugin_uid set** uid

type: plug-in command

message tokens:

plugin_uid ::= 10_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
uid	string	10 characters

usage: This message is set by the mainframe to set the unit identification number of the plug-in. This message should be used only in manufacturing or during service repair. The *uid* parameter specifies the value to which the plug-in unit identification number will be set. During manufacturing or repair at a service center, the unit identification number will be set to the serial number of the plug-in. The user might, but should not, change this value. This value is used during calibration to determine calibration requirements based on configuration changes. This function is enabled or disabled by an internal hardware jumper.

response: The plug-in will send the **plugin_uid status** message in response.

error handling: If the hardware jumper is in the disable position, the plug-in will not change the unit id and will send the plugin_uid status message with the previous value. No error message will be generated.

also see: **plugin_uid status**, **plugin_uid query**

command transactions:

plugin_uid set : plugin_uid status

name: **plugin_uid status**

syntax: **plugin_uid status** uid

type: plug-in status

message tokens:

plugin_uid ::= 10_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
uid	string	10 characters

usage: The plug-in will send this message in response to a **plugin_uid query** or **plugin_uid set** message from the mainframe. The *uid* parameter indicates the present setting of the plug-in's unit identification number. This number is set during manufacturing to the serial number of the plug-in. The user might, but should not, change this value.

response: none

error handling: none

also see: **plugin_uid set**, **plugin_uid query**

command transactions:

plugin_uid set : plugin_uid status
plugin_uid query : plugin_uid status

name: **probe query**

syntax: **probe query** input channel

type: generic plug-in command

message tokens:

probe ::= 15_{16}

query ::= 02_{16}

parameters:

name	**type**	**values**
input	token	**plus** ¦ **minus**
plus	byte	01_{16}
minus	byte	02_{16}
channel	us	1 - 4

usage: This message is sent by the mainframe to request the status of the probe on the specified *channel*. The *input* parameter specifies which input is being reported. The **plus** *input* token indicates the input of a single ended channel or the plus input of a differential input. The **minus** *input* tokens specifies the minus input of a differential channel.

response: The plug-in will send a **probe status** message indicating the status of the probe on the specified *channel*.

error handling: none

also see: **probe status**

command transactions:

probe query : probe status

name: **probe status**

syntax: **probe status** input channel level [type uid]

type: plug-in status

message tokens:

probe ::= 15_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
input	token	**plus** ¦ **minus**
plus	byte	01_{16}
minus	byte	02_{16}
channel	us	1 - 4
level	us	0, 1, 2
type	string	10 characters
uid	string	10 characters

usage: This message will be sent in response to an **SRQ query** message when a probe change has been detected by the plug-in. The plug-in will send an SRQ when it detects a change in any of its probes.

The *channel* parameter indicates the channel for which the change is being reported. The *input* parameter specifies which input is being reported. The **plus** *input* token indicates the input of a single ended channel or the plus input of a differential input. The **minus** *input* tokens specifies the minus input of a differential channel. The *level* parameter specifies the probe level. Level 0 indicates no probe is detected by the plug-in. Level 1 probes use the old 7k resistive encoding scheme and may be either an old 7k probe, a non-Tek probe or a new 11k probe that uses the resistive interface. The probe status message will not include *type* and *uid* parameters when the probe *level* is 1. A *level* 2 probe uses the new TEKPROBE interface. The *type* parameter indicates the probe type. The *uid* parameter specifies the probe serial number.

response: none

error handling: none

also see: SRQ query

command transactions:

SRQ query : probe status

name: **probe_id status**

syntax: **probe_id status** channel

type: generic plug-in status

message tokens:

probe_id ::= $0D_{16}$

status ::= 03_{16}

parameters:

name	**type**	**values**
channel	us	1 - 4

usage: This message is used by generic plug-ins to report the status of the probe id button. This message will be sent whenever the probe id button is pressed.

response: none

error handling: none

also see: **channel_id**

command transactions:

SRQ query : probe_id status

name: **setting recall**

syntax: **setting recall** number

type: smart plug-in command

message tokens:

setting ::= $1A_{16}$

recall ::= 04_{16}

parameters:

name	**type**	**values**
number	us	1 - 10

usage: This message is sent by a mainframe to request a smart plug-in to restore its settings to a previously defined state. The plug-in will set all applicable functions to the operational modes and settings defined by the information previously saved by a **setting save** message with *number* as the parameter. The *number* parameter range is 1 - 10.

response: The plug-in will send the **setting status** message with either the **ready** or the **na** *status* token.

error handling: If the plug-in does not have valid setting data for the requested setting *number*, the plug-in will send the **setting status** message with the **na** *status* token.

also see: **setting status**, **setting store**

command transactions:

setting recall : setting status

name: **setting status**

syntax: **setting status** status

type: plug-in status

message tokens:

setting ::= $1A_{16}$

status ::= 03_{16}

parameters:

name	**type**	**values**
status	token	**ready** ¦ **na**
ready	byte	01_{16}
na	byte	02_{16}

usage: This message is sent by a smart plug-in in response to a **setting save** or **setting recall** message from the mainframe. The **ready** *status* token indicates the plug-in has performed the saving or restoration of settings as requested. The **na** status token indicates the plug-in did not have valid setting data for the setting *number* specified in the **setting recall** message.

response: none

error handling: see text

also see: **setting recall**, **setting store**

command transactions:

setting recall : setting status
setting store : setting status

name: **setting store**

syntax: **setting store** number

type: plug-in command

message tokens:

setting ::= $1A_{16}$

store ::= 02_{16}

parameters:

name	**type**	**values**
number	us	1 - 10

usage: This message is sent by the mainframe to request a smart plug-in to save its present settings in the setting register number *number*. The plug-in will save all applicable operational modes and settings needed to restore operation to its present state. The plug-in is not required to save modes that do not affect the present operation of the plug-in. The *number* parameter range is 1 - 10.

response: The plug-in will send the **setting status** message with the **ready** status token when it has stored the settings.

error handling: none

also see: **setting status, setting recall**

command transactions:

setting store : setting status

name: **smart message**

syntax: **smart message** dest src length message

type: mainframe command

message tokens:

smart ::= 21_{16}

message ::= 01_{16}

parameters:

name	**type**	**values**
dest	char	'L', 'C', 'R'
src	char	'L', 'C', 'R'
length	us	
message	see text	

usage: This message is provided to allow smart plug-ins to communicate with each other without interaction from the mainframe. The *src* parameter specifies the source compartment of the message and is used by the receiving plug-in to return a response. The *dest* compartment parameter specifies to the mainframe where to route the message. The *src* and *dest* parameters are single character compartment values that are reported by the mainframe in the **sys_config status** message.

The *length* parameter specifies the number of following bytes in the message. This parameter may be used by the mainframe to properly pass the message on. The *message* parameter is not interpreted by the mainframe. It is data to be sent to the plug-in identified by the *dest* parameter. The format and meaning of the *message* parameter is not specified in this document. This will be specified by the plug-in designers that use this message.

response: The receiving plug-in will formulate a response using the **smart message** message. The mainframe will pass this response on to the originating plug-in.

error handling: none

also see:

command transactions:

smart message : smart message

name: **SRQ no_report**

syntax: **SRQ no_report**

type: plug-in status

message tokens:

SRQ ::= $0B_{16}$

no_report ::= 01_{16}

parameters: none

usage: This message is sent by a plug-in in response to an **SRQ query** message when the plug-in has no status or commands to report. When the mainframe receives this message it will not send any more **SRQ request** messages until it receives another SRQ from the plug-in.

response: none

error handling: none

also see: **SRQ query**

command transactions:

SRQ query : SRQ no_report

name: **SRQ query**

syntax: **SRQ query**

type: plug-in command

message tokens:

SRQ ::= $0B_{16}$

query ::= 02_{16}

parameters: none

usage: This message is sent by the mainframe when it has recognized a plug-in SRQ. The mainframe will continue to send this message until it receives the **SRQ no_report** message from the plug-in.

response: The plug-in will send either status, commands or the **SRQ no_report** message.

error handling: none

also see: **SRQ no_report**

command transactions:

SRQ query : any valid command or status or **SRQ no_report**

name: **status_disp set**

syntax: **status_disp set** [**DLT1** text1] [**DLT2** text2]

type: mainframe command

message tokens:

status_disp ::= $2B_{16}$

set ::= 01_{16}

parameters:

name	**type**	**values**
DLT1	token	01_{16}
text1	string	50 characters
DLT2	token	02_{16}
text2	string	50 characters

usage: This message is sent by a smart plug-in to cause the mainframe to display a status message. The text of the message may be two lines of up to 50 characters each. The *text1* parameter is the text for line 1. The **DLT1** token delimits the text for line 1. The *text2* parameter is the text for line 2. The **DLT2** token delimits the text for line 2. Either **DLT1** and *text1* or **DLT2** and *text2* or both must be present in this message. At least one line of text must be defined.

response: The mainframe will send the **status_disp status** message as the response.

error handling: none

also see: **status_disp status**

command transactions:

status_disp set : status_disp status

name: **status_disp status**

syntax: **status_disp status** status

type: mainframe/smart plug-in status

message tokens:

status_disp ::= $2B_{16}$

status ::= 03_{16}

parameters:

name	**type**	**values**
status	token	**ready** | **removed**
ready	byte	01_{16}
removed	byte	03_{16}

usage: This message is sent by the mainframe in response to the **status_disp set** message. The **ready** *status* token indicates that the message has been formatted and displayed. This message will also be sent when the mainframe removes the plug-in's status display. In this case, the mainframe will send the **removed** *status* token.

response: The plug-in will send this message with the **removed** *status* token when it receives the **status_disp status removed** message from the mainframe.

error handling: none

also see: **status_disp set**

command transactions:

status_disp set : status_disp status
status_disp status : status_disp status

name: **sys_config query**

syntax: **sys_config query**

type: mainframe command

message tokens:

sys_config ::= 11_{16}

query ::= 02_{16}

parameters: none

usage: This message is sent by a smart plug-in to determine the present plug-in/mainframe configuration.

response: The mainframe will send the **sys_config status** message to identify the system configuration.

error handling: none

also see: **sys_config status**

command transactions:

sys_config query : sys_config status

name: **sys_config status**

syntax: **sys_config status** compartment {report_compart name uid pi_type [disp_channels trig_channels aux_channels] [**minus_coupl**] [**lower_bandw**] [**diff_offset**] [**no_invert**] [**trig_view**] [**trig_out**] [**dig** channels] [**get**]}...

type: mainframe command

message tokens:

sys_config ::= 11_{16}

status ::= 03_{16}

parameters:

name	type	values
compartment	char	'L', 'C', 'R'
report_compart	char	'L', 'C', 'R'
name	string	11000 series nomenclature
uid	string	10 characters
pi_type	token	**generic** | **smart** | **both** | **old** | **empty**
generic	byte	20_{16}
smart	byte	21_{16}
both	byte	01_{16}
old	byte	06_{16}
empty	byte	07_{16}
disp_channels	us	
trig_channels	us	
aux_channels	us	
minus_coupl	token	02_{16}
lower_bandw	token	05_{16}
diff_offset	token	$2A_{16}$
no_invert	token	08_{16}
trig_view	token	26_{16}

trig_out token 03_{16}

dig token 04_{16}

channels us

get token $0F_{16}$

usage: This message is sent by the mainframe in response to a **sys_config query** message from a smart plug-in. This message gives system configuration information. The mainframe will report status for each compartment even if there is not an 11000 Series plug-in installed in that compartment. Configuration for up to three compartments may be reported.

The *compartment* parameter specifies the name of the compartment in which the smart plug-in is installed. It is a single character that the mainframe uses to identify that compartment. The smart plug-in will use this response to form external interface command syntax for its parse tables by prepending this character and the underscore to each external command name.

The *other_compart* parameter indicates the compartment identifier for which the following information is being reported. This parameter is 'L', 'C' or 'R'.

The *name* parameter specifies the 11000 series nomenclature as reported by the plug-in. If the plug-in *type* is **old** or **empty**, this string will be the null string.

The *uid* parameter is the unit identifier reported by the plug-in using the **plugin_uid status** message. It is the null string for non-11000 series plugins or empty compartments.

The *pi_type* parameter indicates the type of plug-in. The **generic** *type* token specifies a generic only plug-in. The **smart** *type* token specifies a smart only plug-in. The **both** *type* token specifies a plug-in that uses both the smart and generic interfaces. The **old** *type* token indicates a non-11000 series plug-in is installed in the compartment. The **empty** *type* token indicates that the compartment does not have any plug-in installed. Some mainframe will report empty compartments as **old** since they cannot differentiate between an empty compartment and one with an old plug-in in it.

The *disp_channels*, *trig_channels* and *aux_channels* parameters specify the number of display, trigger and auxiliary trigger channels, respectively, that are provided by a **generic** or **both** *type* plug-in. The information for these parameters is supplied by each plug-in using the **plugin_config status** message.

The **minus_coupl** token indicates the generic plug-in has differential input channels and supports the minus coupling generic plug-in function. The **coupling set minus** message is legal only for a generic plug-in that reports the **minus_coupl** token in the **plugin_config status** message.

The **lower_bandw** token indicates the generic plug-in supports the lower bandwidth control function. The **bandwidth set lower** message is legal only for a generic plug-in that reports the **lower_bandw** token in the **plugin_config status** message.

The **diff_offset** token indicates that the plug-in supports the differential offset function. The **diff_offset set** and **diff_offset query** messages are legal only for a generic plug-in that reports the **diff_offset** token in the **plugin_config status** message.

The **no_invert** token indicates a generic plug-in that does not have display or trigger signal inversion capability. This affects the operation of the **display set** and **trigger set** messages defined in section **1.16 New Trace**.

The **trig_view** token indicates that the plug-in has a trigger view function to be used for identifying the trigger location. This capability will cause the mainframe to use the trigger view calibration procedure defined in section **5.0 Calibration** and also put a trigger view trace selection in its new trace menu if it has that capability. See section **2.11 Trigger View** for smart plug-ins for more details.

The **trig_out** token indicates the plug-in has a fixed output trigger signal that may be used for display and triggering. A plug-in that uses this function may not have any generic channels. The mainframe will put a selector for this plug-in in its trigger source menu.

The **dig** token indicates the plug-in is a digitizing plug-in and has digitized channels available for display. The mainframe will put selection means in its new trace menu for these traces. See section **2.14 Waveform Transfer** for more details. The *channels* parameter is used only with the **dig** token and indicates how many digitized channels the plug-in has for display.

The **get** token indicates the plug-in has group execute trigger capability. The mainframe will notify these plug-ins each time it receives a group execute trigger message from the external interface. See section **2..9.3 GPIB Codes and Formats** for more details.

A plug-in may not send this message until after it has sent a **plugin_config status** message in response to a **plugin_config query** message from the mainframe.

response: none

error handling: none

also see: **sys_config query, plugin_config query, plugin_config status, plugin_uid query, plugin_uid status**

command transactions:

sys_config query : sys_config status

name: **test begin**

syntax: **test begin**

type: plug-in command

message tokens:

test ::= 13_{16}

begin ::= 02_{16}

parameters: none

usage: This message is sent by the mainframe to request a plug-in to enter self test operation. The plug-in will perform initial kernel tests as defined in section **4.0 Self-Test** then wait for the **test complete** message before completing its testing.

response: The plug-in will send the **test status busy** message indicating it has entered the test mode.

error handling: none

also see: **test complete, test status**

command transactions:

test begin : test status

name: **test complete**

syntax: **test complete**

type: plug-in command

message tokens:

test ::= 13_{16}

complete ::= 04_{16}

parameters: none

usage: This message is sent by the mainframe to a plug-in to request it to complete its self testing. When the plug-in receives this message it may use the voltage source and measurement functions defined by the **cal set_cvr** and **cal make_meas** messages. In addition, a plug-in may also test the auxiliary trigger lines using the **cal cvr_connect** message.

response: The plug-in will send the **test status busy** message to indicate it is completing its tests.

error handling: none

also see: **test status**, **test begin**

command transactions:

test complete : test status busy : {[cal set_cvr : cal cvr_status] : [cal make_meas : cal meas_status]}... [cal cvr_connect : cal cvr_connect] : test status

name: **test end**

syntax: **test end**

type: plug-in command

message tokens:

test ::= 13_{16}

end ::= 01_{16}

parameters: none

usage: This message is sent by the mainframe to request the plug-in to exit the test mode and enter the diagnostic mode.

response: The plug-in will send the **diag status** message to indicate it has entered the diagnostic mode.

error handling: none

also see: **test complete, diag status**

command transactions:

test end : diag status

name: **test status**

syntax: **test status** status [block_id area_id routine_id fault_id]

type: plug-in status

message tokens:

test	::= 13_{16}
status	::= 03_{16}

parameters:

name	type	values
status	token	**ready** ¦ **busy**
ready	byte	01_{16}
busy	byte	02_{16}
block_id	us	1 - 11
area_id	us	1 - 11
routine_id	us	1 - 11
fault_id	char	'0' - '9', 'A' - 'F'

usage: This message is sent by the plug-in to report its test status after completing its self testing or in response to a **test begin** or **test complete** message.

When set in response to a **test begin** or **test complete** message, the *status* token will be **busy** indicating the plug-in has entered the test mode and is performing tests. When the **busy** *status* token is sent, none of the other parameters is included.

When the plug-in has completed its testing, it will send the **test status** message with the **ready** *status* token indicating its test mode status. The following parameters indicate information about the first test that failed. If no tests fail, the *block_id*, *area_id* and *routine_id* parameters will be set to 0 and the *fault_id* parameter will be set to '0'.

The *block_id* indicates the block in which the error was found. The *area_id* indicates the area in which the error was found. The *routine_id* indicates the routine that was executing when the error was detected. The *fault_id* indicates the type of fault that was found.

response: none

error handling: none

also see: **test begin, test complete**

command transactions:

test complete : test status busy : {[cal set_cvr : cal cvr_status] : [cal make_meas : cal meas_status]}... [cal cvr_connect : cal cvr_connect] : test status

name: **trace data**

syntax: **trace data** lmpb [**HDLT** wvfm_id format points x_inc x_zero x_mult x_unit y_zero y_mult y_unit] [**DDLT** data]

type: mainframe command

message tokens:

trace ::= $1E_{16}$

data ::= 01_{16}

parameters:

name	type	values
lmpb	token	**more** ¦ **last** ¦ **ack** ¦ **abort**
more	byte	01_{16}
last	byte	02_{16}
ack	byte	03_{16}
abort	byte	04_{16}
HDLT	token	01_{16}
wvfm_id	us	1 - 8
format	token	**integer** ¦ **fraction**
integer	byte	01_{16}
fraction	byte	02_{16}
points	ui	
x_inc	float	
x_zero	float	
x_mult	float	
x_unit	string	
y_zero	float	
y_mult	float	
y_unit	string	
DDLT	token	02_{16}

data see text

usage: This message is sent by a smart plug-in to transfer acquired waveform data to the mainframe for display. The **HDLT** token delimits the header part of the message which is optional. If the header is not sent, the mainframe either presumes default parameters or uses the values specified by the last **trace data** message that included the header parameters. This message uses the long message protocol defined in section **6.1.4 Long Messages**. See that section for details of the *lmpb* parameter. There are no restrictions on the splitting up of waveform data into messages.

The *wvfm_id* parameter specifies the waveform number for which the data applies. It is from a list of waveform id's specified by a previous **trace status** message from the mainframe.

The *format* parameter specifies the type of data to be transferred. The **integer** *format* token specifies integer values of waveform data. The actual value of a point is obtained by multiplying the waveform point integer by the *y_mult* parameter. The **fraction** *format* token specifies fractional format for waveform data. In this form, the data is supplied with an implied decimal point to the left of the most significant bit. The actual value of a point is obtained by multiplying the waveform point fraction by the *y_mult* parameter. This data format type is useful for transferring data from a digitizer to a display unit.

The *points* parameter indicates the number of points in the waveform being sent. This value with the *format* token specifies the number of bytes of waveform data in the message.

The *x_inc* parameter specifies the time increment between each data point. The *x_zero* parameter specifies the zero point (or the trigger point) in divisions relative to the first data point. The *x_mult* specifies the horizontal units per division. The value *x_mult/x_inc* indicates the number of points per division. The *x_unit* parameter gives the name of the horizontal unit of measurement.

The *y_zero* parameter specifies the vertical zero location in divisions. The *y_mult* parameter specifies the vertical units per division. The *y_unit* parameter gives the name of the vertical unit of measurement.

response: The mainframe will send the **trace status ready** message in response to a **trace data** message.

error handling: none

also see: **trace status, trace request**

command transactions:

trace data : trace status ready

name: **trace request**

syntax: **trace request** status [wvfm_id] [points format]

type: smart plug-in command

message tokens:

trace ::= $1E_{16}$

request ::= 04_{16}

parameters:

name	type	values
status	token	**enable** | **disable**
enable	byte	01_{16}
disable	byte	02_{16}
wvfm_id	us	1 - 12
points	ui	
format	token	**integer** | **fraction**
integer	byte	01_{16}
fraction	byte	02_{16}

usage: This message is sent by the mainframe to a smart plug-in that has identified itself as a digitizing plug-in using the **plugin_config status** message. It is sent when a waveform that has one of the plug-in's traces as a component is created by the user.

The *wvfm_id* parameter specifies the waveform number that the plug-in will use to identify data for that trace. This number is a sequencer slot number. The sequencer is programmed by the mainframe. The data identified by *wvfm_id* is the data acquired by the plug-in when the sequencer is set to slot *wvfm_id*. Although the sequencer is programmed by the mainframe, it is controlled (stepped through the sequence of combinations) by the plug-in.

The *status* parameter indicates the status of the request. The **enable** *status* token requests the plug-in to start acquisition and to begin sending data to the mainframe as it is acquired. This will continue until the mainframe sends the **trace status** message with the **disable** *status* token. At that point, the plug-in will terminate acquisition and waveform data transfer to the mainframe. When the **disable** *status* token is sent, the *points* and *format* parameters are not included in the message.

The *points* parameter specifies the number of points per waveform requested by the mainframe. The plug-in will adjust its acquisition to acquire that many points. If it cannot acquire and send data in exactly the number of *points* requested by the mainframe, the plug-in will choose an appropriate point number and send that number of points in the **trace data** message as indicated in the *points* parameter of that message.

The *format* parameter specifies the type of data to be transferred. The **integer** *format* token specifies integer values of waveform data. The **fraction** *format* token specifies fractional format for waveform data. In this form, the data is supplied with an implied decimal point to the left of the most significant bit. This data format type is useful for transferring data from a digitizer to a display unit.

response: The plugin will send the **trace status ready** message in response.

error handling: none

also see: **trace data**, **trace status**

command transactions:

trace request : trace status ready

name: **trace status**

syntax: **trace status** status

type: mainframe/smart plug-in status

message tokens:

trace ::= $1E_{16}$

status ::= 03_{16}

parameters:

name	**type**	**values**
status	token	**ready**
ready	byte	01_{16}

usage: This message is sent with the **ready** *status* token by the mainframe in response to the **trace data** message and by the plug-in in response to the **trace request** message.

response: none

error handling: none

also see: **trace data**, **trace request**

command transactions:

trace data : trace status ready
trace request : trace status ready

name: **trigger query**

syntax: **trigger query**

type: generic plug-in command

message tokens:

trigger ::= 07_{16}

query ::= 02_{16}

parameters: none

usage: This message requests the plug-in to report the trigger output selection.

response: The plug-in will report the combination list using the **trigger status** message.

execution time: The plug-in will send the first byte of the **trigger status** message within xxms of sending the acknowledge transport packet for the **trigger query** message.

error handling: none

also see: **trigger status**

command transactions:

trigger query : trigger status

name: **trigger set**

syntax: **trigger set** num comb...

type: generic plug-in command

message tokens:

trigger ::= 07_{16}

set ::= 01_{16}

parameters:

name	**type**	**values**
num	us	1 - 12
comb	special	bits 7 6 5 4 3 2 1 0 meaning P4 E4 P3 E3 P2 E2 P1 E1 En = 0 => channel n is off En = 1 => channel n is on Pn = 0 => channel n is +up Pn = 1 => channel n is inverted

usage: This message requests a new trigger output selection. The plug-in will set its trigger sequencer to the specified *comb*inations. The *num* parameter indicates the number of combinations being specified. There may be up to 12 *comb*inations. A complete list of combinations must be sent each time a change is made to any combination. If there are fewer than 12 *comb*inations specified, the plug-in will set the remaining combinations to all 0's. The number of combinations in the list specifies the length of the channel switching sequence. The mainframe must append null combinations (all 0's) as necessary to match the number of combinations to the length of the sequence.

response: The plug-in will report the new combination list using the **trigger status** message.

execution time: The plug-in will send the first byte of either the **trigger status** message within xxms of sending the acknowledge transport packet for the **trigger set** message.

error handling: If there are no *comb*inations, the plug-in will make no change to any combination. If there are more than 12 combinations, the additional combinations will be ignored.

also see: **trigger status**

command transactions:

trigger set : trigger status

name: **trigger status**

syntax: **trigger status** num comb...

type: generic plug-in status

message tokens:

trigger ::= 07_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
num	us	1 - 12
comb	special	bits 7 6 5 4 3 2 1 0 meaning P4 E4 P3 E3 P2 E2 P1 E1 En = 0 => channel n is off En = 1 => channel n is on Pn = 0 => channel n is +up Pn = 1 => channel n is inverted

usage: A generic plug-in uses this message to report the present status of its trigger outputs. It is sent in response to a **trigger set** message or to the **trigger query** message. The plug-in will report the status of all defined *comb*inations. The *num* parameter specifies the number of combinations being reported. The number of combinations reported will match the number specified by the mainframe. There may be up to 12 *comb*inations reported.

response: none

error handling: none

also see: **trigger set**

command transactions:

trigger query : trigger status
trigger set : trigger status

name: **trig_view request**

syntax: **trig_view request** status

type: mainframe command

message tokens:

trig_view ::= 26_{16}

request ::= 04_{16}

parameters:

name	**type**	**values**
status	token	**on** ¦ **off**
on	byte	01_{16}
off	byte	02_{16}

usage: This message is sent by either the mainframe or a smart plug-in to enable or disable the trigger view function. This message will be sent by the mainframe to a smart plug-in when the user has selected the display of the trigger view function from a mainframe menu. This applies only to smart plug-ins that have identified the trigger view function in the **plugin_config status** message. This message will be sent by a smart plug-in that has trigger view capability when the user has selected that function from a smart plug-in menu.

The **on** *status* token requests that plug-in or the mainframe to enable the trigger view function. The **off** *status* token requests the plug-in or the mainframe to disable the trigger view function.

response: The mainframe or the plug-in will send the **trig_view status** message in response to the **trig_view request** message.

error handling: none

also see: **trig_view status**

command transactions:

trig_view request : trig_view status

name: **trig_view status**

syntax: **trig_view status** status

type: mainframe status

message tokens:

trig_view ::= 26_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
status	token	**on** ! **off**
on	byte	01_{16}
off	byte	02_{16}

usage: This message is sent by a smart plug-in or the mainframe in response to the **trig_view request** message. The *status* token indicates the status of the function. The **on** *status* token indicates the function has been enabled. When sent from the plug-in it indicates the plug-in is sending the appropriate signals out the AB-13 and AB-11 interface pins. When send by the mainframe, it indicates the mainframe is displaying the trigger point in a manner appropriate to its display capability. The **off** *status* token indicates the mainframe or plug-in has disabled the trigger view function.

response: none

error handling: none

also see: **trig_view request**

command transactions:

trig_view request : trig_view status

name: **units query**

syntax: **units query** channel

type: generic plug-in command

message tokens:

units ::= 09_{16}

query ::= 02_{16}

parameters:

name	**type**	**values**
channel	us	1 - 4

usage: This message is used by the mainframe to request the *units* for the specified *channel*.

response: The plug-in will send the **units status** message to report the units.

error handling: If a channel number is received as part of a **units query** message that a plug-in does not have, the plug-in will take unspecified action and return a **units status** message indicating the action it took.

also see: **units status**

command transactions:

units query : units status

name: **units status**

syntax: **units status** channel units

type: generic plug-in status

message tokens:

units ::= 09_{16}

status ::= 03_{16}

parameters:

name	type	values
channel	us	1 - 4
units	string	10 char

usage: This message is used by generic plug-ins to report the *units* for the specified *channel*. The *units* are derived from the presently connected probe, or, if no probe is connected or if the probe does not use the new digital probe interface, the units will be "Volts". This message will be sent in response to the **units query** message or whenever a probe change causes the units to be changed.

response: none

error handling: none

also see: **units query**

command transactions:

units query : units status

name: **update request**

syntax: **update request** compartment mode

type: mainframe command

message tokens:

update ::= 25_{16}

request ::= 04_{16}

parameters:

name	**type**	**values**
compartment	char	'L', 'C', 'R'
mode	token	**on** ! **off**
on	byte	01_{16}
off	byte	02_{16}

usage: This message is sent by a smart plug-in to request updates from generic plug-ins. The *compartment* parameter specifies the plug-in compartment for which the smart plug-in is requesting updates. The *mode* parameter specifies whether the plug-in is enabling or disabling updates. The **on** *mode* parameter requests the mainframe to enable the update function for the specified *compartment*. The **off** *mode* token requests the mainframe to disable updates from the specified *compartment*. When the update function is enabled, the mainframe will send to the smart plug-in any status messages received from the generic plug-in in the specified *compartment*. It is the responsibility of the smart plug-in to request initial status of the plug-in using the **generic command** message.

response: The mainframe will send the **update status** message to report the status of the request.

error handling: none

also see: update status

command transactions:

update request : update status

name: **update status**

syntax: **update status** status [compartment function param]...

type: smart plug-in command

message tokens:

update ::= 25_{16}

status ::= 03_{16}

parameters:

name	**type**	**values**
status	token	**status** ¦ **ready**
status	byte	03_{16}
ready	byte	01_{16}
compartment	char	'L', 'C', 'R'
function	token	see text
param	see text	

usage: This message is sent by the mainframe when it has status to report from a compartment for which the update function has been enabled. It is sent with the **status** *status* token to the smart plug-in that requested the update function. The *compartment* parameter indicates the compartment for which the status is being reported. The *function* parameter indicates the function for which status is being reported. This parameter can be any of the function status message tokens defined for generic plug-ins: **gain status**, **offset status**, **diff_offset status**, **coupling status**, **impedance status**, **bandwidth status**, **units status**, **led status**, **display status**, **trigger status**, **channel_id status** and **probe_id status**. When the function parameter is one of these token pairs, the *params* parameter indicates the channel and status of the reported function as defined in section **1.0 Generic Plug-ins**.

This message is also sent by the mainframe in response to the **update request** message or by the **smart** plug-in in response to the **update status** message. In these cases, the *status* token will be **ready** and the rest of the parameters will not be included.

response: When sent to a smart plug-in, the plug-in will send the **update status** message with the **ready** *status* token and no other parameters.

error handling: none

also see: **update request**

command transactions:

update request : update status
update status : update status

autorange request, 70, 153
autorange status, 70, 154
aux_trig query 31, 155
aux_trig set 31, 156
aux_trig status 31, 157

bandwidth query lower, 17, 158
bandwdith query upper, 17, 159
bandwidth set lower, 17, 160
bandwidth set upper, 17, 162
bandwidth status lower, 17, 163
bandwidth status upper, 17, 164
busy query 101, 165
busy status 102, 166

cal acc_query 123, 167
cal acc_status 123, 168
cal const_num 170
cal const_query 82, 171
cal const_query_num 172
cal const_set 83, 173
cal const_status 82, 174
cal cvr_connect, 90, 111, 175
cal cvr_mode, 89, 110, 177
cal cvr_set, 124, 179
cal cvr_status, 124, 180
cal measure, 125, 181
cal meas_set 125, 182
cal meas_value, 125, 183
cal mf_imbalance 104, 184
cal mode 102, 185
cal pi_imbalance 104, 188
cal probe 120, 189
cal probe_act query 120.3, 190.1
cal probe_act set 120.2, 190.2
cal probe_act status 120.3, 190.4
cal probe_nom query 120.2, 190.5
cal probe_nom set 120.2, 190.6
cal probe_nom status 120.2, 190.8
cal req_cal 123, 191
cal status 103, 192
cal tv_nom 122, 194
cal tv_set 122, 195
cal tv_status 121, 196
channel_id status, 24, 198
coupling query minus, 15, 199
coupling query plus, 15, 200
coupling set minus, 14, 201
coupling set plus, 14, 203
coupling status minus, 14, 205
coupling status plus, 14, 206

cursor limits, 53, 207
cursor set, 51, 209
cursor status, 53, 212

diag com_test, 95, 214
diag config, 92, 215
diag enter, 92, 219
diag exec, 93, 220
diag exit, 92, 222
diag halt, 94, 223
diag request, 92, 224
diag status, 94, 225
diff_offset query 13, 227
diff_offset set 12, 228
diff_offset status 13, 229
display query, 30, 230
display set, 29, 231
display status, 29, 233
disp_attr query, 20, 234
disp_attr status, 19, 235

error generic, 78, 238
error req_text, 80, 240
error smart, 78, 241
error text, 80, 243
external fp_button, 68, 245
external fp_mode, 67, 246
external get, 68, 247
external help_query, 65, 248
external help_status, 65, 249
external interf_data, 66, 250
external long_form, 64, 251
external message, 63, 252
external oper_compl, 66, 253
external req_table, 62, 254
external set_data, 65, 255
external set_query, 64, 256
external set_status, 64, 257
external status, 63, 258
external table_data, 63, 259

function select, 69, 261
function status, 69, 263

gain query, 8, 264
gain set abs, 7, 265
gain set coarse, 8, 267
gain set fine, 8, 269
gain status, 8, 271
generic command, 55, 272

impedance query 16, 274
impedance set, 15, 275
impedance status, 16, 276

knob change, 49, 277
knob def, 48, 279
knob delete, 49, 281
knob keypad, 49, 282
knob request, 47, 283
knob status, 47, 284
knob update, 48, 286

led query, 26, 287
led set, 25, 288
led status, 26, 288

mainframe message 59, 290
menu change, 37, 291
menu def_generic, 21, 295
menu def_generic impedance, 24, 297
menu def_generic lower_bandw, 24, 299
menu def_generic minus_coupl, 23, 301
menu def_generic plus_coupl, 23, 303
menu def_generic upper_bandw, 24, 305
menu def_smart, 36, 307
menu delete, 37, 311
menu request, 22, 35, 77, 312
menu restore, 37, 314
menu status, 36, 315
menu touch, 37, 317
mf_display query, 57, 318
mf_display set, 58, 319
mf_display status, 58, 320
mf_id config 100, 322
mf_id report 100, 323
mf_trigger set, 58, 324
mf_trigger status, 58, 326

offset query, 11, 327
offset set abs, 10, 328
offset set coarse, 10, 330
offset set fine, 10, 332
offset status, 10, 334

plugin_config init, 82, 335
plugin_config query, 75, 336
plugin_config status, 76, 337
plugin_id query, 81, 340
plugin_id status, 81, 341
plugin_uid query, 81, 342
plugin_uid set, 82, 343
plugin_uid status, 81, 344
probe query 101, 345
probe status 101, 346
probe_id status, 25, 348

setting recall, 54, 349
setting status, 54, 350
setting store, 54, 351
smart message, 56, 252
SRQ no_report, 83, 353
SRQ query, 83, 354
status_disp set 46, 355
status_disp status 46, 356
sys_config query, 56, 357
sys_config status, 56, 358

test begin, 88, 362
test complete, 87, 363
test end, 90, 364
test status, 88, 365
trace data, 72, 367
trace request, 71, 370
trace status, 72, 372
trigger query, 30, 373
trigger set, 29, 374
trigger status, 30, 376
trig_view request, 70, 377
trig_view status, 70, 378

units query, 18, 379
units status, 18, 380
update request, 59, 381
update status, 60, 382