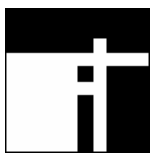


Agile Coherent Synthesizer SCPI Programming Guide



Innovative
Technology

Table of Contents

CALibration Subsystem	4
CALibration [ALL UDOWN UUP LOBAND HIBAND]	4
CALibration:STATE <boolean>.....	4
CALibration:DATA Row, CalValue	5
MEMory Subsystem	6
MEMory:TABLE:CONTrol Row[, (<Numeric List>)].....	6
MEMory:TABLE:CONTrol:DESCription <String>	7
MEMory:TABL:CONTrol:LOOKupmode?	7
MEMory:TABLE:CONTrol:PREamble:ROWs <numeric_value>	8
OUTPut Subsystem	9
OUTPut[:STATE] <Boolean>	9
OUTPut:MARKER1 [NONE SEQuence TABLE]	9
OUTPut:MARKER2 [NONE SEGment TABLE]	10
STATus Subsystem	11
STATus:ENVironment?	11
SYSTem Subsystem	12
SYSTem:ERRor[:NEXT]?.....	12
SYSTem:VERSion?.....	12
TRIGger Subsystem	13
ABORT	13
INITiate:[ALL]	13
INITiate:CONTinuous <Boolean>	14
TRIGger:DISable [NONE SEQuence SEGment 	14
TRIGger:LOOP <numeric_value>	14
TRIGger:SOURce [NONE IMMEDIATE FIBER PARAllel EXTernal 	15
UNIT Subsystem	16
UNIT:FREQuency [GHZ MHZ KHZ HZ].....	16
UNIT:DWELL [S MS US NS].....	16
IEEE-488.2 Command Reference	17
*IDN?	17
*RST	17
*CLS	17
*TST?	17
*OPC?	18
*TRG	18
*STB?	18
Innovative Technology Command Reference	19
IT:MEM:TABLE:ROW <numeric_value>.....	19
IT:DEBUG:MSG [NONE TACiturn VERbose TCP FPGA]	19
Programming Examples	20
CScpiExamples Class	20
void CScpiExample::SampleDataRun() const	20
void CScpiExample::SendSynthWaveform() const	20
void CScpiExample::LoadSynthWaveform	21

void CScpiExample::ReadSynthTable(String& fullData).....	21
void CScpiExample::TcpReadAllData()	21
bool CScpiExample::TcpRead(String& dataRead, int msToWait)	22
void CScpiExample::WaitForOperationComplete()	22
void CScpiExample::Button_CalibrateClick()	22
NScpiMsg Namespace.....	23
ScpiMsg.cpp	23
const String NScpiMsg::CalCmd(int calType)	23
const String NScpiMsg::CalStateCmd(bool calOn)	23
const String NScpiMsg::FreqUnitCmd(bool useMhz)	24
const String NScpiMsg::InitCmds()	24
const String NScpiMsg::LoopCmds(bool continuousOn, int loopCount).....	24
const String NScpiMsg::Marker1Cmd(int marker1Type).....	24
const String NScpiMsg::Marker2Cmd(int marker2Type).....	25
const String NScpiMsg::OutputCmd(bool turnOn).....	25
const String NScpiMsg::PreambleRowCmd(int preambleRow)	25
const String NScpiMsg::TriggerDisableCmd(int disableCmd).....	25
const String NScpiMsg::TriggerSourceCmd(int trigSourceType)	26
const String NScpiMsg::WaveformDescriptionCmd(const String& formDescription)	26
ScpiMsg.h	26
NScpiCommands Namespace.....	28
ScpiCommandConstants.h	28

CALibration Subsystem

CALibration [ALL | UDOWN | UUP | LOBAND | HIBAND]

The command performs a calibration on the instrument to provide a level power output at all frequencies.

- ALL initiates a full calibration of the instrument
- UDOWN calibrates only the UHF Down module
- UUP calibrates only the UHF Up module
- LOBAND calibrates only the .25-1 GHz module
- HIBAND calibrates only the .9-18 GHz module

Example

CAL ALL

Reset Condition

On reset the instrument does not perform a calibration. Any previous calibration data stored in the instrument's non-volatile RAM will be used.

Query

CALibration?

The query returns 1 or 0 into the output buffer.

- 1 is returned if the instrument has a full valid calibration data set
- 0 is returned if full calibration data is not available

Error Message

If the calibration does not succeed error -231, "Data Questionable" occurs.

Notes

This is a non-standard SCPI command. Note that there is no colon allowed between the CALibration and ALL keyword and that there is no query form for this commands that both performs the calibration and returns the status. The query form just returns the status. The calibration requires that a proper load be attached to the output ports of the synthesizer. At the conclusion of each calibration stage the EEPROM is updated with the new calibration data. In certain stages this update can take minutes to complete. Running a full calibration can require up to 15 minutes. The synthesizer is left with the RF output power muted at the completion of the calibration.

CALibration:STATE <boolean>

This command instructs the instrument to use the current calibration data to produce level power output. Currently the ACS does not have a defined operation for running with Calibration Off. Therefore while this command is technically supported, it currently has

no effect on the operation of the ACS.

Example

```
CAL:STATE ON
```

Reset Condition

On reset the instrument the calibration state will be on provided a valid calibration data set is available.

Query

```
CALibration:STATE?
```

- 1 is returned if the calibration state is on
- 0 is returned if the calibration state is off

Error Message

If calibration data is not available error -231, “Data Questionable” occurs.

CALibration:DATA Row, CalValue

This command downloads a single row of calibration data to the instrument. The cal table starts at row 0. The value corresponds to the attenuator setting for the portion of the cal table being downloaded. The structure of the cal table is very specific and it is expected that this command is used only to send back data that was originally read from the synthesizer using the query form. Once the entire table is sent a command with a Row number of -1 forces the instrument to store the downloaded data into EEPROM.

Example

```
CAL:DATA 0, 8;
```

Query

```
CALibration:DATA?
```

This command will put the entire contents of the internal calibration table in the output buffer. A line feed separates the values between each logical band, that is, UHF Up, UHF Down, LO Band, HI Band). Before sending this data back to the synthesizer it must be reformatted with the row number (starting at 0) and a single value per line.

Error Message

If calibration data is not available error -231, “Data Questionable” occurs.

MEMory Subsystem

MEMory:TABLE:CONTROL Row[, (<Numeric List>)]

The MEMory:TABLE:CONTROL command allows the internal control table of the instrument to be filled with user defined values.

The first parameter Row specifies the row in the control table that is being written. The lowest available row is 1 and the maximum value is 124,999. Once all rows have been sent the final row, and size of the table, is designated by using a negative row number.

For example, if rows 1 through 1000 are set then send MEM:TABL:CONT -1000, to specify that the last row of the table is 1000. Note that no numeric list is needed when sending a negative row number.

The <Numeric List> is a string is comprised of a series of comma separated values. Each value corresponds to one of the four values in a row in the instrument's control table. The four values in the numeric list in order are FREQUency, DWELL, ATTENUation, MARKer. The total length of the entire command must be 255 bytes or fewer.

The units for frequency defaults to GHz but can be changed to MHz, KHz or Hz using the UNIT:FREQUENCY command. The default unit for DWELL is microseconds but can be changed to milliseconds, nanoseconds or seconds with the UNIT:DWELL command. The change to the units must be made prior to sending the control table commands.

Attenuation values are specified as an offset to the calibrated data. The offset is specified in dB and must be in the range of + or - 7 dB. The markers are turned on with a 1 and off with a 0. The range of valid values is 1 -3.

Example

Program three frequencies as follows:

Frequency	Dwell	Attenuation	Markers
10.2 GHz	5.5 ms	0	1
17.5 GHz	250 us	6 db	1&2
400 MHz	1.6 ms	0	2

Assuming the UNITs for frequency and dwell are set to defaults:

```
MEM:TABL:CONT 1, ( 10.2, 5500, 0);
```

```
MEM:TABL:CONT 2, (17.5, 250, 6, 3);
```

```
MEM:TABL:CONT 3, (.4, 1600, 0, 2);
```

```
MEM:TABL:CONT -3;
```

All the columns except Frequency are optional. Columns may only be omitted in order starting with the last column (Markers).

```
MEM:TABL:CONT 1, (10.2);
```

MEM:TABL:CONT 2, (17.5);

MEM:TABL:CONT 3, (.4);

This command will specify just the Frequency information for the above example.

Reset Condition

On reset, the control table is cleared.

Query

MEM:TABL:CONT?

The query returns the entire contents of the instrument's control table in the in the output buffer. The table is returned as a comma separated value list with line feed characters delimiting each row.

NOTE

The instrument automatically decides which band to use for downloaded frequencies. If all frequencies can be generated within a single band then that band is used. If the table requires both the LO and HI bands then the default break point is used. If the downloaded table is in a range such that it could be generated by both the LO and HI bands, then the LO band is used.

MEMory:TABLE:CONTrol:DESCription <String>

This command allows up to 200 characters in <String> to describe the current waveform loaded into the control table. The

Example

MEM:TABL:CONT:DESC "Target A1 in mixed mode analysis"

Reset Condition

On reset the description is empty.

Query

MEMory:TABLE::CONTrol:DESC?

The query returns the stored string.

MEMory:TABLE:CONTrol:LOOKupmode?

Queries the ACS for the frequency lookup mode currently in use. The ACS has frequency overlap between the Sub L and high bands. The frequencies in the range of 850 MHz through 1050 MHz can be generated in either band. When the ACS receives a frequency table it attempts to generate all the frequencies in a single band if possible. If not it uses Sub L for frequencies below 1000 and high band for all others. Sometimes it is helpful to know what mode the ACS is in based on the data sent.

The query returns 0, 1 or 2 into the output buffer

- 0 is returned when the lookup mode is normal
- 1 is returned when the lookup mode is forced into Sub L band
- 2 is returned when the lookup mode is forced into High band

Example

```
MEM:TABL:CONT:LOOK?
```

MEMory:TABLE:CONTRol:PREamble:ROWs <numeric_value>

This command specifies how many rows at the beginning of the internal control table of the instrument are part of the preamble section. If this value is set at 0 then the entire table is considered the main table with no preamble.

Example

```
MEM:TABL:CONT:PRE 50
```

Reset Condition

On reset the value is 0.

Query

```
MEMory:TABLE::CONTRol:PREamble:ROWs?
```

The query returns the numeric value as a string.

```
MEMory:TABL:CONTRol:LOOKupmode?
```

OUTPut Subsystem

OUTPut[:STATE] <Boolean>

This command turns on and off the RF output.

Example

OUTP ON

Reset Condition

On reset the RF output is muted (off).

Query

OUTPut[:STATE]?

The query returns 1 or 0 into the output buffer.

- 1 is returned when the output is on
- 0 is returned when the output is off

OUTPut:MARKER1 [NONE] SEQuence | TABLE]

This command controls the pulse on the Marker2 output. Marker1 by default is NONE and no pulse is generated. If SEQuence is selected then the marker is generated at the start of each sequence. If TABLE is used the output of the marker is driven by the internal control table.

Example

OUTP:MARKER1 SEQ

Reset Condition

On reset, MARKER1 is set to NONE.

Query

OUTPut:MARKER1?

The query returns the current setting for MARKER1 either NONE, SEQ or TABLE is returned in the output buffer.

OUTPUT:MARKER2 [NONE| SEGment| TABLE]

This command controls the pulse on the Marker2 output. Marker2 by default is NONE and no pulse is generated. If SEGment is selected then the marker is generated at the start of each segment. If TABLE is used the output of the marker is driven by the internal control table.

Example

```
OUTP:MARKER2 SEG
```

Reset Condition

On reset, MARKER2 is set to NONE.

Query

```
OUTPut:MARKER2?
```

The query returns the current setting for MARKER2 either NONE, SEG or TABLE is returned in the output buffer.

STATus Subsystem

STATus:ENVironment?

Returns the voltages and temperature readings in the ACS. The VICR voltages are reported as pass or fail.

Temperature:	39.2 deg C
3.3VBIT	3.25
2.5VBIT	2.24
15VBIT	14.98
17VBIT	16.78
9VBIT	9.00
9VICR	PASSED
15VICR	PASSED
12VICR	PASSED
5VBIT	5.01
12VBIT	11.93
VCCBIT	3.15

Example

STAT:ENV?

An example of the returned data would be:

Temperature:	39.2 deg C
3.3VBIT	3.25
2.5VBIT	2.24
15VBIT	14.98
17VBIT	16.78
9VBIT	9.00
9VICR	PASSED
15VICR	PASSED
12VICR	PASSED
5VBIT	5.01
12VBIT	11.93

SYSTEM Subsystem

SYSTEM:ERRor[:NEXT]?

Returns the next error number and corresponding error message in the Instrument's error queue. Works as a FIFO

When no error exists +0, "No error" is returned

Example

```
SYST:ERR?
```

SYSTEM:VERSion?

Returns the version of SCPI being supported.

Example

```
SYST:VERS?
```

TRIGger Subsystem

The instrument has three states: IDLE, ARMED, and TRIGGERED. At power on and after reset the instrument enters the idle state. The instrument can be armed with either the INITiate[:ALL] command or the INITiate:CONTinuous ON command. Once armed the system will respond to triggers in a variety of ways as defined in this document. The system will immediately exit the TRIGGERED or ARMED states and return to IDLE by receiving either the *RST or ABORT command. If INITiate:CONTinuous is on the system will remain in the ARMED state after completion of a trigger cycle, otherwise it will return to the IDLE state at the completion of the trigger cycle.

ABORT

The ABORT command resets the trigger system and places the instrument in the IDLE state without waiting for the completion of the trigger cycle.

INITiate:[ALL]

The INITiate command initializes the trigger cycle and places the system in the ARMED state. Once ARMED the instrument will ignore the TRIGger:SOURce command and any other commands that are defined as being in the IDLE layer. Once ARMED, the instrument operates in a partially overlapped mode, meaning passive commands and queries will still be processed. Commands that actively affect the frequency or triggering will be ignored and return error -200. The ABORT command and *RST will return the instrument to IDLE mode. The INITiate command is only needed when TRIGger:SOURce is set to EXTERNAL.

Example

```
INIT
```

Reset Condition

On reset, the system is in IDLE mode.

Query

This command is an event and there is no query form.

INITiate:CONTinuous <Boolean>

This command can configure the instrument to execute a single trigger cycle or continuous trigger cycles. This command has no effect when TRIGger:SOURce is FIBER. If INITiate:CONTinuous is set to off while the system is TRIGGERED the current trigger cycle will complete and the instrument will then enter the IDLE state unless an ABORT or *RST command is received

Example

```
INIT:CONT ON
```

Reset Condition

On reset, initiate continuous is OFF.

Query

```
INITiate:CONTinuous?
```

The query returns 1 or 0 into the output buffer.

- 1 is returned for continuous triggers
- 0 is returned for single triggers

TRIGger:DISable [NONE | SEQuence |SEGment |

This command allows the external triggers to be selectively disabled. This command only has an effect when the TRIGger:SOURce is set to EXTernal.

Example

```
TRIG:DIS SEG
```

Reset Condition

On reset the value is NONE.

Query

```
TRIGger:DISable]?
```

- The query returns NONE, SEQ or SEG

TRIGger:LOOP <numeric_value>

This command defines how many times the main portion of the frequency table will be repeated. If INITiate:CONTinuous is ON this value is ignored.

Example

```
TRIG:LOOP 1000
```

Reset Condition

On reset the value is 1.

Query

```
TRIGger:LOOP]?
```

- The query returns the numeric value as a string.

```
TRIGger:SOURce [NONE | IMMEDIATE | FIBER | PARAllel| EXTernal |
INTERNAL]
```

This command configures the instrument to step through the control table via the specified source.

1) NONE.

Stops the instrument from stepping.

2) IMMEDIATE

The trigger requires no external sources and the duration of each step is determined by the segment's dwell setting. The instrument will start stepping immediately and continuously (free run mode)

3) FIBER

Triggering is controlled from the fiber interface. . The dwell time is controlled from the fiber interface the internal dwell times are not used. The INITiate:CONTinuous and TRIGger:DISable commands ignored.

4) PARAllel

Triggering is controlled from the parallel interface. The dwell time is controlled from the copper interface the internal dwell times are not used. The INITiate:CONTinuous and TRIGger:DISable commands are ignored.

5) EXTernal

Trigger is controlled from the EXTernal trigger connectors.

7) INTERNAL

The trigger source is the *TRG SCPI command. Internal dwell settings from the frequency table are used. No external triggers are available.

Example

```
TRIG:SOURCE EXT
```

Reset Condition

On reset, TRIGger SOURce is set to IMMEDIATE

Query

```
TRIGger:SOURce?
```

The query returns the current setting for the Trigger source

UNIT Subsystem

UNIT:FREQuency [GHZ] MHZ | KHZ | HZ]

This command sets the default frequency unit for values downloaded to the instrument's control table via the MEMory subsystem.

- GHZ – Gigahertz 1×10^9 Hz
- MHZ – Megahertz 1×10^6 Hz

Example

UNIT:FREQ GHZ

Reset Condition

On reset the default units for frequency are GHZ

Query

UNIT:FREQuency?

Will put either “GHZ”, “MHZ”, “KHZ” or “HZ” in the output buffer.

UNIT:DWELL [S] MS | US | NS]

This command sets the default dwell time units for values downloaded to the instruments control table via the MEMory subsystem.

- MS – milliseconds
- US – microseconds
- NS - nanoseconds

Example

UNIT:DWELL MS

Reset Condition

On reset the default units for dwell time is milliseconds.

Query

UNIT:DWELL?

Will put either “S”, “MS”, “US” or “NS” in the output buffer

IEEE-488.2 Command Reference

*IDN?

This command returns the instrument identifier string “Innovative Technology <model num>, <TCP/IP address and subnet mask>, <serial number>, Firmware version, FPGA version.

*RST

This command will place Instrument into a known state.

- Any currently running trigger will abort and the synthesizer will return to idle mode.
- The Memory Control table will be emptied.
- No preamble will be set.
- The memory table description will be blank.
- The RF output power will be muted.
- Markers 1 and 2 will be set to NONE.
- The trigger source will be set to internal
- Both triggers will be enabled.
- The Loop count will be set to 1
- Continuous trigger cycles will be set to false
- Calibration state will be checked.
- Dwell units will be set to milliseconds.
- Frequency units will be set to GHz

*CLS

This command clears the SYSTem:Error queue and status registers

*TST?

This command performs the Built In Test. It will run both the Heartbeat BIT and the Frequency Dependent BIT. This test verifies that all major assemblies in the system are operating properly. Any errors detected will result in a message being placed in the output buffer. The message will have the form:

Assembly number, module tested, signal test, test name, lower voltage limit for test point, upper voltage limit for test point, frequency used for test, chip select of A/D, channel of A/D.

- 0 is returned when the BIT is successful
- 1 is returned when the test fails and SYSTem:ERRor? can be used to retrieve the error message(s).

Running the self test will overwrite the synthesizer's frequency control table with the BIT frequency control table. The synthesizer will be left with the RF Output muted when the test is complete. The self test takes several seconds to run.

*OPC?

This command instructs the instrument to put a "1" in the output buffer when all queued operations are complete.

*TRG

This command instructs the instrument to execute a trigger command when the TRIGger:SOURce is set to BUS. If the TRIGger:SOURce has any other setting Error -211, "Trigger Ignored" is returned. .

*STB?

This command instructs the instrument to put the status byte in the output buffer. A Zero in any position indicates a normal status and a 1 indicates an abnormal condition.

The status byte is defined as follows.

Bit 7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Rsvd	Rsvd	Rsvd	Rsvd	EXTREF	QUEUE	CAL	BIT

Where

BIT: Built In Test – A 1 indicates a failure occurred in the heartbeat BIT, which runs approximately every second. When the instrument is in the idle mode heartbeat BIT errors are not logged to the output queue. When the instrument is armed then messages are logged to the output queue until the instrument again enters the idle state.

CAL: Calibration – A 1 indicates the instrument does not have a valid calibration table.

QUEUE: - A 1 indicates there are messages in the output buffer queue.

EXTREF – A 1 indicates the system has detected that an external reference frequency is in use. Whenever the instrument determines that the external reference state changes a message is logged in the output buffer.

Innovative Technology Command Reference

This section contains special commands that are used for troubleshooting or special operations.

IT:MEM:TABLE:ROW <numeric_value>

This command puts the synthesizer in a special manual triggering mode where it steps to the row specified in the command. This command is not intended for standard operation and it doesn't obey all the settings for triggering and markers. This mode will step to any valid row number specified regardless of whether valid data exists in the waveform table for that row. Using this command to step to row 0 will cause the synthesizer to go to its default state.

Example

```
IT:MEM:TABLE:ROW 1
```

Reset Condition

On reset the instrument is not in the manual table mode.

IT:DEBUG:MSG [NONE|TACiturn|VERbose|TCP|FPGA]

This command configures the synthesizer to send special troubleshooting messages out the serial port. Using this command with any parameter other than NONE (the default) will alter the behavior of the ACS and the VERbose setting can slow down the ACS significantly.

- NONE– Only critical unexpected error conditions generate messages.
- TACiturn – The most useful message from this mode is a HEX dump of the instrument's control table every time a new table is downloaded.
- VERbose – Very wordy, very informative, very slow.
- TCP – Special messages about the processing of the received TCP commands.
- FPGA – Messages about commands and parameters being dispatched to the on-board FPGA.

Example

```
IT:DEBUG:MSG TAC
```

Reset Condition

On reset the instrument is set to NONE.

Programming Examples

```

/*****
/* This code is intended as example code only to show the basic approach to
/* programming the instrument. This code was not compiled. The basic commands
/* were pulled from the instrument's Virtual Front Panel code. As the original code
/* uses classes specific to Borland's C++ Builder IDE the code was changed to remove
/* references to interface widgets and most IDE specific classes. The class String
/* is Borland's version of the AnsiString class and is a reference counted class.
/* These examples use the following files
/* ScpiCommandConstants.h
/* ScpiMsg.cpp
/* ScpiMsg.h
/* These files are included as part of this package and an attempt was made to
/* modify them to be platform independent. They should be useful for any C++
/* implementation and permission is granted to the customer to use these files
/* as they see fit. If the customer uses these files the customer assumes
/* all responsibility for the code contained therein.
/* The following coding standards are used.
/* Class names start with C. Example CScpiExample.
/* Namespaces start with N NSomeNamespace
/* class methods are InitialCapCase
/* member variables are mixedCaseWithTrailingUnderscore_
/* parameters are mixedCase
/* local variables are lower_case_with_underscores
/* enumerated types use a prefix_ where the prefix represents
/* the initials of the type tag e.g. pml_SEQUENCE.
/* Some comments (especially in ScpiMsg.cpp and.h are written in CppDoc format
/* so that automatic documentation can be generated. This format supports HTML
/* and other application specific tags like @param, @throws, @returns etc.
/*****

```

CScpiExamples Class

```

//=====
// SampleDataRun
// Send all the commands necessary to set up the synth for a continuous freq sweep.
//=====
void CScpiExample::SampleDataRun() const
{
    //First abort any run that may be in progress
    SendTcpMsg(NScpiMsg::AbortCmd());
    SendSynthWaveform(); //give the synth a new waveform
    SendTcpMsg(NScpiMsg::TriggerDisableCmd(PTD_NONE)); //enable both triggers
    bool loop_continuous = true;
    int loop_count = 100; //won't do anything since looping continuously
    SendTcpMsg(NScpiMsg::LoopCmds(loop_continuous, loop_count)); //loop continuously
    SendTcpMsg(NScpiMsg::Marker1Cmd(pml_SEQUENCE)); //set marker 1 to sequence
    SendTcpMsg(NScpiMsg::Marker2Cmd(pm2_SEGMENT)); //set marker 2 to segment
    SendTcpMsg(NScpiMsg::CalStateCmd(true)); //turn on cal mode
    SendTcpMsg(NScpiMsg::OutputCmd(true)); //turn on the RF output
    SendTcpMsg(NScpiMsg::TriggerSourceCmd(NScpiMsg::PTS_FREE_RUN)); //Free Run the system
}

//=====
// SendSynthWaveform
// Turn off the RF Output, clear out any error messages, stop the Synth
// Send over the new table and wait for OPC.
//=====
void CScpiExample::SendSynthWaveform() const
{
    SendTcpMsg(NScpiMsg::OutputCmd(false)); // turn off the RF output
    SendTcpMsg(NScpiMsg::SystQueryAllCmd()); // get synth to send any queued errors out
    tcp
    TcpReadIntoMemo(Memo_Status); // put any msgs in memo
    //Abort any run that is going on
    SendTcpMsg(NScpiMsg::AbortCmd());
    //Typcially would loop here and send out all the table entries
    //For this demo we are just hard coding some data
    String scpi_msg = MEM_TABL_CMD + "1" + "4000,100;"; //Note the ; command terminator
    //Can save some TCP overhead by combining multiple rows into single message
    scpi_msg+=MEM_TABL_CMD + "2" + "4010,100;";
    SendTcpMsg(scpi_msg);
    //Tell the Synth we are done downloading a table.
    int rows_in_table = first_row - last_row; //want it as a negative number to signal
    end of table.
    String end_table_cmd = MEM_TABL_CMD+"-2";
}

```

```

SendTcpMsg(end_table_cmd);
//Wait for the Synth to digest and encode the entire table.
WaitForOperationComplete();
//Check and display any errors
SendTcpMsg(NScpiMsg::SystQueryAllCmd());
String response_msg;
TcpRead(response_msg);
printf (response_msg);
}

//=====
// Load back the waveform that currently resides in the Synthesizer.
//=====
void CScpiExample::LoadSynthWaveform
{
String full_data = "";
//First get rid of any data sitting on the TCP
TcpRead(full_data);
full_data = "";
SendTcpMsg(NScpiMsg::WaveformQuery());
ReadSynthTable (full_data);
if (full_data.Length() > 0 )
{
//Do what you're gonna do
}
else
{
ShowMessage("No data could be loaded from the synthesizer.");
}
// end check ok_to_continue
}

//=====
// ReadSynthTable() can be used to read back the tabular data for the waveform
// table or the cal data.
//=====
void CScpiExample::ReadSynthTable(String& fullData)
{
String single_read_data="";
bool end_of_table = false;
while (!end_of_table)
{
if (TcpRead(single_read_data ) )
{
if (single_read_data.Pos(";") == 1) end_of_table = true;
else fullData += single_read_data;
}
else //didn't get any data
{
end_of_table = true;
}
}
}

//=====
// Read all data waiting out on the TCP buffer and just throw it away. In a real
// application this data should go into a display area for the operator
//=====
void CScpiExample::TcpReadAllData( )
{
String rcv_string;
bool data_avail = TcpRead(rcv_string);
while (data_avail)
{
char* p_char;
p_char = strtok(rcv_string.c_str(), "\n");
while (p_char)
{
p_char = strtok(NULL, "\n");
}
//see if there is more data waiting.
data_avail = TcpRead(rcv_string);
}
}

//=====
// This is just a stub, whatever classes are available for TCP needed to be used

```

```

// here. The TCP read command should have some way to wait for data. The wait
// should generally be in the area of a few hundred milliseconds.
//=====
bool CScpiExample::TcpRead(String& dataRead, int msToWait)
{
    const int MAX_BUFFER = 8191;
    // if ( Tcp>WaitForData(msToWait)) //make sure there is something to read
    // {
    //     char rcv_buf[MAX_BUFFER];
    //     int bytes_read = Tcp->ReceiveBuf(rcv_buf, MAX_BUFFER);
    //     if we didn't fill the buffer, assume end of message and add string term.
    //     if (bytes_read < 0) return false;
    //     if (bytes_read < MAX_BUFFER) rcv_buf[bytes_read] = '\0';
    //     dataRead = rcv_buf;
    //     return true;
    // }
    // else
    // {
    //     return false;
    // }
}

//=====
// WaitForOperationComplete. On any lengthy synthesizer operation we will send
// the *OPC? command and keep waiting until we get a "1" back.
// Adding user feedback in this loop and callbacks to allow the app to process
// other messages is a good idea.
//=====
void CScpiExample::WaitForOperationComplete()
{
    const int WAIT_IN_MS = 200;
    String opc_response;
    TcpRead(opc_response); //clear out any data on the tcp output buffer
    bool no_data_avail = true;
    SendTcpMsg(STAR_OPC_QUERY_CMD);
    do
    {
        no_data_avail = !TcpRead(opc_response, WAIT_IN_MS);
        //Application->ProcessMessages(); //The C++ Builder callback
    } while (no_data_avail) ;
}

//=====
// This example shows how to do a full cal. If a partial cal is needed then
// the proper enumerated value (see ePopupCalType in ScpiMsg.h) should be used.
//=====
void CScpiExample::Button_CalibrateClick()
{
    const int MIN_RESPONSE_LEN = 2;
    const int HALF_SEC_WAIT = 500;
    SendTcpMsg(NScpiMsg::CalCmd(NScpiMsg::pct_ALL ));
    WaitForOperationComplete();
    SendTcpMsg(NScpiMsg::SystQueryAllCmd());
    String cal_result="";
    String temp_result="";
    while (TcpRead(temp_result, HALF_SEC_WAIT) )
    {
        cal_result += temp_result;
    }
    if (cal_result.Length() > MIN_RESPONSE_LEN ) printf(cal_result);
}

```

NScpiMsg Namespace

```
//=====
// Namespace that holds the trivial routines that send the proper SCPI strings
// to the Synthesizer for various function controls.
// @see "ScpCommandConstants.h for defs of the commands." <p>
//=====
```

ScpiMsg.cpp

```
#include "ScpiMsg.h"
//-----

//=====
// CalCmd - Send the command to cause the synthesizer to calibrate either
// a portion or the entire bandwidth. You can put a floating reference to
// the files used with {@files}.
// @param calType the popup selection - see enumerated type ePopupCalType
// @return the SCPI command to do the selected cal or an empty string on an error
// @throws nothing
//=====
const String NScpiMsg::CalCmd(int calType)
{
    String scpi_msg = CAL_CMD;
    switch (calType)
    {
        case pct_ALL:
            scpi_msg += " ALL";
            break;
        case pct_UUP:
            scpi_msg += " UUP";
            break;
        case pct_UDOWN:
            scpi_msg += " UDOWN";
            break;
        case pct_LOBAND:
            scpi_msg += " LOBAND";
            break;
        case pct_HIBAND:
            scpi_msg += " HIBAND";
            break;
        default:
            scpi_msg = "";
    }
    return scpi_msg;
}

//=====
// CalStateCmd - User can run calibrated or uncalibrated
// @param calOn bool true is cal on, false is cal off
// @returns the scpi command to set cal to the desired state
//=====
const String NScpiMsg::CalStateCmd(bool calOn)
{
    String scpi_msg = "";
    scpi_msg.sprintf("%s %s", CAL_STATE_CMD, calOn?" ON":" OFF");
    return scpi_msg;
}

//=====
// DwellUnitCmd VFP only supports the MS and US options.
// @param useUs bool true sets default to microseconds false uses milliseconds
// @returns the scpi command to set dwell units to the desired state
//=====
const String NScpiMsg::DwellUnitCmd(bool useUs)
{
    String scpi_msg = UNIT_DWELL_CMD;
    if (useUs) scpi_msg += " US";
    else scpi_msg += " MS";
    return scpi_msg;
}

//=====
```

```

// FreqUnitCmd - VFP only supports the GHZ or MHZ options
// @param useMhz bool true tells system to use MHZ false uses GHZ
//=====
const String NScpIMsg::FreqUnitCmd(bool useMhz)
{
    String scpi_msg = UNIT_FREQ_CMD;
    if (useMhz) scpi_msg += " MHZ";
    else scpi_msg += " GHZ";
    return scpi_msg;
}

//=====
// Send the commands to initialize the synth to a known state and make it
// correspond to the virtual front panel (VFP) settings. Intended to be called
// at startup time from the Main form.
// @returns the scpi string to reset the instrument to the default VFP state
//=====
const String NScpIMsg::InitCmds()
{
    String scpi_msg = STAR_RST_CMD;
    scpi_msg += ";";
    scpi_msg += STAR_CLS_CMD;
    scpi_msg += ";";
    scpi_msg += FreqUnitCmd(true);
    scpi_msg += ";;"; //NOTE the extra colon to start the cmd back at root
    //The scpi standard requires this otherwise after UNIT:FREQ you are still
    //in the UNIT subsystem. The colon makes you start over again at root.
    scpi_msg += OutputCmd(false);
    scpi_msg += ";";
    return scpi_msg;
}

//=====
// LoopCmds - sends both the Loop continues and loop count commands
// @param continuousON bool if true continuous looping is on if false then
// continuous looping is turned off and a set number of loops can be executed
// @param loopCount int if first param false then this sets the number of loops
// @returns the scpi string to set up the desired looping.
//=====
const String NScpIMsg::LoopCmds(bool continuousOn, int loopCount)
{
    String scpi_msg = INIT_CONT_CMD;
    if (continuousOn) scpi_msg += " ON";
    else
    {
        scpi_msg += " OFF";
        scpi_msg += TRIG_LOOP_CMD;
        scpi_msg += " ";
        //If no valid number it will default to 1.
        scpi_msg += loopCount;
    }
    return scpi_msg;
}

//=====
// Marker1Cmd Send out the command for marker 1
// @param marker1Type the popup selection (see enumerated type)
// @return the SCPI command or an empty string on an error
//=====
const String NScpIMsg::Marker1Cmd(int marker1Type)
{
    String scpi_msg = OUTP_MARKER1_CMD;
    switch (marker1Type)
    {
        case pml_NONE:
            scpi_msg += " NONE";
            break;
        case pml_SEQUENCE:
            scpi_msg += " SEQ";
            break;
        case pml_TABLE:
            scpi_msg += " TABLE";
            break;
        default:
            scpi_msg = "";
    }
    return scpi_msg;
}

```

```

//=====
// Marker2Cmd Send out the command for Marker 2
// @param marker2Type the popup selection
// @return the SCPI command or an empty string on an error
//=====
const String NScpiMsg::Marker2Cmd(int marker2Type)
{
    String scpi_msg = OUTP_MARKER2_CMD;
    switch (marker2Type)
    {
        case pm2_NONE:
            scpi_msg += " NONE";
            break;
        case pm2_SEGMENT:
            scpi_msg += " SEG";
            break;
        case pm2_TABLE:
            scpi_msg += " TABLE";
            break;
        default:
            scpi_msg = "";
    }
    return scpi_msg;
}

//=====
// OutputCmd output on/off command AKA MUTE and UNMUTE
//=====
const String NScpiMsg::OutputCmd(bool turnOn)
{
    String scpi_msg = "OUTPUT ";
    if (turnOn) scpi_msg += "ON";
    else scpi_msg += "OFF";
    return scpi_msg;
}

//=====
// PreambleRow Set the preamble Row count
// @param preambleRow int the Row in the table where the preamble ends
// @returns the scpi command to set up the preamble row
//=====
const String NScpiMsg::PreambleRowCmd(int preambleRow)
{
    String scpi_msg = MEM_TABL_CONT_PRE_ROW_CMD;
    scpi_msg += " "; //need whitespace before parameter
    scpi_msg += preambleRow;
    return scpi_msg;
}

//=====
// TriggerDisableCmd - You can disable one or the other or neither trigger.
// @param disableCmd the trigger disable type see enumerated type in ScpiMsg.h
// @return the SCPI command or an empty string on an error
//=====
const String NScpiMsg::TriggerDisableCmd(int disableCmd)
{
    String scpi_msg = TRIG_DIS_CMD;
    switch (disableCmd)
    {
        case ptd_NONE: //The VFP enables both so we disable NONE
            scpi_msg += " NONE";
            break;
        case ptd_SEGMENT: //the VFP enables the SEQUENCE only so disable SEGMENT
            scpi_msg += " SEG";
            break;
        case ptd_SEQUENCE:
            scpi_msg += " SEQ";
            break;
        default:
            scpi_msg="";
    }
    return scpi_msg;
}

//=====
// TriggerSourceCmd uses the local enum to create parameter for TRIG:SOUR cmd
// certain types of trigger source commands will start the synthesizer running so

```

```

// this will typically be the last command sent after sending all the other setup
// commands.
// @param trigSourceType the popup choice item index.
// @return the SCPI command or an empty string on an error
//=====
const String NScpiMsg::TriggerSourceCmd(int trigSourceType)
{
    String scpi_msg = TRIG_SOUR_CMD;
    switch (trigSourceType)
    {
        case pts_FREE_RUN:
            scpi_msg += " IMM";
            break;
        case pts_INTERNAL:
            scpi_msg += " INT";
            scpi_msg += "\n INIT \n*TRG"; //add the arming and trigger cmds
            break;
        case pts_EXTERNAL:
            scpi_msg += " EXT";
            scpi_msg += "\n INIT"; //in external add just the arming cmd
            break;
        case pts_FIBER:
            scpi_msg += " FIBER";
        case pts_PARALLEL:
            scpi_msg += " PARA";
        break;
        default:
            scpi_msg = "";
    }
    return scpi_msg;
}

//=====
// WaveformDescription send the description for the waveform table
// truncating if necessary. If no description, don't send the command
// @param formDescription String reference; the waveform description
// @returns the SCPI command or an empty string on an error
//=====
const String NScpiMsg::WaveformDescriptionCmd(const String& formDescription)
{
    const int MAX_CMD_LEN = 255;
    String scpi_msg = MEM_TABL_CONT_DESC_CMD;
    String user_desc =formDescription.Trim();
    //make sure there is a description to send otherwise you get an incorrect num of
    params
    //for sending the description command but no description (white space isn't
    counted as a parameter
    //by the scpi parser.
    if (user_desc.Length() > 0)
    {
        scpi_msg += " "+user_desc;
        //Limit to max command length in case long description entered
        if (scpi_msg.Length() > MAX_CMD_LEN) scpi_msg =
            scpi_msg.SubString(0,MAX_CMD_LEN-2);
    }
    else
    {
        scpi_msg = "";
    }
    return scpi_msg;
}

```

ScpiMsg.h

```

#ifndef ScpiMsgH
#define ScpiMsgH
//-----
#include "ScpiCommandConstants.h"
//=====
// Namespace that holds the trivial routines that send the proper SCPI strings
// to the Synthesizer for various function controls.
// @see "ScpiCommandConstants.h for defs of the commands." <p>
//=====
namespace NScpiMsg
{

```

```

enum ePopupTrigSource {pts_FREE_RUN, pts_INTERNAL, pts_EXTERNAL, pts_FIBER,
pts_PARALLEL};
enum ePopupTrigDisable{ptd_NONE, ptd_SEGMENT, ptd_SEQUENCE };
enum ePopupCalType {pct_ALL, pct_UUP, pct_UDOWN, pct_LOBAND, pct_HIBAND, };
enum ePopupMarker1 {pm1_NONE, pm1_SEQUENCE, pm1_TABLE};
enum ePopupMarker2 {pm2_NONE, pm2_SEGMENT, pm2_TABLE};
const String AdMonitorCmd() {return IT_AD_MONITOR_CMD;};
const String AdMonitorVoltageCmd(const String& onOffMsg);
const String AbortCmd() {return ABORT_CMD;};
const String BuiltInTestCmd() {return STAR_TST_QUERY_CMD;};
const String CalCmd(int calType);
const String CalStateCmd(bool calOn);
const String DwellUnitCmd(bool useUs);
const String InitCmds();
const String FreqUnitCmd(bool useMhz);
const String HwInfoCmd() {return IT_HW_INFO_CMD;};
const String HwInfoQueryCmd() {return IT_HW_INFO_QUERY_CMD;};
const String LoopCmds(bool continuousOn, int loopCount=1);
const String Marker1Cmd(int marker1Type);
const String Marker2Cmd(int marker2Type);
const String OutputCmd (bool turnOn);
const String PreambleRowCmd(int preambleRow);
const String StatusByteQuery() {return STAR_STB_QUERY_CMD;};
const String SystQueryAllCmd() {return SYST_ERR_ALL_CMD;};
const String TableStepRowCmd(int stepRow);
const String TriggerSourceCmd(int trigSourceType);
const String TriggerDisableCmd(int disableCmd);
const String WaveformDescriptionCmd(const String& formDesc);
const String WaveformQuery() {return MEM_TABL_CONT_QUERY_CMD;};
};
#endif

```

NScpiCommands Namespace

ScpiCommandConstants.h

```

#ifndef ScpiCommandConstants_H
#define ScpiCommandConstants_H
//NOTE: Commands are shown in their complete form. End Use application can remove all
//the lower case letters from all the constants below to increase TCP/IP communication
//speed. This has already been done for the MEM:TABL:CONT command as it can be used in
//very large loops.

namespace NScpiCommands
{
    //-----
    //IEEE488.2 commands
    //numbered 0-13
    const char STAR_CLS_CMD [] = "*CLS";
    const char STAR_ESE_CMD [] = "*ESE";
    const char STAR_ESE_QUERY_CMD [] = "*ESE?";
    const char STAR_ESR_QUERY_CMD [] = "*ESR?";
    const char STAR_IDN_QUERY_CMD [] = "*IDN?";
    const char STAR_OPC_CMD [] = "*OPC";
    const char STAR_OPC_QUERY_CMD [] = "*OPC?";
    const char STAR_RST_CMD [] = "*RST";
    const char STAR_SRE_CMD [] = "*SRE";
    const char STAR_SRE_QUERY_CMD [] = "*SRE?";
    const char STAR_STB_QUERY_CMD [] = "*STB?";
    const char STAR_TST_QUERY_CMD [] = "*TST?";
    const char STAR_WAI_CMD [] = "*WAI";
    const char STAR_TRG_CMD [] = "*TRG";
    //-----
    // Required SCPI commands (see SCPI Standard V1999.0 ch 4.2.1)
    //14-25
    const char SYST_ERR_CMD [] = "SYSTEM:ERRor[:NEXT]?";
    const char SYST_ERR_ALL_CMD [] = "SYSTEM:ERRor:ALL?";
    const char SYST_VERS_CMD [] = "SYSTEM:VERSion?";
    const char STAT_OPER_CMD [] = "STATus:OPERation[:EVENT]?";
    const char STAT_OPER_COND_QUERY_CMD [] = "STATus:OPERation:CONDition?";
    const char STAT_OPER_ENAB_CMD [] = "STATus:OPERation:ENABLE";
    const char STAT_OPER_ENAB_QUERY_CMD [] = "STATus:OPERation:ENABLE?";
    const char STAT_QUES_COND_CMD [] = "STATus:QUEStionable[:EVENT]?";
    const char STAT_QUES_COND_QUERY_CMD [] = "STATus:QUEStionable:CONDition?";
    const char STAT_QUES_ENAB_CMD [] = "STATus:QUEStionable:ENABLE";
    const char STAT_QUES_ENAB_QUERY_CMD [] = "STATus:QUEStionable:ENABLE?";
    const char STAT_PRES_CMD [] = "STATus:PRESet";
    //-----
    //MEMory subsystem
    //26-33
    const char MEM_TABL_CONT_CMD [] = "MEM:TABL:CONT";
    const char MEM_TABL_CONT_QUERY_CMD [] = "MEMory:TABLE:CONTRol?";
    const char MEM_TABL_CONT_PRE_ROW_CMD [] = "MEMory:TABLE:CONTRol:PREAmble:ROWs";
    const char MEM_TABL_CONT_PRE_ROW_QUERY_CMD [] = "MEMory:TABLE:CONTRol:PREAmble:ROWs?";
    const char MEM_TABL_CONT_DESC_CMD [] = "MEMory:TABLE:CONTRol:DESCRiption";
    const char MEM_TABL_CONT_DESC_QUERY_CMD [] = "MEMory:TABLE:CONTRol:DESCRiption?";
    const char MEM_TABLE_CONT_BAND_CMD [] = "MEMory:TABLE:CONTRol:BAND";
    const char MEM_TABLE_CONT_BAND_QUERY_CMD [] = "MEMory:TABLE:CONTRol:BAND?";
    //-----
    // OUTPut subsystem
    //34-39
    const char OUTP_CMD [] = "OUTPut[:STATE]";
    const char OUTP_QUERY_CMD [] = "OUTPut[:STATE]?";
    const char OUTP_MARKER1_CMD [] = "OUTPut:MARKER1";
    const char OUTP_MARKER1_QUERY_CMD [] = "OUTPut:MARKER1?";
    const char OUTP_MARKER2_CMD [] = "OUTPut:MARKER2";
    const char OUTP_MARKER2_QUERY_CMD [] = "OUTPut:MARKER2?";
    //-----
    // TRIG subsystem
    //40-50
    const char TRIG_SOUR_CMD [] = "TRIGger:SOURCE";
    const char TRIG_SOUR_QUERY_CMD [] = "TRIGger:SOURCE?";
    const char TRIG_DIS_CMD [] = "TRIGger:DISable";
    const char TRIG_DIS_QUERY_CMD [] = "TRIGger:DISable?";
    const char TRIG_LOOP_CMD [] = "TRIGger:LOOP";
    const char TRIG_LOOP_QUERY_CMD [] = "TRIGger:LOOP?";
    const char ABORT_CMD [] = "ABORT";
}

```

```

const char INIT_CMD [] = "INITiate[:ALL] ";
const char INIT_IMM_CMD [] = "INITiate:IMMediate";
const char INIT_CONT_CMD [] = "INITiate:CONTinuous";
const char INIT_CONT_QUERY_CMD [] = "INITiate:CONTinuous?";
//-----
// CAL subsystem
//51-56
const char CAL_CMD [] = "CALibration";
const char CAL_QUERY_CMD [] = "CALibration?";
const char CAL_STATE_CMD [] = "CALibration:STATE";
const char CAL_STATE_QUERY_CMD [] = "CALibration:STATE?";
const char CAL_DATA_CMD [] = "CALibration:DATA";
const char CAL_DATA_QUERY_CMD [] = "CALibration:DATA?";
//-----
// UNIT subsystem
//57-60
const char UNIT_FREQ_CMD [] = "UNIT:FREQuency";
const char UNIT_FREQ_QUERY_CMD [] = "UNIT:FREQuency?";
const char UNIT_DWELL_CMD [] = "UNIT:DWELL";
const char UNIT_DWELL_QUERY_CMD [] = "UNIT:DWELL?";
}

```